Architecture-Based and Uncertainty-Aware Confidentiality Analysis

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Sebastian Hahner

Tag der mündlichen Prüfung: 09. Dezember 2024
1. Referent: Prof. Dr. Ralf Reussner, Karlsruher Institut für Technologie (KIT)
2. Referent: Prof. Dr. Jesper Andersson, Linnaeus University (LNU)

“Research is as much about ideas as about communicating them.”

— *author unknown*

Abstract

In this dissertation, we investigate architecture-based confidentiality analysis under uncertainty. We present a classification of uncertainty regarding confidentiality, a catalog of uncertainty sources, an uncertainty impact analysis that propagates uncertainty to predict its impact on the system's confidentiality, and uncertainty-aware confidentiality analyses that identify violations of confidentiality requirements with respect to uncertainty.

In our interconnected world, an increasing amount of data is exchanged, processed and stored every day. This enables a multitude of modern applications in many domains, from mobility to healthcare, but it also presents significant challenges. A particular challenge is confidentiality, which requires that information is not made available to unauthorized individuals or organizations. The rise of cybercrime and the ubiquity of data breaches underline the relevance of confidentiality, as its loss can significantly impact stakeholder wealth and social acceptance. As demanded by approaches like privacy and security by design, confidentiality should be addressed and analyzed early in the development, e.g., on the abstraction of the software's architecture. Here, numerous architecture-based analysis approaches have been proposed. Based on the assessment of architectural models, these analyses can accurately and early identify violations of confidentiality requirements.

However, today's software systems are neither built nor operated in isolation and are subject to uncertainty within the software system, its design, or its environment. Uncertainty, often referred to as a lack of knowledge or information, is inherently unavoidable in modern, interconnected software systems. When not handled comprehensively, confidentiality violations can occur due to uncertainty that voids previous analysis results, e.g., because influences from the system context have not been adequately considered or users and third parties behave differently than assumed. Addressing this problem requires new approaches to identify and document uncertainty sources, as one cannot analyze what one does not know. Software architects face challenges both in the inspection of software architectures and in assessing confidentiality under uncertainty.

Although architecture evaluation under uncertainty has been comprehensively researched, existing approaches focus on other quality properties like performance, cost, or reliability but do not consider confidentiality. On the opposite, existing design time confidentiality analyses fall short of considering the impact of uncertainty on confidentiality. Recent surveys and road maps also find a lack of uncertainty management, accurate representation of uncertainty sources, and uncertainty-aware end-to-end approaches.

In this dissertation, we address this lack of simultaneous consideration of uncertainty and confidentiality in architecture-based analysis. We present an approach to identifying, classifying, propagating, and analyzing uncertainty and its impact on a system's confidentiality using the architectural abstraction. First, we present a classification of uncertainty regarding confidentiality that provides the terminology to understand and discuss the relation between uncertainty and confidentiality. We complement this classification with a catalog of uncertainty sources that simplifies their identification and addresses the lack of awareness. Second, we introduce an architecture-based uncertainty impact analysis. By propagating uncertainty within the software architecture, software architects can predict potential confidentiality violations with minimal effort. Third, we present uncertainty-aware confidentiality analyses. These analyses consider uncertainty sources as first-class entities in the software architecture and identify confidentiality violations due to uncertainty.

We validate our contributions using six evaluation scenarios and by conducting two user studies. In addition to investigating the quality of our classification, we evaluate the usability of our catalog approach and the accuracy of all of our analyses. Furthermore, we inspect the effort reduction of our uncertainty impact analysis and the scalability of the confidentiality analyses under uncertainty. The results of the user studies indicate that combining the classification with our catalog of uncertainty sources helps to identify and describe uncertainty sources. Afterward, the uncertainty impact analysis can accurately predict the impact with an F1-score of 0.88 while greatly reducing the effort by 86% compared to manual analysis. Last, uncertainty-aware confidentiality analyses accurately identify confidentiality violations with high precision and recall while significantly reducing the analysis runtime compared to the state of the art.

Our work benefits software architects by helping them more accurately identify confidentiality violations by considering uncertainty within the software and its context. Applying our tool-supported and automated analyses requires both less effort and less expertise. Last, it has already been shown that our findings can be generalized to other research concerns, e.g., regarding uncertainty propagation and interaction in self-adaptive systems.

Zusammenfassung

In dieser Dissertation untersuchen wir die architekturbasierte Analyse von Vertraulichkeit unter Ungewissheit. Wir präsentieren eine Klassifikation von Ungewissheit mit Bezug auf Vertraulichkeit, einen Katalog von Ungewissheitsquellen, eine Ungewissheitsauswirkungsanalyse, die Ungewissheit propagiert, um ihre Auswirkungen auf die Vertraulichkeit eines Systems vorherzusagen, und Vertraulichkeitsanalysen unter Ungewissheit, die Verletzungen von Vertraulichkeitsanforderungen unter Berücksichtigung von Ungewissheit identifizieren.

In unserer vernetzten Welt werden täglich immer größere Datenmengen ausgetauscht, verarbeitet und gespeichert. Dies ermöglicht eine Vielzahl moderner Anwendungen in vielen Bereichen, von Mobilitätssystem bis zur Gesundheitsbranche, bringt aber auch erhebliche Herausforderungen mit sich. Eine besondere Herausforderung ist die Vertraulichkeit, welche fordert, dass Informationen nicht an unbefugte Personen oder Organisationen weitergegeben werden dürfen. Der Anstieg der Cyberkriminalität und die Allgegenwart von Datenschutzverletzungen unterstreichen die Relevanz von Vertraulichkeit, da ihr Verlust erhebliche Auswirkungen auf den Wohlstand der Betroffenen und die gesellschaftliche Akzeptanz haben kann. Wie von Ansätzen wie Sicherheit und Datenschutz durch Technikgestaltung gefordert, sollte die Vertraulichkeit bereits in der frühen Entwicklung berücksichtigt und analysiert werden, z. B. auf der Abstraktionsebene der Softwarearchitektur. Hier wurden zahlreiche architekturbasierte Analyseansätze vorgeschlagen. Auf der Grundlage der Bewertung von Architekturmodellen können diese Analysen Verstöße gegen Vertraulichkeitsanforderungen genau und frühzeitig erkennen.

Heutige Softwaresysteme werden jedoch weder isoliert entwickelt noch betrieben und unterliegen Ungewissheiten innerhalb des Softwaresystems, seines Entwurfs, oder seiner Umgebung. Ungewissheit, die oft als Mangel an Wissen oder Informationen bezeichnet wird, ist in modernen, vernetzten Softwaresystemen unvermeidlich. Wenn sie nicht umfassend gehandhabt wird, kann es zu Vertraulichkeitsverletzungen kommen, weil die Ungewissheit frühere Analyseergebnisse invalidiert, z. B. weil Einflüsse aus dem Systemkontext nicht angemessen berücksichtigt wurden oder weil sich Benutzer und Dritte anders verhalten als ursprünglich angenommen. Die Lösung dieses Problems erfordert neue Ansätze zur Identifizierung und Dokumentation von Ungewissheitsquellen, da man nicht analysieren kann, was man nicht kennt. Softwarearchitektinnen und Softwarearchitekten stehen vor Herausforderungen sowohl bei der Prüfung von Softwarearchitekturen als auch bei der Bewertung der Vertraulichkeit unter Ungewissheit.

Obwohl die Bewertung von Architekturen unter Ungewissheit umfassend erforscht wurde, konzentrieren sich bestehende Ansätze auf andere Qualitätseigenschaften wie Leistung, Kosten oder Zuverlässigkeit, berücksichtigen aber nicht die Vertraulichkeit. Umgekehrt werden bei bestehenden Analysen der Vertraulichkeit während der Entwurfszeit die Auswirkungen der Ungewissheit auf die Vertraulichkeit nicht berücksichtigt. Jüngste Studien zeigen auch einen Mangel an Ungewissheitsmanagement, genauer Darstellung von Ungewissheitsquellen und Ende-zu-Ende-Ansätzen, die Ungewissheit berücksichtigen.

In dieser Dissertation befassen wir uns mit der fehlenden gleichzeitigen Berücksichtigung von Unsicherheit und Vertraulichkeit in architekturbasierten Analysen. Es wird ein Ansatz zur Identifizierung, Klassifikation, Propagation und Analyse von Ungewissheit und deren Auswirkungen auf die Vertraulichkeit eines Systems unter Verwendung der Architekturabstraktion vorgestellt. Zunächst stellen wir eine Klassifikation der Ungewissheit in Bezug auf die Vertraulichkeit vor, die die Terminologie für das Verständnis und die Diskussion der Beziehung zwischen Ungewissheit und Vertraulichkeit zur Verfügung stellt. Wir ergänzen diese Klassifikation durch einen Katalog von Ungewissheitsquellen, der deren Identifizierung vereinfacht und den Mangel an Bewusstsein behebt. Zweitens führen wir eine architekturbasierte Analyse der Auswirkungen von Ungewissheit ein. Durch die Propagation von Ungewissheit innerhalb der Softwarearchitektur können Softwarearchitektinnen und Softwarearchitekten potenzielle Vertraulichkeitsverletzungen mit minimalem Aufwand vorhersagen. Drittens stellen wir auf Ungewissheit basierende Vertraulichkeitsanalysen vor. Diese Analysen betrachten Ungewissheitsquellen als Entitäten erster Klasse in der Softwarearchitektur und identifizieren Vertraulichkeitsverletzungen aufgrund von Ungewissheit.

Wir validieren unsere Beiträge anhand von sechs Evaluationsszenarien und zwei Nutzerstudien. Zusätzlich zur Untersuchung der Qualität unserer Klassifikation evaluieren wir die Benutzerfreundlichkeit unseres Katalogansatzes und die Genauigkeit aller unserer Analysen. Darüber hinaus untersuchen wir die Aufwandsreduktion unserer Ungewissheitsauswirkungsanalyse und die Skalierbarkeit der Vertraulichkeitsanalysen unter Ungewissheit. Die Ergebnisse der Nutzerstudien zeigen, dass die Kombination der Klassifikation mit unserem Katalog der Ungewissheitsquellen hilft, Ungewissheitsquellen zu identifizieren und zu beschreiben. Anschließend kann die Ungewissheitsauswirkungsanalyse die Auswirkungen mit einem F1-Score von 0,88 genau vorhersagen und gleichzeitig den Aufwand im Vergleich zur manuellen Analyse um 86% reduzieren. Schließlich werden durch die Analyse von Ungewissheiten in Bezug auf die Vertraulichkeit Verstöße gegen die Vertraulichkeit mit hoher Präzision und Ausbeute identifiziert, während die Laufzeit der Analyse im Vergleich zum Stand der Technik erheblich reduziert wird.

Unsere Arbeit kommt Softwarearchitektinnen und Softwarearchitekten zugute, indem sie ihnen hilft, Vertraulichkeitsverletzungen genauer zu identifizieren, indem sie die Ungewissheit innerhalb der Software und ihres Kontexts berücksichtigt. Die Anwendung unserer werkzeuggestützten und automatisierten Analysen erfordert sowohl weniger Aufwand als auch weniger Expertenwissen. Schließlich wurde bereits gezeigt, dass unsere Erkenntnisse auf andere Forschungsfragen verallgemeinert werden können, z. B. in Bezug auf die Propagation und Interaktion von Ungewissheit in selbstadaptiven Systemen.

Danksagung

1560 Tage.

1560 Tage liegen zwischen meinem ersten Arbeitstag als wissenschaftlicher Mitarbeiter am Karlsruher Institut für Technologie (KIT) und der erfolgreichen Verteidigung meiner Dissertation. Zum Glück musste ich den Weg an den allermeisten dieser Tage nicht allein gehen—wofür ich mich nachfolgend bedanken möchte.

Zuallererst und vor allem möchte ich mich bei meiner Frau Alina bedanken, die mich stets unterstützt hat, egal, ob ich gerade übermotiviert und viel zu fokussiert geforscht oder verzweifelt alles infrage gestellt habe. Sie hat mir den bei einer Promotion dringend notwendigen Rückhalt gegeben, jederzeit an mich geglaubt und somit diesen großen Erfolg überhaupt erst möglich gemacht. Ebenfalls möchte ich meinen Eltern von ganzem Herzen für ihre Unterstützung und unermüdliche Ermutigung danken. Bedanken möchte ich mich auch bei meinen Freunden und Kollegen außerhalb der Universität, denen ich auch in den letzten vier Jahren nicht die Aufmerksamkeit geschenkt habe, die sie verdient hätten.

Während meiner Zeit am Lehrstuhl hatte ich die Möglichkeit mit vielen motivierten und äußerst fähigen Studierenden zusammen zu arbeiten. Hier möchte ich mich insbesondere bei Niko bedanken, mit dem ich als Allererstes an meinen neuen Ideen geforscht habe, und bei Felix, der mir am Schluss tatkräftig geholfen hat, die prototypische Umsetzung meiner Konzepte zu finalisieren. Ein weiteres Danke für größere und kleinere Beiträge zu meiner Forschung geht an Gabriel, Tizian, Denis, Daniel, Oliver, Anne-Kathrin, Alexander, Tom, Lilian, Benjamin und Jonas. Ich hatte während meiner Promotion zahlreiche Gelegenheiten zu reisen und meinen Horizont zu erweitern. Ich möchte mich insbesondere bei Simon, Marc und Mario für die neuen Einblicke bedanken. Ebenfalls bedanken möchte ich mich bei den Organisatoren und Teilnehmern des Bertinoro USAS Seminars und der Dagstuhl Seminare, welche mir besonders in Erinnerung geblieben sind.

Natürlich dürfen in dieser Danksagung meine großartigen Kolleginnen und Kollegen am Lehrstuhl nicht fehlen. Zuerst möchte ich mich bei Stephan und Maximilian bedanken, die nicht nur wichtige Vorarbeit für meine Forschung geleistet haben, sondern mich auch herzlich am Lehrstuhl aufgenommen und insbesondere in der Anfangszeit angeleitet haben. Ein weiterer Dank gebührt meinen Bürokollegen Frederik, Sandro, Nicolas und Nils für den regen—und oft genug nicht fachbezogenen—Austausch. Insbesondere bei Nicolas möchte ich mich für die jahrelange extrem erfolgreiche und spaßige Zusammenarbeit bedanken: Wie wir innerhalb so kurzer Zeit von Grund auf ein neues Analyse-Framework samt Team aufgebaut haben, war mindestens genauso ein Highlight wie die vielen gemeinsamen Kaffeepausen und Weinwanderungen. Auch bei Timur und Larissa möchte ich mich bedanken:

Zu dritt haben wir nicht nur fast zeitgleich mit der Promotion begonnen, sondern diese auch fast gleichzeitig abgeschlossen; der regelmäßige Austausch im alltäglichen Wahnsinn hat sehr gutgetan. Zudem habe ich mit Timur—und später Dominik und Robin—zusammen den Plagiatsdetektor JPlag wiederbelebt und von einem Software-Archäologieprojekt in ein erfolgreiches und aktives Open-Source-Projekt verwandelt. Das war eine steile Entwicklung und ich bin extrem stolz darauf, was wir daraus gemacht haben. Steil waren auch die Wände, die ich während meiner Promotion erklommen habe—nicht nur im übertragenen Sinne. Hier möchte ich Tobias danken, der mich nicht nur zu einem neuen Hobby geführt, sondern auch während des Diss-Endspurts unterstützt hat.

Ein weiteres Danke gebührt den übrigen Lehrstuhl-Seniors. Hier möchte ich zuerst Anne, Christopher und Erik für die zahlreichen Hinweise, Hilfestellungen und Möglichkeiten danken. Ebenfalls bedanken möchte ich mich bei Raffaela, welche viele Grundlagen für meine Arbeit erforscht und mir zahlreiche Türen geöffnet hat. Der meiste Dank gebührt hier aber natürlich Robert, welcher mich von Anfang an in meiner Promotion begleitet und betreut hat. Die zahlreichen Diskussionen, die Einbindung in Forschungsprojekte sowie die gemeinsame Publikationsplanung haben eine maßgebliche Auswirkung auf meine Forschung gehabt. Zum Schluss—denn Professoren stehen traditionell am Ende der Autorenliste—möchte ich mich bei meinem Erstgutachter und Betreuer Ralf bedanken. Seine Anregungen, Feedback, und manchmal auch herausfordernden Ideen haben diese Doktorarbeit deutlich beeinflusst. Noch mehr bin ich aber für alles außerhalb der Promotion dankbar; seien es die Einblicke in die Wissenschaft, die zahlreichen Möglichkeiten, mich kreativ einzubringen, oder die stundenlangen Ausflüge in die Philosophie.

Mein Dank gilt auch allen, die mich während meiner Promotion begleitet haben, aber hier nicht direkt oder indirekt genannt wurden. Diese 1560 Tage waren eine herausfordernde Zeit, in der ich extrem viel erleben durfte. Dennoch bin ich froh, dieses Kapitel abgeschlossen zu haben und hinter mir lassen zu können. Danke für alles. Auf die Zukunft!

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Acronyms

ABAC Attribute-Based Access Control 16, 135

ADD Architectural Design Decision 4, 18, 43–48, 52–55, 57f., 69f., 119, 131f., 196, 200, 202, 231f., 239, 246

ADL Architectural Description Language 18f., 35f., 43ff., 48ff., 56, 59, 61, 73ff., 80, 82, 85, 96, 100, 111, 117, 174, 235

AIS Actual Impact Set 83f., 95

BFS Breadth-First Search . . . . . 99f.

CBSE Component-based Software Engineering 18, 234

CIS Candidate Impact Set 83f., 88, 95

CPS Cyber-Physical System 4, 37, 114, 126, 204, 249

CVSS Common Vulnerability Scoring System 55

DAG Directed Acyclic Graph . . . 11, 22f., 59, 61ff., 69f., 75, 79, 98–101, 105, 108, 113, 120ff., 124ff., 144, 148–153, 157ff., 161, 247, 249, 295f.

DFD Data Flow Diagram 4, 19–23, 26, 28, 35, 37f., 40, 42–45, 59–63, 68ff., 74–80, 82, 85, 90, 92, 94–105, 108, 110f., 113, 115, 117, 119–126, 128, 130, 138, 140, 142, 144, 148ff., 158, 168, 190f., 205f., 208ff., 217, 220, 234–237, 244, 248f., 281, 295f.

DFS Depth-First Search 99f., 105, 124, 154

DLT Distributed Ledger Technology 173f., 177

DSL Domain-Specific language 160, 235f., 248

EMF Eclipse Modeling Framework 17, 146

FIS Fuzzy Inference System 135, 137ff., 145, 160, 231

GDPR General Data Protection Regulation 3, 16, 26, 50, 58, 176, 236

GQM Goal Question Metric 179f., 194, 204f., 210f., 217f., 221, 245

IIoT Industrial Internet of Things 4

IoT Internet of Things 3

JSON JavaScript Object Notation 43, 66

KAMP Karlsruhe Architectural Maintainability Prediction 83, 88, 111

LLM Large Language Models 3, 247

MAPE-K Monitor-Analyze-Plan-Execute-Knowledge 34, 108

MDSD Model-Driven Software Development 11, 17f., 23, 233

MOF Meta Object Facility 17

NDFD Nondeterministic Data Flow Diagram . . . . 144, 148–154, 247, 295f.

OCL Object Constraint Language 234

OMG Object Management Group 13f., 17, 225, 227, 240

OWASP Open Worldwide Application Security Project 4, 41, 117, 134

PCM Palladio Component Model . . 18f., 28, 35, 38, 48, 50, 56, 59, 74–77, 80–88, 90–94, 96, 100f., 103, 105f., 111, 113, 117, 120, 122f., 125, 132, 146f., 149, 153, 158, 161, 167–176, 234f., 237, 248, 277, 289

PSUM Precise Semantics for Uncertainty Modeling13ff., 50, 52, 108, 200, 225, 227, 230, 240, 295

RBAC Role Based Access Control 16, 212, 215, 234

SAS Self-Adaptive System . 4, 15, 28, 33f., 41, 45, 47f., 69, 74, 107f., 110, 126, 225–228, 240, 247, 249

SEFF Service Effect Specification . 18, 82, 85–89, 93–96, 100, 104, 140, 168–172, 174f., 277

SIS Starting Impact Set 83f., 95, 97

SLR Systematic Literature Review6, 14, 37, 41, 129, 159, 167, 179, 225, 228, 233, 239f., 295

SUS System Usability Scale 184, 188, 195f., 199, 201f., 204, 218

TEE Trusted Execution Environment 173f., 177

TFG Transpose Flow Graph . . 120–125, 140–144, 149–158, 160, 168–172, 174f., 192, 212ff., 217, 220, 229

UAP Uncertainty Awareness Problem 41ff., 63, 65, 69, 74f., 106f., 113, 187, 194, 218

UFD Uncertainty Flow Diagram 37, 74, 108–113, 115, 159, 161, 244

UIP Uncertainty Interaction Problem74f., 96f., 107f., 110, 112f., 119, 126, 161, 228, 244, 249

UML Unified Modeling Language 17, 19, 75, 85, 91, 108, 111, 230, 233ff., 237

XACML eXtensible Access Control Markup Language 16, 234

XML Extensible Markup Language 16, 43

Part I.

Prolog

1. Introduction

This dissertation is titled “Architecture-Based and Uncertainty-Aware Confidentiality Analysis”. This title comprises the two most relevant aspects that set the scene for this work: *Confidentiality* and *Uncertainty*. Both represent extensively researched and highly relevant topics—but it is the intersection of the two areas that represents a gap in the state of the art. In this dissertation, we[1] address this gap and present new, tool-supported methods to reason about confidentiality under *software-architectural* uncertainty based on architectural modeling and analysis.

In the following, we motivate our work and derive a problem statement. Afterward, we introduce our research goal and our three central research questions. We conclude this section by providing a summary and an outline of the remaining thesis.

1.1. Motivation

Our world is interconnected; data is *flowing* everywhere. Today, more humans[2] and software systems are connected than ever [248]. Extensive exchange and storage of data enable modern applications in all domains, from online shopping to healthcare, from smart home to mobility systems, from the Internet of Things (IoT) to Industry 4.0 [113]. Recent advances like the rise of Large Language Modelss (LLMs) that are trained with trillions of words [170, 262] show the worth of collecting, organizing, and processing data. Not surprisingly, policy makers address this development by both supporting shared and open data but also restricting the use of personal data and strengthening the rights of individuals. The recent discussions about the German mobility data act for open mobility data [30] are a prominent example of the former, while the European General Data Protection Regulation (GDPR) [73] exemplifies the latter. Put simply, our modern society runs on data—with all the opportunities and challenges that come with it.

A particular challenge of this intense exchange of data is *Confidentiality.* Informally speaking, if someone tells you a secret and asks you to not tell anyone, you are asked to keep this secret confidential. Less informal, the ISO/IEC 27000 standard defines confidentiality as the “property that information is not made available or disclosed to unauthorized individuals, entities, or processes” [120]. The challenge of confidentiality—and security in

[1] Although I am the sole author of this dissertation and this thesis represents my research of the last four years, I will use the plural form as it is more common in scientific writing.

[2] In this work, we intentionally include all genders. We use gender-neutral language for inclusivity.

general—is especially visible in large systems of systems [187] that are neither developed nor operated by a single person or team. It becomes even more challenging when dealing with sensitive data, e.g., regarding politics [271] or health [69]. Threats to the confidentiality of data are rising. For instance, the German Federal Criminal Police Office notes a rise in cyber crimes, where less than 30% of incidents are solved. Comparing the 2022 and 2024 Global Automotive Cybersecurity Report [259, 260] indicates that remote attacks are on the rise and now make up 95% of all cybersecurity incidents. Data breaches are common. Well-known examples are the 2013 attack on Yahoo, affecting 3 billion accounts [108], or the LinkedIn leak affecting 700 million users [175]. A lack of privacy cannot only cause costly fines [31] but also harm the users' trust [271], as seen with the Cambridge Analytica scandal [122]. Data breaches have diverse reasons like hacking, malware, sabotage, privilege abuse, or configuration errors [55], have a significantly negative effect on stakeholder wealth [83], and represent a major issue for organizations [13].

Due to the nature of confidentiality as quality property regarding a software system's data, reacting is not enough, and precaution is required [75]. Here, addressing issues earlier during the design of a software system is beneficial because fixing issues in later phases is usually more costly [32, 240]. Moreover, the Open Worldwide Application Security Project (OWASP) lists *insecure design* as one of the top 10 security risks [79]. Here, approaches like *Security by Design* and *Privacy by Design* ask for the early and continuous consideration of security and privacy-related properties like confidentiality [73, 224, 229]. In addition, the concept of *phase containment* requires that problems are tackled in the same development phase in which they arose. Identifying confidentiality violations is a system-wide analysis task that requires more information than the source code alone can provide, e.g., information about the system deployment or its users [233]. Thus, numerous design time confidentiality analyses that consider such information about the software architecture and its context have recently been proposed [36, 193, 196, 228, 233, 264]. The ISO/IEC/IEEE 42010 standard defines software architecture as "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [121]. Using the software architectural abstraction is not only common in larger software systems [207], but also, for instance, in smaller open-source projects [171], and can address the shortcomings of considering only a single diagram types like Data Flow Diagrams (DFDs) [241]. By combining structural, behavioral, and deployment information of a software system, such analyses can search for attack paths [264], check for conformance with the implementation [193] or business processes [196], or identify confidentiality violations even before a system is implemented [236]. This enables software architects to assess the confidentiality of a software system, thereby addressing the aforementioned challenges.

However, today's software systems are neither built nor operated in isolation. Examples like Industrial Internet of Things (IIoT) [9], Cyber-Physical Systems (CPSs) [2], and Self-Adaptive Systems (SASs) [272] show that software has to adapt to its environment. Moreover, with the rise of agile methods, a software's architecture is not graved in stone and is subject to expected but also unanticipated change. This phenomenon is known as *Uncertainty*. The ISO/IEC 27000 standard describes uncertainty as "the state, even partial, of deficiency of information related to, understanding or knowledge of, an event, its consequence, or likelihood" [120]. This lack of information or knowledge can exist both within the system and its environment [195] and degrades—or in the worst case voids—previous analysis results [194]. Put simply, one cannot analyze what one does not know. Software engineers face uncertainty during the development and operation of software systems [258]. Moreover, also the design process can cause uncertainty, e.g., due to a lack of documentation [107], or open Architectural Design Decisions (ADDs) [167]. Garlan [80] proposed more than a decade ago to include uncertainty as a first-class concern in all phases of software development. Even earlier, in 1995, Jackson [123] attributed such problems to the gap between the world and the machine. The naive solutions in architecture-based confidentiality analysis are assuming certainty [233] or denying quality predictions—both are not expedient [258]. Thus, we have to *embrace* uncertainty within the software and its environment and actively consider its potential impact on the validity of the confidentiality analysis results. As we apply architectural modeling and analysis to reason about uncertainty, we are particularly interested in software-architectural uncertainty. *Software-architectural uncertainty* describes uncertainty, that can be represented on architectural abstraction and where early awareness enables considering its impact on quality attributes [102]. To conclude, "there is no point in using exact methods where there is no clarity [...] to which they are to be applied" [176].

1.2. Problem Statement

In this dissertation, we present an approach to uncertainty-aware confidentiality analysis that uses architectural abstraction. As motivated above, many approaches to architecture-based confidentiality analysis exist—however, they do not consider uncertainty as inherent phenomenon of everyday software systems [36, 193, 196, 228, 233, 264]. Numerous works present comprehensive collections of different types of uncertainty sources [48, 195, 202, 255]. Examples for such uncertainties are open design decisions, e.g., regarding deployment locations or component choices, and also environmental factors like the behavior of users or third parties, or the validity of sensor input data. When considering uncertainty in confidentiality analysis, multiple challenges arise [99]. *First*, we need to better understand the relation of uncertainty and confidentiality. We require a trade-off between allowing uncertainty to influence the software architecture while still being able to assess confidentiality and identify violations. The solution to this challenge is non-trivial due to the variety of uncertainty. *Second*, we need ways to identify and represent uncertainty within the software architecture. This includes considering uncertainty as first-class entity within the software architecture [80], but also requires approaches to identify new uncertainty sources within the software system [81]. Here, model-based approaches to software architecture that represent the software system in an abstract form are common [131, 135, 207, 255]. Addressing both challenges shall enable software architects to identify and represent uncertainty in software architectures regarding confidentiality.

However, it is not sufficient only identifying and considering uncertainty within the software architecture, i.e., architectural models that represent the structure, or behavior of software systems. To analyze confidentiality, automated analyses are required as "detecting confidentiality issues manually is not feasible" [234]. As discussed previously, rapid and early feedback enhances the quality and minimizes the cost [207]. Especially with the growing size and complexity of modern software systems, manual analysis becomes bothersome and error-prone[3]. Manually analyzing—and re-analyzing—large software models by hand after each change and with regard to each identified uncertainty is not expedient. Thus, the *third* challenge is to provide accurate yet scalable analyses that identify confidentiality violations with respect to uncertainty. This includes assessing the potential impact of uncertainties on the software system and, thus, on confidentiality. We require accurate predictions of this impact, and to relate it to confidentiality violations. Overestimations can introduce noise and thus degrade the analysis results, while underestimations can cause missed violations and render the analysis ineffective [71]. Furthermore, we require scalability of *what-if* analyses, in order to be applicable during the software design [143]. Last, we require such analyses to be usable by software architects [223]. Addressing this challenge shall enable software architects to reason about confidentiality under uncertainty based on a tool-supported modeling and analysis approach.

We summarize these challenges in our problem statement, comprising the two central problems, also discussed in the motivation:

P1 *Inspection problem*: There is a lack of knowledge and awareness to identify, describe, classify, understand, and represent uncertainty sources and their impact on confidentiality using architectural abstraction. This problem hinders software architects from inspecting uncertainty in software systems.

P2 *Assessment problem*: There is a lack of tool-supported design time analyses that use the software architecture to predict the impact of uncertainty and identify confidentiality violations with respect to uncertainty. This problem hinders software architects from assessing the *actual* confidentiality of software systems.

Both Problems **P1** and **P2** are supported by the literature. While "there is growing consensus on the importance of uncertainty" [115], much is yet unknown regarding the impact of uncertainty on software systems [80]. Hezavehi et al. [115] conducted a survey on uncertainty. They find a "lack of systematic approaches for managing uncertainty" [115] and that uncertainty should already be addressed at design time. This is supported by the work of Troya et al. [255]. They conducted a Systematic Literature Review (SLR) and analyzed 123 papers. They state that "software models are still falling short of explicitly representing uncertainty" [255] and that software engineers require more help "to identify the types of uncertainty that can affect their application domains" [255]. Many security problems at runtime can be traced back to earlier phases [96], and current tools seem not to be appropriate to handle this gap in abstraction [19]. In sum, we need to research at the intersection of software architecture, confidentiality, and uncertainty to address this shortcoming and to enhance the validity of state of the art confidentiality analysis.

[3] To illustrate the challenge of manual analysis, Appendix D shows the structural view of the *Corona Warn App* [222], a contact tracing app that handles sensitive data such as COVID-19 test results, and that was downloaded more than 20 million times during the COVID-19 pandemic [119].

1.3. Research Objective

Having motivated the actuality and relevance of uncertainty regarding confidentiality and the resulting problems of inspecting (**P1**) and assessing (**P2**) the confidentiality of software systems under uncertainty, we define the research goal of this dissertation:

> ❗ **Research Goal:** Define a classification of uncertainty sources regarding confidentiality using the software architectural abstraction. Provide architecture-based analyses that predict the impact of uncertainty sources and assist software architects in identifying confidentiality violations with respect to uncertainty.

A classification provides a terminology for understanding, communicating, and documenting the object under study, i.e., uncertainty sources. It thereby also supports the identification and modeling of uncertainty sources as a first-class concern within the software architecture [80]. This addresses the first Problem **P1** regarding the inspection of uncertainty. The analyses support software architects in understanding the criticality of previously identified uncertainty sources by propagating the effects in the software architecture and thereby predicting the potential impact on the software system. Furthermore, the actual confidentiality of the software system can be analyzed by considering uncertainty sources within the system and its environment. This addresses the second Problem **P2** regarding the assessment of confidentiality under uncertainty. Based on this research goal, we derive three research questions. The first research question is:

> ❓ **Research Question 1:** How to identify, describe and classify software-architectural uncertainty regarding its relation to and impact on confidentiality?

The first question aims to better understand the different uncertainty types, their attributes, and their impact on confidentiality, as discussed with Problem **P1**. According to Shaw [238], this is a *characterization* question, as we are interested in the most important characteristics of uncertainty. The envisioned research results is of *descriptive* nature, a classification of uncertainty tailored to confidentiality. The approach of first classifying uncertainty has also been proposed in related work [115, 255]. Answering this question includes understanding the location, relationship, nature and modeling possibilities of relevant uncertainty types. Besides classifying uncertainty, this question also targets the identification of relevant uncertainty sources in the software system under study. This represents the baseline for all further research regarding the intersection of confidentiality and uncertainty. Our second research question is:

> ❓ **Research Question 2:** How to propagate classified uncertainty sources based on architectural modeling to predict their impact on a system's confidentiality?

The second question considers the propagation of uncertainty based on architectural modeling. This relates to Problem **P2** as we need to understand the impact of uncertainty first in order to analyze its effects. This question asks for a *method for analysis* [238], where the envisioned results is a technique for uncertainty impact prediction. Propagating uncertainty has been proposed in other research areas by related work [50, 115]. In architectural models, propagation means following the potential effects through the software system until no further elements of the architecture are affected [46, 211]. Answering this question includes the understanding of uncertainty propagation in all relevant representations of software architectures to make predictions of confidentiality violations. After having identified relevant terminology to describe and classify uncertainty sources, assessing their impact represents the obvious next step [115]. Last, our third research question is:

> ❓ **Research Question 3:** How to analyze confidentiality requirements using architectural data flow analysis with respect to uncertainty within the model?

This third question asks for an uncertainty-aware analysis of confidentiality. Here, we assume that we already have answers regarding the classification and impact of uncertainty, which were addressed in ❓ **Research Question 1** and ❓ **Research Question 2**, respectively. This question targets both Problems **P1** and **P2**, i.e., the lack of approaches that can model and analyze confidentiality requirements under uncertainty using software-architectural modeling. Similar to the previous question, we ask for a *method for analysis* [238] with the envisioned research result of a new technique. Here, we build on existing approaches to architecture-based confidentiality analysis [233, 264] that lack uncertainty awareness. Answering this question includes understanding how to represent uncertainty in architectural models and how to make statements about confidentiality under uncertainty. This is especially challenging when considering multiple uncertainty sources and their potential interactions [52]. In sum, answering these three questions satisfies the research goal and helps software architects to develop software systems with higher resilience against confidentiality issues like data breaches. Although answering these questions on their own already provides valuable insights in the nature and interaction of confidentiality and uncertainty, combining the results represents the most promising approach. Here, recent work highlights the need for comprehensive end-to-end approaches to uncertainty-aware analysis [273].

1.4. Approach

Our approach to tackle the Problems **P1** and **P2** and to answer the aforementioned research questions is twofold. We present an analysis dualism by combining the analysis of uncertainty with the analysis of confidentiality to assess confidentiality requirements for an uncertainty-afflicted software architecture. This enables us to not only make statements about confidentiality in a deterministic world but to include the nondeterministic impact of uncertainty [263]. As discussed previously, the classification of uncertainty serves as the foundation for both analyses. This approach is visualized in Figure 1.1. In the following, we summarize the three parts of our approach that also represent the three contributions **C1**, **C2**, and **C3** of this dissertation. They will be detailed later in this thesis.

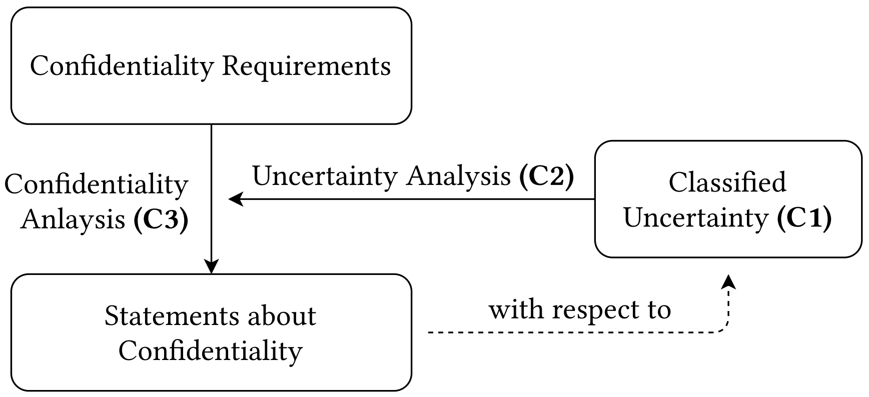

Confidentiality Requirements
Confidentiality Anlaysis (C3)
Uncertainty Analysis (C2)
Classified Uncertainty (C1)
Statements about Confidentiality
with respect to

Figure 1.1.: Schematic and informal visualization of the presented approach.

Classifying Uncertainty Classifying uncertainty provides the required terminology to inspect and assess uncertainty regarding confidentiality. By identifying relevant uncertainty types and grouping uncertainty sources into classes, we bootstrap the further analysis process. This part also includes considering the identification of uncertainty sources by addressing the lack of awareness and expert knowledge of software architects. Put simply, we first have to investigate the nature of the problem before developing a solution. This represents an answer to ❓ **Research Question 1** and also our first Contribution **C1**.

Uncertainty Analysis We build on these findings on the types and attributes of uncertainty sources that affect the confidentiality of a software system to develop an analysis of uncertainty. Here, we first define an impact analysis to propagate uncertainty. Based on the relations of the elements of a software architecture, we propagate the effects of uncertainty sources on confidentiality through the architectural model. Comparable to change impact analysis [46, 110, 211], this analysis yields an impact set that predicts the potential impact of unanticipated change, i.e., uncertainty. This does not only serve software architects as an early prediction but also shows the effects of uncertainty sources that can subsequently be used in confidentiality analysis. Put simply, we have to analyze uncertainty first before analyzing confidentiality under uncertainty. This represents an answer to ❓ **Research Question 2** and also our second Contribution **C2**.

Confidentiality Analysis Confidentiality analyses take confidentiality requirements as input to make statements about the confidentiality of a software architecture, see Figure 1.1. Based on our previous findings, we extend existing analysis approaches [36, 236] to consider uncertainty. Here, black-box [266] and white-box [101] analysis extensions are possible. After software architects have modeled sources of uncertainty as part of the architecture model, the uncertainty analysis determines which confidentiality requirements are violated due to which uncertainties. Here, considering multiple interacting uncertainty sources is especially challenging, as their effects can add up or cancel each other out [50, 52]. This represents an answer to ❓ **Research Question 3** and also our third Contribution **C3**.

As discussed previously, all contributions can be used independently or in conjunction. We also provide tooling that supports the modeling and the analysis based on established frameworks [205, 207]. In this thesis, we motivate, present, and evaluate all contributions independently. We transparently enumerate assumptions and limitations. Last, we provide a comprehensive data set including our tooling and all raw evaluation data [98].

1.5. Outline and Reading Paths

This dissertation consists of four parts: The prolog, the contributions, their validation, and the epilog. The remainder of the prolog comprises the presentation of foundations in Chapter 2 and the introduction of a running example in Chapter 3. This running example does not only serve to exemplify the scientific problems addressed in this thesis, but will also be used intensively throughout the thesis. Afterward, we present the contributions of this dissertation. We start with an overview that continues the introduction of our approach in Chapter 4. Then, we introduce our classification of uncertainty (**C1**) in Chapter 5, the uncertainty impact analysis (**C2**) in Chapter 6, and four uncertainty-aware confidentiality analysis approaches (**C3**) in Chapter 7. We show the scenarios used in the validation of the contributions in Chapter 8 and present our comprehensive evaluation thereafter in Chapter 9. Last, the epilog concludes this thesis with an overview of related work in Chapter 10 and a summary and outlook on future work in Chapter 11. We would also like to point out the additional information in the Appendix and in the data set [98]. There are multiple ways to read this dissertation:

- **Beginner readers**, who are not interested in the technical details of the contributions but want to understand the bigger picture, continue with reading the foundations in Chapter 2. Afterward, every chapter ends with a section called *In Simpler Words*. There, we explain the most important concepts and findings of each chapter without using scientific terminology but with reference to everyday life. For your convenience, this means Sections 1.7, 2.7, 3.4, 4.6, 5.10, 6.10, 7.9, 8.9, 9.6, 10.5, and 11.4[4].
- **Expert readers**, who want to gain an overview of our insights, will find gray highlighted boxes titled *Findings* throughout the thesis. A concise yet technical overview of the contributions is given in Chapter 4, and the conclusion in Chapter 11 provides a summary. If the information provided there is not sufficient, the contribution chapters 5, 6, and 7 come with their own introduction and summary sections.
- **Particularly interested readers** are welcome to read the entire work in one piece. Furthermore, we recommend suitable basic literature in Chapter 2 and provide all relevant references to own publications at the beginning of each chapter.

[4] This dissertation is meant to be read digitally. We make extensive use of references, e.g., between chapters or enumerable items like research questions, contributions, and evaluation goals. We recommend using a shortcut to jump back from following a reference. Furthermore, all paragraphs, figures, and tables are optimized for the DIN A4 version of this dissertation. If you happen to have the DIN A5 or the book version at hand, you can find the other versions until the link expires at `https://thesis.abunai.dev/`.

1.6. Summary and Outlook

In this chapter, we introduced the topic of this thesis. First, we motivated the actuality and relevance of confidentiality, uncertainty, and software architecture. Without considering uncertainty as a first-class concern within the software architecture, the validity of statements about confidentiality can be degraded or completely voided [99]. Afterward, we presented three challenges of architecture-based and uncertainty-aware confidentiality analysis. They revolve around understanding the relationship between uncertainty and confidentiality, representing uncertainty in architectural models, and providing automated analyses. The challenges of representing and analyzing uncertainty in the software architecture are also supported by the literature [115, 255]. We summarized these challenges in our problem statement as *inspection problem* (**P1**) and *assessment problem* (**P2**).

Based on these problems, we presented our research goal and three research questions that map to the three Contributions **C1** – **C3** of this dissertation. First, we define a novel classification of uncertainty regarding confidentiality. Second, we use this classification in uncertainty impact analysis, thereby propagating the effects of uncertainty within the software architecture and calculating the potential impact. Third, we include the impact of uncertainty within confidentiality analysis to identify confidentiality violations with respect to uncertainty. We summarized the relations of these research questions and our contributions by introducing this dissertation's approach. Last, we presented the outline and the intended reading paths for the different types of readers of this thesis.

Next, in ❯ **Chapter 2: Foundations**, we explain the foundations required for understanding this thesis, e.g., Directed Acyclic Graphs (DAGs), or Model-Driven Software Development (MDSD). Afterward, we introduce the running example of this thesis in ❯ **Chapter 3: Running Example**. Both chapters also help to understand the context and the general assumptions of this dissertation. In ❯ **Chapter 4: Overview**, we continue the presentation of our approach by relating common activities in the procedure of handling uncertainty in software architectures to our contributions and our tool support.

1.7. In Simpler Words

Our world runs on data. Think of social media, online shopping, advertisement, smart home, or artificial intelligence—all of these systems require large amounts of data to provide their functionality. In this dissertation, we focus on the confidentiality of this data. Confidentiality demands that data is not shown to unauthorized persons or organizations. It is a quality property of software systems and is related to security and privacy. In this dissertation, we research how to analyze the confidentiality of software systems while considering uncertainty. Put simply, uncertainty is the lack of knowledge, i.e., not knowing something for sure, e.g., because an engineer cannot assess environmental conditions or the behavior of third parties.

We present an approach to better understand the potential impact of uncertainty on confidentiality by using a high-level view of the software system, i.e., software architecture, the big picture of a software system. Therefore, we first have to better understand uncertainty—like you need to understand the rules of a board game to be able to play better. Afterward, we build an analysis that propagates uncertainty within a software architecture to better understand its impact. This is comparable to introducing a drop of dye into a stream of water and then observing the coloration. In the end, we enable software architects to identify confidentiality violations early and to build better software systems that protect the data of their users.

2. Foundations

In this chapter, we explain the foundations of this thesis. As stated in the title, the topic of this thesis revolves around software architecture, confidentiality, and uncertainty. We present common terminology, approaches, and underlying concepts that will be used throughout this thesis. For every section, we also provide references for further reading.

The remainder of this chapter is structured as follows: First, we introduce the central concerns of this thesis, i.e., uncertainty, and confidentiality. Afterward, we explain the concept of model-driven development and how we use it in architectural modeling and analysis. Last, we present the foundations of modeling data flows of software systems.

2.1. Uncertainty

Uncertainty has many definitions: Walker et al. [263] refer to uncertainty as "being any departure from the unachievable ideal of complete determinism" [263]. Weyns [272] defines uncertainty as "any deviation of deterministic knowledge that may reduce the confidence of adaptation decisions made based on the knowledge" [272]. In architecture evaluation, Sobhy et al. [243] define it as "the lack of full knowledge about the outcomes of deploying the architecture options" [243]. Uncertainty is also related to doubt, error [128], or risk [120]. In the ISO/IEC 27000 standard, uncertainty is defined as "the state, even partial, of deficiency of information related to, understanding or knowledge of, an event, its consequence, or likelihood" [120]. The current version of the Object Management Group (OMG) Precise Semantics for Uncertainty Modeling (PSUM) standard refers to Zhang et al. [278], describing uncertainty as "the lack of confidence (i.e., knowledge) about the timing and nature of inputs, the state of a system, a future outcome, as well as other relevant factors" [184]. In software design, the *cone of uncertainty* describes the lack of knowledge about a software system in early development phases [167]. Here, uncertainty about open design decisions can impact cost estimation.

These definitions share four common aspects. First, they refer to a lack of information regarding a system. Second, they mention sources of uncertainty within the system or its environment [2]. Third, they name consequences of uncertainty, e.g., increased risk, which may impact the quality of the system. Fourth, the majority of these definitions consider even small changes in certainty, using terms like "any deviation", "full knowledge", or "unachievable ideal", highlighting how omnipresent or ubiquitous uncertainty is. For the scope of this thesis, we adopt these four aspects and refer to uncertainty as the commonly occurring lack of information that can negatively impact the quality of a software system.

Available Categories	Available Options
Location: Describes where uncertainty originates from or where it manifests itself within the system or model [41, 162, 184, 195, 255, 263]	**Context**: system boundaries [195, 263], user input [41], execution context [162, 184], environment [184]; **Model structural**: existence of elements [255], elements and their relationship [195, 263], structural differences [162], components and their properties [41]; **Model technical**: software and hardware [263]; **Input**: input types [263], input values [195], measurement deviation [184, 255], geographical location [184], time [184]; **Parameters**: parameter calibration [195, 263]; **System behavior**: actual behavior [41], including parameters and actions [255]; **Belief**: uncertain statements about system and environment [255]
Level: Describes how much is known about the uncertain influence and how the uncertainty can be described [11, 41, 162, 184, 195, 263]	**Statistical**: Statistical data available [162, 263]; **Scenario**: Possible scenarios available without statistical data [11, 162, 263]; **Recognized ignorance**: Awareness of uncertainty, but cannot be further described [263], can be supported by evidence, e.g., empirical evidence, or theorem proving results [184]; **Total ignorance**: Lack of awareness of uncertainty [263]; **Orders of Uncertainty**: No uncertainty (0th), known uncertainty (1st), lack of awareness, i.e., unknown unknowns (2nd), lack of awareness and process (3rd), meta-uncertainty (4th) [11, 41, 195]; **Perspective**: Subjective, based on observation, or objective, independent of any observing agency [184]

Table 2.1.: Categories and options to classify the location and level of uncertainty.

Existing literature proposes different approaches to deal with uncertainty, involving activities like identification, management, and mitigation [115, 184, 194, 195, 243, 272]. Another common approach to uncertainty is its classification, resulting in a multitude of taxonomies [11, 41, 71, 115, 162, 184, 195, 202, 255, 263]. These taxonomies describe many dimensions that shall help to better understand uncertainty, e.g., by distinguishing between sources of uncertainty or classifying the size of the knowledge gap. Examples of such uncertainties are the lack of knowledge of the structure of a software system, the behavior of external systems, or a sensor's input value. At the time of writing, uncertainty represents a commonly researched topic with novel findings and challenges ahead [273].

In the following, we give an overview of the current state of classifying and describing uncertainty. We investigated existing taxonomies [11, 41, 71, 162, 195, 202, 263], as well as recent Systematic Literature Reviews (SLRs) and surveys on uncertainty [115, 255] and the most recent version of the OMG PSUM standard [184], which is in beta status. To unify the terminology, we only use the terms *classification*, *category*, and *option*.

Table 2.1 shows the first two available categories regarding the *location* and the *level* and all available options. Uncertainty is often described with its *location* which describes where the uncertainty manifests itself. Many classifications provide options like context, input, or

Available Categories	Available Options
Nature: Describes the essence and character of the uncertainty [41, 162, 184, 195, 255, 263]	**Aleatory**: Uncertainty due to inherent variability or randomness [41, 162, 184, 195, 255, 263]; **Epistemic**: Uncertainty due to a lack of knowledge [41, 162, 184, 195, 255, 263], can be further described as indeterminacy, e.g., due to insufficient resolution, or missing information [184]
Manageability: Describes whether the uncertainty can be reduced [71, 184, 263]	**Reducible**: Uncertainty can be fully reduced after acknowledgement [71, 184, 263]; **Partially reducible**: No full certainty, but uncertainty can be reduced [184]; **Irreducible**: Uncertainty cannot be further reduced at this point in time [71, 184, 263]
Emerging time: Describes at which state of the software development uncertainty arises [115, 162, 184, 195, 202, 255]	**Requirements time**: As requirements are defined [202, 255]; **Design time**: As the software system is designed [115, 162, 202, 255]; **Verification**: At verification of software models [255]; **Testing**: During software testing [115, 255]; **Implementation**: As the software gets implemented [255]; **Run time**: During software execution [115, 162, 195, 202, 255], can be described using patterns like periodic, persistent, or random [184]

Table 2.2.: Categories and options to classify the nature, manageability, and emerging time of uncertainty.

behavior. The *level* category is used to describe how much is known about the uncertainty. Here, three different approaches exist. Some classifications define the level based on the description of uncertainty, e.g., by using statistical means, or scenarios [162]. Others refer to the orders of uncertainty [11] and the distinction between known unknowns and unknown unknowns. However, even in this order, different nuances exist, e.g., recognized ignorance compared with statistical data. Last, the level can be described based on the perspective being subjective or objective [184].

Table 2.2 shows the categories regarding the *nature*, *manageability*, and *emerging time*. The *nature* of uncertainty with the two options *aleatory* and *epistemic* represents the most uniformly used category among all classifications. However, this category is questioned [71, 137] as it depends on the point of view and is often not clearly distinguishable. Thus, some use the category *manageability* that focuses on reducibility. The distinction between reducible, partially reducible and irreducible is also supported by the PSUM standard. The category *emerging time* describes the time when uncertainty arises. The options reflect the common phases of software engineering from requirements time to run time.

Table 2.3 shows the categories regarding the *impact on quality*, *relationship* of uncertainty, and uncertainty *sources*. The *impact on quality* is important as not all uncertainties affect a system's quality. The *relationship* of uncertainty represents influences of one uncertainty source to another. Last, several publications list *sources* of uncertainty as not all taxonomies directly aim at classifying sources. We list some examples here and also provide a comprehensive catalog of sources as part of the thesis' data set [98].

Available Categories	Available Options
Impact on Quality: Describes how uncertainty affects quality properties [115, 184]	**Performance**: Impact on performance [115]; **Resources**: Impact on resource consumption [115]; **Safety**: Impact on a system's safety [115]; **Risk**: General effect of uncertainty on system objectives [184]
Relationship: The relation between uncertainties [184, 202]	**Directed**: Directed relationship or influences between uncertainties [202]; **Related**: Unspecified relationship between uncertainties [202]; **Effect**: An uncertainty can be caused by another uncertainty [184]
Source: Potential sources of uncertainty [71, 162, 184, 195, 202]	Several publications contain comprehensive lists of uncertainty sources, e.g., human in the loop [195], abstraction [195], missing requirements [202], inadequate design [202], ...

Table 2.3.: Categories and options to classify the impact on quality, relationship, and source of uncertainty.

Further reading: For a comprehensive introduction to the topic of uncertainty in Self-Adaptive Systems (SASs), we recommend Weyns [272]. A more recent overview of the research community is given by Hezavehi et al. [115].

2.2. Confidentiality

Confidentiality is defined as the "property that information is not made available or disclosed to unauthorized individuals, entities, or processes" [120]. It is named together with integrity and availability as part of information security [120]. It is also often referred to in the context of privacy [38, 39, 224, 252]. Confidentiality requirements specify what should be accessible by whom. Properties of such requirements are, for instance, stakeholders, the protected data, and the purpose [189]. Confidentiality is part of legal regulations such as the General Data Protection Regulation (GDPR) [73]. Exemplary confidentiality requirements are the protection of personal information or payment data [36].

Common mechanisms to ensure confidentiality are access control and data encryption [230]. Access control provides "means to ensure that access to assets is authorized and restricted based on business and security requirements" [120]. For instance, Role Based Access Control (RBAC) uses the member roles of an organization to specify permissions. An example is that only members with the administrative role shall be able to directly access a database. Attribute-Based Access Control (ABAC) extends this and supports the evaluation of complex rules comprising different attributes [117]. The eXtensible Access Control Markup Language (XACML) extends the Extensible Markup Language (XML) and represents an open industry standard to formulate and enforce ABAC policies [182]. Encryption is the mapping of a readable plain text to an unreadable cipher text, using an encryption algorithm and a key [18]. Without knowing the key, the plain text shall be hard or impossible to determine. In sum, both mechanisms help to prevent the access of unauthorized entities to protected data and thus help to ensure confidentiality.

Further reading: Schumacher et al. [230] provides a comprehensive introduction to security in the development of software systems, covering an overview of the field of security and a variety of security patterns.

2.3. Model-Driven Software Development

Model-Driven Software Development (MDSD) is a concept where models play a central role in the development of software systems [246]. Nowadays, using models in software development is common, e.g., as shown by the popularity of the Unified Modeling Language (UML). However, compared to only using models for documentation purposes, model-driven approaches give models an equivalent or higher importance than code. This includes techniques like tailored modeling languages, model transformations, and code generation. This thesis uses design time models for analysis purposes, which is related to MDSD.

Stachowiak [245] names three central properties of models: representation, abstraction, and pragmatics. First, a model represents the modeled entity, e.g., a UML diagram can represent the structure or behavior of a real software system. Second, a model abstracts from the modeled entity, e.g., a UML activity diagram does not represent every single line of code. Third, a model has pragmatics, it follows a purpose. The models we use are generally about representing a software system in a simplified yet analyzable way.

One of the most important aspects of MDSD is meta modeling. A meta model describes the structure of a model, including available elements, their relationship, and modeling constraints [246]. Put simply, a meta model states the rules on how a model has to be constructed. An example is the UML providing many diagram types like activity diagrams,

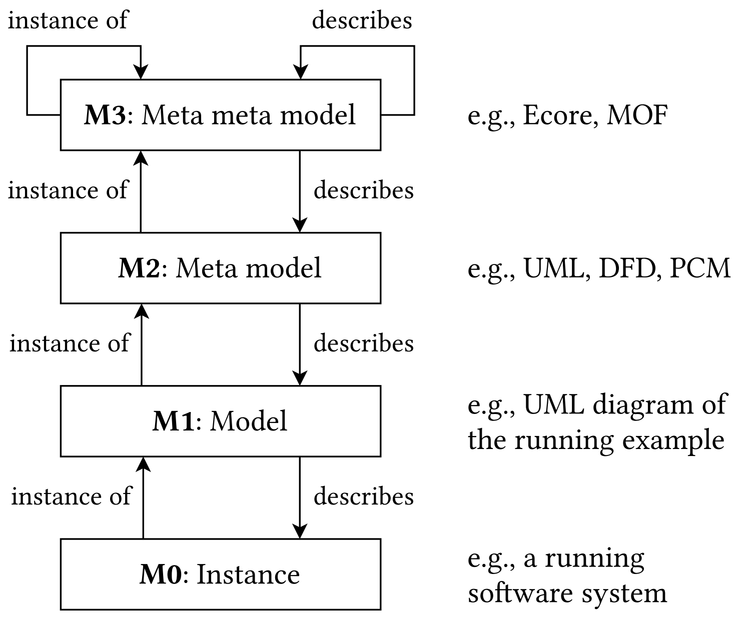


Figure 2.1.: The four meta levels with examples from our domain, based on Stahl et al. [246].

class diagrams, or deployment diagrams. The meta model itself can be described following the same concept using a meta meta model, which states the rules of the meta model. Common meta meta models are the OMG Meta Object Facility (MOF) [188] and Ecore from the Eclipse Modeling Framework (EMF) [249]. Due to the high level of abstraction, these models can describe themselves. This is visualized in Figure 2.1. There, we depict the description and instance relation of all meta levels and show examples from the domain of this thesis, e.g., Ecore, the UML meta model, and UML models. This includes the instance level, representing an entity of the real world, e.g., the modeled software system.

Further reading: We recommend the comprehensive introduction to Model-Driven Software Development (MDSD) by Stahl et al. [246].

2.4. Software Architecture

As stated in the title of this thesis, this work uses the architectural representation of software systems as a baseline. Software architecture comprises the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [121], according to the ISO/IEC/IEEE 42010 standard. Software architecture can be also interpreted as a set of Architectural Design Decisions (ADDs) [124, 125]. Another approach to software architecture is to describe it by its views, e.g., a logical view, or a development view [149]. As discussed in the previous section, we used model-driven techniques. Here, Architectural Description Languages (ADLs) provide "means of expression, with syntax and semantics, consisting of a set of representations, conventions, and associated rules intended to be used to describe an architecture" [121]. As a meta model for this work, we use the Palladio Component Model (PCM) [207], which we describe in more detail in the following.

The Palladio approach [207] enables modeling, simulating, and analyzing software architectures regarding quality dimensions like performance, reliability, or maintainability. Besides comprehensive tool support [205], the approach provides a meta model for software architecture, the PCM [206, 207]. Palladio builds on the concept of Component-based Software Engineering (CBSE), which divides software architectures into reusable and composable components. The PCM meta model has been used previously in security analysis, e.g., regarding confidentiality [233], access control analysis [196], or attacker propagation [264]. Here, the shared underlying meta model simplifies the reuse of architectural models, which minimizes costs. The PCM is partitioned into five submodels:

- The *Component Repository Model* contains components and interfaces. The components' behavior is described using Service Effect Specifications (SEFFs) that abstract from the control flow. Components can be composed both vertically, and horizontally. The repository belongs to the structural viewpoint. The inter-component behavior description with SEFFs belongs to the behavioral viewpoint. For instance, components could be the user management or database of an online learning platform.

- The *System Model* describes the assembly, i.e., the wiring of the components of the *Component Repository Model.* This belongs to the structural viewpoint. For instance, the previously introduced user management component could be wired to the database component for the persistence of user data.
- The *Execution Environment Model* defines hardware resources and the network. These resources represent the deployment locations of components. For instance, resources can be an on-premise server or a cloud service.
- The *Component Allocation Model* describes the deployment of assembly contexts that represent components from the system model, i.e., shows the allocation of the components in use. This model belongs to the deployment viewpoint. For instance, the database component can be deployed on-premise.
- The *Usage Model* specifies the user interaction with the software system. By modeling expected calls to the components, usage scenarios can be expressed. For instance, users interact with the user management when registering on the online platform.

In sum, the PCM enables the system-independent modeling of components and their behavior and of the executing environment. Furthermore, engineers describe the system-specific assembly context and the allocation context of the components, and also the usage context of the software system. Based on these submodels, a multitude of simulations can be executed in the Palladio Bench [205]. For instance, using the information on expected usage scenarios together with the allocation of components, performance bottlenecks can be identified. For the sake of this thesis, we only focus on confidentiality. Thus, we build on the PCM as comprehensive ADL without relating to other analysis approaches of different quality dimensions like performance and reliability.

We choose the PCM as foundation of our work due to its maturity, wide adoption, and reception in the community [146]. However, we want to stress that the contributions presented in this dissertation can also be realized with other ADLs, e.g., the UML. Here, using the PCM does not represent a "vendor lock-in", as its meta models are related to other well-known diagram types like UML component diagrams or UML activity diagrams.

Further reading: Reussner et al. [207] present the Palladio approach in detail. Additionally, they provide an overview of software architecture as a discipline with a focus on architectural modeling and analysis.

2.5. Data Flow Diagrams

DeMarco [64] introduces Data Flow Diagrams (DFDs) to represent software systems "from the point of view of the data" [64]. By focusing on data flows instead of control flows, the data-oriented analysis of issues within software systems is simplified. Regarding security analysis, DFDs represent a simple yet powerful representation of software systems [226]. They are used both for documentation and discussion [226, 241], and also automated analysis [6, 24, 53, 236, 256, 257]. Especially regarding confidentiality, the use of DFDs is expedient, as “problems tend to follow the data flow, not the control flow” [239]. In the following, we briefly introduce the original syntax of DFDs [64]. Afterward, we introduce the unified modeling primitives for DFDs [235], used throughout this thesis, and the representation of DFDs as directed graphs [15, 66].

Further reading: For a historically relevant, original definition of Data Flow Diagrams (DFDs), we refer to DeMarco [64]. To learn more about the fundamentals, see the introduction to graph theory by Diestel [66].

2.5.1. Elements of Data Flow Diagrams

In describing the conventions of data flow diagrams, DeMarco [64] states that “the data flow diagram shows flow of data, not of control” [64]. Resulting from this, loops are excluded from DFDs as they represent control flow, of which “the data are unaware of” [64]. The graphical notation of DFDs comprises four elements:

- *Data sources and sinks* are represented by boxes. They depict an entity like a person or an organization that is outside of the context of the system under study. Examples are the user of a software system or an external data base.
- *Processes* are represented by circles. They depict the processing or transformation of data within the scope of the system under study. Examples are the processing of user input or the calculation of results.
- *Files* are represented by straight line. They depict internal and temporary repositories of data within the scope of the system under study. An example is the temporary storage of intermediate data processing results.
- *Data flows* are represented by named arrows. They represent data paths or pipelines of data within the software system and connect all other elements. Multiple arrows between two elements are possible if the data flow has more than one purpose or type. Examples are the flow of data from a source or the flow between two processes.

Figure 2.2 shows the DFD of a simplified online shop scenario. The *User* represents a data source, the *Data base* represents a data sink, the *Shipping fees* represent internal storage, and the internal processing represents processes. All elements are connected by named arrows, i.e., data flows. The user input comprising an order from the shop is first processed.

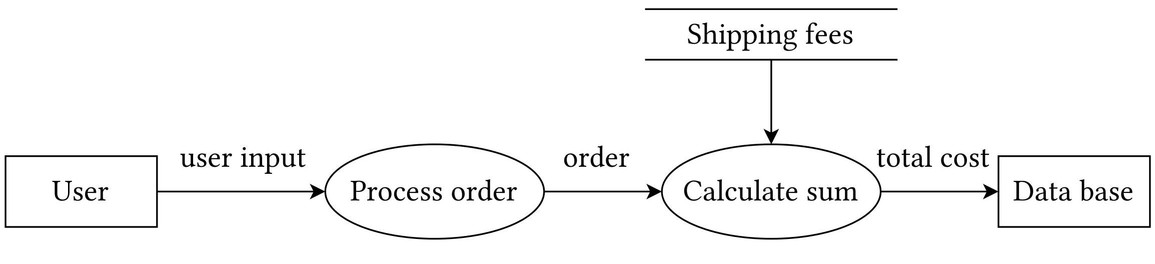


Figure 2.2.: A Data Flow Diagram (DFD) showing all four element types in a simplified online shop scenario.

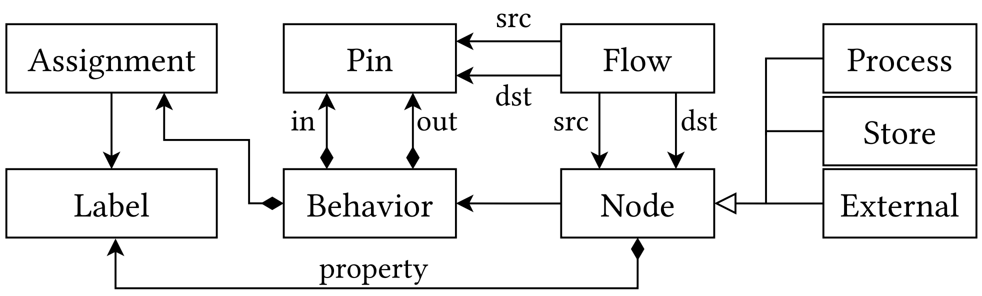


Figure 2.3.: Meta model of unified modeling primitives, based on Seifermann et al. [235].

Afterward, the total cost of the order is calculated, which includes applying the correct shipping fee. The result is stored in the data base. This figure shows several conventions of DeMarco [64], e.g., the arrows of files have no description as their purpose is unambiguous. Inspired by related work [233], we use two horizontal lines to represent files throughout this thesis to increase readability.

2.5.2. Unified Modeling Primitives

The DFD syntax is simple and easy to understand. However, this simplicity causes ambiguity, especially regarding security analysis. Sion et al. [241] name several weaknesses of DFDs for security-related analysis: They lack the means to represent security concepts, lack precision in describing the details of flowing data, and do not express deployment information. To address this, Seifermann et al. [235] present the *unified modeling primitives* of DFDs by combining multiple notations of security-focussed DFD notations from other work [234, 256]. We use these primitives in this thesis to represent DFDs.

Figure 2.3 shows the meta model of the unified modeling primitives. It comprises the element types known from DFDs by DeMarco [64], i.e., *External* nodes, *Processes*, and internal *Stores*. These *Nodes* are connected by *Flows*. To reduce the ambiguity of data flows, *Pins* are used to represent the incoming and outgoing data of a *Node*. Alternative *Pins* can be used to represent required inputs, e.g., the calculation of the sum in Figure 2.2 requires both the order and the shipping fees. Multiple ingoing flows to a single *Pin*, or multiple outgoing flows from a single *Pin* represent alternative flows. *Pins* are used to decouple the *Behavior* from a *Node*. A *Behavior* describes the data processing of a node using *Assignments*. *Assignments* alter *Labels* that represent characteristics of flowing data, e.g., whether data is encrypted or personal. For instance, the order processing in Figure 2.2 removes all personal information of the user and only forwards the cost of purchased items, which can be represented using an *Assignment* within the *Behavior* of the *Process*. This addresses the shortcoming regarding behavior descriptions [241]. Last, *Labels* are used to describe the properties of *Nodes*, e.g., their deployment.

Using the unified modeling primitives, we can analyze transitive data flows by tracing *Labels* in DFDs. The procedure of following *Labels* through the system is called *label propagation* [233, 235, 236]. To identify confidentiality violations, the propagated *Labels* can be compared to the property *Labels* of a node. This enables the specification of confidentiality requirements as data flow constraints [100, 105]. We reuse conventions for the behavior and constraint description, e.g., the two-headed arrow $\twoheadrightarrow$ represents the forwarding of data in a DFD. The crossed-out arrow $\nrightarrow$ is used to represent a forbidden data flow. For instance, the data flow constraint of personal data that shall *never flow* to the data base can be written using *Labels* as $\textit{personal} \nrightarrow \textit{database}$. For a more detailed explanation, see the work of Seifermann [233], and Boltz and Hahner et. al [36].

2.5.3. Directed Acyclic Graphs

As stated previously, DFDs use arrows to depict the flow of data. These arrows represent directed connections between nodes, usually without forming cycles [64]. Thus, DFDs can be interpreted as graphs where vertices are represented by processes, files, data sources, or data stinks. The data flows form the edges of the graph. In graph theory [15, 66], such graphs are referred to as a Directed Acyclic Graph (DAG).

A DAG is a pair $G = (V, E)$ of sets, where V represents the vertices and $E \subseteq [V]^2$ represents the edges. We assume that both represent nonempty and finite sets and $V \cap E = \emptyset$. We write $|G|$ to represent the order of G, i.e., the number of its vertices. Furthermore, we require the graph to be directed and free of cycles. Note that control flow graphs with cycles can be transformed into acyclic DFDs [148]. As we use DAGs only to represent data flows, we do not use the usual notation of an edge that concatenates its vertices name. Instead of writing ab to depict an edge from a to b, we use the flow syntax of $a \longrightarrow b$. Besides representing a simple DFD, Figure 2.2 also satisfies the requirements of a DAG. All data flows are directed and there are no cycles with $V = \{\textit{User}, \textit{Process order}, \textit{Calculate sum}, \textit{Shipping fees}, \textit{Data base}\}$ and, for instance, $\{\textit{User} \longrightarrow \textit{Process order}, \textit{Process order} \longrightarrow \textit{Calculate sum}\} \subset E$.

In DAGs, data flows can be represented as strict partial ordering, being irreflexive, asymmetric, and transitive [141]. For all $a, b, c \in V$, this means that $\neg(a < a)$, i.e., there is no data flow of a vertex to itself. Additionally, if $a < b$, i.e., if there is a flow from a to b, then $\neg(b < a)$, i.e., there is no flow back, which would create a cycle. Last, data flows are transitive, i.e., if $a < b$ and $b < c$, then trivially also $a < c$. Figure 2.2 exemplifies all three properties: It is irreflexive and asymmetric, without any cyclic flows or flows of data from a vertex to itself. Last, it shows transitive flows, e.g., the flow from the *User* to the *Data base*. In sum, DAGs are a simple yet powerful notation to represent DFDs.

2.6. Summary and Outlook

In this chapter, we explained the most relevant foundations of this thesis. This includes the topics of this thesis, i.e., confidentiality, uncertainty, and software architecture. We briefly repeat the central definitions based on ISO/IEC 27000 and ISO/IEC/IEEE 42010. Uncertainty is defined as “the state, even partial, of deficiency of information related to, understanding or knowledge of, an event, its consequence, or likelihood” [120]. Confidentiality is defined as the “property that information is not made available or disclosed to unauthorized individuals, entities, or processes” [120]. Last, software architecture can be defined as “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution” [121].

Furthermore, we introduced model-driven approaches to software engineering as we build on model-based techniques in our contributions. Last, we also discussed multiple DFD representations: The original definition of DeMarco [64], the unified modeling primitives by Seifermann et al. [235], and the representation of DFDs as DAGs. We use all three notations throughout this thesis. The example of an online shop, briefly introduced in this chapter, is used as a running example in ❯ **Chapter 3: Running Example**. Afterward, in ❯ **Chapter 4: Overview**, we give an overview of the thesis and its contributions, which are based on the aforementioned foundations.

2.7. In Simpler Words

Our research builds on the work of others. We build on many findings of other researchers from the last decades. The most important foundations of our work are presented in this chapter. First, we introduce the topic of uncertainty. You can think of uncertainty as the opposite of certainty, i.e., the lack of knowledge about something. For instance, you may not know what you are going to eat tomorrow—you are unsure, or uncertain. Next, we introduce confidentiality. Put simply, if I tell you a secret and ask you not to tell it to anyone else, I ask for confidentiality. This confidentiality is especially important in software systems that handle large amounts of sensitive data.

We also introduce model-driven approaches to software development and software architecture, known as Model-Driven Software Development (MDSD). Models can be thought of as diagrams of software systems that show, for example, the structure of a system. In this thesis, these diagrams are central to our way of thinking about software systems. We thereby focus on the software architecture, i.e., a higher abstraction of the system under study. We are not interested in every single line of code but we focus on the bigger questions of how the system is constructed and deployed.

Last, we discuss Data Flow Diagrams (DFDs), as we are especially interested in the confidentiality of data. Here, we present different notations of such diagrams that differ in their nature and complexity. For instance, we refer to the original definition by DeMarco [64] from 1979, and a more recent notation by Seifermann et al. [235]. We also explain a more formal way to draw such diagrams, called Directed Acyclic Graphs (DAGs). It is important for us to have appropriate means to express data flows, as the remainder of this thesis will build on them.

3. Running Example

In this chapter, we introduce the running example that will be used throughout this thesis to illustrate our findings. The running example is based on internationally operating e-commerce businesses known from everyday life. To become easy to understand, we simplify the software architecture compared to real-world applications. Nevertheless, the confidentiality requirements and potential uncertainty sources can be transferred to large-scale systems. The simplified *Online Shop* running example has been used in various publications [101, 102, 105]. Similar case studies exist, e.g., in the context of the Common Component Modeling Example, CoCoME [203].

In the following, we shortly introduce the online shop software architecture, its intended functionality, and confidentiality requirements. Then, we illustrate the variety of uncertainty and problems that can be caused by this uncertainty regarding confidentiality.

Literature: This chapter is based on the following (co-) authored publications: [ECSA-C 2021], [IEEE/ACM SEAMS 2023], [ACM/IEEE MODELS-C 2024]

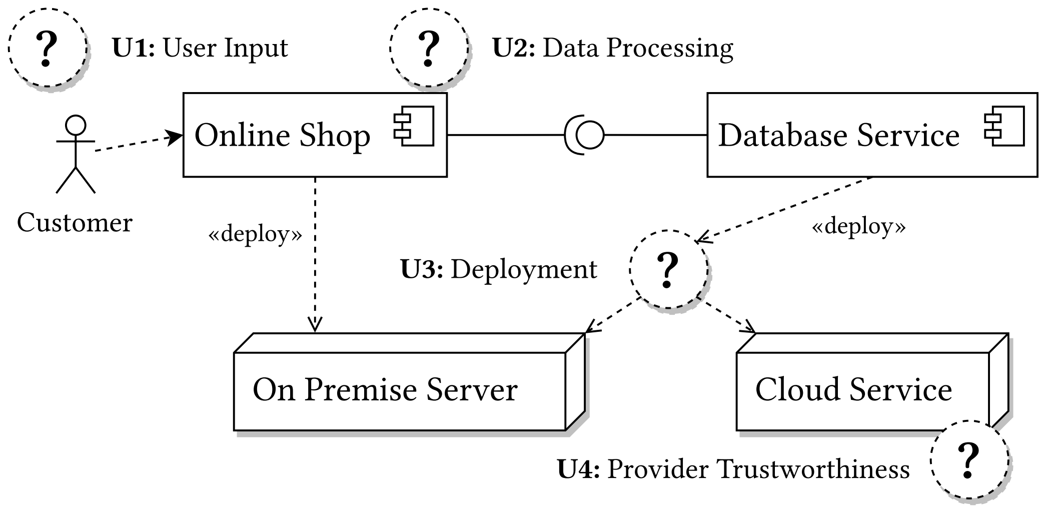


Figure 3.1.: Combined component and deployment diagram showing the software architecture of the online shop running example with annotated uncertainty sources depicted as circled question marks.

3.1. Online Shop Software Architecture

The software architecture is depicted in Figure 3.1. It consists of two components and two deployment locations. The *Online Shop* component serves as the interface for customers while the *Database Service* is used to persist information about customers, purchases, and the shop's inventory. Both components are connected through an interface which is provided by the *Database Service*. The *Online Shop* component is deployed on an *On Premise Server* while the *Database Service* can either be deployed locally on the same server or alternatively on a *Cloud Service* for better scalability.

Customers interact directly with the *Online Shop* component. The running example comprises three usage scenarios. First, customers input search details and request to view available items. Second, they select an item to purchase and enter payment and shipping details. In both scenarios, the input is processed in the *Online Shop* component first. Afterward, data is exchanged with the database, e.g., to query the inventory, or to store the purchase history. The third usage scenario considers users that request static information like support contact addresses. This information is provided directly by the *Online Shop* component without any communication with the database.

We consider two types of data in the running example: The online shop's public information, and the user's private purchase details. The former is publicly available and does not have to be protected—on the contrary, the online shop benefits from the public availability of this information. The latter can be classified as personal or sensitive information for which confidentiality requirements apply. Public availability of the customers' shopping and payment details would harm their privacy and might cause fraud or theft. The European General Data Protection Regulation (GDPR) demands that personal data of European citizens is only allowed to be stored and processed on European servers or on servers which ensure "an adequate level of protection" [73, Art. 45]. In the following, we assume that the online shop is operated in Europe, which implies that data processing on the *On Premise Server* complies with this confidentiality requirement. Depending on the cloud's server location and the trustworthiness of the provider, the storage on the *Cloud Service* might violate this requirement. Additional confidentiality requirements can include the user input, which has to be neither erroneous nor malicious, and also has to be validated and encrypted as part of the *Online Shop* processing.

We illustrate the behavior and data flows of the running example as Data Flow Diagram (DFD) in Figure 3.2. In addition to the DFD elements introduced in Section 2.5, we depict the users and components using gray boxes [233]. The upper half of the diagram represents the flow of query requests from the user via the online shop to the database and the response containing available items from the database to the user. Additionally, it shows the request for support information. In our simplified example, both data flows do not contain sensitive information. Thus, no confidentiality requirements apply. In the lower half, the customer purchases an item. The purchase is processed on the online shop and then stored in the database. This data flow is subject to confidentiality requirements.

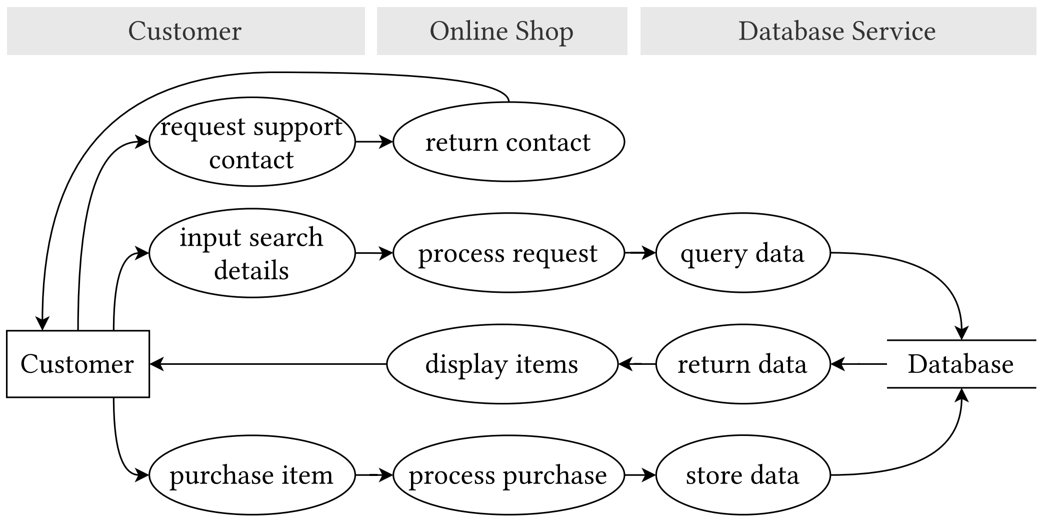


Figure 3.2.: Exemplary Data Flow Diagram (DFD) illustrating possible data flows through the online shop running example. The gray boxes indicate the user and components of the software architecture.

3.2. Exemplary Uncertainty Sources

As described previously in Section 2.1, uncertainty sources can exist within the software system and its environment [2]. In Figure 3.1 and in the following, we represent uncertainty sources in diagrams as circled question marks. We introduce four exemplary yet common sources of uncertainty to our running example:

U1 The users' input is a source of uncertainty. Although certain behavior regarding entered information can be expected already at design time, it cannot be guaranteed.

U2 The data processing is still uncertain. This can be the case due to an open design decision or due to the black box nature of a third-party off-the-shelf component.

U3 The component deployment is an uncertainty source. This can be caused by missing design decisions or dynamic reconfiguration at runtime, e.g., due to load balancing.

U4 The trustworthiness of a resource provider, which is used as a deployment location for a component or service, is uncertain and can only be assured, e.g., with policies.

We annotate the first Uncertainty **U1** to the customer using the online shop. Humans are a common uncertainty source within the literature, often referred to as *human in the loop* [195, 202]. Although the user input is uncertain, we can assume different input classes, e.g., valid input as expected, erroneous input due to human failure, or malicious input.

The second uncertainty source **U2** considering the data processing is annotated to the *Online Shop* component and its connection to the *Database Service*. This could be uncertain due to the black-box principle of software components [207] or due to a yet-to-be-defined design decision [159]. In our example, this could affect confidentiality if we do not know whether all requests to the *Database Service* are encrypted before sending. Additionally, input validation could be part of the processing, which analyzes the users' input to counter Uncertainty **U1**. Also, combinations of input validation and encryption are imaginable.

The third uncertainty source **U3** affects the deployment of the *Database Service* and is annotated on the associated arrow. Deployment decisions could be still uncertain at design time [167], or depend on the runtime situation, e.g., in the context of Self-Adaptive Systems (SASs) [272]. In Figure 3.1, we show two possible deployment locations, with the *Database Service* being either deployed on-premise or in the cloud.

The fourth uncertainty source **U4** is annotated to the *Cloud Service* and questions the providers' trustworthiness. Providers could void confidentiality either intentionally, e.g., by misusing data [60], or unintentionally, e.g., due to the lack of proper security requirements and appropriate measures [77]. To that end, we can only assume the trustworthiness and categorize whether providers are trustworthy enough or not.

Other and additional uncertainty sources are also possible within the running example, see ARC^{3}N. We selected these uncertainties as they demonstrate different locations, types, and origins, while still keeping the running example simple and easy to understand.

3.3. Summary and Outlook

In this chapter, we introduced a running example that will be used throughout this thesis. Already a software system of the size of our running example shows the challenge of dealing with multiple uncertainty sources. Uncertainty sources and impact locations arise at different points within the software system and its environment, which might cause complex interrelations and interactions [49] regarding confidentiality. Especially in larger software systems, analyzing them manually is not expedient. Thus, we need proper modeling, analysis, and mitigation strategies to ensure confidentiality.

Note, that the DFD shown in Figure 3.2 also depends on the outcome of the uncertainty sources **U1** – **U4**. Uncertainty **U1** could alter the customer process *input search details*, or Uncertainty **U3** could affect the structure of the DFD. Additionally, the concrete form of the DFD depends on the level of abstraction [64]. Building on previous advances in uncertainty representation [80, 255], we find that:

> **Finding:** Even small software systems like our running example can have many corresponding data flow diagrams under uncertainty. Uncertainty hinders precise modeling and analysis when not included properly within the model.

The presentation of the running example was based on DFDs and uncertainty which both have been introduced in **Chapter 2: Foundations**. Next, we use the running example to give an overview of all contributions in **Chapter 4: Overview**. We modeled the software architecture of the running example based on the Palladio Component Model (PCM). The model files can be found in our data set [98]. An overview and first impression is given in Appendix A.

3.4. In Simpler Words

A simple running example makes it easier to understand complex ideas. We introduce the running example of an online shop known from everyday life. Online shops deal with customer data and are thus required to protect that data and keep it confidential. Additionally, we presented different sources of uncertainty in our example. Examples are the behavior of customers or the trustworthiness of cloud service providers. Even in this simple example, we can see how this uncertainty harms confidentiality. For example, if we are unsure whether providers behave maliciously, customers' stored data could be threatened. One approach to face the challenge of analyzing confidentiality under uncertainty will be presented in the remainder of this thesis.

Part II.

Contributions

4. Overview

In this chapter, we give an overview of the contributions of this dissertation. This shall serve both as an introduction and as a high-level summary of the detailed explanations in the following chapters. All contributions can be located in the research area of design time analysis and architecture-based quality prediction [207]. Here, common benefits are simplifying the work of software architects and enhancing existing analysis approaches, e.g., regarding their applicability, scalability, accuracy, or usability [142].

The central research concern of this thesis is confidentiality analysis under uncertainty by using architectural modeling [99]. We divide the contributions of this thesis into three parts: First, the identification and classification of uncertainty regarding confidentiality. Second, the uncertainty impact analysis by propagating uncertainty within architectural models and data flow diagrams. Third, the architecture-based confidentiality analysis under uncertainty. Each of these contributions can be used independently and enhances the state of the art in its area. However, they can also be combined to form a comprehensive end-to-end approach which is an aspired goal in the community [115, 273].

In the following, we describe the general procedure for uncertainty-aware analyses and how our contributions are positioned within it. We briefly introduce these contributions afterward. Last, we give an overview of the tool support that has been developed to complement the research.

Literature: This chapter is based on the following (co-) authored publications: [EMLS 2021], [ECSA-C 2021], [ACM MODELS-C 2022], [IEEE ICSA-C 2023], [Springer ICETE 2023], [IEEE/ACM SEAMS 2024], [ACM/IEEE MODELS-C 2024]

4.1. Procedure for Uncertainty-Aware Analyses

Incorporating uncertainty in the analysis of software systems—thus achieving uncertainty awareness—has been discussed within the literature [80, 243, 255]. Garlan [80] proposed to include uncertainty as "first-class concern in the design, implementation, and deployment" of software systems. Hezavehi et al. [115] conducted a community survey and proposed a reference process for uncertainty management. They include phases like identification, modeling, impact analysis, and assessment. However, their process aims at Self-Adaptive Systems (SASs) that include run time strategies that are beyond the scope of this thesis. Regarding model-based analyses at design time [2], we consider four central activities:

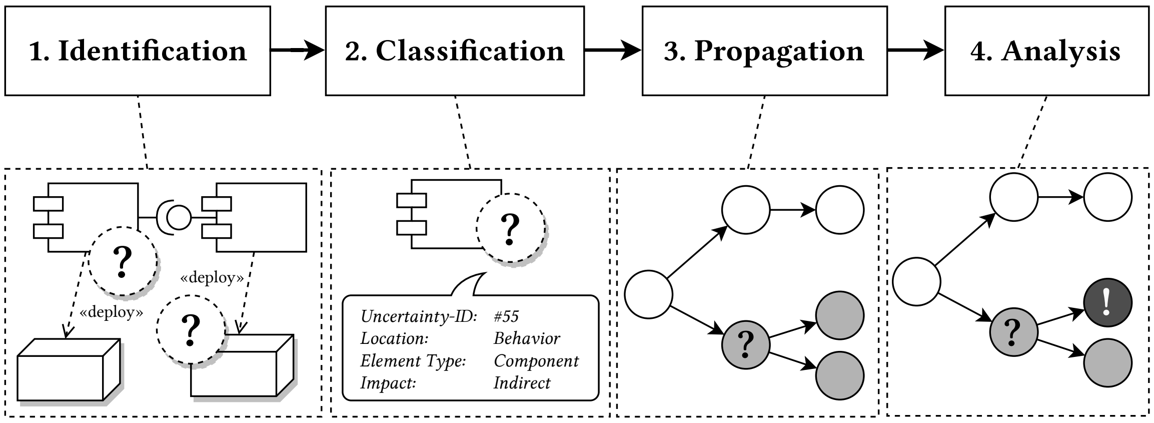


Figure 4.1.: Informal overview of the central activities of uncertainty-aware analyses

1. **Identification and awareness**: To include uncertainty sources in the analysis, they must be known first. Thus, raising awareness to recognize the presence of uncertainty in a system is the necessary first step.
2. **Classification**: To better understand the type of uncertainty sources and their properties, they can be classified. To that end, classifications and taxonomies provide the foundations for the documentation and the discussion of identified uncertainty.
3. **Propagation**: To assess the impact of identified and classified uncertainty sources, they can be propagated through the architectural model. Estimating the potential impact early helps in making more precise statements and decisions.
4. **Analysis**: To apply appropriate mitigation strategies, the effect of uncertainty on the software system's quality has to be analyzed. In our case, this means identifying confidentiality violations due to the identified, classified, and propagated uncertainty.

Figure 4.1 illustrates these four activities in an informal diagram. *First*, uncertainty sources have to be identified, e.g., within the software system or its environment. In Figure 4.1, we annotate these uncertainty sources in the architectural model similar to the running example presented in Chapter 3. In the *second* activity, these uncertainty sources have to be described according to an appropriate classification [104]. This does not only help in the understanding of the underlying problem but also enables an uniform handling in the later activities. The *third* activity is the propagation of uncertainty sources to estimate their potential impact. Here, software architects develop an understanding of which parts of the software systems could be affected and which parts can be safely ignored. The *fourth* activity is the confidentiality analysis under uncertainty. This activity yields confidentiality violations in those parts of the software system that have been impacted by uncertainty.

Finding: Literature proposed processes for uncertainty management. Common steps are the identification, classification, propagation, and analysis of uncertainty. By applying these activities to confidentiality analysis, we can provide a comprehensive approach to identify confidentiality violations under uncertainty.

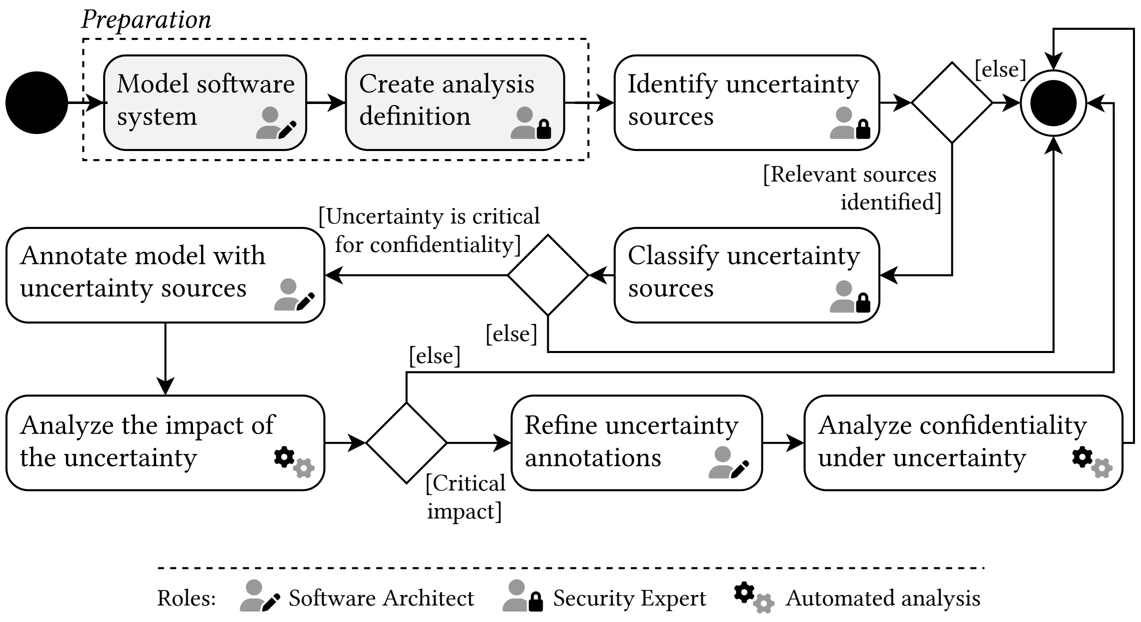

Preparation
Model software system
Create analysis definition
Identify uncertainty sources
[else]
[Relevant sources identified]
Classify uncertainty sources
[Uncertainty is critical for confidentiality]
Annotate model with uncertainty sources
[else]
[else]
Analyze the impact of the uncertainty
Refine uncertainty annotations
Analyze confidentiality under uncertainty
[Critical impact]
Roles: Software Architect Security Expert Automated analysis

Figure 4.2.: Proposed analysis procedure showing the modeling and analysis activities and assigned roles.

A high-level description like the informal overview introduced in Figure 4.1 serves as an overview or introduction. However, due to the high abstraction, two important details are left out. *First*, the procedure does not need to be linear. Sometimes, not all activities are required, e.g., because a critical subsystem has to be always analyzed despite the results of the propagation. Additionally, some activities or the procedure might be repeated, either because of the understanding of the uncertainty or because the system evolves [273]. Especially with regard to uncertainty and SASs, approaches tend to be iterative and incremental, as in continuous architecture evaluation [243], or in the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) loop [136, 274]. *Second*, the informal overview lacks role descriptions. Specifying expertise and required knowledge helps in understanding the application of analysis procedures.

To address these shortcomings, we provide a more detailed procedure with Figure 4.2. We reuse the roles proposed for architectural confidentiality analysis [233], i.e., *software architect* and *security expert*. Note, that we do not introduce a role for *uncertainty experts*. With appropriate means like tool-supported identification, classification, and analysis approaches, only minimal additional knowledge shall be required. Thus, we do require expert knowledge of uncertainty classification or uncertainty sources in general.

The procedure starts with the preparation phase which comprises modeling the software system [234, 237] and creating the analysis definition [105]. These steps originate from architectural data flow analysis [233]. An architectural model has to be defined by a software architect using an Architectural Description Language (ADL) like the Palladio Component Model (PCM) or Data Flow Diagrams (DFDs). Afterward, confidentiality requirements serve as input for the security expert to create the confidentiality analysis definition. This enables identifying confidentiality violations without considering uncertainty.

First, uncertainty sources have to be identified by the security expert. Although we do not require expert knowledge of uncertainty, security expertise helps estimate the potential impact on confidentiality. Uncertainty sources can be identified in discussions, architecture reviews, or using checklists, guidelines, or catalogs [103]. If no relevant sources have been identified, the procedure ends until new knowledge about uncertainty is gained. This step represents the first activity of identification and awareness.

In the next step, the identified uncertainty sources are classified according to an appropriate classification [104]. This step is performed by the security expert with the help of existing catalogs that bootstrap the classification. If only uncertainty sources have been identified that clearly have no impact on confidentiality, the procedure ends. This step represents the second activity. Based on the result of the classification, the uncertainty sources are annotated to the architectural model. This does not require security expertise but knowledge of the ADL in use and is thus driven by the software architect.

Afterward, the uncertainty impact analysis propagates the uncertainty source within the software system and calculates the potential impact [102]. This step represents the third activity and is fully automated and thus requires neither knowledge about security nor software architecture. If no relevant uncertainty impact is identified, e.g., because only non-critical parts of a software system are affected, the procedure ends. Otherwise, the confidentiality under uncertainty has to be analyzed which is the fourth activity. This includes a more detailed specification of the uncertainty sources for a precise analysis result. While the refinement of the information about the uncertainty is conducted by the software architect, the analysis is fully automated. Afterward, the procedure ends.

Note that the end of the procedure does not imply the end of the design and analysis process. If the knowledge about uncertainty or the software systems changes, the procedure can be restarted. Additionally, all artefacts like the modeled software architecture, the analysis definition, collected uncertainty sources and their classification and annotation can be reused. Thus, we consider this procedure to be iterative. As stated previously, we do not require expert knowledge about uncertainty. The classification and the automated analyses, i.e., the uncertainty impact analysis and the uncertainty-aware confidentiality analysis encapsulate the required knowledge [101]. The procedure has to be incremental as the existence of uncertainty is inherent to software systems. It can never be assured to be aware of all uncertainty sources, as *unknown unknowns* [195] might exist. In this way, uncertainty is similar to bugs in software systems, where testing can only be used to show the “presence of bugs, but never to show their absence” [67]. Additionally, the procedure requires exit points after each activity when more knowledge is gained about the impact of uncertainty on the software system’s confidentiality.

ℹ **Finding:** Due to the nature of uncertainty, it is impossible to ensure its absence. However, not every uncertainty source causes confidentiality violations. This depends on many aspects like the uncertainty itself, confidentiality requirements, security measures, or the software’s architecture. A procedure for uncertainty-aware confidentiality analysis has to be iterative and incremental.

4.2. Overview of the Contributions

In the following, we introduce the central contributions of this thesis and relate them to the procedure presented hereabove. We divide the contributions into three main parts:

C1 **Identification and classification**: Recognizing and understanding the types of uncertainty and their relation to confidentiality from an architectural point of view.

C2 **Uncertainty impact analysis**: Propagating uncertainty using architectural models and DFDs to predict the uncertainty impact on the system's confidentiality.

C3 **Uncertainty-aware confidentiality analysis**: Extending architecture-based confidentiality analysis to consider uncertainty as a first-class concern in both modeling and analysis to identify confidentiality violations under uncertainty.

First, with Contribution **C1**, we aim to support software architects in identifying and classifying relevant uncertainty sources with a potential impact on confidentiality. This is an elemental first step, as awareness about uncertainty sources and their properties is key in documentation, modeling, discussion, and analysis [81]. This combines the activities of identification and classification in one contribution, as both are interdependent and support each other. We contribute a classification of software-architectural uncertainty [104] and support the collaborative identification of uncertainty sources [103]. During this research, we also examine Cyber-Physical Systems (CPSs) as a relevant type of uncertainty-afflicted software systems [2]. Our findings answer ❓ **Research Question 1**, which is related to the identification and description of uncertainty with regard to confidentiality.

Our second Contribution **C2** addresses the impact analysis of uncertainty, which supports the early prediction and assessment of the uncertainty impact. Similar to change impact analysis [210, 211], we propagate uncertainty through the software architecture. Besides the theoretical foundations of modeling and propagating uncertainty in architectural models and DFDs, this contribution also includes a fully automated analysis [102]. Based on our concept of uncertainty propagation in DFDs, additional research led to the formulation of Uncertainty Flow Diagrams (UFDs) [49]. They support software architects in understanding and analyzing uncertainty interactions [50]. In sum, these findings provide an answer to ❓ **Research Question 2**, which relates to architectural uncertainty propagation and impact analysis.

In our third Contribution **C3**, we develop confidentiality analysis approaches that respect uncertainty. To that end, we extend an existing architecture-based confidentiality analysis which utilizes DFDs [233, 236]. We provide approaches for different types of uncertainty, e.g., environmental uncertainty [37], or structural uncertainty [266]. By defining an analysis framework for architectural data flow analysis, we also provide the baseline for more comprehensive approaches [36]. We use this framework to research the expressiveness, accuracy, and scalability of data flow-based confidentiality analysis under uncertainty [101]. These approaches address ❓ **Research Question 3**, which asks how to consider uncertainty in architectural confidentiality analysis.

4.3. Tool Support

In addition to answering the research questions of this thesis, we provide comprehensive tool support. For each contribution, we realized tooling to demonstrate the applicability of the concepts. The prototypical implementation is not only used in the evaluation but also shall support further research [36]. A recent Systematic Literature Review (SLR) found a lack of replication packages in software architecture research [142]. To address this, all tool support is open source, publicly available and also part of this thesis' data set [98].

Automated analyses support software architects because they encapsulate expert knowledge, e.g., annotation and propagation rules [104]. Especially in large software systems or systems of systems, manual analysis is also not feasible and error-prone [234]. In our iterative procedure, it cannot be assumed that software architects manually evaluate hundreds of DFD nodes [104] without mistakes. Additionally, after each change to the annotated uncertainty sources or the software system, the manual work would have to start all over again. Thus, it is important to ensure that the developed concepts can be automated to a high degree without manual effort required from software architects. Similar to the division into three contributions, we provide three tooling artifacts. We introduce and shortly summarize their functionality in the following[1].

🔧 ARC^{3}N stands for rese**arc**h **arc**hive for Software-**arc**hitectural u**n**certainty. This is a web-based interactive catalog of uncertainty sources that can affect confidentiality [103]. It simplifies the identification by connecting a classification [104] to a catalog approach. All data is publicly available, extensible by other researchers, and can be integrated into existing analyses. This catalog of uncertainty sources supports Contribution **C1**.

🔧 UIA means **u**ncertainty **i**mpact **a**nalysis and is a tool for propagating uncertainty sources within architectural models and DFDs. It extends the data flow analysis framework [36]. The tool support is realized as an Eclipse plugin and can be integrated into the existing Palladio tooling [207]. The analysis yields an impact set for uncertainty sources which can be annotated to the software architecture. Besides the annotation, no additional effort is required for existing PCM models as the propagation is fully automated. See Appendix B for an exemplary impact analysis result. This supports our second Contribution **C2**.

🔧 Abunai is short for **a**rchitecture-**b**ased and **un**certainty-**a**ware confidentiality analys**i**s[2]. Similarly to the uncertainty impact analysis, this tooling is realized as an Eclipse plugin and extends the data flow analysis framework [36] within the Palladio approach [207]. This analysis enables software architects to refine the uncertainty annotations by providing possible scenarios [101]. Afterward, the automated analysis considers the uncertainty impact in the data flow analysis and yields confidentiality violations with respect to uncertainty. See Appendix A and Appendix C for exemplary analysis inputs and analysis results. This is one of the analysis approaches that support Contribution **C3**.

[1] At the time of writing—and hopefully a long time to come—the most up-to-date version of all tool support is online available here: `https://abunai.dev/`. Once the link is broken, please refer to the data set [98].

[2] Furthermore, the Japanese word *abunai* translates to dangerous, risky, or uncertain. How fitting!

4.4. Illustration using the Running Example

We demonstrate the procedure, the contributions, and the tool support briefly using the running example introduced in Chapter 3. We assume a team of software architects and security experts that already modeled the software system using PCM and created the analysis definition to analyze the system's confidentiality. See the preparation phase in Section 4.1 for more details.

First, relevant uncertainty sources have to be identified by security experts. As part of our first Contribution **C1**, the tool support 🔧 ARC^{3}N provides a catalog of uncertainty sources that serve as a starting point. The catalog also contains all uncertainty sources which have been annotated to the running example, e.g., the user input (**U1**), or the provider's trustworthiness (**U4**). As the uncertainty sources in this catalog already are classified, the subsequent step of uncertainty classification is simplified or can be skipped. All of the contained sources are relevant regarding confidentiality. The software architect annotates the uncertainty sources within the architectural model in the running example. For example, the user input uncertainty (**U1**) is annotated to the *Customer*, and the data processing uncertainty (**U2**) is annotated to the *Online Shop* component. In Figure 3.1, we illustrate this annotation using circled question marks.

Using the architectural models and the annotations as input, the impact analysis calculates the potential impact on confidentiality. This is part of our second Contribution **C2**, and fully automated with 🔧 UIA. In the running example, the impact set of all uncertainty sources contains the *Database Service* component. As the database stores confidential data, we interpret this impact as being critical continue with the step of uncertainty source refinement. The precise modeling of uncertainty sources and scenarios and the analysis are part of our third Contribution **C3**. In the running example, the software architects describe all relevant scenarios, e.g., the possible deployment locations of the *On Premise Server* or the *Cloud Service*. Afterward, the automated analysis 🔧 Abunai yields confidentiality violations due to single or multiple uncertainty scenarios. In our running example, the lack of proper data processing (**U2**) and the deployment in the cloud (**U3**) without proper security measures violate confidentiality. After addressing these issues, the procedure can be repeated. This also happens whenever the software system changes, e.g., due to the introduction of a new encryption component, or the knowledge about the uncertainty changes, e.g., because the scenario of the lack of any data processing can be excluded.

4.5. Summary and Outlook

In this chapter, we discussed the procedure of uncertainty-aware analysis and summarized the contributions of this thesis. First, we investigated multiple proposals for required activities in uncertainty-aware analysis. We combined them to propose a procedure for architecture-based confidentiality analysis under uncertainty. This procedure contains the central activities of identification, classification, propagation, and analysis. Besides these activities, our proposed procedure also contains modeling activities carried out by software architects and security experts. The analyses are fully automated and thus shall not require additional knowledge about uncertainty. The procedure is both iterative and incremental to react to changes in the system, its environment, and its uncertainty sources. Put simply, uncertainty is uncertain, and we have to deal with it.

Afterward, we gave an overview of the central contributions of this thesis. They cover the aforementioned activities of identification and classification with Contribution **C1**, the propagation and uncertainty impact analysis with Contribution **C2**, and the uncertainty-aware confidentiality analysis with Contribution **C3**. All contributions are tool-supported with three research artifacts: 🔧 ARC^{3}N collects and explains uncertainty sources and their classification, 🔧 UIA automatically propagates uncertainty in an architectural model to calculate the potential impact, and 🔧 Abunai provides uncertainty-aware confidentiality analysis based on architectural modeling and DFDs. Last, we illustrated the tool-supported contributions and their interrelationships using the running example. The application of our prototypical tooling is also shown in Appendix A, Appendix B, and Appendix C.

Throughout this chapter, we stressed the lack of an explicit role for uncertainty experts in the proposed procedure. Ideally, automated analyses and other artifacts should encapsulate the required knowledge about uncertainty. We will return to this idea in the following chapters and explain the consequences in more detail. The illustration in this chapter was based on ❯ **Chapter 3: Running Example**. In the following three chapters, we present the contributions in detail. In ❯ **Chapter 5: Identification and Classification of Uncertainty Regarding Confidentiality**, we show the first Contribution **C1**, in ❯ **Chapter 6: Uncertainty Propagation to Enable Uncertainty Impact Analysis**, we introduce the second Contribution **C2**, and in ❯ **Chapter 7: Uncertainty-Aware Data Flow Analysis to Identify Confidentiality Violations**, we discuss the third Contribution **C3**.

4.6. In Simpler Words

This chapter gives an overview that helps to better understand the relation of the individual contributions of this thesis. Our central research concern is analyzing confidentiality under uncertainty. Uncertainty can threaten the confidentiality of sensitive user data like names or addresses. An example is the uncertainty about the encryption of this data. If we cannot guarantee the correct encryption, we cannot guarantee confidentiality.

We propose four activities which help to act upon such uncertainty. First, the uncertainty sources have to be identified and described. We developed tools that contain catalogs and descriptions of uncertainty sources. This minimizes the effort and expertise requirements of software architects and security experts. Afterward, we assess the impact of uncertainty on confidentiality using an automated analysis. For example, the lack of encryption could impact confidentiality. Our contributions help to identify that this uncertainty exists and how great its impact is. If the impact is relevant, our automated analysis helps to find confidentiality violations that can be caused by this uncertainty.

5. Identification and Classification of Uncertainty Regarding Confidentiality

In this chapter, we present the first Contribution **C1**. This contribution covers the first two activities shown in Section 4.1, i.e., the identification and classification of uncertainty. We focus on confidentiality as a central quality property for this thesis.

As introduced in Chapter 1, confidentiality demands that "information is not made available or disclosed to unauthorized individuals, entities, or processes" [120]. With the growing size and connections of today's software systems, ensuring confidentiality becomes a major challenge. We describe in Chapter 4 that we use design-time confidentiality analysis [236, 256] to identify flaws early and to avoid costly repairs of running systems [32]. Here, data flow-oriented analyses became common because "problems tend to follow the data flow, not the control flow" [239]. However, especially in early development and in complex systems of systems, the software architecture is subject to uncertainty. This does not only affect decision making—also known as the *cone of uncertainty* [167]—but even blurs which decisions should be prioritized. When not managed properly, the lack of awareness of uncertainty can void a system's confidentiality. Also, the OWASP Top 10 [192] lists issues like *insecure design* as top security risks. Thus, we have to actively address the phenomenon of uncertainty and its impact on the confidentiality analysis results.

Managing uncertainty includes activities like the identification and classification [2, 115, 272]. Here, multiple taxonomies were defined to better understand the nature of uncertainty [195, 202, 263]. However, they mostly originate from the domain of Self-Adaptive Systems (SASs) and do not focus on confidentiality. The consequences are a lack of applicability and an increase in ambiguity. The relation of software architecture, confidentiality and uncertainty remains unclear [99]. And while "there is growing consensus on the importance of uncertainty" [115], much is yet unknown regarding the impact of uncertainty on software systems [80]. Hezavehi et al. [115] conducted a survey on uncertainty. They find a "lack of systematic approaches for managing uncertainty" [115] and that uncertainty should already be addressed at design time. This statement is supported by the work of Troya et al. [255]. They conducted a Systematic Literature Review (SLR) and analyzed 123 papers. They state that "software models are still falling short of explicitly representing uncertainty" [255] and that software engineers require more help "to identify the types of uncertainty that can affect their application domains" [255].

Uncertainty can arise within software systems, e.g., design choices, or reconfiguration at runtime, and their environment, e.g., sensor data, or humans in the loop [2, 195]. To enhance existing confidentiality analysis approaches, software architects require already known uncertainty sources, which are also called *known unknowns* or first-order uncertainty [195], e.g. known variation of sensors or user behavior. Second-order uncertainty, also called *unknown unknowns*, cannot be analyzed due to the lack of knowledge about its sources, e.g., due to unforeseen environmental conditions, which limits the development of mitigation strategies [273]. This Uncertainty Awareness Problem (UAP) impedes the comprehensive analysis of software systems which poses security threats that could have been resolved at design time. However, it is a highly subjective phenomenon, as it only depends on the knowledge or awareness of the software architect conducting the analysis. Put simply, one cannot analyze what one does not know.

The UAP and the related need for collecting uncertainties are supported in the literature. Recent surveys and research agendas highlight this problem of dealing with unanticipated change [272], the need to share knowledge of uncertainty [115], and the development of "reusable methods to assess and manage uncertainties" [273]. A class between known and unknown uncertainty was proposed, as not every change is completely unanticipated and can be addressed by flexible designs using intentional over-engineering or meta-adaptation [81]. To address this, classification approaches alone are not sufficient but require additions like collections and surveys [202, 255], or expert knowledge [70, 159]. Classifications only describe properties of uncertainty which does not resolve the awareness problem, publication-based collections are hard to extend dynamically and are thus prone to become outdated and incomplete, and relying on experts is not expedient. Developing a joined approach of identification and classification is a step towards end-to-end approaches that promise a higher impact than incrementally enhancing existing analyses [273].

In sum, we aim to address both the identification and classification problem to include uncertainty in confidentiality analysis based on Data Flow Diagrams (DFDs). This breaks the limitation of previous approaches that were only able to analyze confidentiality with perfect knowledge, i.e., by excluding uncertainty about software systems and their data [233]. We summarized this research concern in the first research question:

Research Question 1: How to identify, describe and classify software-architectural uncertainty regarding its relation to and impact on confidentiality?

The remainder of this chapter is structured as follows: First, we summarize the problem statement. We revisit the relation of uncertainty, confidentiality, and software architecture. We investigate existing uncertainty classifications and their appropriateness to addressing confidentiality. Then, we define a new classification of uncertainty regarding confidentiality and discuss, based on this classification, how to represent uncertainty in data flow diagrams. Afterward, we build on the classification to present an uncertainty catalog approach that helps in the identification of uncertainty and addresses the UAP. We close the chapter with known assumptions and limitations, and also a summary and an outlook.

Literature: This chapter is based on the following (co-) authored publications: [EMLS 2021], [ECSA-C 2021], [ACM MODELS-C 2022], [Springer ICETE 2023], [ACM/IEEE MODELS-C 2024]

5.1. Problem Statement

We summarize the problems **P1** – **P5** addressed by Contribution **C1**. Finding solutions to these problems helps to provide a comprehensive answer to ❓ **Research Question 1**.

P1: Knowledge about the relation of uncertainty and confidentiality We need to better understand the role of uncertainty and its relation to confidentiality in software architecture. This includes considering uncertainty at design time and runtime and also relating uncertainty to Architectural Design Decisions (ADDs), which is related to architectural decision-making under uncertainty [159]. We see this as a precondition to defining precise classifications of uncertainty regarding confidentiality [99]. The solution to this problem is considered to be non-trivial due to the variety of uncertainty.

P2: Classification of uncertainty tailored to confidentiality We require classifications tailored to confidentiality and software architecture. There exist classifications from other domains [195, 202, 263], but they fall short in representing the impact of uncertainty on confidentiality. Additionally, they do not relate the classified uncertainty to software architecture and ADDs. Here, a classification scheme to aid software architects in understanding the impact of uncertainty on confidentiality. This shall raise awareness of properties of uncertainty and their relevance for choosing appropriate ADDs and mitigation strategies. We intentionally speak of a classification rather than a taxonomy because we focus on a subset of uncertainty, i.e., known uncertainty on architectural abstraction.

P3: Representation of uncertainty as first-class concern in DFDs Garlan [80] raised the need to represent uncertainty as a first-class concern in the design of software systems. This applies to ADLs as well as to DFDs. Whether in tool-supported analysis or in discussions between software architects, uncertainty must be represented appropriately to be taken into account. This includes defining a notation for uncertainty impact in DFDs that works for all uncertainty types of the classification. Only limited work towards considering uncertainty in DFDs exists [68], which only considers selected uncertainty types.

P4: Expert knowledge required to understand uncertainty sources Due to space limitations in publications, previous approaches only explain classifications briefly without comprehensive data sets [202]. We stress the importance of explainability [26], e.g., using examples, interactive visualizations, and context information from the underlying classification. Additionally, classifications provide dimensions or categories to describe uncertainties but often fall short in relating multiple uncertainty types which impairs usability [104]. It should be possible to filter, search, and navigate between uncertainties, which requires defined relations between them. This is especially relevant as uncertainty sources are rarely independent and interactions can occur [49]. We do not require an *uncertainty expert* role in our procedure, see Section 4.1. Therefore, our contributions to the identification and classification must be understandable without extensive training.

P5: Extensible uncertainty source catalogs to support the identification The Uncertainty Awareness Problem (UAP) states that we need awareness of uncertainty sources to consider them in the uncertainty mitigation process. One approach to this problem is a collection of uncertainty sources, building on the solutions to the aforementioned problems. However, publication-based collections are hard to reuse and extend [202] and thus prone to become outdated and incomplete. Knowledge can be scattered between researchers or organizations which complicates collaboration [251]. Additionally, research artifacts often are unavailable [142] or partially broken [59, 86]. To address this, an open-source and publicly available catalog is required, which shall be easy to discuss and extend by researchers and practitioners. To enable the seamless integration into uncertainty-aware analyses and to enable end-to-end approaches [273], it shall be possible to transform the catalog into a machine-readable format like JavaScript Object Notation (JSON) or XML. Related collections of architectural knowledge [86], or privacy patterns [59] have shortcomings regarding usability, extensibility, and longevity of the contained information.

5.2. Understanding the Relation of Uncertainty, Confidentiality, and Software Architecture

We address the first Problem **P1** by defining software-architectural uncertainty, providing a distinction of uncertainty source and uncertainty impact, and relating the phenomenon of uncertainty to the *cone of uncertainty* [167], confidentiality, and ADDs.

5.2.1. Defining Software-Architectural Uncertainty

We understand *software-architectural uncertainty* as uncertainty, that can be described on architectural abstraction and where (early) awareness enables considering its impact on quality attributes like confidentiality. We do not only refer to *known unknowns*, as this only implicates awareness which is too imprecise. Additionally, whether an uncertainty sources is known or unknown depends on the knowledge or awareness of the software architect—and is not directly related to the uncertainty as such. We require architectural abstraction, e.g., as part of an architectural model with a specified impact on software-architectural elements, e.g., software components, interfaces, or hardware resources. While requiring architectural abstraction, we do not limit the representation to a specific notation but consider multiple model types like ADLs or DFDs. We exclude higher orders of uncertainty [195] as their impact cannot immediately be expressed due to the lack of awareness. However, awareness can be raised with increasing knowledge, e.g., by asking a domain expert [194], by using a classification scheme for systematic treatment [263], or by using uncertainty catalogs [103]. We stress that this subset can be accurately assessed using architecture-based analysis—and architectural methods in general. Uncertainty that is not covered by this definition is not in the scope of this thesis.

When speaking about uncertainty in software architecture, we propose to distinguish between the uncertainty *source* and the uncertainty *impact*. Uncertainty *sources* can exist within the software system or its environment [2], e.g., due to a lack of knowledge or natural variability. As such, it is hard to both resolve uncertainty sources directly, and also to analyze their relevance without considering the system under analysis. However, the *impact* of uncertainty can be analyzed and mitigated [2]. Speaking of uncertainty in terms of *impact* rather than only considering the uncertainty's type or source enables software architects to focus on its mitigation during design, e.g., to enhance confidentiality. Most importantly, the *source* and the *impact* of the uncertainty do not have to be located in the same element—or even the same component or part—of a software architecture. Understanding the impact locations of uncertainty helps in understanding uncertainty sources and, ultimately, in uncertainty mitigation. This potentially large spatial separation of *source* and *impact* especially motivates the need for uncertainty propagation [49, 102].

ℹ **Finding:** Software-architectural uncertainty can be represented and analyzed on architectural abstraction, e.g., as a first-class concern in Architectural Description Languages (ADLs) or Data Flow Diagrams (DFDs). We distinguish between the uncertainty source, which can be within the software system or its environment, and the uncertainty impact, which is the location within the software system where the uncertainty affects a quality property like confidentiality.

In the running example presented in Chapter 3, all four uncertainty sources represent software-architectural uncertainty. The user input (**U1**) and the provider trustworthiness (**U4**) are in the environment of the software system while the data processing (**U2**) and the deployment (**U3**) lie within the software system. Because all four represent software-architectural uncertainty, we can annotate the uncertainty sources within the architectural model, e.g., to a component or resource. Here, we can also see the difference between source and impact: The uncertainty source of the user's input (**U1**) is annotated to the *Customer* However, the actual negative impact on confidentiality might happen within the *Database Service* component that is depicted on the other side of the system. We illustrate the system context boundary and the impact of Uncertainty **U1** in Figure 5.1.

5.2.2. Existing Notions of Uncertainty

Multiple definitions of uncertainty exist, see Section 2.1. Regarding software-architectural uncertainty, we consider two major points of view. First, the concept of uncertainty stemming from the research community of SASs [272, 273]. Here, uncertainty is often referred to as unanticipated change, fuzziness, noise, or incomplete knowledge [273] and located within the system context or environment [272]. Uncertainty is seen as a challenge which has to be considered within the system design and "engineering software-intensive systems that can handle uncertainty is complex" [272]. Second, from a software engineering point of view, the *cone of uncertainty* describes the inherent phenomenon of the lack of knowledge about a software system in early development phases [167]. Here, uncertainty is constituted as part of every software project and is reduced over time,

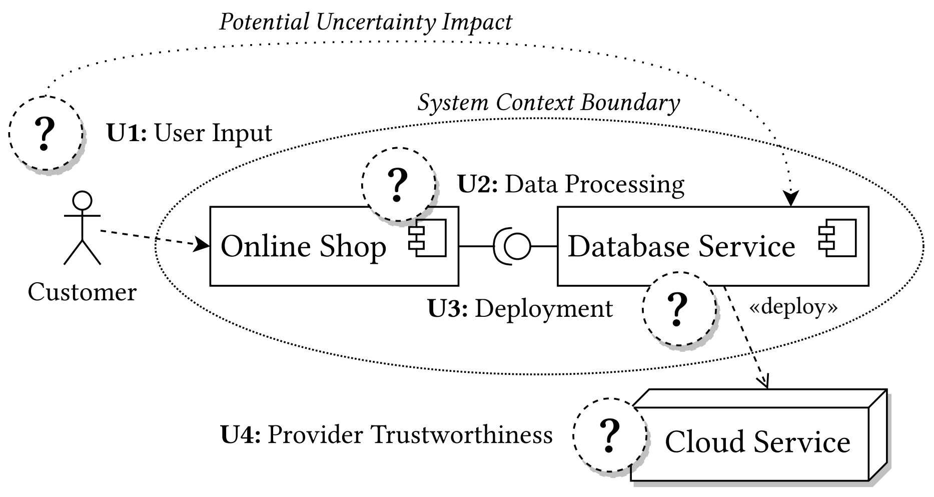


Figure 5.1.: Combined component and deployment diagram of the running example with annotated uncertainty sources, the system context boundary, and the impact of the user input uncertainty source.

e.g., by making design decisions like ADDs. In our running example, the user input (**U1**) as human in the loop and the provider trustworthiness (**U4**) as uncertain environment represent uncertainty sources known from SASs. Opposite to this, the data processing (**U2**) and the deployment (**U3**) represent design decisions that would reduce the *cone of uncertainty*. Regarding software-architectural uncertainty, we do not distinguish between these two perspectives, as both are relevant regarding confidentiality. As long as we can represent the uncertainty sources in architectural models and analyze their impact, we can asses their effect on confidentiality.

When dealing with software-architectural uncertainty, considering ADDs helps to structure the design process. At the beginning of this process, much is yet unknown or imprecise and ADDs are made under assumptions [167], e.g., that the provider is trustworthy in Uncertainty **U4**. Making this uncertainty explicit can help to mark decisions as challenged [150] and consider backtracking[1]. While some uncertainty only exists due to not yet decided ADDs, others cannot be reduced immediately [195], e.g., Uncertainty **U1** regarding the user input. Still, creating awareness of the uncertainty source can help refine the architecture and make more informed statements about confidentiality. This way, uncertainty can be understood, modeled, analyzed, and—eventually—managed.

There are multiple relevant properties of ADDs that help in the mitigation of uncertainty. The number of solutions of related ADDs [124] can help to estimate whether the uncertainty can already be fully reduced at design time, e.g., by design space exploration [70, 144]. We distinguish between closed sets that could at least partially be analyzed and open sets with a potentially infinite number of solutions or configurations. In our example, Uncertainty **U3** relates to the ADD of the deployment of the *Database Service* and repre-

[1] A couple of years ago, I had the possibility to discuss this very point with Philippe Kruchten. He stressed the importance and costs of backtracking due to false assumptions and design faults. This was another motivation to research the early identification and impact analysis of software-architectural uncertainty.

sents a closed set. But even with a closed set of alternatives, one cannot guarantee that a given ADD might not be challenged in the future [150] due to changes in requirements or the system's execution context. Thus, when speaking about decisions under uncertainty, considering the probability, possibility and costs of revisions can help to quantify the risk. This awareness also helps in the prioritization of ADDs and deciding whether existing mitigation is sufficient.

However, only considering ADDs to understand the impact of uncertainty is not enough because uncertainty might not be directly connected to a single decision, e.g., to resolve Uncertainty **U4**. We argue to also consider which architectural elements are affected, rather than only considering this transitively via the impact of ADDs. This does help in understanding the consequences of uncertain influences regarding confidentiality and also helps to connect these to architectural analyses to identify confidentiality violations.

Finding: Different notions of uncertainty exist, e.g., the *cone of uncertainty* or uncertainty in Self-Adaptive Systems (SASs). We do not limit software-architectural uncertainty to one of these notions, as all of these uncertainties can affect confidentiality. Regarding the software architecture, considering Architectural Design Decisions (ADDs) helps for early assessment and prioritization.

5.2.3. Investigating Existing Uncertainty Classifications

Based on our understanding of the relation of uncertainty, software architecture, and confidentiality presented in the previous section, we investigate existing uncertainty classifications stemming from related work. Hereby, we address Problem **P2** regarding the need for a classification tailored to confidentiality. As discussed previously and also in Section 2.1, there are many types, notions, and dimensions of uncertainty. In the following, we follow a top-down approach by discussing shortcomings of classifications and taxonomies while focusing on confidentiality.

We gathered and assessed categories, i.e., dimensions, or characteristics, and options, i.e., entries of a category in Section 2.1. Here, the first category was the *location* of uncertainty, see Table 2.1. Although this category is used in many taxonomies [41, 162, 184, 195, 255, 263], is has several shortcomings. There is no common distinction between the source and the impact location of uncertainty. Also, there is no common understanding of the term *model* as the taxonomies originate from different research areas. This impedes the common understanding of uncertainty regarding confidentiality. In the running example, Uncertainty **U3** about the deployment could be classified as *context*, *model structural*, *model technical*, and *belief uncertainty*. Such ambiguity can invalidate the purpose of a classification. The second category was the *level* of uncertainty, see Table 2.1. In the running example, all uncertainty sources represent known unknowns, i.e., first-order uncertainty, because we are aware of the sources and can annotate them in the architectural model. However, the concrete representation might be different, e.g., using scenarios to describe the deployment of Uncertainty **U3**. As discussed previously, we find distinguishing between a *known* and *unknown* uncertainty to be not expedient as it only depends on the viewpoint of the software architects whether they are aware of the uncertainty source.

The next three categories are the *nature*, *manageability*, and the *resolution time*, see Table 2.2. In our running example, the uncertainty sources in the software system, i.e., Uncertainty **U2** about the data processing and Uncertainty **U3** about the deployment, represent epistemic uncertainty. The uncertainty sources in the system context, i.e., Uncertainty **U1** about the user input and Uncertainty **U4** about the provider trustworthiness can be classified as epistemic or aleatory, depending on the point of view. This shows again the ambiguity of this category. All uncertainties in the running example can be described using scenarios and resolved between design and run time.

The last categories are the *impact on quality* of uncertainty and the *relationships* between uncertainties. To focus analysis capabilities, software architects must know about potential impacts and their severity. Although we focus on confidentiality, the analysis of multiple properties is possible, e.g., by using software architecture evaluation approaches under uncertainty [243]. In the running example, all four uncertainty sources can have an impact on confidentiality. Uncertainty relationships can cause hard-to-find problems regarding quality properties [50]. However, they often require further analysis based on the software system under study [49]. In the running example, there is such a relationship between Uncertainty **U3** about the deployment and Uncertainty **U4** about the provider's trustworthiness. The latter only affects the confidentiality of the software system if the cloud provider is chosen in the first place. As described above, such uncertainty interactions have to be further analyzed [49], and cannot be generally described using a classification.

Finding: The literature comprises many classifications and taxonomies of uncertainty that propose categories like location, level, or nature. Despite being mostly relevant regarding confidentiality, they use varying terminology, contain ambiguity, and do not focus on uncertainty sources. Thus, they have to be adapted to classify software-architectural uncertainty sources.

5.3. Classification of Uncertainty Regarding Confidentiality

Based on the categories and options we identified in existing classifications of uncertainty, we define a novel classification scheme of uncertainty regarding confidentiality. This allows us to build on existing knowledge from the domain of SASs while aligning the classification to software-architectural uncertainty and confidentiality. Additionally, we focus on uncertainty *sources* as the primary concern regarding identification and early mitigation [195, 202], see Section 5.2. This addresses our second Problem **P2** regarding the need for a classification tailored to confidentiality.

Our classification scheme consists of 8 categories with a total of 27 options and is shown in the tables 5.1 – 5.8. The categories are based on taxonomies of uncertainty [41, 71, 162, 184, 195, 202, 263], related work on ADDs [124, 150] and ADLs like the PCM [207]. A category-based classification helps to group uncertainties and identify similar characteristics and

Location: Describes where an uncertainty manifests itself within the architecture and which view or concern of the software architecture is primarily applicable. System-Independent, Nominal, Immutable	
System Structure	The uncertainty is related to the structure of the software system and becomes visible in structural views, e.g., component diagrams, or class diagrams.
System Behavior	The uncertainty is related to the behavior of the software system and becomes visible in behavioral views, e.g., activity diagrams, or sequence diagrams.
System Environment	The uncertainty is related to the environmental context of the software system and becomes visible when considering external factors, e.g., in deployment diagrams.
System Input	The uncertainty is related to the input to the software system and becomes visible when considering external interface specifications, or usage descriptions.

Table 5.1.: Uncertainty classification regarding confidentiality. Category: Location.

mitigation approaches. Once classified, the information can be reused across different software architectures. To create this classification, we assessed and adapted existing categories and combined or refined their options, see Subsection 5.2.3. We repeated this process until each category fulfilled its purpose, i.e., being able to describe and partition uncertainty sources with respect to confidentiality in software architectures.

In this work, we focus on confidentiality requirements during the design time. Thus, the categories should be interpreted from an architectural point of view, e.g., while modeling a software system. In the following, we explain each category in detail. We provide information about the rationale and possible benefits of applying each category. We also define whether the options are unordered, i.e., *nominal*, or ordered without defined distance, i.e., *ordinal*. Last, we specify if the gained knowledge by classification can be reused. Here, a category can be specific for the uncertainty source and thus *system-independent*, or specific for the software architecture under investigation and thus *system-specific*. This also concerns the question of whether the specific classification of a source of uncertainty is *immutable* and does not change after the identification or *mutable* and can be changed if more information is obtained during modeling and analysis.

5.3.1. Classification Category: Location

The first category shown in Table 5.1 is concerned with the *location* of the uncertainty source. Previous taxonomies [41, 162, 195, 263] already considered "where the uncertainty manifests itself within the model" [195] but did not explicitly relate to an ADL. Since the location of the source is one of the most important properties for design time modeling

Architectural Element Type: Describes the type or class of elements to which an uncertainty can be assigned when considering its impact on a software system's confidentiality. System-Independent, Nominal, Immutable	
Component	The uncertainty is assignable to software components. This represents the nodes of a software system.
Connector	The uncertainty is assignable to wires between components, or their communication. This represents the edges between the nodes of a system.
Interface	The uncertainty is assignable to interfaces of components. This represents the contact point of nodes and edges in a software system.
External	The uncertainty is assignable to external resources or their properties. This represents annotating the nodes of a software system.
Behavior	The uncertainty is assignable to behavior descriptions. This represents specifying the behavior of nodes within a software system.

Table 5.2.: Uncertainty classification regarding confidentiality. Category: Architectural Element Type.

analysis, we define four locations as inspired by the viewpoints of the PCM [207]. Compared to existing taxonomies, this enables more precise description and mitigation because we can model uncertainty and its relation to existing architecture elements.

The option *system structure* describes uncertainty in components and their assembly, which becomes visible in structural views like component diagrams. The option *system behavior* describes uncertainty in the communication, e.g., related to the handling of data within a component or the communication between components. The option *system environment* describes uncertainty in the system's context including hardware resources and the external situation. The option *system input* describes uncertainty in inputs provided by external actors, e.g., people using the software system. Note that we distinguish between the input, i.e., the user behavior, and the system behavior. This is due to the special role—and rights—of humans with regard to confidentiality, e.g., due to legal requirements like the GDPR [73], and shall also help in collaboration with legal analysis [39]. Because the location is defined on the level of an ADL meta model, it is system-independent and immutable. The category is nominal, as its options only represent viewpoints without applicable order.

5.3.2. Classification Category: Architectural Element Type

Table 5.2 shows the second category *Architecture Element Type*. While the *Location* is on the abstraction of the viewpoint (e.g., structure, or behavior), the *Architecture Element Type* describes the concrete elements where the uncertainty arises. Similarly to the previous category, the options are inspired by software architecture element types known from ADLs like the PCM. We consider this to be the central category for further modeling and analysis of uncertainty, as it specifies a starting point to consider the effect of an uncertainty source. Compared to existing classifications, we provide five concrete options

Type: Describes how much is known about uncertainty and how it is described on a scale from only being aware to having precise knowledge. This may change with growing knowledge. SYSTEM-SPECIFIC, ORDINAL, MUTABLE	
Statistical Uncertainty	The uncertainty can be described with statistical means, e.g., related to the probability of certain outcomes.
Scenario Uncertainty	The uncertainty can be described with distinct scenarios but there is a lack of knowledge to apply statistical means.
Recognized Ignorance	There is awareness of the uncertainty but no knowledge about potential scenarios or lack of a description strategy. This is the most general form of an identfied uncertainty source.

Table 5.3.: Uncertainty classification regarding confidentiality. Category: Type.

rather than only speaking of uncertainty within the model [195]. The need for such precision is supported by more recent approaches to classifying and propagating uncertainty [2, 49] and also in the recent PSUM standard [184]. By understanding which elements and viewpoints are affected, software architects can assess responsibilities and evaluate mitigation methods.

The option *component* describes uncertainty assignable to software components, e.g., related to their allocation or other component-wide decisions or properties. The option *connector* describes uncertainty assignable to, e.g., wires between components and thus communication and its properties. The option *interface* describes uncertainty assignable to interfaces, e.g., signatures, parameters, and return values. The option *external* describes uncertainty assignable to hardware resources, e.g., servers, and external actors like users of the system. Note that we renamed this category compared to the original publication [104] to take into account that users also represent external entities. The option *behavior* describes uncertainty assignable to behavior descriptions, e.g., algorithms, user behavior and input, data processing, and persistence. This option also helps to underline the difference between *Architecture Element Type* and *Location*: Although behavior descriptions of users and the software system may be similar, the viewpoints of *System Behavior* and *System Input* are different. Both categories are orthogonal and cannot be replaced by each other. The category is immutable and system-independent and can thus be reused across architectural models. The category is nominal as there is no order between location or element types.

5.3.3. Classification Category: Type

The third category is called *Type* and describes how much is known about the uncertain influence as shown in Table 5.3. Other taxonomies [41, 195] only specify this in terms of levels on a scale from knowledge to ignorance [11], which is too imprecise to classify uncertainty for later mitigation. With the *Type*, we describe how much is known about the uncertainty, based on the definitions by Walker et al. [263]. As discussed previously, we

Manageability: Describes to which extent the uncertainty can be managed, reduced, or mitigated. This only provides a first estimate and may change with growing knowledge. System-Specific, Ordinal, Mutable	
Fully Reducible	The uncertainty can be fully resolved with appropriate means that mitigate its potential impact.
Partially Reducible	The uncertainty is at least partially reducible which reduces the potential impact severity or limits critical outcomes by appropriate mitigation techniques.
Irreducible	The uncertainty cannot be further reduced as there is no reasonable way to achieve the required knowledge at this point in time.

Table 5.4.: Uncertainty classification regarding confidentiality. Category: Manageability.

do distinguish between known or unknown unknowns as this depends on the viewpoint of the software architect and not the classified uncertainty source.

Statistical uncertainty implies that the uncertainty is describable with statistical means, e.g., stochastic expressions, or probability distributions. *Scenario* uncertainty implies that the uncertainty can be represented with distinct scenarios depending on the uncertain outcome but without statistical means. *Recognized ignorance* only implies awareness of uncertainty sources, without scenario-based or statistical means to describe their form. These options can be ordered depending on the amount of knowledge and thus are ordinal. Having statistical information enables more precise analysis results than just knowing scenarios—which is still better than simple awareness. The category is system-specific and mutable. During the software design and realization, more information may be gained, enabling a more precise description of the uncertainty, e.g., because more statistical information is collected or some scenarios can be excluded.

5.3.4. Classification Category: Manageability

The fourth category shown in Table 5.4 covers the *manageability* of uncertainty. *Manageability* states whether we can control or reduce the impact of the uncertainty at design time or are only aware of it [71]. Early identification of potential threats to security-related properties like confidentiality is also required by *Data Protection by Design* [73, Art. 25]. We do not consider the nature of the uncertainty [263] because the manageability is closer to the uncertainty's impact on the software system [71]. Representing the manageability on a scale from reducibility to irreducibility is also supported by the PSUM standard [184].

Fully reducible uncertainty can be fully reduced by taking appropriate actions, e.g., an ADD at design time, or comprehensive simulation. This can include "collecting additional information until achieving full certainty" [184]. *Partially reducible* implies that full certainty cannot be reached. Nevertheless, the uncertainty can be reduced by collecting more information, or by applying scenario-based mitigation strategies, or system-wide constraints and policies. *Irreducible* uncertainty cannot be further reduced, e.g., due to its

Resolution Time: Describes the rough time span in the development process where the uncertainty is usually expected to be fully resolved. System-Independent, Ordinal, Immutable	
Requirements Time	The uncertainty is expected to be resolved as soon as the requirements are defined.
Design Time	The uncertainty is expected to be resolved as soon as the software system is designed.
Realization Time	The uncertainty is expected to be resolved as soon as the software system or parts of it are implemented and deployed.
Runtime	The uncertainty is expected to be resolved as knowledge is gained from testing and system operations, or not at all.

Table 5.5.: Uncertainty classification regarding confidentiality. Category: Resolution Time.

aleatory nature [184, 195]. Similar to the type category, this category is ordinal because the options can be ordered. The classification depends on the architecture under investigation and the current knowledge. Thus, this option is system-specific and mutable. However, many uncertainty types tend to be categorized similarly across architectures.

5.3.5. Classification Category: Resolution Time

The *Resolution Time* shown in Table 5.5 is based on the phases of software development and can help to narrow down sources and responsibilities. Since we focus on the impact of uncertainty on confidentiality, we consider the expected full resolution time rather than the emerging time [115, 195, 202, 255]. Also, we only include phases that are relevant from the point of view of design time modeling and analysis.

Requirements time implies that the uncertainty resolves as soon as requirements are defined, e.g., related to confidentiality requirements or security policies. *Design time* implies that the uncertainty resolves as soon as the system is designed, e.g., because the uncertainty is related to ADDs, the system structure, or software components. *Realization time* implies that the uncertainty resolves as the system or parts of it are implemented and deployed. Here, we combine implementation and deployment as a distinction is not expedient from an architectural point of view. *Runtime* implies that the uncertainty resolves at runtime, e.g., because knowledge is gained from testing or dynamic analysis like monitoring or profiling. This category depends on the uncertainty source and not on the system under study and is thus system-independent and immutable. It is ordinal because we can order the options along the phases of software development from requirements to runtime.

Reducible by ADD: Describes whether the uncertainty is resolvable or treatable by an architectural design decision, i.e., a decision that specifies or restricts a software systems's structure or behavior, thereby limiting the design space. SYSTEM-INDEPENDENT, NOMINAL, IMMUTABLE	
Yes	The uncertainty can be reduced by making an architectural design decision.
No	The uncertainty is not resolvable by taking an architectural design decision.

Table 5.6.: Uncertainty classification regarding confidentiality. Category: Reducible by ADD.

Impact on Confidentiality: Describes the impact on confidentiality requirements. Initially, this only provides a first, system-independent estimate. SYSTEM-SPECIFIC, ORDINAL, MUTABLE	
Direct	The uncertainty has a direct impact on the software system's confidentiality.
Indirect	The uncertainty only has an indirect impact on the software's confidentiality that usually relies on other uncertainties or other contextual factors.
None	The uncertainty is expected to have no impact on confidentiality at all.

Table 5.7.: Uncertainty classification regarding confidentiality. Category: Impact on Confidentiality.

5.3.6. Classification Category: Reducible by ADD

Table 5.6 shows the category *Reducible by ADD* that specifies whether the uncertainty source can at least be partially mitigated by a design decision on architectural abstraction, i.e., an ADD. Making the connection between ADDs and uncertainty explicit [159] helps to prioritize, e.g., check whether multiple or critical uncertainty sources can be tackled by a single decision. ADDs are seen as crucial aspect of software architecture [5, 124, 150].

The option *Yes* describes uncertainty which can be addressed on architectural abstraction by making appropriate decisions, i.e., by designing the system in a way that the impact of the uncertainty is (partially) mitigated This implies that the uncertainty can—and probably should—be addressed in the architectural design and falls into the *cone of uncertainty* [167]. The option *No* describes uncertainty that is not resolvable or treatable by taking an ADD, e.g., because the uncertainty is outside the scope of the designed software system, or cannot be properly addressed within the design process, e.g., due the behavior of a third-party. An uncertainty that is not resolvable by an ADD does not imply a resolution later than design time. The uncertainty could also exist due to a lack of knowledge or could be addressed on a lower level than architectural abstraction—but not by an ADD. Both categories are orthogonal. Reducibility is nominal, system-independent and immutable.

5.3.7. Classification Category: Impact on Confidentiality

The last two categories are used to quantify the impact of uncertainty on confidentiality requirements. Table 5.7 shows the category *Impact on Confidentiality*. It represents a

Severity of the Impact: Describes the severity if the uncertainty is not mitigated. Initially, this only provides a first, system-independent estimate. System-Specific, Ordinal, Mutable	
High	The uncertainty can cause a total loss of confidentiality, e.g. due to a data breach.
Low	The uncertainty can cause information leaks, but the damage is limited.
None	The uncertainty is expected to cause no loss of confidentiality at all.

Table 5.8.: Uncertainty classification regarding confidentiality. Category: Severity of the Impact.

first estimate on whether the uncertainty can directly impact confidentiality or requires additional conditions to result in confidentiality violations.

The option *Direct* describes uncertainty with a direct impact that can void confidentiality even without taking other factors, decisions, or uncertainties into account, e.g., by directly affecting input, processing, or storage of sensitive data like user data. The option *Indirect* describes uncertainty that is related to ADDs, security measures, or other uncertainty sources. The option *None* describes uncertainty without impact on confidentiality, e.g., if it is related to handling non-sensitive data or well-secured system parts or because the uncertainty has no effect on any data processing at all, e.g., related to the color of a button. Note that this category only describes the type of impact but not its severity. An indirect impact can be as severe as a direct impact but is potentially even harder to identify. This category only provides a first, mutable estimate and is highly system-specific. We consider the options to be ordinal as they can be sorted from direct to no impact.

5.3.8. Classification Category: Severity of the Impact

To prioritize uncertainty with a potentially critical impact, we combine the previously introduced impact type with its severity. Table 5.8 shows the last category of the classification that is named *Severity of the Impact*. This category is based on the confidentiality impact metrics of the open industry standard Common Vulnerability Scoring System (CVSS) [78]. We reuse the definition as it is broadly used, and known by security experts.

The option *High* refers to uncertainty that can cause the total loss of confidential data or access to restricted information. This can be the case because the uncertainty is either related to highly sensitive data like certificates, encryption keys, or an admin password or because the uncertainty is related to central security measures like input validation and sanitization. The option *Low* refers to uncertainty that could cause the loss of restricted information, but the damage is limited, e.g., because the input validation only has limited effects due to insufficient security policies. The option *No* refers to uncertainty where no loss of confidentiality is expected at all, e.g., because the uncertainty is related to parts of a software system that do not deal with sensitive information. As stated in the previous category, there is no clear dependency between an impact's type and its severity. Only uncertainty sources that have no impact on confidentiality at all are also expected to have no severity. Other than that, any combination of high or low, and direct or indirect can happen. This category as well provides a first, mutable estimate that is system-specific. Nevertheless, both categories help to estimate the potential impact and plan further steps toward analyzing the uncertainty. The knowledge gained by this classification can help in clustering and prioritizing uncertainty and related ADDs. This category is ordinal.

5.3.9. Building on the Uncertainty Classification

In sum, our classification provides eight categories that do not classify uncertainty in isolation but also relate the uncertainty sources to software architecture and confidentiality, as discussed in Section 5.2. These categories help in the early understanding of uncertainty sources and serve the discussion of software architects and security experts. We split the categories into system-independent and system-specific. System-specific categories are always mutable, as the knowledge about the uncertainty source can change over time.

ℹ **Finding:** Uncertainty classifications are a well-suited entry point to understanding uncertainty sources. However, classification can change over time when more knowledge is gained or the system or environment changes. Some categories of the classification depend on the architecture under study. Still, many uncertainty sources tend to be categorized similarly across architectures.

This classification represents the baseline for further modeling and analysis activities, as outlined in Section 4.1. Many categories help in the early assessment of the criticality of an uncertainty source, even without systematically analyzing the software architecture. However, to gain detailed insights on the impact of uncertainty and to identify confidentiality violations, architecture-based analysis is expedient. Here, we benefit from the category *Architectural Element Type*, which relates uncertainty sources to concrete element types of software architectures and can be used to extend existing ADLs like the PCM.

ℹ **Finding:** Centering an uncertainty classification around the location of the uncertainty source simplifies the later modeling, analysis, and mitigation. Thus, we see the category *Architectural Element Type* and its options *Component*, *Connector*, *Interface*, *External*, and *Behavior* as central dimension of our classification.

This finding is supported by comparable approaches [2, 49] that analyze the impact of uncertainty. Acosta et al. [2] define the category *Locus* that describes the location of uncertainty within the model and serves as starting point of uncertainty propagation. Due to the importance of this category, we will use it throughout the remainder of this thesis and refer to it directly, e.g., by speaking of *Component* uncertainty, *Behavior* uncertainty, or *External* uncertainty.

Category	**U1**: User Input	**U2**: Data Processing
Location	System Input	System Behavior
Architectural Element Type	Connector	Behavior
Type	Scenario Uncertainty	Scenario Uncertainty
Manageability	Partially Reducible	Fully Reducible
Resolution Time	Runtime	Design Time
Reducible by ADD	Yes	Yes
Impact on Confidentiality	Indirect	Direct
Severity of the Impact	High	High

Table 5.9.: Exemplary classification of the uncertainty sources U1 and U2 in the running example.

5.4. Applying the Classification to the Running Example

We illustrate the classification introduced in the previous section by applying it to the running example from Chapter 3. The running example, which shows a simplified online shop, comprises four uncertainties: The user input, data processing within the *Online Shop* component, the deployment of the *Database Service*, and the provider trustworthiness of the *Cloud Service* resource. In the following, we present one possible classification using the aforementioned classification scheme. While some categories like the location are system-independent, others like the severity regarding confidentiality are system-specific and are based on our interpretation of the running example. We provide our reasoning and also explain alternative classifications. All classified uncertainty sources are part of our data set [98]. Additionally, they are listed in ARC^{3}N with the proposed classification[2].

Table 5.9 shows our classification of the first two uncertainty sources, i.e., the user input (**U1**) and the data processing (**U2**). The Uncertainty **U1** about the user input is located in the system input and can be annotated at a connector. We annotate the uncertainty to a connector instead of a behavior description because it relates to the question of how the user communicates with the system. In our discussion in Section 3.2, we enumerated scenarios like normal or malicious behavior, thus we already reached the knowledge of scenario uncertainty. With mitigation strategies like proper input validation, we can at least partially reduce the uncertainty—more optimistic experts could even argue for full reducibility. The uncertainty persists until run time when actual humans use the system under study. Until then, it is reducible by appropriate design decisions like the aforementioned input validation. The input on confidentiality in our example is indirect as it depends on the data processing, and thus, it depends on Uncertainty **U2** whether confidentiality could be violated. If we do not have any means of input validation, the impact could be direct. We classify the severity as high because confidentiality cannot be guaranteed under this uncertainty.

[2] At the time of writing, ARC^{3}N is online available here: `https://arc3n.abunai.dev/`. Currently, the uncertainty sources of the running example **U1** – **U4** are listed as #52, #35, #59, and #48.

Category	**U3**: Deployment	**U4**: Provider Trustworthiness
Location	System Structure	System Environment
Architectural Element Type	Component	External
Type	Scenario Uncertainty	Scenario Uncertainty
Manageability	Partially Reducible	Partially Reducible
Resolution Time	Realization Time	Runtime
Reducible by ADD	Yes	Yes
Impact on Confidentiality	Direct	Indirect
Severity of the Impact	High	High

Table 5.10.: Exemplary classification of the uncertainty sources U3 and U4 in the running example.

The Uncertainty **U2** about the data processing is located in the system behavior and can be represented in a behavior description. Similarly to Uncertainty **U1**, we already provided scenarios in Section 3.2, thus, it represents scenario uncertainty. The uncertainty about the data processing is an uncertainty due to an open ADD as described by the *cone of uncertainty* [167]. Thus, it is fully reducible already at design time. We classify the impact on confidentiality as direct and high, because improper processing directly and severely affects the confidentiality, e.g., because of missing validation or encryption.

Table 5.10 shows our classification of the last two uncertainty sources, i.e., the deployment (**U3**) and the provider trustworthiness (**U4**). The Uncertainty **U3** about the deployment is located in the system structure and can be annotated to a component that must be deployed. In Section 3.1, we describe two scenarios of this scenario uncertainty, i.e., deployment on premise, or in the cloud. In our interpretation, this uncertainty is only partially reducible by a proper ADD at design time as it resolves at realization time, i.e., when the component gets deployed after the architectural design of the software system. In our running example, the impact on confidentiality is direct and high. If the confidentiality requirements demand selected deployment locations due to legal requirements like the GDPR [73], any deviation causes a confidentiality violation.

The Uncertainty **U4** about the trustworthiness of the cloud service provider is located in the system environment and classified as external uncertainty in the system's context, see Figure 5.1. Although we cannot control the provider's behavior, we can describe scenarios, i.e., trustworthy or not trustworthy behavior. With further knowledge or experience, experts could quantify the probability of these scenarios to reach statistical uncertainty, but in the running example, that is out of scope. With proper ADD, we can at least limit the impact of this uncertainty. Thus, it is partially reducible. In our running example, the impact on confidentiality is high but indirect. The impact depends on the form of data processing (**U2**) and also the deployment location (**U3**). Put simply, the cloud provider's trustworthiness does not affect the confidentiality if nothing is deployed in the cloud.

In sum, this illustration shows how our classification can be applied to better understand uncertainty sources. Compared to other classifications, we focus on confidentiality and thus provide appropriate categories for this purpose[3]. Describing the uncertainty in these categories helps to better understand the relation and criticality of uncertainty sources. In our running example, we can see the importance of proper data processing (**U2**) even without applying automated architectural analyses. We also see the importance of choosing the right deployment locations and can prioritize this ADD. Last, classifying uncertainty always has some level of interpretation. Thus, the classification of similar uncertainty sources in other software systems could be different. Moreover, our classification could change at a later stage, e.g., when enough knowledge about the uncertainty sources and the probability of certain scenarios is gained for statistical analysis.

5.5. Representing Uncertainty in Data Flow Diagrams

Analyzing confidentiality with Data Flow Diagrams (DFDs) [228, 236, 256] has been proposed because "problems tend to follow the data flow, not the control flow" [239]. Thus, Problem **P3** is concerned with the representation of uncertainty in DFDs. We address this by connecting DFDs to the *Architectural Element Type* category of our classification. As concluded previously in Section 5.3, this category is crucial for modeling and analyzing uncertainty on architectural abstraction. This applies to ADLs like the PCM [207] as well as to DFDs. The following section builds on the foundations about DFDs, DAGs, and the unified modeling primitives [235] presented in Section 2.5.

5.5.1. Mapping Uncertainty to Data Flow Diagrams

DFDs consist of processes, data flows, files, data sources, and data sinks—put simply, different types of nodes and edges. The unified modeling primitives [235] extend this notation and consist of nodes, pins, flows, behavior and labels with their assignments. Nodes represent structural elements of software systems, e.g., processes, stores, or external entities. Pins represent their interfaces and flows are used to connect multiple nodes through their pins. Nodes have a defined behavior that assigns labels, e.g., based on labels at the node's input pins, constants, logical expressions or a combination of the above. In our running example presented in Chapter 3, we can represent the data processing of the *Online Shop* component as a process node. The node has one input pin for the incoming data flow from the user and one output pin for the outgoing, processed data flowing to the *Database Service*. The deployment of the *Online Shop* component can be represented as a *On Premise* label that is assigned to the process node. Last, the behavior of the data processing node can be described using an assignment that assigns the label *Encrypted* to all outgoing data to represent the encryption of all processed data. For details on the unified modeling primitives of DFDs, see Section 2.5 or the primary publication [235].

[3] I discussed this question with Raffaela Mirandola, who co-authored one of the investigated classifications [195]. We concluded that there is no such thing as a *best* classification—only appropriate and less appropriate classifications for a specific *purpose*.

Uncertainty Type	Modeling Primitives	Effect of the Uncertainty
Component	Flow and node	Existence and use of nodes
Connector	Flow and assignment	Existence and targets of flows
Interface	Flow and pin	Existence and forms of pins
External	Label	Existence of a node's labels
Behavior	Assignment	Existence of a node's assignments

Table 5.11.: Mapping of the five uncertainty types to the unified modeling primitives.

In Table 5.2, we describe the category *Architectural Element Type* which comprises the options component, connector, interface, external, and behavior. Speaking in terms of nodes and edges, these options represent uncertainty, e.g., within a node in the case of component uncertainty, or at the contact point of a node and an edge in the case of interface uncertainty. We build on this finding and the fact that DFDs are a viable representation of software architectures [226] to map the five uncertainty types to DFDs. Table 5.11 shows the mapping of the five uncertainty types of the category *Architectural Element Type* to the unified modeling primitives.

Component uncertainty is assignable to software components that are represented by one or multiple nodes in a DFD. Thus, uncertainty regarding components affects the existence and use of nodes. In our running example, Uncertainty **U3** about the deployment can be represented as two alternative nodes where only one is in use.

Connector uncertainty is assignable to the communication between entities, i.e., to the edges between nodes in a DFD. This affects the existence and targets of data flows. Additionally, this uncertainty affects how and which data flows through an edge or data flow. In our running example, we model Uncertainty **U1** about the user input as connector uncertainty. Here, different kinds of data, represented using assignments, can flow into the *Online Shop* component, e.g., invalid or malicious data.

Interface uncertainty is assignable to interfaces of components which are represented as pins in a DFD. This affects the existence and forms of pins. Additionally, this uncertainty affects the data flows as an alternative interface can imply different processing which is represented as an alternative target of the data flow. We do not model interface uncertainty in our running example; an example would be alternative signatures of the *Database Service* which can be swapped and offer different data processing.

External uncertainty represents annotations to nodes of software systems, e.g., components or resources, which are represented as nodes in DFDs. This affects the assignment of labels to nodes. In our running example, Uncertainty **U4** about the provider's trustworthiness is an external uncertainty. In a DFD, the trustworthiness can be represented as a label that is or is not annotated to all nodes that represent the processes of the provider.

Behavior uncertainty is assignable to behavior descriptions in architectural models which translate to behaviors and assignments following the unified modeling primitives [235]. In our running example, Uncertainty **U2** about the data processing can be represented as alternative behaviors, e.g., just forwarding the data or applying validation or encryption. In sum, all uncertainty types are applicable to architectural models and also in DFDs that represent those models from the view of the data [234].

Following the unified modeling primitives for DFDs by Seifermann et al. [235], we see how the *Architectural Element Type* fits to the available elements types. However, this is not obvious: The five options of this category, i.e., the five uncertainty types, were collected by investigating existing uncertainty classifications and ADLs. Nevertheless, applying a bottom-up approach and starting with DFDs as underlying formalism yields the same five uncertainty types to represent uncertainty sources in all available element types. The only exception is the element type *Behavior*. However, this type only acts as a container in the original definition of the unified modeling primitives Seifermann et al. [235].

Finding: The five uncertainty types described in the classification category *Architectural Element Type* are derived from existing uncertainty classifications and Architectural Description Languages (ADLs) but also cover all elements of the unified modeling primitives for Data Flow Diagrams (DFDs). Although the representations are different, the five uncertainty types remain appropriate and applicable. Thus, the results of the bottom-up and top-down approaches match.

5.5.2. Mapping Uncertainty to Directed Acyclic Graphs

In the following, we discuss how to include uncertainty as a first-class entity in DFDs, following the proposal of Garlan [80]. We simplify the representation of uncertainty by only considering DAGs. DAGs can be used as simple formalism to represent DFDs, see Section 2.5. Put simply, processes, files, data sources, and data sinks are denoted as vertices, and data flows are denoted as edges. We go into more detail when we include uncertainty in architectural analyses—in this chapter, we remain at the conceptual level.

Looking at the affected modeling primitives in Table 5.11, we see that the former three uncertainty types affect flows while the latter two do not. Speaking of DFDs in terms of vertices and edges of a DAG, this is no coincidence. Component uncertainty affects the choice of vertices and, therefore, also affects the edges to these vertices, and connector and interface uncertainty always affect the edges between vertices. In contrast, external and behavior uncertainty directly affect the vertices, in terms of the unified modeling primitives, by affecting labels or assignments. They do not affect edges[4].

Based on this observation, we divide the five uncertainty types into two groups, which we call *primary* and *secondary* uncertainty. *Primary* uncertainty affects vertices and can directly be annotated to a vertex with an uncertainty source. *Secondary* uncertainty affects edges between vertices and can be annotated to an edge by adding alternative targets. We

[4] For you, dear reader, this may seem like an obvious or insignificant insight: Some uncertainties affect flows and others affect nodes. But, among others, this finding ultimately enabled us to create analyses that scale orders of magnitude better than the state of the art.

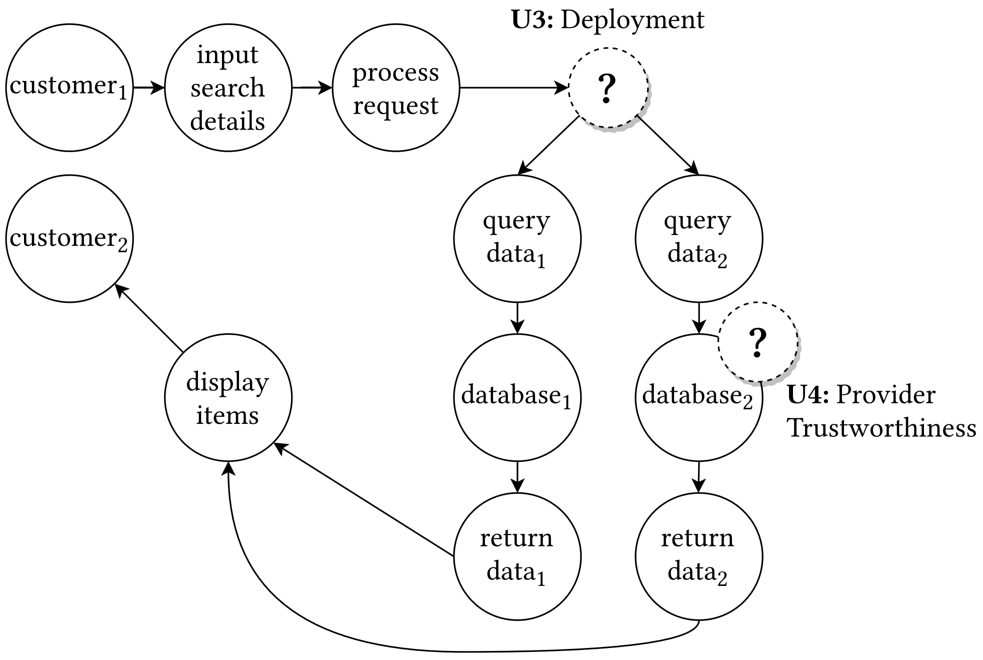


Figure 5.2.: Data flow of the running example as Directed Acyclic Graph (DAG) with annotated primary and secondary uncertainty, denoted by question marks.

use the terms primary uncertainty and secondary uncertainty to avoid ambiguity. Direct or indirect uncertainty, structural uncertainty, higher-order uncertainty, and first-order or second-order uncertainty are already used in existing classifications [11, 195, 263].

Figure 5.2 shows the data flow of viewing available items from the running example, represented as DAG. We annotate the vertices and edges with primary and secondary uncertainty. This example also illustrates how to transform a DFD into a DAG, as introduced in Section 2.5. Cycles are replaced, e.g., the customer's sending and receiving activities are represented by separate vertices [53]. The data flow of viewing available items is affected by two uncertainty sources: The uncertain deployment (**U3**) and the provider's trustworthiness of the *Cloud Service* (**U4**). The deployment (**U3**) is classified as component uncertainty and thus falls into the group of *secondary* uncertainty. In this example, this affects the flow from the *process request* vertex to the *query data* vertices and all following vertices that represent the *Database Service*. The left flow from *query data$_1$* until *return data$_1$* represents the component being deployed on-premise, and the right flow represents the component being deployed in the cloud. Here, we have to additionally consider the uncertainty about the provider's trustworthiness (**U4**). This uncertainty source is classified as external uncertainty and thus represents a *primary* uncertainty. For the sake of simplicity, we annotated this uncertainty only to the *database$_2$* vertex.

A DAG $G = (V, E)$ consists of vertices V and edges E. Vertices affected by *primary* uncertainty form a subset $V_u \subseteq V$. We do not further restrict the size of V_u. The subset is empty if no primary uncertainty exists, but it can also contain every vertex of V. In the running example, $V_u = \{database_2\}$. *Secondary* uncertainty affects the edges of a

DAG. Without limiting the generality, we can assume that the DAG contains all known alternative scenarios, e.g., the deployment on-premise and in the cloud. Then, each secondary uncertainty can be represented as a subset $E_{u_i} \subseteq E$. All secondary uncertainties form a set $E_u = \{E_{u_1}, \dots, E_{u_n}\}$ with size $|E_u| = n$. Similarly, we do not further restrict its size but we require each edge to be contained at most once in a subset of a secondary uncertainty, i.e., $\forall e \in E, \forall i, j \in \{1, 2, \dots, n\} : (e \in E_{u_i} \land i \neq j) \Rightarrow e \notin E_{u_j}$. Otherwise, we would allow for the interference of secondary uncertainties. In the running example, $E_{u_1} = \{\mathit{process\ request} \longrightarrow \mathit{query\ data}_1, \mathit{process\ request} \longrightarrow \mathit{query\ data}_2\}$ and $E_u = \{E_{u_1}\}$.

Already an example of this size demonstrates the benefits of including uncertainty as a first-class entity in the design [80]. It helps to better understand the locations of uncertainty sources and even to investigate simple uncertainty interactions [49]. In this example, Uncertainty **U4** is only relevant if the *Cloud Service*, which is represented by the right data flow, is selected as deployment in Uncertainty **U3**. We consider this fact in the classification of the uncertainty sources in Section 5.4. Here, we classify the impact of Uncertainty **U3** as direct and the impact of Uncertainty **U4** as indirect as it depends on another uncertainty, i.e., the Uncertainty **U3**. Representing uncertainty as primary and secondary uncertainty in DAGs visualizes such effects. To this end, we actively manage a model with uncertainty instead of modifying the model to exclude uncertainty [194].

Finding: Including uncertainty sources as first-class entity in Data Flow Diagrams (DFDs) simplifies their investigation, modeling, and analysis. Even simple DFD representations like Directed Acyclic Graphs (DAGs) are sufficient to gain insights about the direct and indirect effects of uncertainty. Here, we partition the five uncertainty types of the classification category *Architectural Element Type* into two groups: *Primary* uncertainty directly affects vertices of DAGs while *secondary* uncertainty affects edges between vertices.

5.6. Uncertainty Catalogs to Support the Identification

Although classifying uncertainty is necessary to understand the differences in nature and type [195], it is not sufficient to enable the early identification of uncertainty sources. Our classification shown in Section 5.3 and also model-based confidentiality analyses require the uncertainty sources to be known to the software architect who conducts the assessment. Uncertainty sources unknown to the architect, can neither be represented nor considered. We motivated this Uncertainty Awareness Problem (UAP) already at the beginning of Chapter 5 and also in Problem **P4**.

We propose a solution based on collecting and relating uncertainty sources and providing context information [103]. Figure 5.3 shows the metamodel of an *Uncertainty Source Catalog*. The *Uncertainty Source Catalog* consists of *Uncertainty Sources* and *Examples* that demonstrate these sources. Each source consists of a unique id, a name, a description, and a literature reference for additional information about its origin. It can be related to an arbitrary number of other sources and can have children which represent inheritance

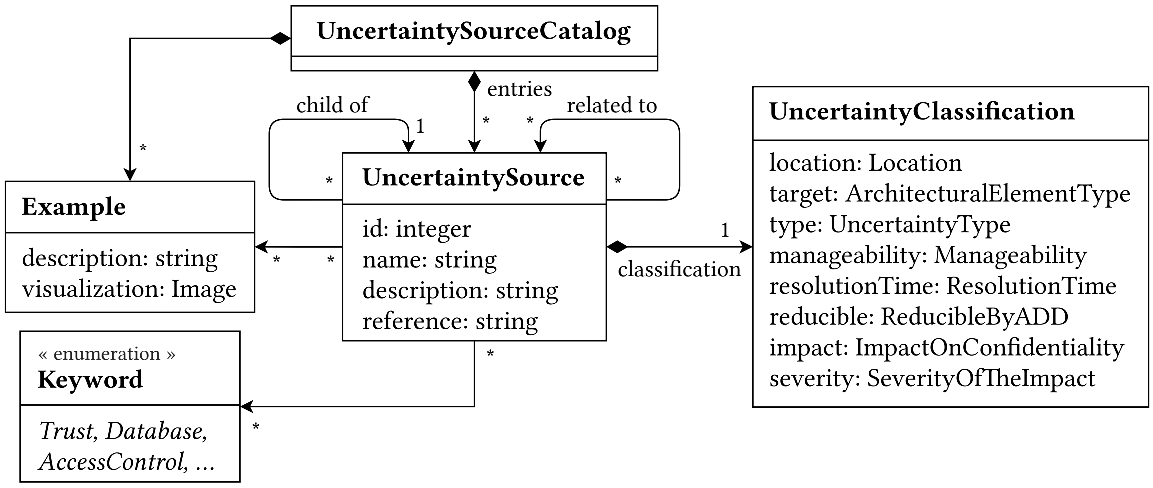


Figure 5.3.: Simplified metamodel of the uncertainty source catalog. To simplify the illustration, we leave out all types of the UncertaintyClassification and refer to their comprehensive definition in Section 5.3.

between sources. To classify the source, it is described by an *Uncertainty Classification* using the categories of our classification [104], presented in Section 5.3. Last, sources reference appropriate *Examples* that consist of a description and a visualization. *Examples* can be used to explain more than one *Uncertainty Source*, and a source can be illustrated with more than one example. Last, *Uncertainty Sources* can be further described by *Keywords* to group uncertainty sources beyond their classification.

The *Uncertainty Classification* represents the application of our classification to an uncertainty source, as exemplified in Section 5.4. In our running example presented in Chapter 3, Uncertainty **U4** describes the unknown trustworthiness of the cloud service provider. The *Uncertainty Classification* shows the selected classification options, e.g., the location of the system environment, or the manageability of partially reducible, see Table 5.10. Besides referencing the classification, the *Uncertainty Source* has an id, a name, a description, and literature references for further information [104, 202]. In our initial population of the catalog [98], we use the following example to explain the source: "In a cloud-based application, the uncertainty about the deployment provider's trustworthiness involves questioning whether the chosen cloud service ensures data security and system availability or whether or not data is made available to third parties such as governmental institutions." The *Uncertainty Source* is tagged with the keyword *Trust*.

In addition to its own properties, classification, examples, and keywords, *Uncertainty Sources* can have relations and parent uncertainties. In our example, *Provider Trustworthiness* is a child of the uncertainty source *Unpredictable Environment* [202], and related to uncertainty sources of our initial population regarding encrypted communication, component trustworthiness, and deployment location. This is illustrated in Figure 5.4.

Combined, this information provides comprehensive context knowledge for software architects, raises awareness, and simplifies the assessment. By attaching this information and also linking related uncertainty types, we address the organizational challenges of

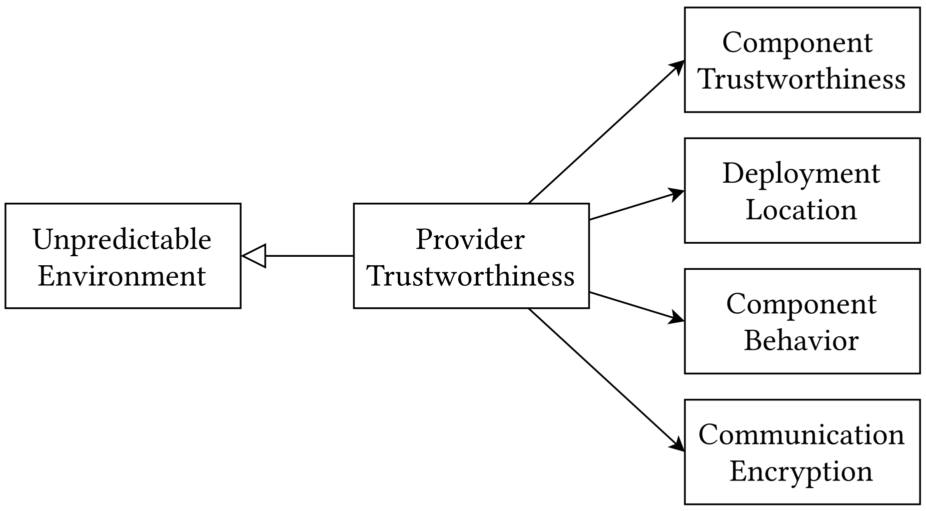
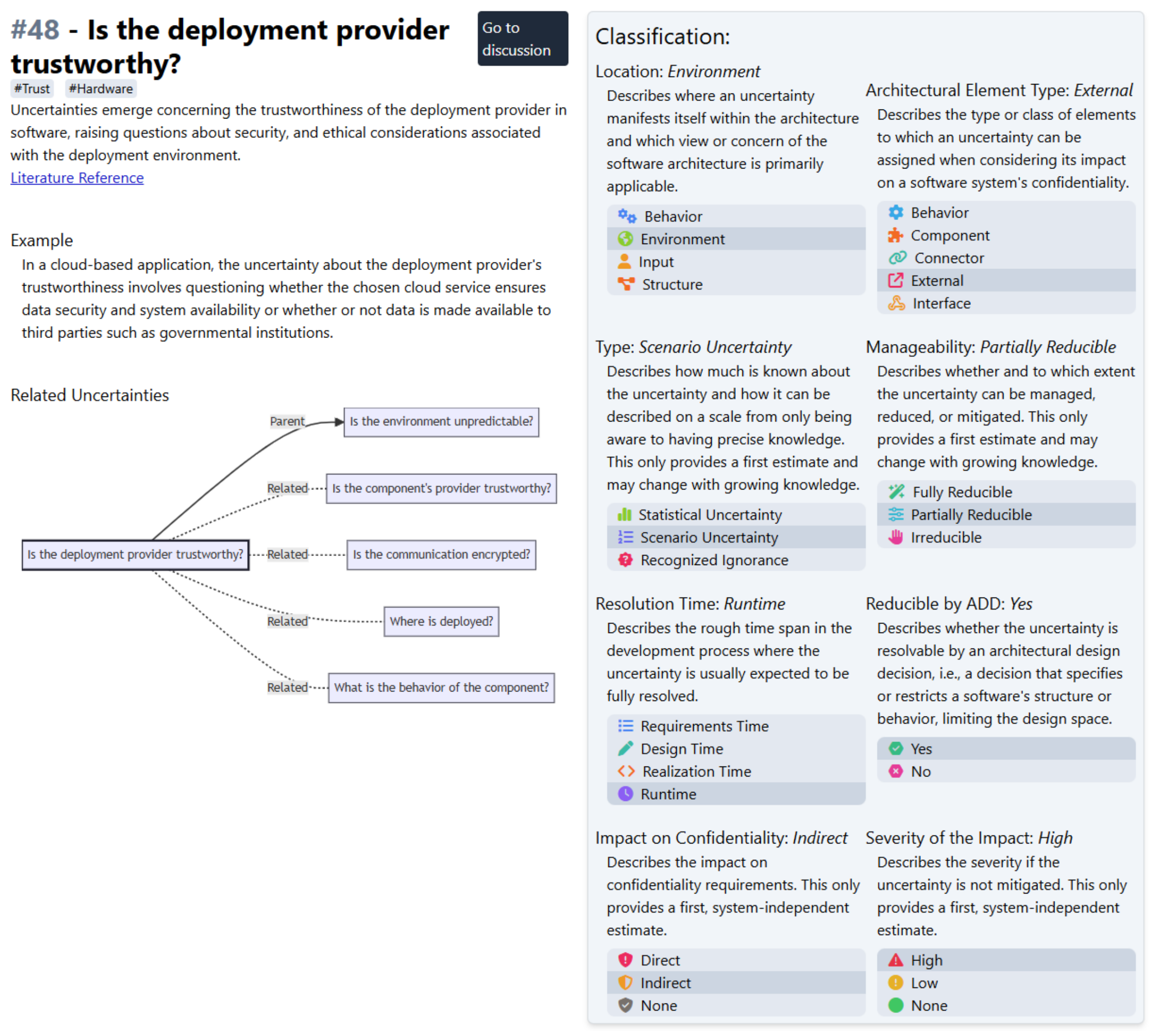


Figure 5.4.: Uncertainty source about the “Provider Trustworthiness” with its parent uncertainty source “Unpredictable Environment” and related uncertainties.

collecting uncertainty sources and enable to search and filter for uncertainty sources. Additionally, there are no space limitations regarding examples, visualizations, and discussions. To demonstrate this, we enriched 51 uncertainty sources from multiple classifications [104, 202] with context information, examples, and also uncertainty relations and inheritance as part of our data set [98]. In sum, this addresses Problem **P4** and enables non-experts to reuse expert knowledge on uncertainty sources.

Note that the current state of research cannot create a generally applicable and comprehensive answer to identifying all possible uncertainty sources [273]. However, processing, persisting and relating information about uncertainty sources provides a pragmatic first step, especially for practitioners [103]. To this end, approaches like this uncertainty source catalog are steps towards also addressing unanticipated changes that are not totally unforeseeable [80]. Following the discussion about the orders of uncertainty [11], we hereby lower the order of uncertainty from the *third* to the *second* order. Although there can be a lack of awareness of an individual, we provide means to identify some uncertainty sources, which previously were unknown to this individual [195].

Finding: Knowledge about uncertainty sources can be reused across software architectures. Extending the classification of uncertainty sources with descriptions, examples, keywords, and relations between uncertainty sources helps in the identification and simplifies the assessment. This reuse of expert knowledge addresses the Uncertainty Awareness Problem (UAP) and helps to deal with the ever-present challenge of unanticipated change.



#48 - Is the deployment provider trustworthy?

Go to discussion

#Trust #Hardware

Uncertainties emerge concerning the trustworthiness of the deployment provider in software, raising questions about security, and ethical considerations associated with the deployment environment.

Literature Reference

Example

In a cloud-based application, the uncertainty about the deployment provider's trustworthiness involves questioning whether the chosen cloud service ensures data security and system availability or whether or not data is made available to third parties such as governmental institutions.

Related Uncertainties

Is the deployment provider trustworthy? — Parent → Is the environment unpredictable?

Is the deployment provider trustworthy? — Related — Is the component's provider trustworthy?

Is the deployment provider trustworthy? — Related — Is the communication encrypted?

Is the deployment provider trustworthy? — Related — Where is deployed?

Is the deployment provider trustworthy? — Related — What is the behavior of the component?

Classification:

Location: *Environment*

Describes where an uncertainty manifests itself within the architecture and which view or concern of the software architecture is primarily applicable.

- Behavior
- Environment
- Input
- Structure

Architectural Element Type: *External*

Describes the type or class of elements to which an uncertainty can be assigned when considering its impact on a software system's confidentiality.

- Behavior
- Component
- Connector
- External
- Interface

Type: *Scenario Uncertainty*

Describes how much is known about the uncertainty and how it can be described on a scale from only being aware to having precise knowledge. This only provides a first estimate and may change with growing knowledge.

- Statistical Uncertainty
- Scenario Uncertainty
- Recognized Ignorance

Manageability: *Partially Reducible*

Describes whether and to which extent the uncertainty can be managed, reduced, or mitigated. This only provides a first estimate and may change with growing knowledge.

- Fully Reducible
- Partially Reducible
- Irreducible

Resolution Time: *Runtime*

Describes the rough time span in the development process where the uncertainty is usually expected to be fully resolved.

- Requirements Time
- Design Time
- Realization Time
- Runtime

Reducible by ADD: *Yes*

Describes whether the uncertainty is resolvable by an architectural design decision, i.e., a decision that specifies or restricts a software's structure or behavior, limiting the design space.

- Yes
- No

Impact on Confidentiality: *Indirect*

Describes the impact on confidentiality requirements. This only provides a first, system-independent estimate.

- Direct
- Indirect
- None

Severity of the Impact: *High*

Describes the severity if the uncertainty is not mitigated. This only provides a first, system-independent estimate.

- High
- Low
- None

Figure 5.5.: Screenshot of the detail view of our tooling 🔧 ARC^{3}N. It shows the description, explanation, and classification of the uncertainty source regarding the trustworthiness of a resource provider.

5.7. A Collaborative Approach for Uncertainty Catalogs

Our classification presented in Section 5.3 provides the terminology to understand and discuss uncertainty sources, and the catalog meta model presented in Section 5.6 simplifies the identification of relevant uncertainty sources and sharing of knowledge. However, to be applicable, knowledge reuse must be simplified and should not be limited to individuals or institutions. With Problem **P5**, we stressed the importance of an open and publicly available catalog of uncertainty sources. Collecting uncertainty sources only within publications [202] or data sets [104] lacks reusability, extensibility, and availability. To address this, we propose a web-based catalog approach in the following.

We realized our meta model and catalog approach as 🔧 **ARC^{3}N**, which stands for *Rese**arch** **arch**ive for Software-**Arc**hitectural **Un**certainty*[5]. Our tooling revolves around a table of uncertainty sources and their classification which allows for quick navigation, search, and filtering by keywords or classification categories. By selecting a source, software architects can see its description, examples, related sources, and its classification. Because there is no space limitation, the selected options of the classification can be explained in detail, as shown in Figure 5.5. We also provide further explanation of all classification categories and relate known uncertainty sources similar to Figure 5.4. Software architects can quickly navigate through uncertainty sources, examples, and classification. Last, the full catalog can be downloaded in a machine-readable JSON format.

All four uncertainty sources of our running example are contained in the initial population of our catalog. Thus, security experts could become aware of these uncertainty sources by using our tool support. To identify the uncertainty of the provider's trustworthiness (**U4**), they could search for trust-related uncertainty sources or navigate to this uncertainty by its related uncertainty sources, e.g., the unpredictable environment or the encrypted communication. They could even identify the uncertainty by investigating the open deployment decision with Uncertainty **U3** as both uncertainty sources are related.

We chose a web-based approach to minimize the effort required to set up and use the tooling. We use a GitHub [90] git repository as the backend for adding new uncertainty sources as well as discussions between participants. New uncertainty sources can be added directly from within our tool support and are then retrieved by using the open GitHub API [90]. This enables lightweight and openly accessible storage of all required information. We choose GitHub to minimize the risk of link decay, which is especially prominent on institutional websites [91]. The information retrieval is automated using a DevOps pipeline based on GitHub Actions [90]. Once the information has been updated, our tooling works completely autonomously and can be self-hosted or used locally. This enables the easy integration of other information sources and also simplifies switching to other platforms like GitLab.

Our tooling 🔧 ARC^{3}N collects uncertainty sources in a central place instead of relying only on publication-based collections which simplifies the collaboration of researchers and practitioners. We foster two types of collaboration: 🔧 ARC^{3}N benefits the communication between different roles with different knowledge, e.g., security experts and software architects. It also benefits peers with the same role as a shared catalog for exchange and documentation. New uncertainty sources can be proposed by everyone based on the open-source GitHub repository. All required information is stored as part of the tool support and can be downloaded in third-party analyses. In sum, this approach provides an easy-to-use, extensible, and open collaborative approach to address Problem **P5**. Note that we do not claim comprehensiveness regarding uncertainty sources with this catalog. We do also not claim long-term availability over decades. Nevertheless, this approach is a proposal towards an open and available catalog to foster collaboration [251].

[5] 🔧 ARC^{3}N is open-source and available online: `https://arc3n.abunai.dev/`. It is also archived in our data set [98]. Any similarities by name with extensively praised jewels are completely coincidental.

5.8. Assumptions and Limitations

In this section, we discuss the assumptions and limitations of the identification and classification approaches presented in this chapter. We partition these into five groups: The focus on confidentiality, the use of software architecture-based approaches at design time, the lack of automated analysis, the cooperation required for collaborative approaches, and the ever-present challenge of unanticipated change.

Focus on confidentiality First, we only focus on confidentiality as a central quality attribute of our classification. While this reduces the generalizability, we do this intentionally to obtain more precise results for the modeling and analysis of uncertainty sources. As demonstrated, the focus on confidentiality enables the connection of the uncertainty types to DFDs, which simplifies later data flow-based confidentiality analysis. Additionally, we assume that the concepts can at least be applied partially to related quality attributes like integrity [36, 233]. Using the location as a central classification category and representing uncertainty in flow diagrams were the inspirations for other work [2, 49]. Nevertheless, we do not claim generalizability beyond confidentiality.

Using the software architectural abstraction Similarly, we limit ourselves to software-architectural uncertainty, as defined in Section 5.2. This was also an explicit decision due to fit existing modeling [207, 235] and analysis [234, 236, 256] approaches for confidentiality. Still, most categories are general enough to be used even without explicit models, e.g., *Type*, *Manageability*, or *Resolution Time*. We assume that software architects and security experts already have enough knowledge about the software system to assess uncertainty at design time. Such assumptions exist in many architecture-based analyses [233, 264].

Transitive impact of uncertainty Third, the approaches for identification and classification provide no assistance for the transitive impact of uncertainty. The uncertainty source is often not the location where the uncertainty affects confidentiality and where it can be mitigated. We presented the relation of uncertainty sources and their impact in Section 5.2, e.g., with the transitive impact of the Uncertainty **U1** about the user input to the *Database Service* shown in Figure 5.1. However, to face such propagation effects, a precise description of uncertainty—such as our classification—is required in the first place. Additionally, the modeling and analysis of uncertainty for design-time confidentiality analysis require tool support as "detecting confidentiality issues manually is not feasible" [234]. We will address this limitation in the following chapters.

Collaboration for joint uncertainty catalogs Our web-based approach 🔧 ARC^{3}N represents a publicly available uncertainty catalog. Here, we assume that researchers and practitioners alike benefit from sharing knowledge about uncertainty sources, as known from legal sciences [251]. However, companies or institutions could try to keep the information private, especially regarding sensitive topics like security. The lack of willingness to cooperate limits the benefits of a shared knowledge base. Nevertheless, ARC^{3}N could also be used in a closed environment but with limited usefulness.

Unanticipated change We do not claim to have solved the challenge of unanticipated change. A recently published research agenda [273] names the challenge of end-to-end approaches that can operate in changing real-world conditions. We do not provide a solution to such unanticipated change. However, our identification approach represents a pragmatic approach towards reducing the effect to the individual by sharing knowledge about them, i.e., making them known. In sum, our catalog approach can be seen as a first step or a potential part of a comprehensive solution.

5.9. Summary and Outlook

In this chapter, we presented our classification of *software-architectural uncertainty* and our approach to address the identification of uncertainty. This represents our first Contribution **C1** and provides the answer to **Research Question 1**.

First, we presented our understanding of the relation of uncertainty, confidentiality, and software architecture to address Problem **P1**. We highlighted the distinction between an uncertainty source and its impact. The former represents the origin of uncertainty in the software system or its environment while the latter represents the actual location of the impact of the uncertainty source on confidentiality. We also related the notion of uncertainty to the terminology known from Self-Adaptive Systems (SASs) [272], the *cone of uncertainty* [167], and Architectural Design Decisions (ADDs) [124, 150].

Based on this discussion, we investigated existing uncertainty classifications [41, 162, 195, 202, 263] and defined our own classification tailored to the impact on confidentiality, thereby addressing Problem **P2**. Our classification contains 8 categories with a total of 27 options. They classify uncertainty in categories like location, manageability, or resolution time. The central category is the *Architectural Element Type* as this category describes where to annotate an uncertainty source in architectural models. Last, we illustrated the classification using our running example.

Building on this classification, we addressed Problem **P3** of representing uncertainty as a first-class concern in DFDs. Our first finding was that the five uncertainty types of the aforementioned classification match the model elements of DFDs. We then partitioned the five uncertainty types category into *primary* and *secondary* uncertainty. This enables a simple and straightforward representation of uncertainty in DAGs, which can be used to represent DFDs. Primary uncertainty affects vertices, and secondary uncertainty affects edges between vertices.

Afterward, we extended the classification and presented a meta model for *uncertainty source catalogs*, which addresses Problem **P4**. We proposed the enrichment of classified uncertainty sources with descriptions, keywords, and examples. Additionally, uncertainty sources are usually not independent [50] and thus can have relations and inheritance. By collecting such information in a central archive, the UAP can be partially addressed.

Last, we presented our tooling 🔧 ARC^{3}N to create an open catalog of uncertainty sources and to address Problem **P5**. Our initial population contains 51 uncertainty sources with descriptions, examples, and relations. It is publicly available under an open-source license and can help to identify relevant uncertainty sources that were previously unknown to security experts. By combining classification and identification, we support the first two activities presented in Section 4.1.

❓ **Research Question 1** asked about the identification, description, and classification of software-architectural uncertainty regarding confidentiality. Our comprehensive answer to this question comprises the inspection of existing classifications and the definition of a novel classification regarding the impact of uncertainty on confidentiality. We derive five uncertainty types that fit DFDs, which are often used to analyze confidentiality [235]. By applying these uncertainty types to DAGs, we close the gap between uncertainty sources and confidentiality. Furthermore, our classification provides the means to describe and discuss uncertainty. By extending the classification to form tool-supported uncertainty source catalogs, we ultimately also address the question of description and identification.

The benefits of our Contribution **C1** include precise terminology to discuss and understand uncertainty regarding confidentiality. This supports both software architects and security experts in modeling and analyzing software-intensive systems. As proposed in Chapter 4, we do not require an additional *uncertainty expert* role, as the required knowledge is contained in the classification and the uncertainty source catalog. Here, our tooling 🔧 ARC^{3}N presents a useful starting point. By identifying and assessing uncertainty sources early, the reasoning and prioritization of ADDs is simplified and costly backtracking is minimized. This is especially true regarding uncertainty interactions [50], which represent uncertainty impacts that are particularly hard to find and mitigate. Last, our classification lays the foundations for further integration of software-architectural uncertainty in the architecture-based analysis of confidentiality.

We will revisit the idea of representing uncertainty sources and their impact as first-class entities in DFDs and DAGs in the following chapters. In ❯ **Chapter 6: Uncertainty Propagation to Enable Uncertainty Impact Analysis**, we discuss uncertainty propagation in architectural models and DFDs alike. This will help to better understand the actual impact of uncertainty sources on confidentiality beyond the early assessment using our classification. This also addresses the limitation of the sole classification regarding the transitive impact of uncertainty. It also defines the foundations for later uncertainty-aware analysis in ❯ **Chapter 7: Uncertainty-Aware Data Flow Analysis to Identify Confidentiality Violations**. Here, we will also see again the distinction between primary and secondary uncertainty and how it can be used to speed up data flow analysis. Last, the evaluation of this chapter will be presented in ❯ **Chapter 9: Evaluation**.

5.10. In Simpler Words

In this chapter, we focus on two things: The classification and the identification of uncertainty. This is an important first step because we first need to better understand the threat to confidentiality, which is uncertainty. We define clear terms for software architects and security experts in the form of a classification. Afterward, we show how to extend this classification in order to identify more uncertainty sources.

A classification helps to understand important properties of a classified element—in our case, uncertainty. Many researchers already defined many classifications of uncertainty. However, they do not fit our purpose, relating uncertainty to confidentiality. Thus, we define our own classification based on the existing ones. We propose different categories like the location, the manageability, or the severity of the impact. For instance, the uncertainty regarding user behavior can be located in the system input as users operate a software system. We cannot control their input but we can try to think of possible behavior in advance and thus partially reduce this uncertainty. If the user behaves maliciously and tries to attack our system, the severity of the impact can be high. In total, we define eight categories like these three. They help us to better understand uncertainty, and group uncertainty sources together. We apply this knowledge to data flow diagrams, a diagram type that is often used to reason about confidentiality. By connecting this uncertainty terminology to data flow diagrams, we lay the foundations for the automated analyses we will present in the following chapters.

To help software architects and security experts even more, we also discuss how to identify uncertainty sources. If you had not read the previous paragraph, you might not have known about the uncertainty of the user's behavior. This is the nature of unexpected changes—often, you do not even know that you do not know about the change. But now that I told you, you know about the uncertainty of user behavior. You still do not know how the user will behave in the end, but you are at least aware of the issue. We present a software tool that helps to collect uncertainty sources and make software architects and security experts aware of them. This also helps to connect different uncertainty sources, as they often have relations. For instance, if we apply input validation to address the uncertain user input, we can address this uncertainty. But if we do not know how this input validation works and whether it always works correctly, the validation itself is an uncertainty source. Then, both uncertainties are connected—this is called *uncertainty interaction*. Uncertainty interactions are particularly dangerous and hard to find. Our contribution regarding the identification and classification can also help here.

6. Uncertainty Propagation to Enable Uncertainty Impact Analysis

In this chapter, we present the second Contribution **C2**. This contribution covers the third activity shown in Section 4.1, i.e., the propagation of uncertainty. We present means to model and analyze uncertainty in software architectures in order to enable the early assessment of the potential impact of uncertainty on confidentiality.

As motivated throughout this thesis, uncertainty can quickly become critical, especially regarding security-related properties like confidentiality. For proactive decision-making as implied by *Privacy by Design* [224] and adaptation to a changing environment, uncertainty should be analyzed as early as possible. This requires means to model and analyze uncertainty that build on existing Architectural Description Languages (ADLs) and do not require extensive knowledge or modeling and analysis effort. As implied by the term *what-if* analysis, we aim to quickly understand the potential impacts of uncertainty without having to re-model—or even worse, re-implement—a software system.

Much is yet unknown about potential uncertainty sources and their effects [99], e.g., in early design due to abstract requirements and open decisions [167], or in systems of systems because of unpredictable behavior and complex dependencies and interactions [50, 190]. Here, uncertainty in a software system (e.g., component choices) or its environment (e.g., user input) can void assumptions [2]. Design time confidentiality analysis can help to identify potential confidentiality violations early [236], but is either not suited to deal with uncertainty [233], or requires to much additional modeling effort [37, 266]. This lack is supported by the literature. Hezavehi et al. [115] recently found a “lack of systematic approaches for managing uncertainty” [115]. This also becomes visible in a recent study by Troya et al. [255], which highlights the opportunity of providing useful notations and tool support to model and analyze uncertainty.

We aim to address this problem with a tool-supported approach for *Uncertainty Impact Analysis* regarding confidentiality. Based on software-architectural modeling, we propagate uncertainty sources through the software system to identify impact locations that subsequently can be analyzed and mitigated accordingly. This approach fills the gap between identifying uncertainty sources and understanding their actual impact on a system’s confidentiality by using the information of the software’s architecture. This also addresses the limitation of understanding the transitive impact of uncertainty described in Section 5.8. To achieve this, we build upon the concept of architecture-based change impact analysis [110, 210, 211]. Such analyses trace changes, e.g., replacing components or altering interfaces, and predict the impact on the software system. To analyze the impact of uncertainty on confidentiality, we combine this structural propagation with the propagation along extracted data flows. This enhances the precision of the impact, especially regarding confidentiality, as “problems tend to follow the data flow, not the control flow” [239]. By calculating the impact of uncertainty, software architects can identify potential issues without laborious modeling and analysis of architectural variations [101, 265].

As defined in Section 5.2, we stress the importance of distinguishing an uncertainty *source* and its *impact*. Only considering uncertainty sources rather than their impact impedes precise mitigation. Moreover, similarly to confidentiality analysis [234], manual propagation of uncertainty is not feasible, especially in large systems of systems. Thus, the provided concepts need tool-supported modeling and analysis to aid software architects. Here, we build on the classification of uncertainty, which was presented in the previous chapter, see Chapter 5. We summarize this research concern of uncertainty propagation and impact analysis in the second research question:

Research Question 2: How to propagate classified uncertainty sources based on architectural modeling to predict their impact on a system’s confidentiality?

We also present one extension to this work. As acknowledged in Section 5.5, sources of uncertainty are rarely independent, and their interactions can affect the overall system in subtle and unpredictable ways [52]. Similarly to the discussion presented in Chapter 5, this Uncertainty Interaction Problem (UIP) demands representing uncertainty as a first-class concern. To consider heterogeneous uncertainty sources, i.e., uncertainty which differs in representation and nature, and their propagation and interaction, we generalize the propagation concept to define Uncertainty Flow Diagrams (UFDs) [49]. Building on our findings on Data Flow Diagrams (DFDs) and uncertainty propagation presented throughout this thesis, we present an initial, yet more general approach to propagate uncertainty that is not limited to confidentiality. This extension positions our work in the research area of Self-Adaptive Systems (SASs) [273] and shows further application possibilities.

The remainder of this chapter is structured as follows: First, we present the problem statement. We discuss how to represent uncertainty in architectural models using the ADL Palladio Component Model (PCM) [207]. Afterward, we present uncertainty propagation separately for architectural models and DFDs and then combine these findings to define a joint uncertainty impact analysis regarding confidentiality. Building on that, we combine the analysis with the identification approach presented in Section 5.6, to address the Uncertainty Awareness Problem (UAP) of uncertainty impact analysis. Last, we generalize the propagation concept to define UFDs. We close the chapter with a discussion of assumptions and limitations and give a summary and an outlook.

Literature: This chapter is based on the following (co-) authored publications: [ACM MODELS-C 2022], [IEEE/ACM SEAMS 2023], [IEEE/ACM SEAMS 2024], [ACM/IEEE MODELS-C 2024]

6.1. Problem Statement

We summarize the problems **P1** – **P4** addressed by Contribution **C2**. Finding solutions to these problems helps to provide a comprehensive answer to ❓ **Research Question 2**.

P1: Representation of uncertainty in architectural models In order to consider uncertainty in software architecture, we require means to describe uncertainty in the architectural abstraction. We benefit from the restriction to software-architectural uncertainty of our classification [102] that can be used as a baseline. Nevertheless, we need to relate the different uncertainty types to elements of the software architecture [255]. The representation of uncertainty in architectural models can be realized by annotating uncertainty to ADLs like the PCM [207]. Last, we need to understand the representation of the impact of uncertainty both in PCM and DFDs.

P2: Propagation of uncertainty for uncertainty impact analysis We require rules to propagate uncertainty in all models that are considered in design time confidentiality analysis. This includes architectural models described with ADLs and also DFDs. For both representations, we need to understand how uncertainty can impact confidentiality, and how this impact can be calculated. We also need to relate both representations to define a comprehensive uncertainty impact analysis. Last, analysis automatization is required as manually propagating uncertainty in large software systems is not feasible.

P3: Identifying uncertainty for uncertainty impact analysis Similar to other architecture-based analyses under uncertainty [101, 266], uncertainty impact analysis suffers from the Uncertainty Awareness Problem (UAP). We cannot assume that software architects are aware of all relevant uncertainty sources. We can also not assume that software architects have enough expertise to annotate the correct uncertainty sources in the architectural model. These shortcomings severely limit the accuracy of the uncertainty impact analysis as only identified and correctly annotated uncertainty sources can be propagated. Addressing this problem also helps to move towards end-to-end approaches [273].

P4: Uncertainty propagation to address uncertainty interactions The Uncertainty Interaction Problem (UIP) can be addressed using uncertainty propagation. This requires appropriate notations to follow the flow of uncertainty in software systems. These notations need to support not only homogeneous uncertainty but also heterogeneous uncertainty of different type and nature. Another key requirement for such a notation is that it should be able to capture how uncertainty propagates both horizontally, i.e., within the same level of abstraction, and vertically, i.e., across different levels of abstraction [50].

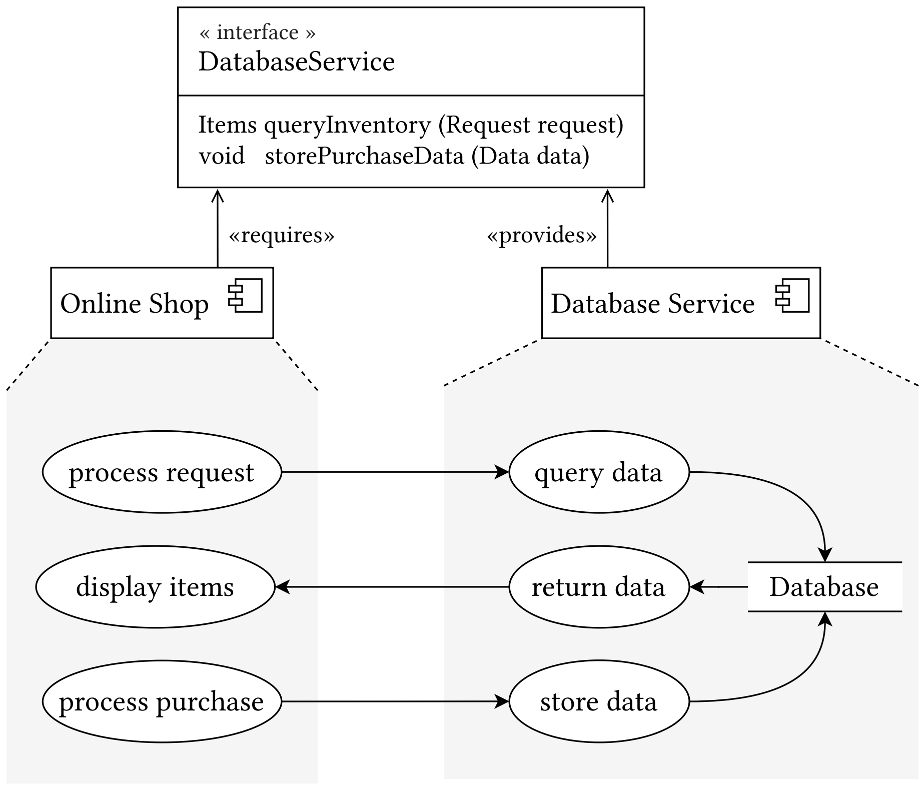


Figure 6.1.: Simplified mapping of the components of the running example to a Data Flow Diagram (DFD).

6.2. Representing Uncertainty in Architectural Models

In order to enable architecture-based uncertainty impact analysis regarding confidentiality, we first need to understand the representation of uncertainty sources and their impact both in architectural models and DFDs. Here, we build on the findings presented in Section 5.5 about representing uncertainty sources in DFDs and Directed Acyclic Graphs (DAGs). In this section, we discuss the relation of software architectural models and DFDs and the role of uncertainty in both representations of a software system. Afterward, we define a meta model for representing the propagated uncertainty impact in DFDs and derive which elements of a software architecture can be annotated with uncertainty sources. We build on the PCM as it represents a mature and widely-used ADL. Nevertheless, the findings of this section are also applicable to other representations of a software architecture like Unified Modeling Language (UML) diagrams [186]. This addresses our first Problem **P1** about the representation of uncertainty in architectural models.

6.2.1. Modeling Uncertainty Sources and Uncertainty Impacts

We start by explaining the relation of a high-level structural view like a PCM component repository model and DFDs using our running example. Figure 6.1 shows the simplified mapping of the two components *Online Shop* and *Database Service*, the interface that

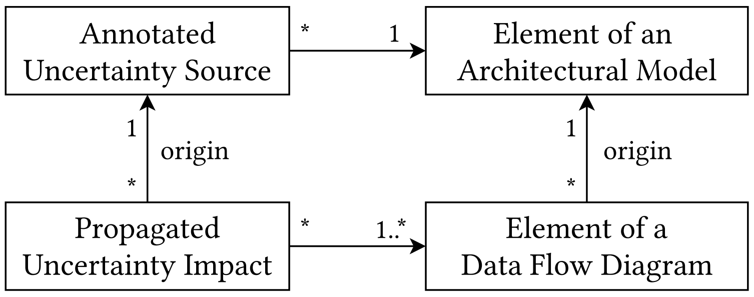

Annotated Uncertainty Source
Element of an Architectural Model
Propagated Uncertainty Impact
Element of a Data Flow Diagram
*
1
origin
1..*

Figure 6.2.: Simplified view on the relation of software-architectural uncertainty sources, their propagated impact, software architectural models, and Data Flow Diagrams (DFDs).

connects those components, and a DFD showing the data flows of querying and purchasing items. This mapping has been precisely defined by Seifermann [233], which includes the automated extraction of DFDs from the PCM.

The components of Figure 6.1 are connected by an interface and additionally encapsulate their own behavior. A PCM component can have multiple intended purposes, e.g., the *Online Shop* component is used to process item requests and to process purchases. This is represented in the signatures of an interface and also becomes visible when looking at the DFD. Additionally, a single interface signature's behavior can be represented by multiple DFD nodes, e.g., querying items with the signature *queryInventory* involves the *query data*, *Database*, and *return data* nodes. Last, some information of the architectural model is not mapped to DFDs, e.g., every information related to the details of signatures and interfaces. We find that there is no one-to-one mapping between an architectural model of the PCM and a DFD. Both models can represent the same software system but show different concerns and abstraction levels.

When we discussed the difference of uncertainty *sources* and the *impact* of uncertainty on confidentiality, we stated that the locations can differ. Figure 5.1 showed an uncertainty source annotated to the customer using the *Online Shop* and its potential impact to the *Database Service*. This relation also becomes visible in this mapping, illustrated in Figure 6.1. An uncertainty source in the input of the *process request* node could affect the *query data* node and transitively affect all other nodes in the flow, starting with the *Database*. Put simply, unspecified user input could lead to confidentiality violations in all of these nodes. We find that a single uncertainty source can impact multiple nodes of a DFD and that the representation of DFDs is suited to express this impact.

These findings are summarized in Figure 6.2. An element from the architectural model can be represented by zero, one, or many elements of DFD. Our example shows all three cases, as discussed previously. When extracting DFDs from PCM with the mapping of Seifermann [233], every DFD element can be traced back to one originating element of the software architecture.

Software architects annotate identified uncertainty sources to the architectural model as this enables them to stay on the architectural abstraction [97]. Although every uncertainty source is annotated to exactly one element of the software architecture, there is no limit how many uncertainty sources can be annotated to a single element. Furthermore, there also can be elements without any annotated uncertainty, either because it has not yet been identified, or because it is currently not relevant. See the procedure of uncertainty-aware analysis described in Section 4.1 for more details.

As shown in the discussion in Figure 6.1, a single annotated uncertainty source can have multiple impact locations. Similarly to the relation of architectural elements and DFD elements, every impact can be traced back to one originating uncertainty source. Put simply, once the confidentiality has been violated, software architects can trace the violation back to its cause, e.g., an overlooked uncertainty source. A propagated uncertainty impact affects at least one DFD element—or more, as seen in the running example. Nevertheless, not every element of a DFD has to be impacted by uncertainty.

This relation of uncertainty sources, their propagated impact, architectural models, and DFDs lays the foundation for all further discussions in this chapter. Note that this is only one view on the relation of these concepts. For instance, one could argue that a single uncertainty source can affect multiple elements of an architectural model[1]. For the sake of consistency, we are keeping this view for the remainder of this chapter.

ℹ **Finding:** Architectural models are a suitable representation to be annotated with software-architectural uncertainty sources and Data Flow Diagrams (DFDs) are suitable to represent their impact on confidentiality. Multiple DFD elements can represent a single element of the architectural model, and similarly, multiple elements of a DFD can be impacted by a single uncertainty source.

6.2.2. A Meta Model of the Uncertainty Impact in Data Flow Diagrams

In Section 5.3, we defined our classification of uncertainty that is tailored to confidentiality. The category *Architectural Element Type* is used as the primary category for annotating architectural models. It offers five uncertainty types, i.e., five possible target elements of uncertainty. Component uncertainty represents uncertainty in software components, connector uncertainty represents uncertainty in the wiring between components, interface uncertainty represents uncertainty in interfaces, external uncertainty represents uncertainty in resources, or users, and behavior uncertainty represents uncertainty in behavior descriptions. See Table 5.2 for details on the classification and Table 5.11 for details on the relation of these options and DFD elements. We do not limit our modeling to a certain type of uncertainty, e.g., environmental uncertainty [37], or structural uncertainty [266] in the following, but aim to support all five uncertainty types.

While these uncertainty types and their associated software-architectural elements are sufficient to document uncertainty sources, additional effort is required to also represent

[1] When I came back from SEAMS ’23 in Australia where I presented this work, I had this very discussion with Shmuel Tyszberowicz. I conclude that allowing one uncertainty source to only affect a single element does not limit the generality as there could be infinite uncertainty sources that affect different elements and that are related and analyzed together.

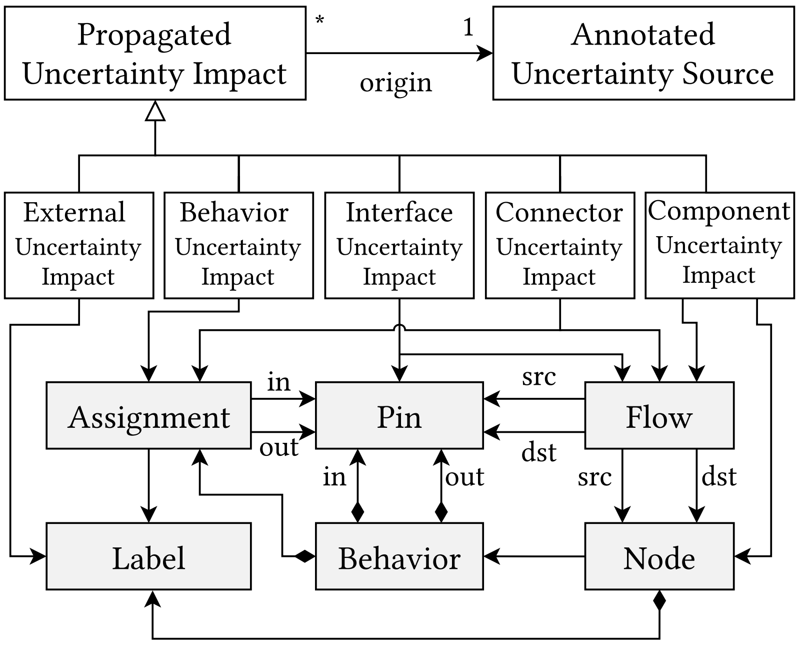

Propagated Uncertainty Impact
*
1
origin
Annotated Uncertainty Source
External Uncertainty Impact
Behavior Uncertainty Impact
Interface Uncertainty Impact
Connector Uncertainty Impact
Component Uncertainty Impact
Assignment
in
out
Pin
src
dst
Flow
in
out
src
dst
Label
Behavior
Node

Figure 6.3.: Meta model of Data Flow Diagrams (DFDs) with the five different uncertainty impact types.

their impact, see Figure 6.2. As discussed, confidentiality and the impact of uncertainty on confidentiality is best investigated using DFDs. To this end, we extend the unified modeling primitives of DFDs [235], which were introduced in Section 2.5 and related to uncertainty in Section 5.5.

Figure 6.3 shows our meta model that combines the DFD primitives (highlighted gray) and the five uncertainty types. We extend the relation of uncertainty sources and their propagated impact to DFD elements introduced in Figure 6.2 based on the mapping shown in Table 5.11. One *Annotated Uncertainty Source* can have any number of DFD elements affected by the *Propagated Uncertainty Impact*. We explain the relation of the five different uncertainty impact types and DFD elements in the following.

Nodes represent the system's structure and are affected by *Component* uncertainty, e.g., Uncertainty **U3** regarding the deployment of the *Database Service* component in the running example would affect all nodes that represent this component. We exemplified how to represent this in DAGs in Figure 5.2. *Flows* connect these nodes by transmitting data and are affected by *Connector Uncertainty*. Additionally, *Connector* uncertainty affects *Assignments* of confidentiality-related labels, e.g., whether the user input is erroneous or malicious in Uncertainty **U1** of the running example. *Flows* are also affected by *Component* uncertainty, as altering components can change the flow of data, and also by *Interface* uncertainty. Primarily, *Interface* uncertainty affects *Pins* that decouple flows from a node and function as interfaces in the unified modeling primitives for DFDs [235]. In Section 5.5, we called these three uncertainty types *secondary* uncertainty. Also, in this meta model, we can see their broad impact on DFD elements. All *secondary* uncertainty affects two different DFD elements and can have a structural impact on the overall system.

In comparison, *primary* uncertainty, i.e., *Behavior* uncertainty and *External* uncertainty, only affects one DFD element. Although this impact is more direct, we cannot make any claims on the severity or probability of confidentiality violations due to such uncertainty compared to *secondary* uncertainty—this depends on the software system, the environment, and the uncertainty sources in both. *Behavior* uncertainty impacts the assignments that are used to represent the behavior of a *Node*. In our running example, Uncertainty **U2** about the data processing could be represented as such an uncertainty source. This would affect whether the behavior of the data processing node assigns the *Labels* for validation or encryption. Ultimately, this can lead to confidentiality violations similarly as all other uncertainty sources. Last, *Nodes* can also be described by *Labels* to represent their confidentiality-related properties, which are affected by *External* uncertainty. In our running example, Uncertainty **U4** can be represented as *External* uncertainty affecting a label that expresses the trustworthiness of said DFD node.

Our metamodel and its graphical representation in Figure 6.3 demonstrate again the good fit of the five uncertainty types introduced by the category *Architectural Element Type* of our classification to DFDs. Every uncertainty impact affects another element type of the unified modeling primitives. Additionally, all *secondary* uncertainty types affect the flow, as discussed in Section 5.5. The only exception to this rule is the *Behavior*, which has no associated uncertainty type because it only acts as a container for pins and assignments. In sum, this meta model enables us to connect the uncertainty types from the software architecture to their corresponding impact type. The representation based on DFD elements simplifies the subsequent task of uncertainty propagation.

Finding: The propagated uncertainty impact can be represented by the five uncertainty types of the classification category *Architectural Element Type*. There is a direct relation of these types to the unified modeling primitives of Data Flow Diagrams (DFDs), which lays the foundation for the precise impact of uncertainty regarding confidentiality in software architectures.

6.2.3. Annotating Uncertainty Sources in the Palladio Component Model

By now, we have discussed the relation of annotated uncertainty *sources* and their propagated *impact* with the more abstract view of architectural models and the more concrete view of DFDs. However, to be applicable, one piece of the puzzle introduced in Figure 6.2 is missing: The mapping of uncertainty sources to concrete elements of the architectural model. Instead of only referring to architectural concepts like components, connectors, and interfaces, we discuss concrete annotation targets based on the ADL PCM [207] in the following. These annotation targets can be derived by following the transformation [233] from PCM to DFDs. Put simply, we investigate, which elements of the PCM are relevant for DFDs—and thus, for confidentiality—and annotate uncertainty sources to them. This not only concludes the relation discussed in this section but also lays the foundation for tool-supported modeling and automated analysis presented hereafter.

Uncertainty Type	PCM elements that can be annotated
Component	AssemblyContext
Connector	AssemblyConnector, ProvidedDelegationConnector
Interface	Signature, Interface
External	ResourceContainer, UsageScenario
Behavior	ExternalCallAction, EntryLevelSystemCall, SetVariableAction, BranchAction

Table 6.1.: Mapping of the five uncertainty types to Palladio Component Model (PCM) elements.

For the remainder of this chapter, we use the terminology known from the PCM. See Section 2.4 for an overview of PCM, or look into its comprehensive documentation [21, 206, 207]. To minimize confusion, we use the original terminology [206, 207] although there are slight deviations in the current Palladio implementation [205]. However, this does only affect two PCM element types used in the following: *Signature* compared to *OperationSignature*, and *Interface* compared to *OperationInterface*. Note that there are also slight differences between the PCM elements used for uncertainty impact analysis and those used to analyze confidentiality under uncertainty in later chapters because we only *annotate* uncertainty source instead of altering the architectural model. However, this is for purely pragmatic reasons that do not affect the underlying concepts.

Table 6.1 shows the mapping of the uncertainty types to elements of the PCM that can be annotated with this uncertainty source. *Component* uncertainty sources affect instances of components and are annotated to the *AssemblyContext*. *AssemblyContexts* refer to component types from the PCM component repository model and instantiate them within the concrete wiring of the PCM system model. Put simply, they represent concrete components and not component types and are thus suited to represent *Component* uncertainty sources visible to software architects. In our running example, we have two repository components, *Online Shop* and *Database Service*, which are both instantiated once. To express uncertainty sources on type level, refer to *Interface* uncertainty which annotates elements from the PCM component repository model.

Connector uncertainty affects inter-component structures, i.e., the wiring visible in the PCM system model by using *AssemblyConnectors*. These connectors link concrete component instances, i.e., *AssemblyContexts*, together. In our running example, the connection of the *Online Shop* and the *Database Service* component is realized using such connectors. Additionally, *Connector* uncertainty can also affect *ProvidedDelegationConnectors*. These connectors represent system interfaces that can be used by users outside of the modeled system, e.g., in *UsageScenarios*. We support both connector types because both can be used to impact the usage of components, which is expressed by *Connector* uncertainty.

Interface uncertainty affects the interfaces that are defined in the PCM component repository model. Here, we support two levels of granularity: Either a single *Signature* of an interface or the complete *Interface*, i.e., all of its signatures, can be affected. This enables software architects to minimize the annotation effort, e.g., to express that every communication over a selected interface is subject to uncertainty. Note that in contrast to the first two uncertainty types, *Interface* uncertainty is specified on a type level and not in the PCM system model. In our running example, the two components offer interfaces with multiple signatures. One of these interfaces is illustrated in Figure 6.1.

External uncertainty affects the context of a software system, e.g., resources, and users. It can be annotated to *ResourceContainers*, which are part of the PCM resource environment model, and *UsageScenarios*, which are part of the PCM usage model. Both elements are part of the system environment and both can transitively affect the software system, e.g., because components are deployed to uncertainty-afflicted resources in the PCM component allocation model. Selecting these two elements to express environmental concerns that can affect confidentiality is inspired by architecture-based confidentiality analysis [234]. In our running example, the *On Premise Server* and the *Cloud Service* represent *ResourceContainers*. The *Customer* has multiple *UsageScenarios*, e.g., browsing for items, or purchasing items.

Behavior uncertainty affects the behavior of the software system and its users. The behavior of the system is described in the form of Service Effect Specifications (SEFFs) in a PCM component repository model. Here, uncertainty can be annotated to *ExternalCallActions* that represent components calling each other, *SetVariableActions* that represent internal activities, and *BranchActions* that can change the control flow. All of these elements can affect the confidentiality of data flowing through them and are thus relevant for *Behavior* uncertainty. Last, *Behavior* uncertainty can also be annotated in *EntryLevelSystemCalls* that represent the user calling the system interface and are located in the PCM usage model. Although the location of classified uncertainty differs, i.e., uncertainty in the *System Input* instead of the *System Behavior*, the *Architectural Element Type* option stays the same for impact analysis. In our running example, the *Customer* requesting the purchase of an item is represented by an *EntryLevelSystemCall*, the storing of data called from the *Online Shop* is realized with an *ExternalCallAction*, and the data processing can be modeled with a *SetVariableActions*. For the sake of simplicity, we do not include branches.

In sum, all five types of uncertainty sources can be annotated to PCM elements. Based on the Palladio [205] tool support, existing modeling tools can be extended to support these annotations. While uncertainty sources are best represented in the architectural model, their corresponding impact types are best represented in DFDs. The uncertainty types of the category *Architectural Element Type* originate from investigating existing classifications and DFD notations. Note that while these are generally applicable, alternative annotation targets other than the presented PCM elements are possible [23]. This is due to the nature of models representing reality, i.e., PCM modeling software architecture [245].

ℹ **Finding:** The source of the five uncertainty types of the category *Architectural Element Type* can be annotated to the Palladio Component Model (PCM). This Architectural Description Language (ADL) can be extended to annotate uncertainty within the software system, e.g., its structure and behavior, and the system environment, e.g. usage scenarios and hardware resources. Relating the five uncertainty types to concrete PCM elements enables software architects to represent uncertainty sources on architectural abstraction.

6.3. Uncertainty Impact Analysis for Architectural Models

Building on the representation of uncertainty sources within architectural models, we can propagate the uncertainty to better understand its impact. Starting from the annotation targets discussed in the previous section, we calculate this impact to define an *uncertainty impact analysis* [115]. Regarding confidentiality, this analysis requires two steps [102]. First, the propagation within the architectural model, and second, the propagation in extracted DFD. As the uncertainty sources are annotated within the architectural model, we start with this part of the impact analysis. As introduced previously, we will use the ADL PCM to detail the propagation rules. This addresses Problem **P2**.

6.3.1. Adapting Change Impact Analysis to Uncertainty Propagation

The name "uncertainty impact analysis" is inspired by architecture-based *change impact analysis* [110, 210, 211]. Change impact analysis helps to predict the impact of changes in software architectures, e.g., changing interfaces, or replacing components [210]. The early assessment and planning of change requests support software architects in software evolution. The underlying concept of architecture-based change impact analysis is the propagation of changes in architectural models to identify potentially affected elements. This requires the definition of propagation rules for each type of change, e.g., renaming a signature in an interface requires the providing component to also adapt its naming.

Regarding the PCM, we build on Karlsruhe Architectural Maintainability Prediction (KAMP) [46, 110, 112, 210, 211], as it provides a comprehensive foundation on change propagation. It has been evaluated for many different use cases, e.g., architectural models [211], business processes [210], or automation systems [112]. KAMP also supports the transitive propagation of changes, i.e., changes that occur due to other changes in the model, also called ripple effects [33]. It has been used for other domain-spanning analyses [110], e.g., to define architecture-based attacker propagation [264, 267, 268]. Uncertainty is often also referred to as *unanticipated change* [115, 272]. Thus, it should not be surprising that the concept of change propagation can be reused[2].

Finding: In architectural models, uncertainty can be treated like unanticipated changes. This enables the leverage of propagation rules of architectural change impact analysis for the propagation of uncertainty.

Change impact analysis defines three sets of elements [33, 46]. The Starting Impact Set (SIS), also sometimes called change set, contains the initial elements of the change request. The Actual Impact Set (AIS), also sometimes called affected set, contains all elements that

[2] But in fact, it was. I have been trying to assess the impact of uncertainty on confidentiality for quite some time without making much progress. In 2021, after helping in an oral exam that covered the topic of change impact analysis, I asked myself: what if we could treat uncertainty like change in terms of its impact? Yes, we can, and the result is presented in this section. Call this serendipity if you want.

Uncertainty Source	Annotated PCM element	Exemplary name
User input (**U1**)	ProvidedDelegationConnector	*PurchaseInterface*
Data processing (**U2**)	SetVariableAction	*processUserData*
Component deployment (**U3**)	AssemblyContext	*DatabaseService*
Provider trustworthiness (**U4**)	ResourceContainer	*CloudService*

Table 6.2.: Starting Impact Set (SIS) representing the annotated uncertainty sources in the running example.

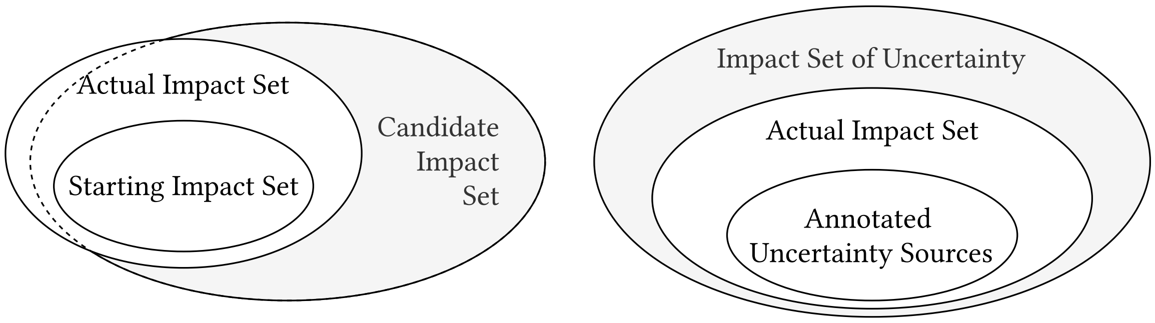


Figure 6.4.: Informally illustrated relation of the different sets of change impact analysis (left) and uncertainty impact analysis (right).

actually have to change due to the change request. However, change impact analysis can only estimate this set [46]. The Candidate Impact Set (CIS), also sometimes simply called impact set, represents this estimation. Here, the goal of change impact analysis is to keep the estimation of the CIS as close to the AIS as possible.

We build on this terminology to apply these sets to *uncertainty* impact analysis. The SIS represents the elements of the software architecture with annotated uncertainty sources. Similarly to change impact analysis, this represents the starting point for any propagation. Additionally, these elements are already affected by uncertainty, which could lead to confidentiality violations even without any further propagation. In the running example, the SIS contains the four elements from the software architecture that are annotated with the four uncertainty sources. Table 6.2 shows the annotation of the uncertainty sources to the PCM elements. Note that these annotation targets only represent exemplary locations and could be modeled differently, see the discussion above. We also include exemplary names comparable to those used in the running example model from our data set [98].

The AIS represents the elements, that are actually impacted by the uncertainty sources in the SIS and can violate confidentiality. This set includes all elements from the SIS, but can additionally contain elements that are directly or indirectly affected. However, we can only estimate this set. Similarly to change impact analysis, an uncertainty impact analysis yields a CIS, which represents an estimation of potentially impacted elements. Here, we require a more strict relation. While change impact analysis may underestimate the CIS compared to the AIS, uncertainty impact analysis regarding confidentiality shall always overestimate the CIS. Put simply, an ignored impact of uncertainty could lead to overlooked confidentiality violations. Especially regarding security-related quality properties like confidentiality, high recall can be more important than high precision [102]. Additionally, the uncertainty impact analysis is used for an *early* prediction and assessment of uncertainty, not for precise analysis, see Section 4.1. This has to be considered in defining the propagation rules for uncertainty. For the sake of simplicity, we refer to the AIS as the actual impact set and to the CIS simply as *impact set* of uncertainty. The relation of the terminology of both analyses is illustrated in Figure 6.4.

In the following, we describe propagation rules for all annotation targets of uncertainty sources in the PCM. They have to be described separately for each relevant PCM element as they are specific to the ADL PCM. Nevertheless, the general concept of architectural uncertainty propagation is generalizable [49]. The goal of uncertainty propagation in the architectural model is to identify all uncertainty-afflicted elements of the software architecture, which are later represented by the extracted DFD [233]. In most cases, this means starting with propagation rules from change impact analysis and then identifying relevant SEFF actions, as these are relevant to the DFD extraction of Seifermann [233]. This step is important because of the abstraction gap between the representations of DFDs and the PCM, see Section 6.2. By identifying those elements, the coupling of the architectural impact analysis with the propagation of uncertainty in DFDs is simplified. Only propagating uncertainty in DFDs is not sufficient due to the missing information compared to architectural models.

Finding: Uncertainty impact analysis regarding confidentiality in architectural models consists of two steps. First, identifying directly and indirectly affected elements, following the propagation rules of change impact analysis. Second, identifying originating elements of the transformation to DFDs, which are relevant for further propagation.

6.3.2. Uncertainty Propagation of Component Uncertainty

Component uncertainty can only be annotated to *AssemblyContexts*, which simplifies the propagation logic. Algorithm 6.1 shows the propagation algorithm. Every propagation algorithm is called with the annotated uncertainty source contained in the parameter *uncertainty* and the PCM model contained in the parameter *model*. We always start by creating an empty list of results and retrieving the model element that has been annotated with the uncertainty source, following the relation specified in Figure 6.2.

In the case of *AssemblyContexts*, which represent instantiated components, we retrieve the component type from the PCM component repository model in Line 4 and then query for its SEFFs in Line 5. A component provides one SEFF for each signature of a provided interface. For instance, in our running example, the *Database Service* provides SEFFs for the operations *queryInventory* and *storePurchaseData*, see Figure 6.1.

From the perspective of the change impact analysis, these SEFFs are impacted by a change to the component. To close the gap to the representation in DFDs, we need to retrieve the initial actions of all SEFFs that are represented as DFD nodes [233]. We show this

Algorithm 6.1 Algorithm for component uncertainty propagation

```
1:  procedure PROPAGATECOMPONENTUNCERTAINTY(uncertainty, model)
2:      result ← ∅
3:      assemblyContext ← GETANNOTATEDELEMENT(uncertainty, model)
4:      component ← GETREPOSITORYCOMPONENT(assemblyContext, model)
5:      seffs ← GETSEFFS(component, model)
6:      for seff ∈ seffs do      ▷ Component uncertainty affects all SEFFs of a component
7:          startAction ← RETRIEVESTARTACTION(seff, model)
8:          result ← result ∪ {startAction}
9:      end for
10:     return APPLYTOASSEMBLY(result, assemblyContext)
11: end procedure
```

Algorithm 6.2 Algorithm for retrieving the initial StartAction of a SEFF

```
1:  procedure RETRIEVESTARTACTION(seff, model)
2:      actions ← GETACTIONS(seff, model)
3:      for action ∈ actions do
4:          if TYPEOF(action) = StartAction then
5:              startAction ← action          ▷ The first start action is the start of the SEFF
6:              return startAction
7:          end if
8:      end for
9:  end procedure
```

separately in Algorithm 6.2 because we reuse this functionality several times in other algorithms. A SEFF contains actions that represent its behavior, comparable to UML activity diagrams. The initial actions have the PCM element type *StartAction*, see Line 4. Using the functionality described in Algorithm 6.2, we iterate over all SEFFs, starting in Line 6. Once we found this action, we add it to the results list in Line 8. The algorithm ends after all initial *StartActions* of all SEFFs have been identified and returns the result.

Note that we skip some Palladio [205] implementation details. First, we do not show required type casts, e.g., from *RepositoryComponent* to *BasicComponent* to retrieve the SEFFs. Second, we hide method calls, e.g., to retrieve all actions, a detour from the SEFF through its behavior is required. Third, we skip the handling of the system context. This step is required because SEFFs are defined in the PCM component repository model and could be instantiated in multiple assemblies. However, we are only interested in those actions representing the *AssemblyContext* that is annotated with the component uncertainty. We illustrate this with a method call of *applyToAssembly* in Line 10. Last, we assume well-formed PCM models without errors. All of these details would only greatly increase the size of the algorithms without contributing to their understandability. To see the detailed implementation of all algorithms, please look into the data set [98].

Algorithm 6.3 Algorithm for retrieving all StartActions that describe a signature

```
1:  procedure retrieveStartActionsBySignature($signature$, $model$)
2:      $result \leftarrow \emptyset$
3:      $allComponents \leftarrow$ getAllRepositoryComponents($model$)
4:      for $component \in allComponents$ do
5:          $seffs \leftarrow$ getSEFFs($component$, $model$)
6:          for $seff \in seffs$ do                                ▷ Find called StartActions
7:              if getDescribedSignature($seff$, $model$) = $signature$ then
8:                  $startAction \leftarrow$ retrieveStartAction($seff$, $model$)
9:                  $result \leftarrow result \cup \{startAction\}$
10:             end if
11:         end for
12:     end for
13:     return $result$
14: end procedure
```

6.3.3. Uncertainty Propagation of Interface Uncertainty

Interface uncertainty can be annotated to full interfaces or selected signatures. Uncertainty affecting interfaces results in a wide impact due to their central role in software systems. The propagation of interface uncertainty requires the most extensive algorithm. We split the propagation into three parts. First, we identify all components and their actions that implement the interface in Algorithm 6.3. Second, we identify all actions that represent calls to the interface in Algorithm 6.4. Third, we identify all system calls from *UsageScenarios* to the interface in Algorithm 6.5. These three steps are required as we cannot make any assumptions on the usage of the interface, e.g., to wire components or to represent a user interface. By combining these three steps, we specify the algorithm for interface uncertainty propagation in Algorithm 6.6.

Interface uncertainty affects the PCM component repository model and is not limited single assemblies like *Component* uncertainty. Thus, the impact set can be even bigger, as all *AssemblyContexts* of a component can be affected. Put simply, if a component that is used multiple times in a software system has uncertainty in its interface, it affects all instances of this component and also all components that depend on these instances.

Algorithm 6.3 shows the algorithm to identify all SEFFs that implement a signature of an interface. Regarding change impact analysis, this resembles the adaption of all implementations of an interface, e.g., due to a rename of a signature. We iterate over all SEFFs of all components in the PCM component repository model to find matching *StartActions*. The retrieval logic is similar to *Component* uncertainty with two important distinctions. First, the retrieval is not limited to a single *AssemblyContext* but includes all components. Second, the algorithm only considers SEFFs that implement the given signature of the interface. Not necessarily all SEFFs of a component have to implement an interface because components can also provide multiple interfaces. This filter is realized in Line 7. Afterward,

Algorithm 6.4 Algorithm for retrieving all ExternalCallActions that call a signature

```
1:  procedure RETRIEVEEXTERNALCALLSTOSIGNATURE($signature$, $model$)
2:      $result \leftarrow \emptyset$
3:      $allComponents \leftarrow$ GETALLREPOSITORYCOMPONENTS($model$)
4:      for $component \in allComponents$ do               ▷ Iterate over all available actions
5:          $seffs \leftarrow$ GETSEFFS($component$, $model$)
6:          for $seff \in seffs$ do
7:              $actions \leftarrow$ GETACTIONS($seff$, $model$)
8:              for $action \in actions$ do                 ▷ Find calling ExternalCallActions
9:                  if TYPEOF($action$) = ExternalCallAction then
10:                     if GETCALLEDSIGNATURE($action$, $model$) = $signature$ then
11:                         $result \leftarrow result \cup \{action\}$
12:                     end if
13:                 end if
14:             end for
15:         end for
16:     end for
17:     return $result$
18: end procedure
```

we retrieve the *StartAction* in Line 8 using the logic described with Algorithm 6.2. In our running example, retrieving the *StartAction* of the SEFF that implements the signature *queryInventory* would yield a *StartAction* of the component *Database Service* because this component provides the interface that contains the signature, see Figure 6.1.

Algorithm 6.4 shows the algorithm to identify all system actions that call a signature of an interface. In the PCM, calls between components are modeled in SEFF using *External-CallActions* [21]. Regarding change impact analysis, this resembles adapting the call to an interface after the interface has changed. To identify the impacted *ExternalCallActions*, we have to iterate over all available actions starting in Line 4. This is achieved by iterating over all components, all of its SEFFs and over all of their actions. If the action represents an *ExternalCallAction*, we check its called signature in Line 10. All actions that fit these conditions are added to the result. These would also be part of the CIS of the change impact analysis KAMP [46]. In our running example detailed in Figure 6.1, an *ExternalCallAction* is used after the processing of the purchase in the *Online Shop* to initiate the storing of the data in the *Database Service*. To minimize costly iterations, concrete implementations of this algorithm can cache identified *ExternalCallActions* or use other query methods, which return all occurrences of an element type within a model [98]—we show the generic approach of iterating over all components to keep the algorithm as simple as possible.

Algorithm 6.5 shows the algorithm to identify all calls from a *UsageScenario* to a signature of an interface. In the PCM, calls to the the system are modeled using *EntryLevelSystem-Calls*. As discussed above, this step is required as we cannot assume that an interface is only used between components. This step is also performed by change impact analysis

Algorithm 6.5 Algorithm for retrieving all EntryLevelSystemCalls that call a signature

```
1:  procedure RETRIEVEENTRYCALLSTOSIGNATURE(signature, model)
2:      result ← ∅
3:      allUsageScenarios ← GETALLUSAGESCENARIOS(model)
4:      for usageScenario ∈ allUsageScenarios do
5:          actions ← GETACTIONS(usageScenario, model)
6:          for action ∈ actions do                    ▷ Find calling EntryLevelSystemCalls
7:              if TYPEOF(action) = EntryLevelSystemCall then
8:                  if GETCALLEDSIGNATURE(action, model) = signature then
9:                      result ← result ∪ {action}
10:                 end if
11:             end if
12:         end for
13:     end for
14:     return result
15: end procedure
```

[46]. To identify relevant *EntryLevelSystemCalls*, we iterate over all actions of all *UsageScenarios*, starting in Line 4. We first check for the type of the action in Line 7, and if it represents an *EntryLevelSystemCall*, we check the called signature in Line 8. Afterward, the actions are added to the result. This procedure closely resembles the retrieval of impacted *ExternalCallActions*. Here, the main difference is that we do not analyze the behavior of a component but the user behavior that is modeled in the PCM usage model. In our running example, all calls of the *Customer* to the *Online Shop* are modeled using *EntryLevelSystemCalls*, e.g., the querying of items.

We combine the algorithms 6.3, 6.4, and 6.5 to specify the propagation algorithm for *Interface* uncertainty in Algorithm 6.6. After retrieving the element that has been annotated with uncertainty in Line 4, we perform a type check in Line 5. If the annotated element is an *Interface*, we decompose the interface into its signatures and add it to the list of affected signatures. If the annotated element is a *Signature*, this list only consists of this single signature. This resembles one propagation step from change impact analysis [46, 210] and enables the treatment of both annotations targets similarly in the following. Put simply, annotating a signature instead of an interface increases the precision of the result but only has a negligible effect on the propagation logic. Afterward, we iterate over all signatures in Line 10. For each signature, we perform the three retrieval tasks, i.e., retrieving affected *StartActions*, *ExternalCallActions*, and *EntryLevelSystemCalls*. The combined result is returned from the algorithm. Note that this does not cause any typing problems, as we expect *any* action as a result, similarly to all other propagation algorithms.

In our running example, an *Interface* uncertainty that has been annotated to the interface between the *Online Shop* and the *Database Service* components would yield affected elements from both components. This includes *StartActions* from SEFFs of all implementations of operations specified in the interface and also *ExternalCallActions* to these operations.

Algorithm 6.6 Algorithm for interface uncertainty propagation

```
1: procedure propagateInterfaceUncertainty($uncertainty$, $model$)
2:     $result \leftarrow \emptyset$
3:     $affectedSignatures \leftarrow \emptyset$
4:     $annotatedElement \leftarrow$ getAnnotatedElement($uncertainty$, $model$)
5:     switch typeOf($annotatedElement$) do
6:         case Signature
7:             $affectedSignatures \leftarrow \{annotatedElement\}$
8:         case Interface                                  ▷ Decompose the interface
9:             $affectedSignatures \leftarrow$ getSignatures($annotatedElement$, $model$)
10:    for $signature \in affectedSignatures$ do         ▷ Combine the three retrieval steps
11:        $result \leftarrow result \cup$ retrieveStartActionsBySignature($signature$, $model$)
12:        $result \leftarrow result \cup$ retrieveExternalCallsToSignature($signature$, $model$)
13:        $result \leftarrow result \cup$ retrieveEntryCallsToSignature($signature$, $model$)
14:    end for
15:    return $result$
16: end procedure
```

However, as this interface is only used between components, no *EntryLevelSystemCalls* would be directly affected. Note that there is no call to *applyToAssembly* as we do not limit the impact of uncertainty to a single assembly. For instance, if the *Database Service* component would be used in multiple locations in the system, all instances would be affected by the uncertainty.

Similarly to the propagation of *Component* uncertainty shown in Algorithm 6.1, we hide implementation details. This includes chained method calls, e.g., to retrieve all actions of a *UsageScenario*, a detour over its behavior is required [205]. Although all algorithms presented in this section contain several loops, we do not expect problems regarding their scalability in practice. First, the sets to iterate over are usually small, e.g., because a properly defined interface only contains a small number of signatures [164]. Second, there are many ways to speed up the implementation, e.g., by using caches, or by integrating the extracted DFDs early in the propagation to filter potential candidates. We apply many of these optimizations in our implementation that are available in the data set [98].

6.3.4. Uncertainty Propagation of Connector Uncertainty

Connector uncertainty can be annotated to *AssemblyConnectors* and *ProvidedDelegationConnectors*. Algorithm 6.7 shows the propagation algorithm. Uncertainty affecting connectors behaves comparable to *Interface* uncertainty. It impacts both the called component and the calling entity, which can be another component or a user. The main difference is that connectors are specific to assemblies and modeled in the PCM system model. We can reuse

Algorithm 6.7 Algorithm for connector uncertainty propagation

```
 1: procedure PROPAGATECONNECTORUNCERTAINTY(uncertainty, model)
 2:     result ← ∅
 3:     connector ← GETANNOTATEDELEMENT(uncertainty, model)
 4:     interface ← GETINTERFACE(connector, model)
 5:     signatures ← GETSIGNATURES(interface, model)
 6:     calledAssembly ← GETPROVIDINGASSEMBLYCONTEXT(connector, model)
 7:     for signature ∈ signatures do                         ▷ Find called StartActions
 8:         actions ← RETRIEVESTARTACTIONSBYSIGNATURE(signature, model)
 9:         result ← result ∪ APPLYTOASSEMBLY(actions, calledAssembly)
10:     end for
11:     switch TYPEOF(connector) do
12:         case AssemblyConnector
13:             callingAssembly ← GETREQUIRINGASSEMBLYCONTEXT(connector, model)
14:             for signature ∈ signatures do          ▷ Find calling ExternalCallActions
15:                 actions ← RETRIEVEEXTERNALCALLSTOSIGNATURE(signature, model)
16:                 result ← result ∪ APPLYTOASSEMBLY(actions, callingAssembly)
17:             end for
18:         case ProvidedDelegationConnector
19:             delegatedContext ← GETDELEGATEDCONTEXT(connector, model)
20:             for signature ∈ signatures do        ▷ Find calling EntryLevelSystemCalls
21:                 actions ← RETRIEVEENTRYCALLSTOSIGNATURE(signature, model)
22:                 result ← result ∪ APPLYTOASSEMBLY(actions, delegatedContext)
23:             end for
24:     return result
25: end procedure
```

the logic defined for *Interface* uncertainty hereabove but we have to relate the results to the correct *AssemblyContexts*.

A *Connector* represents the wiring of two entities within a software system, expressed, e.g., using the ball and socket notation of UML component diagrams as shown in Figure 3.1. All connectors are based on an interface that we retrieve in Line 4. In the PCM, we support *AssemblyConnectors* between two components and *ProvidedDelegationConnectors* that can be referred to outside the system in *UsageScenarios*.

Independent of its type, a connector always has a providing *AssemblyContext*, representing the component that implements the provided interface. This called *AssemblyContext* is retrieved in Line 6. Afterward, we find all called *StartActions* beginning in Line 7. Here, we can reuse the logic defined for *Interface* uncertainty in Algorithm 6.3. The only difference is that we are only interested in those *StartActions* that represent the connected *AssemblyContext*. This is realized in Line 9, similarly to *Component* uncertainty.

Afterward, we address the calling side, starting in Line 11. In the case of an *Assembly-Connector*, we retrieve the calling assembly that requires the interface represented by the connector in Line 13. We use the logic defined for *Interface* uncertainty in Algorithm 6.4 to find the according *ExternalCallActions* in Line 14. Again, we have to apply the correct *AssemblyContext*. This time, this is achieved using the calling assembly in Line 16. In the case of a *ProvidedDelegationConnector*, we proceed with the retrieval of *EntryLevelSystemCalls*. Here, we are only interested in the provided delegation of the system interface, which is retrieved in Line 19. After receiving candidate *ExternalCallActions* in Line 20, we apply again the correct context in Line 22.

Put simply, to propagate *Connector* uncertainty we proceed as if the interface that is represented would be affected by *Interface* uncertainty. Every potentially impacted action is then applied to the correct *AssemblyContext*, either called or calling. Regarding change impact analysis, this resembles the required changes both in the component that implements an interface and the component requiring an interface. Besides the handling of *AssemblyContexts* and indirection from the *Connector* to interfaces, another difference is the distinction of the connector type. When propagating *Interface* uncertainty, we only use the PCM component repository model and we cannot make assumptions on the interface use, i.e., whether the interface is only used between components or also in *UsageScenarios*. Opposite to this, *Connectors* are part of the PCM system model that models this information. Thus, we are not required to handle all cases and can either retrieve *ExternalCallActions* or *EntryLevelSystemCalls*, but not both. In our running example, the results of an *Interface* uncertainty and a *Connector* uncertainty annotated between the *Online Shop* and *Database Service* components are identical because both components are only referred by a single *AssemblyContext*. Similarly to the other algorithms, we simplified some chained method calls and we refer to the data set for more details [98].

6.3.5. Uncertainty Propagation of External Uncertainty

External uncertainty can be annotated to *ResourceContainers* and *UsageScenarios*. Algorithm 6.8 shows the propagation algorithm. *ResourceContainers* are part of the PCM resource environment model and *UsageScenarios* are part of the PCM usage model. Although the propagation logic differs, it states the same: Find all actions that belong to the element that has been annotated with *external* uncertainty. Put simply, if a resource container or a user is affected by uncertainty, we cannot trust any of their actions.

The algorithm starts with a large switch statement in Line 4. In the case of an annotated *UsageScenario*, we iterate over all of its actions, starting in Line 6. This is similar to the identification of relevant *EntryLevelSystemCalls* of *Interface* or *Connector* uncertainty. The major difference is that we do not filter for a specific signature, but add all identified *EntryLevelSystemCalls*, as they could all be affected by uncertainty. We only consider *EntryLevelSystemCalls* as they represent the only user actions that are considered in the extraction of DFDs [233].

Algorithm 6.8 Algorithm for external uncertainty propagation

```
1: procedure propagateExternalUncertainty($uncertainty$, $model$)
2:     $result \leftarrow \emptyset$
3:     $annotatedElement \leftarrow$ getAnnotatedElement($uncertainty$, $model$)
4:     switch typeOf($annotatedElement$) do
5:         case UsageScenario
6:             $actions \leftarrow$ getActions($annotatedElement$, $model$)
7:             for $action \in actions$ do                          ▷ Find all EntryLevelSystemCalls
8:                 if typeOf($action$) = EntryLevelSystemCall then
9:                     $result \leftarrow result \cup \{action\}$
10:                end if
11:            end for
12:        case ResourceContainer
13:            $allAssemblyContexts \leftarrow$ getAllAssemblyContexts($model$)
14:            for $context \in allAssemblyContexts$ do                      ▷ Find all actions
15:                if getAllocation($context$, $model$) = $annotatedElement$ then
16:                    $component \leftarrow$ getRepositoryComponent($context$, $model$)
17:                    $seffs \leftarrow$ getSEFFs($component$, $model$)
18:                    for $seff \in seffs$ do
19:                        $actions \leftarrow$ getActions($seff$, $model$)
20:                        $result \leftarrow result \cup$ applyToAssembly($actions$, $context$)
21:                    end for
22:                end if
23:            end for
24:    return $result$
25: end procedure
```

In the case of an annotated *ResourceContainer*, we have to identify all *AssemblyContexts* that are allocated on this container. We retrieve all *AssemblyContexts* from the PCM system model in Line 13 and iterate over them. For each assembly, we check its allocation using the PCM allocation model in Line 15. If an *AssemblyContext* is deployed on the *ResourceContainer* that has been annotated with the uncertainty, we add all of its actions to the result list in Line 20. This step is comparable to the propagation of *Component* uncertainty. However, we do not limit the propagation to *StartActions* as every action could be affected by uncertainty.

In the running example, an *external* uncertainty annotated to the *ResourceContainer* where the *Online Shop* gets deployed would affect all of its actions. In the simplified illustration shown in Figure 6.1, this would include *process request*, *display items*, and *process purchase*. *External* uncertainty can have a particularly large impact if many *AssemblyContexts* are deployed on a single *ResourceContainers* affected by uncertainty. The same applies to *UsageScenarios* with many *EntryLevelSystemCalls*. Again, we simplified some methods shown in the algorithm to increase understandability. Interestingly, *external* uncertainty

Algorithm 6.9 Algorithm for behavior uncertainty propagation

```
1: procedure PROPAGATEBEHAVIORUNCERTAINTY($uncertainty$, $model$)
2:     $result \leftarrow \emptyset$
3:     $annotatedElement \leftarrow$ GETANNOTATEDELEMENT($uncertainty$, $model$)
4:     switch TYPEOF($annotatedElement$) do
5:         case EntryLevelSystemCall
6:             $result \leftarrow result \cup \{annotatedElement\}$
7:         case ExternalCallAction
8:             $result \leftarrow result \cup \{annotatedElement\}$
9:         case SetVariableAction
10:            $result \leftarrow result \cup \{annotatedElement\}$
11:        case BranchAction
12:            $actions \leftarrow$ GETACTIONS($annotatedElement$, $model$)
13:            for $action \in actions$ do
14:                if TYPEOF($action$) = StartAction then
15:                    $result \leftarrow result \cup \{action\}$
16:                end if
17:            end for
18:    return $result$
19: end procedure
```

requires the connection of all PCM models, namely the usage model, system model, component repository model including SEFFs, the resource environment model, and the allocation model. However, this is not surprising, as *external* uncertainty affects the system context in different places and is propagated into the software system.

6.3.6. Uncertainty Propagation of Behavior Uncertainty

Behavior uncertainty can be annotated to *EntryLevelSystemCall*, *ExternalCallActions*, *SetVariableAction*, and *BranchActions*. Algorithm 6.9 shows the propagation algorithm. Despite being the uncertainty source with the most possible annotation targets, the propagation algorithm is simple. The algorithm is realized using one big switch statement starting in Line 4. In the case of *EntryLevelSystemCalls*, *ExternalCallActions* and *SetVariableActions*, no propagation is required and the annotated elements can directly be added to the result. The reason for this simplicity is that all three elements are directly represented in the extracted DFD [233]. Thus, no further architectural propagation is required as the transitive impact can be calculated purely in the DFD-based uncertainty propagation. In the running example, a *behavior* uncertainty annotated to the *ExternalCallAction* to trigger the storing of data thus only contains the action itself as a result of the architectural propagation.

Only uncertainty annotated to *BranchActions* requires additional architectural propagation, starting in Line 11. These actions encapsulate inner sequences of actions, called

BranchTransitions, that represent the behavior of the SEFF if the conditions for one branch are met. There is no upper limit for the number of *BranchTransitions*. Thus, we retrieve all actions that are part of all transitions of the *BranchAction* and filter for *StartActions* that are added to the result. Because all actions are part of the PCM component repository model, no handling of *AssemblyContexts* is required. Note that this does *not* imply that *Behavior* uncertainty generally has a narrower impact on confidentiality. This only implies that this uncertainty type has a narrower impact using uncertainty propagation in architectural models. To comprehensively analyze the impact of uncertainty on confidentially, additional DFD-based propagation is required, as explained in Section 6.2.

6.3.7. Applying Architectural Uncertainty Impact Analysis

As explained previously in Section 6.2, we combine uncertainty impact analysis in architectural models and in DFD to get comprehensive insights on the impact of uncertainty on confidentiality. In this section, we introduced all algorithms for architectural propagation of uncertainty. To conclude, we apply these algorithms on the four annotated uncertainty sources **U1** – **U4** from our running example presented in Chapter 3. We introduced the terminology of SIS, AIS, and CIS at the beginning of this section. The result of the uncertainty impact analysis can be considered as a partial CIS, or partial *impact set*, because we still lack the second part of the impact analysis, i.e., the propagation in DFDs. Nevertheless, the identified elements of the architectural propagation represent the starting point for the DFD-based propagation. We present the results of the architectural impact analysis based on the exemplary SIS presented in Table 6.2.

Uncertainty **U1** affects the user input and is modeled as *Connector* uncertainty. This uncertainty source is annotated to the *ProvidedDelegationConnector* of the *PurchaseInterface*, which also represents the SIS. We apply the connector uncertainty propagation described in Algorithm 6.7. The interface that is represented by the *ProvidedDelegationConnector* consists of one signature: *purchaseItem*, to buy items from the online shop. This operation is implemented in the *Online Shop* component and used in a single *AssemblyContext* of this component. Thus, there is only one *StartAction* to identify, i.e., the *StartAction* of the SEFF that also contains the processing of the purchase prior to its storage in the database. As *StartActions* are usually not named in SEFFs [205], we refer to it as *purchaseItem.start*. Following the propagation in the calling direction, we identify the *EntryLevelSystemCall* in the *UsageScenario* that represents a customer purchasing items from the shop. *EntryLevelSystemCalls* are usually named after the interface and signature they call, i.e., *PurchaseInterface.purchaseItem*. To conclude, our partial impact set after the architectural propagation is $CIS_{U1} = \{purchaseItem.start, PurchaseInterface.purchaseItem\}$. Both actions can be directly mapped to the DFD [233] for further propagation.

Uncertainty **U2** affects the data processing and is modeled as *Behavior* uncertainty. It is annotated to a *SetVariableAction* of the SEFF for processing the purchases of customers in the *Online Shop* component, called *processUserData*. We apply the behavior uncertainty propagation described in Algorithm 6.9. In the case of a *SetVariableAction*, the result of the architectural propagation is the action itself. Thus, the partial impact set is similar to the SIS, i.e., $CIS_{U2} = \{processUserData\}$. This action can directly be mapped to DFDs.

Uncertainty **U3** affects the deployment of the *Database Service* and is modeled as *Component* uncertainty. It is annotated to the *Database Service* component—to be more precise, to the single *AssemblyContext* that represents this component. We apply the component uncertainty propagation described in Algorithm 6.1. The architectural propagation of this uncertainty includes identifying all *StartActions* of all provided SEFFs. As shown in Figure 6.1, this component provides one interface with two operations: *queryInventory* to query the information about available items stored in the database, and *storePurchaseData* to store details about purchases in the database. Similarly to Uncertainty **U1**, we name the start actions of these two SEFFs *queryInventory.start* and *storePurchaseData.start*. Thus, the partial impact set is $CIS_{U3} = \{queryInventory.start, storePurchaseData.start\}$.

Uncertainty **U4** represents the provider's trustworthiness of the *CloudService* and is modeled as *External* uncertainty. It is annotated to the *ResourceContainer* that represents the *CloudService*. We apply the external uncertainty propagation described in Algorithm 6.8. In this propagation algorithm, we first identify all potentially allocated *AssemblyContexts*. In the running example, this is the single *AssemblyContext* representing the *Database Service* component. Afterward, we retrieve all of its actions, as all actions could be potentially affected by the uncertainty. This includes *StartActions*, *ExternalCallActions*, *SetVariable-Actions*, and *StopActions*. This causes the biggest partial impact set, that we briefly show as $CIS_{U4} = \{queryInventory.start, queryData, storeData, \ldots, storePurchaseData.stop\}$. See Appendix B, or the data set [98] for the comprehensive result.

These partial impact sets show the versatility of the impact of uncertainty on confidentiality. They also demonstrate the importance of considering the transitive impact of uncertainty. For instance, Uncertainty **U2** is propagated only to a single action using the architectural uncertainty impact analysis. However, invalid data processing can cause other confidentiality violations later in the data flow, e.g., in the database. Theoretically, this propagation could be realized in architectural models like the PCM. However, the algorithms would be hard to define, complex, and require the handling of many special cases. Thus, we reuse the mapping from the PCM to DFDs to simplify the further propagation.

Another interesting insight is that some of the impact sets share elements, e.g., *query-Inventory.start*. This is not surprising as these elements represent the data flow and can affect confidentiality. Regarding the impact analysis, we can freely join single impact sets for further propagation or continue with the result of a single annotated uncertainty. This decision depends on the software architect's need for a precise or comprehensive impact analysis result. As we do not alter the modeled software architecture but only annotate uncertainty, we can ignore the UIP in this analysis.

Finding: The Uncertainty Interaction Problem (UIP) does not affect architectural uncertainty impact analysis. The propagation results of single uncertainty sources can be merged into joint impact sets. These sets partially represent the impact on confidentiality and require further propagation along the data flow.

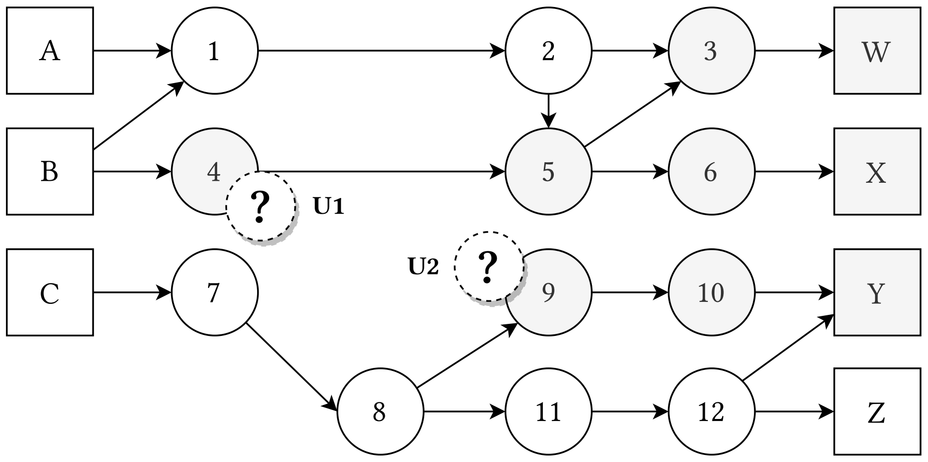


Figure 6.5.: A simple yet versatile Data Flow Diagram (DFD) with sources (A–C), processes (1–12), sinks (W–Z), and data flows. Uncertainty is denoted by question marks (U1, U2) and the impact set is colored gray.

6.4. Uncertainty Impact Analysis for Data Flow Diagrams

In this section, we discuss how uncertainty can be propagated in DFDs to analyze its impact on confidentiality. This represents the second half of our architecture-based uncertainty impact analysis [102] and can be used together with the propagation of uncertainty in architectural models, presented in the previous section. The result of this data flow-based impact analysis is the final *impact set* of uncertainty, an estimation of the actual impact of uncertainty on confidentiality, as discussed in Section 6.2. In the following, we discuss the propagation by referring to DFD nodes. Although we build on the mapping of Seifermann [233] from PCM to DFDs, the impact analysis could also be applied to DFDs that are manually defined or that originate from other ADLs. This addresses Problem **P2**.

The propagation of uncertainty in DFD starts at selected nodes, e.g., the results of the uncertainty impact analysis in architectural models or the results of an expert's manual investigation. Following the terminology introduced in Section 6.3, these nodes form a set similar to the SIS. The central propagation rule of uncertainty in DFD is simple to define[3]: Starting at each impacted node, follow the direction of each outgoing flow until a sink is reached. Add each node on this way to the impact set. We will present the formal foundations and algorithms for this propagation in the following subsections. For now, we stay on this informal level to explain the reasoning behind this propagation rule.

Figure 6.5 shows a simplified DFD with multiple processes, sources, sinks, and flows, demonstrating the versatility of this notation. We use this graph instead of the running example because it contains more special cases, e.g., forking and joining data multiple times. The figure also shows two starting points of the uncertainty propagation, depicted with question marks, named **U1** and **U2**. The annotated nodes represent the starting nodes, i.e., $start_{U1 \cup U2} = \{4, 9\}$. The nodes contained in the impact set are colored gray. The

[3] This central propagation rule might be simple to define, but it was hard to find. It took several student's theses [23, 35, 200, 250] and more than one year of research before a consensus was reached.

informally described rule of uncertainty propagation in DFDs becomes visible here. Every node that can be reached from the starting nodes is contained in the uncertainty impact set. In this simplified example, $impact_{U1} = \{3, 4, 5, 6, W, X\}$ and $impact_{U2} = \{9, 10, Y\}$. Trivially, nodes can also be contained in multiple impact sets if the impact of multiple starting nodes overlaps. Because the UIP does also not affect uncertainty impact analysis in DFDs, we can merge the impact sets to form the full impact set $impact_{U1 \cup U2} = \{3, 4, 5, 6, 9, 10, W, X, Y\}$.

If we interpret uncertainty as unanticipated change, it can only affect confidentiality at every point that is affected by the change. Regarding data flows, this requires data to be processed or forwarded by an affected node. Nodes that have no contact with certain flowing data cannot affect the data's confidentiality, regardless of uncertainty. After a data flow has been impacted by uncertainty, we do not make any further assumptions about confidentiality, as any node could or could not violate confidentiality. In our running example, an uncertain data processing could affect confidentiality directly in the affected processing action, or in a subsequent validation check, or when the data flows to the *Database Service* component, or when the data is stored in the database. This depends on the modeled software architecture and the specified confidentiality requirements. However, we can safely exclude any node that cannot be reached by an impacted node following the flow of data. This includes all previous nodes to the impacted node in the data flow but also all other nodes that cannot be reached. As software systems usually contain many, independent data flows [233], this highly reduces the potential size of the impact set.

Finding: Uncertainty in Data Flow Diagrams (DFDs) propagates along data flows, starting from an impacted node until a sink is reached. Preceding nodes and nodes that cannot be reached following the flow of data are not affected.

6.4.1. Formal Foundation for Uncertainty Impact Analysis

As introduced in Section 2.5 and discussed previously in Section 5.5, DFDs can be represented using DAGs [53]. Regarding uncertainty impact analysis, we do not require the distinction between primary and secondary uncertainty. For impact analysis, we only consider DFD nodes—or DAG vertices—that are either contained or not contained in an impact set. Thus, we do not alter the DAG but only reference a subset of its nodes as an impact set. We explained these sets at the beginning of this section with Figure 6.5.

A DAG $G = (V, E)$ consist of vertices V and edges E. These represent nodes and data flows, respectively. Two vertices $u, v \in V$ are strictly partially ordered $v \prec u$ if there exists a path from v to u, i.e., a data flow. This data flow can also be transitive because a strict partial order is irreflexive, asymmetric, and transitive [141]. For instance, in Figure 6.5, $4 \prec 5$ and also $4 \prec 6$ but $4 \nprec B$ and $4 \nprec 9$ because there is no (transitive) data flow. An induced subgraph $G[V']$ consists of a subset of vertices $V' \subseteq V$ and all edges that have both endpoints in V' [66]. Put simply, in our DAGs, we start with a vertex and add all vertices that can be reached from this vertex and all the required edges to reach these vertices to the subgraph. For instance, in Figure 6.5, $V' = \{12, Y, Z\}$ induces the subgraph $G' =: G[V']$ with vertices V' and edges $\{12 \longrightarrow Y, 12 \longrightarrow Z\} = E' \subset E$.

To propagate uncertainty in DFDs, we start with a vertex $v \in V$ and add it to the impact set $S \subseteq V$. Then, we iteratively add all vertices that are in the direction of the data flow, i.e., all $u \in V$ where $v \prec u$. We add all visited edges to the subset $E' \subseteq E$. The propagation ends once we reached all reachable data sinks, i.e., vertices without outgoing edges. S then induces a subgraph $G[S]$ with all impacted vertices and all directed edges E' between them. This subgraph itself forms a DAG $G' = (S, E')$. Note that we do not limit the size of S. In severe cases and wide uncertainty impacts, $S = V$.

In our current example introduced in Figure 6.5, these induced subgraphs are represented by the nodes that are colored gray. Uncertainty **U1** causes the induced subgraph $G_{U1} =: G[S_{U1}]$ with $S_{U1} = \{3, 4, 5, 6, W, X\}$ and edges $E_{U1} = \{\mathit{4} \rightarrow \mathit{5}, \mathit{5} \rightarrow \mathit{3}, \mathit{5} \rightarrow \mathit{6}, \mathit{3} \rightarrow W, \mathit{6} \rightarrow X\}$. Uncertainty **U2** causes the induced subgraph $G_{U2} =: G[S_{U2}]$ with $S_{U2} = \{9, 10, Y\}$ and edges $E_{U2} = \{\mathit{9} \rightarrow \mathit{10}, \mathit{10} \rightarrow Y\}$. S_{U1} and S_{U2} represent the impact sets presented in the beginning of this section. They also represent the final impact set of uncertainty regarding confidentiality, i.e, the result of our uncertainty impact analysis. Note that impact sets can be a subset of another impact set as well as the induced subgraphs can be subgraphs of other induced subgraphs. Due to our construction of induced subgraphs, both conditions imply each other. For instance, for the a subgraph $G[V'']$ induced by $V'' = \{10, Y\}$ with the edge $\{\mathit{10} \rightarrow Y\}$, we see $V'' \subset S_{U2}$ and $G[V''] \subseteq G_{U2}$.

Finding: The impact of uncertainty in Directed Acyclic Graphs (DAGs) follows the strict partial order of the data flow. Uncertainty impact sets contain the affected vertex and all vertices that can be reached via directed edges. They induce subgraphs of the DAG representing the uncertainty impact.

6.4.2. Algorithm for Uncertainty Impact Analysis in Data Flow Diagrams

Based on the findings presented above, we define an algorithm for uncertainty impact analysis in DFDs. Here, we see the benefits of using simple graphs like DAGs for the propagation. Due to the lack of cycles, we can apply simplified versions of Depth-First Search (DFS) or Breadth-First Search (BFS) [141] without the need for testing for already visited vertices. The resulting sets represent the final impact sets without further steps required regarding the propagation. They can be used to induce subgraphs of the DAG.

Algorithm 6.10 shows the algorithm for uncertainty propagation in DFDs. We first initialize the stack used for the search and the resulting impact set, starting in Line 2. While the stack is not empty, we add the identified vertices to the impact set in Line 6 and continue with all neighbors in the direction of the data flow, i.e., all successors in Line 7. The stack runs empty when no further vertices can be found, i.e., when we have found all reachable data sinks. Last, the resulting impact set is returned in Line 11.

We discussed the one-to-many relation between an uncertainty source and its impact in Section 6.2. While this relation becomes visible already in the architectural propagation, it is common in the data flow-based propagation. If at least one vertex is affected by uncertainty that is not a data sink, the number of impacted elements grows larger than

Algorithm 6.10 Algorithm for uncertainty propagation in data flow diagrams

```
1:  procedure PROPAGATEUNCERTAINTYINDFD($start$, $graph$)
2:      $stack \leftarrow [start]$                              ▷ Initialize stack with the start vertex
3:      $impactset \leftarrow \emptyset$
4:      while NOTEMPTY($stack$) do
5:          $vertex \leftarrow$ POP($stack$)
6:          $impacset \leftarrow impactset \cup \{vertex\}$                 ▷ Fill the impact set
7:          for $successor \in$ GETSUCCESSORS($vertex$, $graph$) do
8:              PUSH($stack$, $successor$)
9:          end for
10:     end while
11:     return $impactset$
12: end procedure
```

the number of annotated elements. Based on this graph-oriented point of view, we can see again the need for uncertainty impact analysis to understand the potential impact of uncertainty on confidentiality.

6.5. Uncertainty Impact Analysis regarding Confidentiality

We combine the propagation of uncertainty in architectural models with the propagation of uncertainty in DFDs to complete our uncertainty impact analysis regarding confidentiality. On the one hand, the architectural propagation uses modeled information to retrieve the uncertainty impact that is missing in DFDs. On the other hand, the architectural propagation, which is based on change impact analysis [46, 211], requires handling special cases, has more complex propagation rules, and is specific to concrete ADLs like the PCM. Uncertainty propagation in DFDs is ADL-independent and can easily be defined based on DFS or BFS. Thus, we connect both impact analyses to combine their benefits. This concludes our approach to address Problem **P2**.

To achieve this, we reuse the mapping of PCM to DFDs, defined by Seifermann [233]. We first calculate the partial impact set of the architectural uncertainty impact analysis. This impact set consists of different elements of the PCM, namely *StartActions*, *StopActions*, *SetVariableAction*, and *ExternalCallActions* that are described in SEFFs and also *EntryLevelSystemCalls* that are specified in the *UsageScenario*. All of these actions have counterparts in the extracted DFDs and there is a one-to-one mapping between these actions and their corresponding DFD nodes. This satisfies the relation presented in Figure 6.2 as every DFD node is a unique origin in the architectural model.

6.5.1. Coupling the Uncertainty Impact Analysis Approaches

We extend the notation introduced in Section 6.4 to present the coupled analyses. We use DAGs $G = (V, E)$ to represent DFDs with the strict partial order $u, v \in V, u \prec v$ representing data flows. Let $A = \{a_1, \ldots, a_n\}$ be the set of all architectural elements like components, or interfaces and let $S = \{s_1, \ldots, s_n\}$ be the set of all uncertainty sources. We name the annotation of an uncertainty source to an architectural element $a : S \rightarrow A$. For instance, in our running example, $\{\mathit{PurchaseInterface}, \mathit{processUserData}\} \subset A$, and $S = \{\mathit{U1}, \mathit{U2}, \mathit{U3}, \mathit{U4}\}$. Then, we can specify the annotations as $a(\mathit{U1}) = PurchaseInterface$, or $a(\mathit{U4}) = CloudService$, as shown in Table 6.2. We reuse the mapping [233] from an architectural element to its corresponding vertices of the DFD as $m : A \rightarrow V$.

The uncertainty impact analysis can be defined as function $u : S \rightarrow X \subseteq V$, where S represents all uncertainty sources. X induces a subgraph $G[X]$ of the DFD, i.e., the part of the software system that is affected by the annotated uncertainty sources. The analysis consists of three steps: First, we conduct the architectural propagation of the uncertainty based on the by adapting the propagation rules defined by change impact analysis. For example, altering an interface does affect both its caller and the callee. We define the architectural propagation as $p_A : A \rightarrow A$. Second, we apply the previously defined mapping $m : A \rightarrow V$ from all affected architectural elements to their corresponding vertices of the DFD. Third, we define the propagation along the data flow as $p_D : V \rightarrow X \subseteq V$. The previously mapped vertices represent the first affected nodes in the direction of the data flow, so that $\forall x \in X \subseteq V, \exists a \in A : m(a) = x \vee m(a) \prec x$. The induced subgraph $G[X]$ represents the full impact set including transitive effects. In sum, we define the uncertainty impact analysis as $u = p_D \circ m \circ p_A \circ a$.

Put simply, starting with the annotation function a, we receive all annotated elements of the software architecture, e.g., the *PurchaseInterface*. The architectural propagation p_a yields all impacted elements of the software architecture, e.g., the *EntryLevelSystemCall* to purchase items. We apply the mapping m to find the DFD node corresponding to this *EntryLevelSystemCall*. Then, we use the data flow-based propagation p_D to identify all nodes where data from this node flows. Speaking in terms of DAGs, these represent a subset of all vertices $X \subseteq V$ that induce a subgraph of the DAG. This represents the final result of the uncertainty impact analysis, showing the potential impact of uncertainty on the software system regarding confidentiality.

6.5.2. Algorithm for the Coupled Uncertainty Impact Analysis

This foundation enables the combination of uncertainty impact analysis in architectural models and DFDs. Algorithm 6.11 shows the resulting propagation algorithm for analyzing the impact of uncertainty on confidentiality in PCM models. This algorithm combines all algorithms that have been introduced in Section 6.3 and Section 6.4 and realizes the aforementioned coupling of the uncertainty impact analysis approaches.

Algorithm 6.11 Algorithm for uncertainty impact analysis regarding confidentiality

```
procedure propagateUncertainty(uncertainty, model)
    candidates ← $\emptyset$
    switch typeOf(uncertainty) do          ▷ Propagation in the architectural model $p_A$
        case ComponentUncertainty
            candidates ← propagateComponentUncertainty(uncertainty, model)
        case InterfaceUncertainty
            candidates ← propagateInterfaceUncertainty(uncertainty, model)
        case ConnectorUncertainty
            candidates ← propagateConnectorUncertainty(uncertainty, model)
        case ExternalUncertainty
            candidates ← propagateExternalUncertainty(uncertainty, model)
        case BehaviorUncertainty
            candidates ← propagateBehaviorUncertainty(uncertainty, model)
    impactset ← $\emptyset$                    ▷ Mapping to the data flow diagram $m$
    graph ← mapToDataFlowDiagram(model)
    candidates ← mapToVertices(candidates, model, graph)
    for vertex ∈ candidates do             ▷ Propagation in the data flow diagram $p_D$
        impactset ← impactset ∪ propagateUncertaintyInDFD(vertex, graph)
    end for
    return impactset
end procedure
```

We start with an empty set of candidates in Line 2. Depending on the type of the annotated uncertainty source, we choose the matching algorithm for the architectural propagation in Line 3. This represents the function p_A. The result of this propagation is one or more elements from the architectural model, i.e., PCM elements. Which and how many candidate elements are identified in the architectural propagation depends on the architectural model and the type of uncertainty. For instance, *Behavior* uncertainty yields one element in most of the cases, as specified in Algorithm 6.9. On the contrary, *Interface* uncertainty always yields multiple candidate elements, as described in Algorithm 6.6.

Afterward, we map the architectural model to a DFD and all candidate elements to their corresponding vertices, starting in Line 15. We do not elaborate the functions *mapToDataFlowDiagram* and *mapToVertices* but refer to the definition of this mapping [233, 234] and to the examples provided in Figure 3.2 and Figure 6.1. The result of this mapping in Line 16 is a set of vertices representing the starting points for the uncertainty impact analysis in DFDs. This represents the function m. We propagate the uncertainties in the DFDs, starting in Line 17. We have to repeat this step for every candidate vertex as there can be multiple candidates. The impact set is the union of all individual propagation results. This represents the function p_D. The final impact set is returned in Line 20.

	Annotated element	Affected PCM elements, Mapped DFD nodes	Uncertainty Impact set
U1	Purchase-Interface	PurchaseInterface.purchaseItem, purchaseItem.start	PurchaseInterface.purchaseItem, purchaseItem.start, processPurchase, processUserData, …
U2	process-UserData	processUserData	processUserData, DBService.storePurchaseData, storePurchaseData.start, storeData, …
U3	Database-Service	queryInventory.start, storePurchaseData.start	queryInventory.start, queryData, Database, returnData, …
U4	Cloud-Service	queryInventory.start, queryData, storeData, …	queryInventory.start, queryData, Database, returnData, …

Table 6.3.: Shortened results of all steps of the uncertainty impact analysis regarding confidentiality.

6.5.3. Applying Uncertainty Impact Analysis Regarding Confidentiality

To exemplify the full uncertainty impact analysis, we apply it to the running example introduced in Chapter 3. This example contains four uncertainty sources that demonstrate different uncertainty types, namely *Connector* uncertainty (**U1**), *Behavior* uncertainty (**U2**), *Component* uncertainty (**U3**), and *External* uncertainty (**U4**). In the first step, software architects annotate the model with these uncertainty sources, as described in Figure 4.2. This represents the annotation function a. We showed exemplary annotation targets, i.e., elements from the PCM model in Table 6.2 and repeat them in Table 6.3 for convenience.

Afterward, Algorithm 6.11 propagates the annotated uncertainty sources within the architectural model, using the propagation algorithm that fits the annotated uncertainty. This represents the architectural uncertainty impact analysis p_A. We showed exemplary propagation results, i.e., elements of the software architecture that could be affected by the annotated uncertainty in Section 6.3 and summarize them in Table 6.3.

The affected elements of the software architecture are mapped to corresponding DFD nodes. To achieve this, we use the tracing information from the extraction of DFDs based

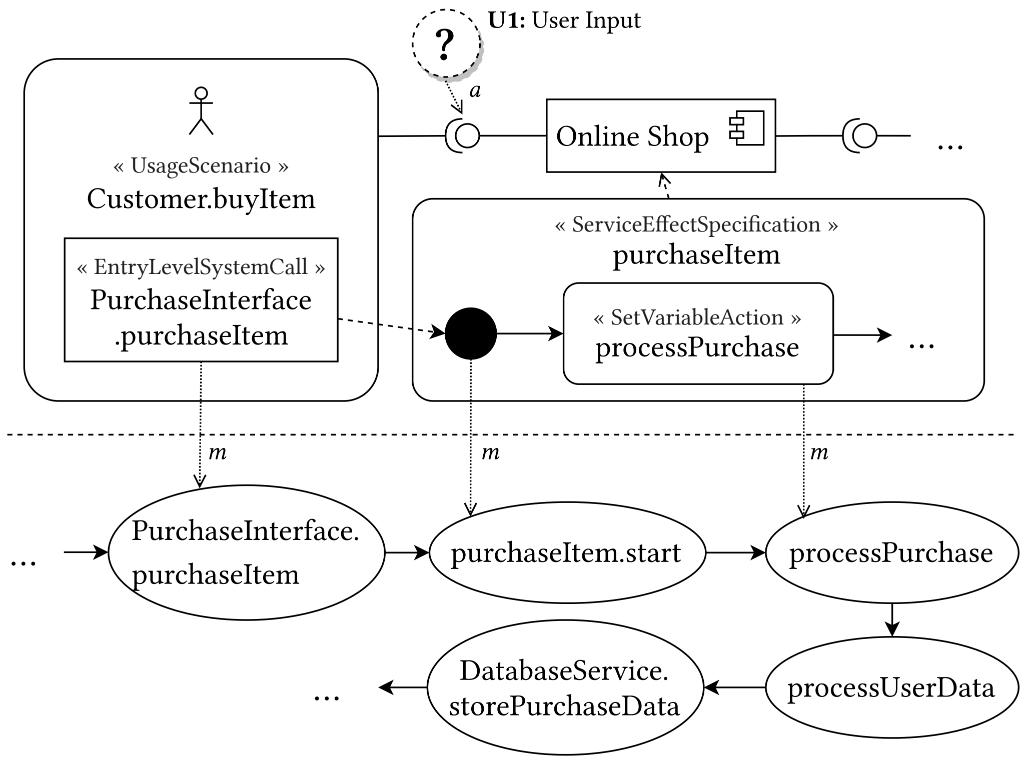


Figure 6.6.: Informal illustration of the propagation of an uncertainty source in the architectural model (top) and a Data Flow Diagram (DFD) (bottom) using the annotation function a and the mapping function m.

on the mapping of Seifermann [233]. Every DFD node has an origin within the PCM model, see Figure 6.2. This represents the mapping function *m*. We show the results of this mapping in Table 6.3. Note that the nodes' names are identical to elements of the software architecture. This is not by accident but to simplify the later identification of uncertainty-afflicted elements and confidentiality violations.

Last, we propagate the uncertainty within in the DFD, starting at each vertex retrieved from the mapping of affected architecture elements, as shown in Algorithm 6.11. This represents the uncertainty impact analysis in DFDs p_D. The result of this analysis step is the final uncertainty impact set that is shown to the software architect. We shorten the results of this step in Table 6.3. The full impact set can be found in Appendix B.

To illustrate the propagation path of an uncertainty source, Figure 6.6 informally shows the propagation of Uncertainty **U1**. We chose this uncertainty source as it is annotated to the *Customer* of the *Online Shop* and thus has the longest path until it reaches a data sink. For the sake of brevity, we leave out many details and focus only on the propagation path. The full example can be found in our data set [98], and in Appendix B. Uncertainty **U1** represents the user input and is annotated to a *ProvidedDelegationConnector* called *PurchaseInterface*. It propagates to an *EntryLevelSystemCall* in the *UsageScenario* named *Customer.buyItem* and to the *StartAction* of a SEFF in the architectural model. The actions are mapped to DFD nodes. Afterward, the uncertainty is propagated in the DFD until a sink is reached. All DFD nodes are part of the impact set of Uncertainty **U1**.

```
Behavior Uncertainty Impact on SEFFActionSequenceElement with
   ID _oEBNYDIXEe-m4c0ChzWfPg (represeting a SetVariableAction).
Origin of this impact: Behavior Uncertainty annotated to
   SetVariableAction "UserDataProcessing" (_oEBNYDIXEe-m4c0ChzWfPg).

All affected elements (1):
SEFFActionSequenceElement (UserDataProcessing, _oEBNYDIXEe-m4c0ChzWfPg)

Impact set (1):
0: SEFFActionSequenceElement (UserDataProcessing)
CallingSEFFActionSequenceElement / calling (DatabaseStoreInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Ending updateInventory)
CallingSEFFActionSequenceElement / returning (DatabaseStoreInventory)
...
```

Listing 6.1: Shortened output of the uncertainty impact analysis showing the impact of Uncertainty U2.

6.5.4. Tool Support for Uncertainty Impact Analysis

In the introduction of this chapter, we discussed the need for automated analyses to aid the software architectural design process. Manual propagation of uncertainty is not feasible, especially in large systems of systems with hundreds of architectural elements or DFD nodes [102]. This need for automated analysis is also described in Problem **P2**.

We realize the algorithms described in this chapter as part of our tooling UIA. Here, we build on the Palladio tooling [205, 207] that offers a meta model, the PCM, and graphical editor support. Our Java-based open-source implementation of the propagation logic is integrated into the Eclipse-based tooling, and part of the data set [98].

Here, we included many of the already discussed optimization strategies to speed up the propagation, e.g., by caching query results or using the extracted DFD to filter affected architecture elements. Using this approach, all propagation algorithms can be realized as DFS, which is in $\mathcal{O}(V + E)$, where V represents the DFD nodes and E represents the edges [141]. Put simply, we test for each node whether or not it is affected by uncertainty, and then propagate the uncertainty along the data flow.

Software architects annotate uncertainty sources by calling the analysis interface and providing the element's identifier. The analysis performs a type check to ensure the validity of the annotation and then propagates the uncertainty without requiring any further user interaction. The result comprises the annotated elements, the affected elements in the PCM model, and the complete impact set. Listing 6.1 shows an exemplary but shortened output of UIA for Uncertainty **U2** with uncertainty type, annotation target, and analysis results. The full output can be found in Appendix B. This output can be used to further enhance the display of the impact, e.g., by highlighting affected areas of DFDs in graphical viewers or editors [36].

We also implemented additional processing to enhance the user experience. As discussed in Section 6.4, impact sets of uncertainty sources can be a subset of impact sets of other uncertainty sources. In our running example, the impact set of **U2** is a subset of the impact set of **U1** because it affects a subgraph of the DAG representing the user input's processing. We identify such relations and only return the largest impact sets that are not subsets of another subset. Formally speaking, the analysis yields an antichain [66] with the partial order of the subset relation, in which all impact sets are pairwise incomparable.

Finding: The uncertainty impact analysis can be fully automated based on the Palladio Component Model (PCM) and the existing tooling of the Palladio approach. Propagating uncertainty requires minimal additional effort by software architects, as the analysis only requires annotating uncertainty sources.

6.6. Addressing the Uncertainty Awareness Problem

Our approach to uncertainty impact analysis supports software architects in the early assessment of the impact of uncertainty on confidentiality. However, it suffers from the Uncertainty Awareness Problem (UAP), like many other uncertainty-aware analyses [115]. As introduced in Section 5.6, the UAP hinders the analysis of uncertainty due to the lack of knowledge about relevant uncertainty sources. Failing the correct identification, classification, and annotation limits the validity of analysis results, as they can lack both precision and comprehensiveness. In this section, we demonstrate how to tackle this problem regarding our uncertainty impact analysis. We connect ARC^{3}N, which was introduced in Section 5.7, to our impact analysis tooling UIA. This addresses Problem **P3** and is a step toward end-to-end analysis approaches [273].

To tackle the UAP in uncertainty impact analysis, we need to solve two underlying problems. First, the analysis and its tooling must receive information about relevant uncertainty sources and their classification. Second, software architects operating the analysis must become aware of potential sources and identify appropriate annotation targets. Both problems arise before the automated analysis is performed. Thus, we extend the uncertainty impact analysis by an additional identification step prior to the propagation.

After our tooling UIA has been initialized and all PCM models are loaded, we request a current list of uncertainty sources from ARC^{3}N. This is possible because ARC^{3}N provides the uncertainty source catalog in a machine-readable format, as presented in Section 5.7. This catalog also includes the classification of each source, as shown with the underlying meta model in Figure 5.3. Software architects use ARC^{3}N to identify relevant uncertainty sources. As all uncertainty sources in the uncertainty source catalog have a unique identifier, they can provide this identifier to the extended impact analysis. After resolving the uncertainty source and the classification category *Architectural Element Type*, the analysis queries all PCM models to identify matching annotations targets. These are displayed to the software architects in an interactive interface.

```
Enter an id of an uncertainty to check for:
> #48

Select one of these elements.
1) "OnPremiseServer" (_qvz80ITgEeywmO_IpTxeAg)
2) "CloudServer" (_upfkIITgEeywmO_IpTxeAg)

Enter line number:
> 2

Analysis completed. Result:
...
```

Listing 6.2: Exemplary interaction of a software architect with the extended uncertainty impact analysis.

Section 6.6 shows an exemplary interaction prior to the propagation of uncertainty. In the running example, we have two *ResourceContainers* that can be affected by uncertainty about the provider's trustworthiness (**U4**), represented as uncertainty number 48 in ARC^{3}N. After annotating this uncertainty source to the *CloudServer*, the uncertainty impact analysis is executed. The full result, including the impact set, is returned, similar to Listing 6.1.

This addresses the UAP by providing software architects with a list of possible uncertainty sources to choose from. Additionally, less expert knowledge about the classification is required as the interpretation is done automatically. Last, the manual effort of searching the architectural model to identify potential elements to annotate with uncertainty is reduced. In sum, this demonstrates the feasibility of extending an existing analysis, even without altering the core propagation algorithms. We stress that other uncertainty-aware analyses [101, 265, 266] could similarly benefit from such extension.

Finding: By connecting the identification and classification of uncertainty to model-based uncertainty analysis, the Uncertainty Awareness Problem (UAP) can be addressed. Although this does not fully resolve the challenge of unanticipated change, it reduces the severity. Approaches like this represent a step toward the end-to-end analysis of uncertainty.

6.7. Uncertainty Propagation in Uncertainty Flow Diagrams

We conclude this chapter by discussing the generalization of the presented concepts to tackle related problems from the community of SASs [273]. In the recent past, the Uncertainty Interaction Problem (UIP) has received increased attention [50, 52, 273]. This problem arises because uncertainties in software-intensive systems are rarely independent [52]. Addressing this problem goes beyond handling single uncertainty sources but poses additional challenges regarding modeling, analysis, and mitigation [50]. Handling the UIP requires notations to represent and analyze the interaction of *heterogeneous* uncertainty sources, i.e., uncertainty that differs in type and nature. Additionally, this notation must support horizontal propagation in the same level of abstraction, and vertical propagation across abstraction levels. This is also summarized in Problem **P4**.

Uncertainty propagation has been identified early as a possible approach towards handling uncertainty interactions [50]. However, the few existing approaches tackling uncertainty propagation focus on *homogeneous* uncertainties, i.e., uncertainties that are similar in nature, admit the same representations and are amenable to similar reasoning mechanisms [49]. In the following, we build on the knowledge about uncertainty propagation in DFDs, previously presented in this chapter. We define a new notation called Uncertainty Flow Diagram (UFD) that enables the representation and propagation of heterogeneous uncertainty sources. Thereby, we generalize the propagation concept introduced in this chapter and make a first step towards comprehensive modeling and analysis of the UIP[4].

In our running example, we see an uncertainty interaction between Uncertainty **U3** and **U4**. We discussed this interaction when we introduced the difference between primary and secondary uncertainty in DAGs in Section 5.5. If Uncertainty **U3** resolves to a deployment location that is not in the cloud, Uncertainty **U4** about the cloud provider has no effect. Otherwise, the combination of both uncertainty sources is relevant. This represents a simple uncertainty interaction where the outcome can be understood by investigating all possible scenarios. However, other interactions could lead to results that go further and introduce new and unanticipated changes [49, 50]. An example is Znn.com [57], where uncertainty about the sensor input interacts with the discretization of the input. Here, the interaction of both uncertainties can lead to false adaption, although both uncertainty sources on their own would not have affected the system behavior [49].

Figure 6.7 shows the simplified UFD meta model [49]. This represents a simplified version of DFDs [64] or UML activity diagrams [186], extended with information about uncertainty, based on PSUM [184]. The top-level element is the *Activity*, i.e., a graph whose nodes and edges are *ActivityNodes* and *ControlFlows*, respectively. The graph represents how the information flows through the computations performed by a program. *Actions* represent behavior. *Actions* are represented by squares with rounded corners and have *Pins* (small white squares) that represent the types of input/output parameters. Each *Pin* has a *Type*. *Actions* may have an associated *Behavior*, such as the invocation of a method that implements the behavior to transform the action's inputs into outputs. This approach is similar to the unified modeling primitives [235].

To enable hierarchical modeling and vertical uncertainty propagation, each *Action* can also be refined by one or multiple *Activities* that represent its inner workings. At the highest abstraction level, the whole system can be represented by a single node with input and output pins, i.e., a black box. Either a *Behavior* or a set of refining internal *Activities* can be specified for an *Action*, but not both. It is also possible to specify multiple internal

[4] When I presented the research about uncertainty propagation regarding confidentiality at SEAMS '23, I was asked by Danny Weyns about the application of these concepts to SASs. My answer comprised the application of models at runtime and the integration into the analysis phase of MAPE-K. However, in hindsight, Uncertainty Flow Diagrams (UFDs) would have been the perfect answer.

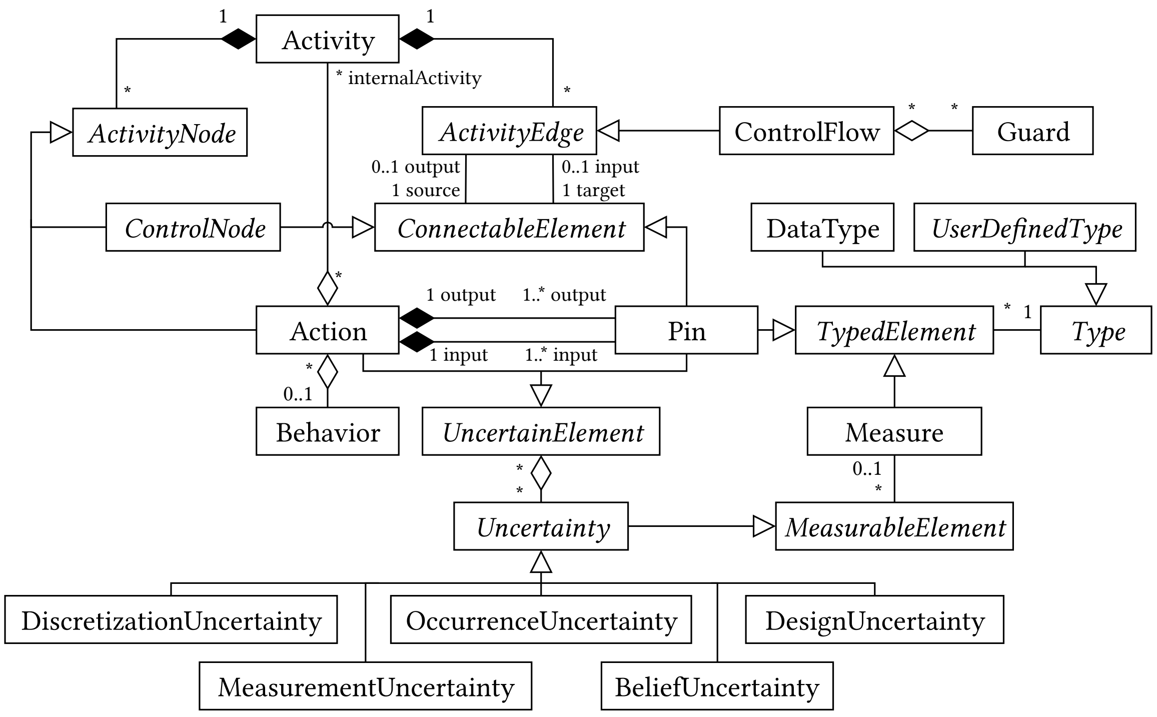


Figure 6.7.: Simplified meta model of the Uncertainty Flow Diagram (UFD) notation.

activities that represent alternatives. This enables the expression of structural uncertainty as variations, which is common to represent design uncertainty [255].

A *ControlFlow* is represented by a directed arrow that connects the outgoing *Pin* of an *Activity* with the incoming *Pin* of another *Activity*. *ControlFlows* may include *Guards* that have to be satisfied for the information to flow between the connected *Pins*, or *ControlNodes*. When more than one *Guard* is specified, they all need to be satisfied for the *ControlFlow* to take place. UFDs enable the explicit representation of uncertainty, with the goal of dealing with the interactions between uncertainties that happen when making computations. UFDs allow the specification of individual uncertainties associated with *Pins* or with *Actions*. Each uncertainty can be of a different type. This includes the common uncertainty types *Measurement Uncertainty*, *Discretization Uncertainty*, *Occurrence Uncertainty*, *Design Uncertainty*, and also *Belief Uncertainty* [255]. Note that these types differ from the uncertainty types when only focusing on confidentiality. Each uncertainty can have an associated *Measure*, which also has a *Type*. One way to assess a *Measurement Uncertainty* is in terms of the accuracy of the measurement, which is normally expressed by means of a real number that represents the possible variation of the nominal value of the parameter, i.e., its estimated standard deviation. *Belief Uncertainty* is normally expressed by a real number between 0 and 1 representing the likelihood that the stated fact is true, expressed as a probability. Alternatively, uncertainty can be expressed by defining multiple *Internal Activities* as a variation of the behavior of a single *Action* which is used to express *Design Uncertainty*. This also enables the expression of additional uncertainty types if they can be denoted either quantitatively using an associated *Measure* or structurally using alternative internal *Activities*.

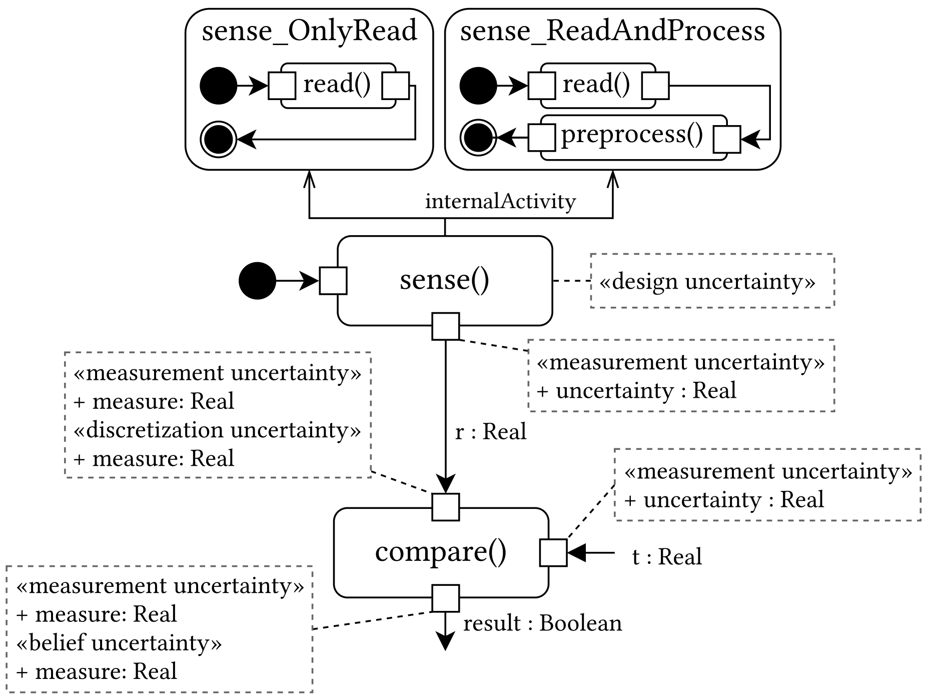


Figure 6.8.: An Uncertainty Flow Diagram (UFD) showing heterogeneous uncertainty sources and hierarchical modeling.

Figure 6.8 shows an exemplary UFD. The *compare Action* compares a sensor value *r* to a threshold *t* to decide whether or not the adaption is triggered. As described previously, both values are affected by *Measurement Uncertainty*. The input *r* is additionally affected by *Discretization Uncertainty*, and the uncertain correctness of the *result* is thus subject to *Belief Uncertainty*. These uncertainties are attached to the matching *Pins* and can be expressed using *Measures*, e.g., the degree of belief, or the variation of the measurement [184]. Additionally, this example shows *Design Uncertainty* in the *sense Action*. Here, two versions are possible, depending on whether additional preprocessing is used. This can affect the *Measurement Uncertainty* in *r* and is represented using *Internal Activities*.

In sum, this notation enables the expression of heterogeneous uncertainty sources and their propagation, both horizontally and vertically. Thereby, UFDs are still close to the unified modeling primitives [235] or DFDs[5]. The propagation of uncertainty in UFDs is similar to the propagation in DFDs, i.e., following all edges of a node until a sink is reached. Although UFDs are a promising approach to generalize the propagation concept to analyze more than confidentiality, this still represents an early proposal [273]. Future research in this direction is required to create comprehensive and tool-supported modeling and analysis approaches to address the UIP.

[5] This research started at the 2023 Bertinoro research seminar on uncertainty in SASs. Our group consisted of Javier Camara, Diego Perez-Palacin, Antonio Vallecillo, Maribel Acosta, Nelly Bencomo, Radu Calinescu, Simos Gerasimou, and myself. After discussing the Uncertainty Interaction Problem (UIP) for days, we were surprised how concise Uncertainty Flow Diagrams (UFDs) can be.

6.8. Assumptions and Limitations

In this section, we discuss the assumptions and limitations of the different approaches to uncertainty propagation and impact analysis presented in this chapter. We also provide our reasoning on whether limitations can become critical when applying the approaches.

Using data flow diagrams We use DFDs to investigate confidentiality as proposed by other work [226, 236, 241]. Although this simplifies the propagation algorithms and provides a widely used and well-known foundation for our work, it limits the generalizability. The uncertainty propagation in DFDs is only defined for this notation and the architectural propagation yields elements that can be represented by DFD elements. This limitation is similar to the limitation of focusing on confidentiality, discussed in Section 5.8. However, we argue that our findings about the propagation of uncertainty are at least partially generalizable, as shown with the definition of UFDs in Section 6.7.

Choice of architectural description language We choose the ADL PCM to define our algorithms for the architectural uncertainty impact analysis. The PCM is well known [243] and easy to extend. Nevertheless, this limits the generalizability to other descriptions of software architecture like UML diagrams [186]. We argue that the foundation of the propagation, our classification defined in Chapter 5, is general enough to also define propagation rules for other ADLs. Additionally, the architectural impact analysis is limited by the specified annotations targets of uncertainty sources. Here, the same argument can be made, that extending the analysis to other architectural elements based on the provided propagation algorithms is possible and should require reasonable effort.

Correctness of propagation rules and of the mapping We assume the correctness of the propagation rules for architecture-based change impact analysis from the KAMP approach [46, 210, 211]. We also assume the appropriateness of the mapping from the PCM to DFDs [233, 234, 236]. Erroneous propagation or mapping rules could decrease the validity of the results of our uncertainty impact analysis. However, both represent well-validated and well-published research approaches.

Overestimation of the impact set Our uncertainty impact analysis overestimates the uncertainty impact set, as discussed in Section 6.3. This is intentional as we argue that a high recall is more important than a high precision in the early assessment of uncertainty [102]. However, the usefulness of the impact set depends on the analyzed architectural model. Architectural models with only a few independent data flows are more likely to yield imprecise results, as the majority of the software system would be affected by the uncertainty sources. We argue that realistic software systems comprise enough independent data flows [227] to apply the analysis with satisfying results.

Propagating uncertainty until a sink is reached The uncertainty impact analysis in DFDs propagates uncertainty starting at a specified node by following all outgoing data flows until a sink is reached. Similarly to the architectural uncertainty impact analysis, this overestimates the impact of uncertainty. For instance, if at one point in the DFD, every data flow is encrypted, this limits the impact of previous uncertainties about the encryption. This is comparable to approaches to reduce uncertainty during operation, e.g., robots that use visual markers to reduce the uncertainty about their location [51]. However, we ignore such uncertainty reduction and propagate uncertainty until the end of each data flow. Considering such interdependence of uncertainty and confidentiality would increase the precision but also require to already consider confidentiality requirements in the uncertainty propagation, which would increase the manual modeling effort. Nevertheless, this represents an interesting direction for future research.

Propagating unknown uncertainty sources Similarly to the limitation of classifying unknown uncertainty sources discussed in Section 5.8, we can also not propagate them. Software architects have to be at least aware of an uncertainty source in order to annotate and to propagate the uncertainty. The connection of our identification approach, presented in Section 5.6, to our uncertainty impact analysis can partially address this limitation. Providing software architects with an uncertainty source catalog that can be propagated reduces the impact of the unknown. Nevertheless, we cannot assume that all required information is available as the underlying challenge of unanticipated change remains.

Querying annotation targets By connecting our identification approach, presented in Section 5.6, to our uncertainty impact analysis, we simplify the annotation of uncertainty sources. However, the querying of elements of the software architecture that match a selected uncertainty source only considers the category *Architectural Element Type*. Including further categories like the *Location* could enhance the precision of annotations targets recommended to the software architect, but could also cause false negatives. Further research is required to enhance the recommendation and explanation of uncertainty [26]. We argue that our initial filter already greatly reduces the number of matching uncertainty sources and is thus helpful.

Solving the uncertainty interaction problem UFDs represent a notation to express and propagate different uncertainty types to tackle the UIP. Nevertheless, this does not fully solve the challenge of uncertainty interactions [50]. Challenges regarding tool-supported modeling, the transformation between different notions, and automated analysis remain. However, UFDs represent a first step for further research in this direction [49, 273].

6.9. Summary and Outlook

In this chapter, we presented different approaches to uncertainty propagation to define an *uncertainty impact analysis* regarding confidentiality. This represents our second Contribution **C2** and provides an answer to ❓ **Research Question 2**.

First, we discussed the representation of uncertainty in architectural models to address Problem **P1**. We presented a distinction between uncertainty *sources* and their *impact* and related this terminology to architectural models and DFDs. We introduced a meta model of DFDs under uncertainty that relates the different uncertainty impact types to DFD elements. Last, we discussed where uncertainty sources can be annotated within the PCM to enable architectural uncertainty propagation.

To address Problem **P2**, we combined uncertainty propagation in architectural models with propagation along the data flow in DFDs. First, we discussed the relation of architecture-based change impact analysis [46] and uncertainty impact analysis. Afterward, we presented propagation algorithms for each uncertainty type described in our classification, introduced in Chapter 5. We described how to map the propagation results to DFDs and to further propagate uncertainty following the flow of data. Hereby, we also provided a formal foundation based on DAGs and induced subgraphs that represent the impact of uncertainty. We explained this impact analysis using our running example and introduced our tooling 🔧 UIA to automate uncertainty impact analysis.

Problem **P3** describes the UAP regarding uncertainty impact analysis, i.e., the lack of knowledge about uncertainty sources to propagate. We addressed this problem by combining the identification approach, introduced in Section 5.6, with the uncertainty impact analysis. Regarding the tool support, this meant combining 🔧 ARC^{3}N with 🔧 UIA. The resulting analysis simplifies both the identification of relevant uncertainty sources and the identification of annotation targets.

Last, we generalized the findings on the propagation of uncertainty in DFDs to better understand the UIP, as discussed with Problem **P4**. To analyze uncertainty interactions, we defined the notation of UFDs that can express heterogeneous uncertainty sources and support both horizontal and vertical propagation of uncertainty. This notation provides a foundation for further modeling and analysis approaches to tackle the UIP.

❓ **Research Question 2** asked about the propagation of previously classified uncertainty sources to predict their impact on confidentiality. We observe that a comprehensive approach requires both the propagation in architectural models and DFDs. Our answer thus comprises means for modeling uncertainty in both representations and propagation algorithms for architectural models based on the PCM and also DFDs. To connect the resulting uncertainty impact analysis regarding confidentiality to classified uncertainty sources, we discussed the connection of this approach to uncertainty source catalogs. In sum, this enables software architects to assess the impact of uncertainty in existing architectural models with regard to confidentiality.

The central benefit of early impact analysis is cost reduction due to the early detection of potential problems [32]. Here, uncertainty impact analysis can be applied even earlier than design time confidentiality analysis [236], because propagating uncertainty does not require details about data processing or confidentiality requirements. Nevertheless, propagating uncertainty helps software architects in mitigating uncertainty early [2]. Architecture models can be annotated with uncertainty sources from existing collections [104] which helps in the documentation and to raise awareness. The analysis helps predict and mitigate confidentiality violations. Using a confidentiality analysis for this purpose would require software architects to understand and model the impact of uncertainty manually which requires more effort and expertise. Note that we do not state that uncertainty impact analysis replaces the need for confidentiality analysis. However, it can be used for an initial, coarse-grained assessment, as discussed in our procedure, introduced in Section 4.1. Last, the calculated models of our analysis can also be used for regression testing or to handle uncertainty at runtime [65].

To conclude, we want to briefly discuss another application of uncertainty propagation. We investigated uncertainty in coupled models [2], e.g., in the automotive domain, where experts from different areas work together. They require different views, e.g., a system architecture, an electrical topology with hardware components, or simulation results. To keep these views consistent, the underlying models are coupled using consistency specifications [139]. Based on the classification of uncertainty in Cyber-Physical System (CPS), we derived the category *Locus* that is similar to *Architectural Element Type*. This category also defines where uncertainty can be annotated and how it propagates through the models. Although this is based on consistency specifications and not architectural change impact analysis, uncertainty propagation can be applied and helps in mitigation.

ℹ **Finding:** Interpreting uncertainty as unanticipated change enables its propagation not only based on change impact analysis, but also using other methods of change propagation, such as consistency specifications. Uncertainty propagation is a powerful technique to better understand the effect of unanticipated changes, or even to detect new uncertainty, e.g., due to uncertainty interactions.

We will revisit uncertainty propagation and interaction in ❯ **Chapter 7: Uncertainty-Aware Data Flow Analysis to Identify Confidentiality Violations**. There, we present different approaches to confidentiality analysis under uncertainty that build on the findings of this chapter. Especially the concept of uncertainty interactions helps in optimizing the uncertainty-aware analysis of confidentiality. Both the uncertainty impact analysis and confidentiality analysis under uncertainty are based on our uncertainty classification, presented in ❯ **Chapter 5: Identification and Classification of Uncertainty Regarding Confidentiality**. An overview of the procedure that connects all of these steps is given in ❯ **Chapter 4: Overview**. Last, ❯ **Chapter 9: Evaluation** presents the evaluation of all contributions, including the uncertainty impact analysis presented in this chapter.

6.10. In Simpler Words

Uncertainty has many definitions. Some define uncertainty as a lack of knowledge about a software system and its environment, while others refer to uncertainty as unanticipated change, i.e., changes in a software system that have not been foreseen. Our goal in this chapter is to define an *uncertainty impact analysis*. When experts become aware of an uncertainty source, they want to better understand its potential impact. Not all uncertainty sources have severely bad effects, many do not even affect confidentiality. Our classification, defined in Chapter 5, helps in an early inspection of an uncertainty source. However, to really understand the impact, they need to look at the software system, and analyze its software architecture and data flows. Our contribution in this chapter supports the experts in this activity and even automates the majority of it.

We call the core idea of uncertainty impact analysis *uncertainty propagation*—both are so closely related that we even use them synonymously. Propagation refers to the uncertainty being forwarded or transmitted through a software system. For instance, if the data processing of a system part is uncertain, this may also affect other system parts where the data flows. The uncertainty of the data processing propagates through the software system. Here, we build on the finding that uncertainty can be understood as unanticipated change, as described above. Other researchers defined architecture-based change impact analysis that propagates change through software systems. For instance, if we rename a signature in an interface, we must also rename the calls to this signature in all methods. This propagation, from a changed element to other elements that also have to adapt, is called change propagation, and the result of such propagation is called *impact set*. Change impact analysis can estimate the impact set—and we build on this to estimate the impact set of uncertainty.

We define many algorithms to propagate uncertainty within the software architecture. These algorithms precisely describe which elements of a software architecture can be indirectly affected by an uncertainty source. Here, we build on the classification category *Architectural Element Type*, presented in Chapter 5. Afterward, we move from the software architecture to Data Flow Diagrams (DFDs) and propagate the impact of uncertainty along all potentially affected data flows. This helps us avoid overlooking potential confidentiality violations due to impacted data flowing to another part of the software system. Remember the example of the data processing that could cause problems somewhere else. We express the final impact set using the DFD of the software system.

Building on this analysis, we describe several extensions. For example, we relate the previously discussed awareness problem of uncertainty to the analysis. Although our impact analysis helps to better understand the effect of uncertainty, it requires experts to be aware of uncertainty sources. We address this limitation by providing them with uncertainty source catalogs, which can be propagated. Last, we discuss the generalization of our uncertainty propagation. We propose Uncertainty Flow Diagrams (UFDs) to represent different uncertainty types in a single diagram type. This notation also helps to better understand *uncertainty interactions*, which have been introduced in Chapter 5. In the future, approaches like this will hopefully be helpful in many related research directions.

7. Uncertainty-Aware Data Flow Analysis to Identify Confidentiality Violations

In this chapter, we present the third Contribution **C3**. This contribution covers the last activity of the procedure shown in Section 4.1, the confidentiality analysis with respect to uncertainty. We discuss the representation of uncertainty as a first-class entity in data flow analysis and introduce four approaches for uncertainty-aware confidentiality analysis.

Software systems are becoming increasingly complex, e.g., in Industry 4.0 [34] or automotive systems [2]. As stated previously, ensuring security-related quality properties like confidentiality becomes a major challenge. Violations cannot only have legal consequences [122] but also affect user acceptance [271]. As proposed by *Privacy by Design* [224], confidentiality should be considered early to avoid costly repairs [32]. This can also seen with the new "Insecure Design" category from the OWASP Top 10 [192] list, which contains the top 10 categories of security problems for web applications.

This problem has been addressed with design time confidentiality analysis. By analyzing data flows [236] or potential attack paths [267, 268] in modeled software architectures, confidentiality requirements [105] can be evaluated early. We discussed these approaches in previous chapters, especially in Chapter 6. There, we built on the transformation from the Architectural Description Language (ADL) Palladio Component Model (PCM) to Data Flow Diagrams (DFDs) in order to propagate uncertainty regarding confidentiality. This transformation originates from an architecture-based data flow analysis to identify confidentiality violations already at design time [233, 234, 236]. Based on an architectural description of the software system, violations of provided confidentiality requirements [105] can be identified. This involves the model-based propagation of characteristics within the software architecture, which describe the flowing data and its properties regarding confidentiality. In sum, these approaches enable software architects to assess the confidentiality of software systems.

However, in early development and in complex systems of systems, uncertainty exists about the software architecture and its environment [2]. Our running example from Chapter 3 shows multiple uncertainty sources, e.g., user input (**U1**), data processing (**U2**), or deployment (**U3**). When ignored, these uncertainties affect both the precision and comprehensiveness of the results of confidentiality analysis [97, 99, 104]. The lack of information reduces the quality of the model's representation of the software system. This results in less accurate or even wrong predictions, made on the basis of the model. Although uncertainty-aware analyses of software architectures exist [70], they usually focus on other quality properties like performance [243].

To address this limitation, we present multiple approaches to architectural confidentiality analysis under uncertainty. We build on the findings about DFDs and uncertainty from the previous chapters to define data flow analyses that identify violations of confidentiality requirements with respect to uncertainty. Taking known uncertainty into account helps in making more comprehensive statements about a software system's confidentiality. The handling of uncertainty can either be included as part of the analysis in a white-box manner or delegated to an uncertainty-aware framework [2]. Here, we keep the architectural abstraction to enable design time analysis. Similarly to the previous chapter, we also propose automated analysis and tool-supported modeling as "detecting confidentiality issues manually is not feasible" [234].

In sum, to define uncertainty-aware data flow analysis, advances both in modeling and analysis of confidentiality and uncertainty are required. As proposed by Garlan [80], we thereby include uncertainty as a first-class concern in the system design. We present two approaches for confidentiality analysis that are tailored to single uncertainty types, e.g., environmental, or structural uncertainty, as well as two uncertainty type-agnostic approaches. We also discuss common challenges and solutions in representing uncertainty in data flow analysis. We summarize this research concern of uncertainty-aware data flow analysis regarding confidentiality in the third research question:

Research Question 3: How to analyze confidentiality requirements using architectural data flow analysis with respect to uncertainty within the model?

The remainder of this chapter is structured as follows: First, we discuss the problem statement. We then introduce a framework for architectural data flow analysis. This framework builds on the architecture-based data flow analysis of Seifermann [233] and Seifermann et al. [237], but extends and simplifies both the modeling and the analysis. It lays the foundation for the further analysis approaches presented in this section. Afterward, we discuss how to represent uncertainty in data flow analysis. Depending on the approach and the use of existing information, different statements about confidentiality and uncertainty can be derived. Building on this, we present two approaches for uncertainty type-specific data flow analysis. The first uses fuzzy inference [140] together with data flow analysis to consider environment uncertainty. The second extends the architectural optimization approach PerOpteryx [143, 144] to consider the relation of structural uncertainty and confidentiality. Additionally, we present an approach to tracing uncertainty in data flow analysis and an uncertainty type-agnostic approach to data flow analysis. Last, we discuss the complexity of the different uncertainty-aware data flow analysis approaches. We close the chapter by presenting assumptions and limitations and giving a summary.

Literature: This chapter is based on the following (co-) authored publications: [IEEE SEAA 2022], [Springer ECSA 2022], [IEEE ICSA-C 2023], [Springer ECSA 2024]

7.1. Problem Statement

We summarize the problems **P1** – **P3** addressed by Contribution **C3**. Finding solutions to these problems helps to provide a comprehensive answer to ❓ **Research Question 3**.

P1: Extending data flow analysis to represent uncertainty To consider uncertainty in architectural data flow analysis, we first require an extensible framework for data flow analysis [36]. This framework lays the foundations for any type of uncertainty-aware data flow analysis that goes beyond black-box approaches [266]. By black-box approaches, we mean approaches to confidentiality analysis that do not consider uncertainty as a first-class entity but encapsulate an existing analysis into an uncertainty-aware framework. Additionally, we need to understand how uncertainty can be represented in confidentiality analysis [101]. This requires relating previous findings on modeling and propagating uncertainty in architectural models to data flow analysis. We stress that there is no single solution to this problem but various solutions that exhibit different benefits and drawbacks. A common baseline is envisaged to understand their characteristics and applications.

P2: Data flow analysis tailored to specific uncertaint types To better understand the relation of uncertainty and confidentiality on architectural abstraction, we first strive to define architecture-based data flow analysis tailored for specific uncertainty types. Here, we focus on uncertainty types that are common in software architectures. This includes structural uncertainty due to Architectural Design Decisions (ADDs) [124, 266] and also environmental uncertainty that can indirectly affect confidentiality, e.g., due to access control decisions under uncertainty [37, 41]. Although such focused analysis approaches are unable to analyze all uncertainty types that can affect confidentiality, they still provide precise results regarding single uncertainty types, especially compared to uncertainty-unaware confidentially analysis [233]. Additionally, they present an important step towards uncertainty type-agnostic data flow analysis [101].

P3: Uncertainty type-agnostic data flow analysis Ultimately, a comprehensive approach is required to analyze all uncertainty types that are relevant regarding confidentiality. Building on the findings from **P1** and **P2**, this approach shall be able to handle all uncertainty types described in the uncertainty classification, presented in Chapter 5. This uncertainty type-agnostic approach needs modeling support for expressing uncertainty in architectural models that goes beyond the annotation of uncertainty sources required for uncertainty impact analysis, discussed in Chapter 6. A common approach to expressing uncertainty in architectural models is modeling variation scenarios [255, 265]. However, analyzing variations—and especially combinations of variations—can quickly jeopardize the scalability [264, 268]. Building on previous findings regarding the Uncertainty Interaction Problem (UIP) in DFDs can reduce the analysis complexity.

7.2. A Framework for Architectural Data Flow Analysis

In this section, we build on an approach for architecture-based data flow analysis [232, 233, 234, 235, 236] to present our extensible framework for architectural data flow analysis [36]. This lays the foundation for further analysis approaches—which are not limited to uncertainty only—and also addresses Problem **P1**.

In general, there are two ways to approach uncertainty-aware analysis [2, 194]. We use the common differentiation between *white-box* analysis and *black-box* analysis composition [111, 253]. A black-box approach extends an analysis without knowing about the implementation details of the analysis and only uses its interfaces. A white-box approach can use the internal details and data structures for the composition. Both approaches have benefits and drawbacks, e.g., regarding expressiveness and maintainability. Regarding uncertainty, we favor white-box analysis approaches as they enable the representation of uncertainty as a first-class entity within the analysis [80]. However, this does not imply that black-box approaches are not feasible [266]. To enable white-box analysis, we require an extensible and stable analysis framework.

Regarding the architectural data flow analysis, the original Prolog-based implementation of Seifermann [233] was hard to maintain and had a high resource demand, which severely limits the applicability for large software systems. Although the analysis already used a model of a DFD [64] as an intermediate representation, it did not continue to follow the idea of using DFDs as the primary analysis artifact. With appropriate tool support, DFDs represent a powerful and commonly used mechanism for threat analysis [25] that helps in correctly identifying security-related issues [226]. Thus, we revisited all aspects of the analysis like the transformation of PCM models to DFDs, and the label propagation, and also reimplemented the analysis from scratch [231]. This results in a more scalable and also easier-to-extend analysis framework [36]. Additionally, the focus on DFDs as the primary analysis artifact simplifies the connection to uncertainty, as discussed in Section 5.5. We introduced some of the underlying concepts already in Chapter 2. For the scope of this work, we summarize the internal data structures and the analysis procedure and refer to our publications for more details [36, 105, 231].

7.2.1. Representing Data Flows in Data Flow Analysis

As discussed in Section 5.5, DFDs can be represented as Directed Acyclic Graphs (DAGs). Using a simple graph structure simplifies the reasoning and the automated analysis regarding confidentiality. We support the transformation of DFDs, which are based on the unified modeling primitives [235], and also of PCM models [207] to DAGs. The former enables the manual definition of DFDs, and the latter extends the integration of architectural models in the data flow analysis [234].

To represent data flows and their relations in DAGs, we use the notion of a Transpose Flow Graph (TFG). Flow graphs represent the flow from one source to one or multiple sinks [141]. However, regarding confidentiality, we are more interested in the relations of

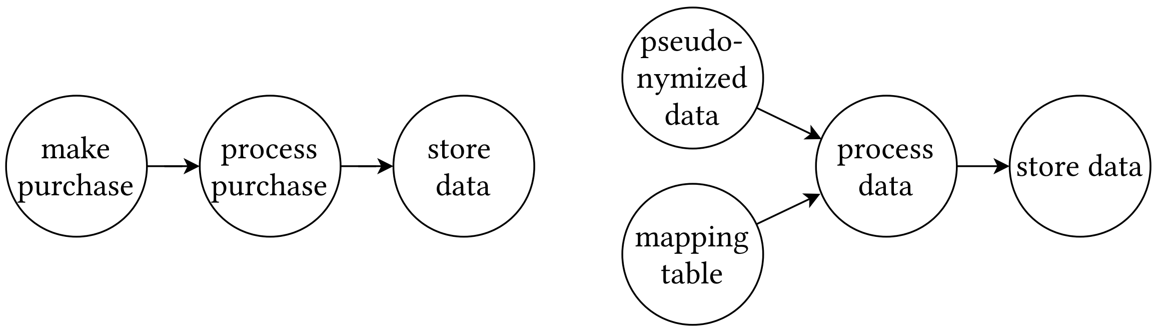


Figure 7.1.: Two Transpose Flow Graphs (TFGs) representing two independent data flows.

data flows from multiple sources to a single sink. An example is the joint flow of multiple sensitive data flows into a single sink, e.g., pseudonymized data flowing together with information about the relation of the data to its originating users, which could violate confidentiality. We use these transposed semantics of a flow graph, thus creating TFGs. Reasoning about multiple, smaller TFGs instead of one large DAG provides further benefits. It represents a divide and conquer approach, as the data flows represented by TFGs are independent and can thus be analyzed independently [233]. This simplifies the analysis and increases its scalability [231]. It is also more intuitive because confidentiality analysis represents identifying critical data flows to a single destination, i.e., critical TFGs.

Figure 7.1 shows two TFGs. The left TFG represents a simple data flow from the running example presented in Chapter 3. The right TFG represents the combination of data flows from two sources into a single sink, as discussed above with pseudonymized data. We can see, that the data flows in TFGs inherit all properties from DAG, i.e., being irreflexive, asymmetric, and transitive. Additionally, starting from the single sink, all other vertices are reachable without cycles. We can also see, that the partition in multiple, independent TFGs simplifies the reasoning. In the example shown in Figure 7.1, both *store data* processes could be represented by the same call to a data base. However, both data flows are independent, represent different data types and data processing, and also could cause different confidentiality violations. For instance, storing data without encryption could violate the confidentiality in the *store data* process in the left TFG, while *process data* could violate the confidentiality due to a broken pseudonymization in the right TFG.

Finding: Transpose Flow Graphs (TFGs) show independent data flows of a Directed Acyclic Graph (DAG). Following the principle of divide and conquer, this simplifies reasoning about the variety of data flows in a system.

To enable confidentiality analysis, we connect the concept of DAGs and TFGs to the unified modeling primitives [235], introduced in Section 2.5. Here, *Nodes* and *Flows* can be represented directly by vertices and edges, respectively. Additionally, vertices can be annotated with *Labels* to represent their characteristics regarding confidentiality, called *node labels*. Examples of such labels are the deployment location or the role of a user. The *Behavior* of a *Node*, represented by one or multiple *Assignments* can also be directly mapped to the vertices of a TFG. *Assignments* can change the characteristics of flowing

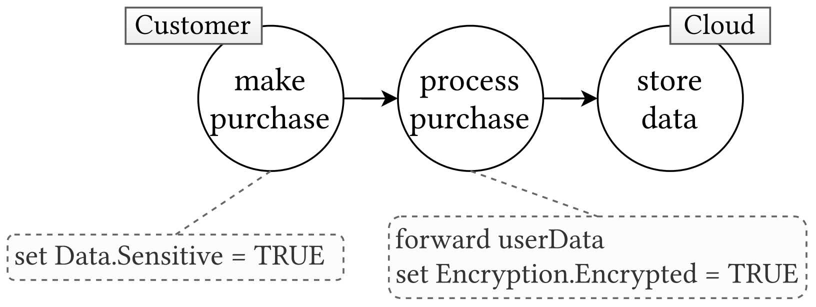


Figure 7.2.: A simple Transpose Flow Graph (TFG), annotated with node labels and node behaviors.

data, represented by *data labels*. For example, the behavior of the *process purchase* vertex in Figure 7.1 could include setting an *encrypted* label to represent the encryption of all data flows. Last, *Pins* are not directly represented in TFGs but influence their construction. For instance, multiple *Flows* out of a single *Pin* represent alternatives that yield a TFG for each alternative flow. Again, for the scope of this work, we do not detail the transformation of DFDs to TFGs, but refer to the original publications [36, 236].

Figure 7.2 shows the left TFG from Figure 7.1, annotated with node labels and behaviors, following the notation of the data flow analysis framework [36]. The exemplary *node labels* show which vertices represent functionality of the *Customer*, or of components deployed in the *Cloud*. Note that there is no general rule on which characteristics can and should be represented by node labels—any information, that can be relevant for analyzing confidentiality, can be included within the model [233]. The exemplary node behaviors show how the *data labels* can be altered. First, the *make purchase* vertex sets the label *Data.Sensitive*. Then, this label is forwarded in the *process purchase* vertex, where also the label *Encryption.Encrypted* is added. Both labels flow to the *store data* vertex. In natural language, this simple TFG represents sensitive user data that flows from a user into a database that is deployed in the cloud, while being encrypted first. This notion simplifies reasoning about data flows regarding confidentiality and can also be applied to large DFD with thousands of nodes [231].

Confidentiality requirements can be formulated in the form of data flow constraints [105]. These constraints restrict selected *data labels* to never flow to selected *node labels*. In the example shown in Figure 7.2, an exemplary data flow constraint could be that there shall not be a data flow of sensitive but unencrypted data to the cloud. Due to the encryption in the vertex *process purchase*, this data flow constraint is satisfied. Thus, this constraint would not cause a confidentiality violation. More complex constraints that consider the multiple flows and their properties are also possible [36, 105].

Finding: Transpose Flow Graphs (TFGs) can be annotated with characteristics that represent system properties relevant for confidentiality analysis. These characteristics comprise node labels, which are annotated to vertices, and data labels, which are altered by a node's behavior. Confidentiality requirements can be represented as data flow constraints that compare node and data labels.

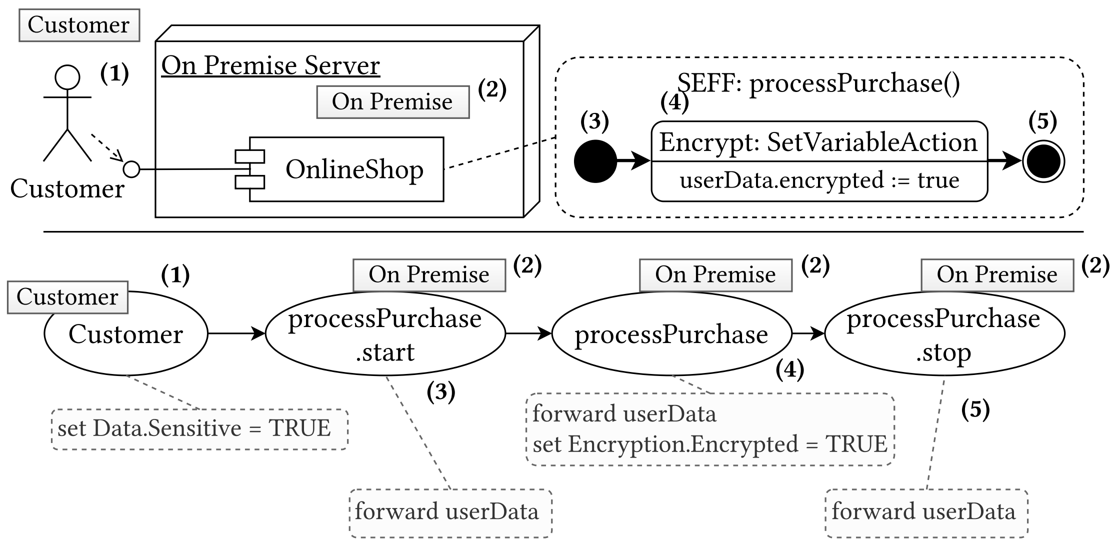


Figure 7.3.: Simplified Palladio Component Model (PCM) instance and the corresponding Transpose Flow Graph (TFG) with annotated node labels, data labels, and numbered transformation traces.

Although the manual modeling of DAGs with confidentiality characteristics is possible [36], we focus on the automated extraction of DAGs from PCM models. We introduced this extraction in Section 6.3. For instance, *ExternalCallAction*, *SetVariableAction*, and *EntryLevelSystemCalls* are represented by DFD nodes and thus also by TFG vertices. Additionally, we extract all confidentiality-related characteristics from the PCM and represent them as node labels and data labels.

Figure 7.3 shows a simplified PCM model based on the running example. The model is annotated with confidentiality-related labels that represent characteristics of data storage like *On Premise* and data processing like the encryption of *userData* in the *SetVariableAction*. In the lower half, we show the extracted TFG. We annotate numbers to represent the transformation traces from the PCM to the TFG. For every node, we perform a lookup of node labels, which can be annotated, e.g., to *ResourceContainers*, or *UsageScenarios*. An exemplary lookup in the PCM model goes from the *processPurchase* vertex to the *Online Shop* component via the deployment to the *On Premise Server* resource which is annotated with *On Premise*. Additionally, we convert the modeled system behavior to assignments representing the node's behavior, e.g., the encryption of *userData*. The default is the forwarding of labels, which represents the flow of data without further effects. The transformation considers all information that is relevant for confidentiality analysis, e.g., data processing. Other information is not transformed, e.g., components and servers do not cause additional elements in the TFG. This enables a system view from the perspective of the data. We store all traces to the originating PCM elements during the transformation. This enables the evaluation of more advanced constraints in the data flow analysis.

In sum, TFG represent a simple yet powerful way to reason about data flows, that can be extracted from PCM models. By adding node and data labels, they can represent the characteristics of the software system relevant for confidentiality analysis. Data flow constraints can compare these labels to identify confidentiality violations.

Algorithm 7.1 Algorithm for vertex evaluation in Transpose Flow Graphs (TFGs)

```
1:  procedure EVALUATEVERTEX(vertex, graph)
2:      vertex.nodelabels ← EVALUATENODELABELS(vertex)            ▷ Collect node labels
3:      predecessors ← GETPREDECESSORS(vertex)
4:      vertex.datalabels ← $\emptyset$
5:      for predecessor ∈ predecessors do              ▷ Collect data labels using recursion
6:          vertex.datalabels ← vertex.datalabels ∪ EVALUATEVERTEX(predecessor, graph)
7:      end for
8:      result ← EVALUATENODEBEHAVIOR(vertex, vertex.datalabels)
9:      return result
10: end procedure
```

Algorithm 7.2 Algorithm for label propagation in Transpose Flow Graphs (TFGs)

```
1: procedure LABELPROPAGATION(tfgs)
2:     for tfg ∈ tfgs do                            ▷ Iterate over all independent TFGs
3:         sink ← GETSINK(tfg)                       ▷ Start the evaluation at each sink
4:         EVALUATEVERTEX(sink, tfg)
5:     end for
6:     return tfgs
7: end procedure
```

7.2.2. Label Propagation to Enable Scalable Confidentiality Analysis

Annotated TFGs help to reason about data flows in software systems. However, DFDs of realistic software systems contain several hundreds of nodes [102]. *Label propagation* has been proposed to automate the analysis, as manual analysis is not feasible [234]. Put simply, label propagation propagates all data labels through all nodes of all TFGs with respect to the behavior of the nodes.

Algorithm 7.1 shows the recursive algorithm of the vertex evaluation in label propagation, which resembles DFS. For each vertex, we first retrieve the vertex node labels in Line 2. To simplify the algorithm, we denote the side effect of storing the labels in the vertex fields. Afterward, we calculate the data labels that represent the output of the vertex and also store them in the corresponding vertex. This is achieved by recursively calling the calculation logic of all previous vertices and using the hereby calculated output labels as input, as shown in Line 6. Before returning, we apply the behavior of the evaluated vertex to the input data labels in Line 8, e.g., altering the encryption label. Note that we do not consider cycles in the propagation logic because TFGs represent DAGs. Based on the evaluation of vertices, Algorithm 7.2 shows the label propagation. Here, we handle each TFG separately as every TFG represents independent data flows, and use the *evaluateVertex* algorithm for the single sink of each TFG.

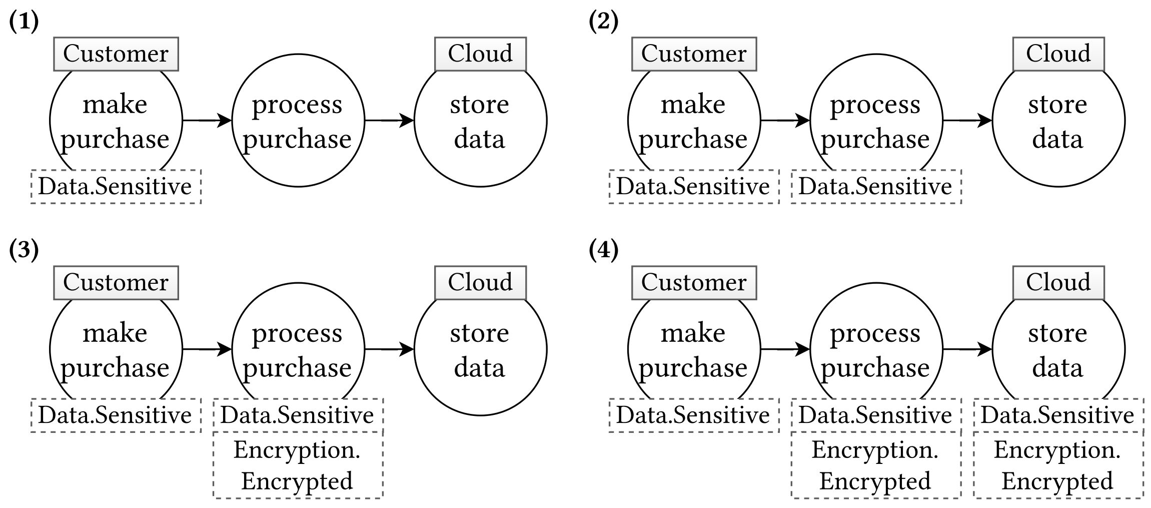


Figure 7.4.: Initial state, and intermediate and final results of the label propagation in a simple Transpose Flow Graph (TFG). Node labels are highlighted gray, data labels have a dashed border.

Figure 7.4 shows four states of the label propagation in the simple TFG introduced with Figure 7.2. First, we only display all node labels as these are not subject to label propagation but fixed, and also the label added in the *make purchase* vertex. This label is propagated to the *process purchase* vertex in the second step. By evaluating the behavior of this vertex, the label *Encryption.Encrypted* is added in the third step. In the fourth step, both labels are forwarded to the *store data* vertex and the label propagation terminates. After the label propagation, data flow constraints can compare node and data labels on each vertex without having to iterate over other vertices. The resulting confidentiality analysis is scalable and has been evaluated on large DFDs with thousands of vertices [231].

Finding: Label propagation in Transpose Flow Graphs (TFGs) represents a simple and scalable way to automate data flow analysis regarding confidentiality.

7.3. Representing Uncertainty in Data Flow Analysis

The data flow analysis framework [36] presented in the previous section provides us with means for scalable confidentiality analysis and also with the notion of TFGs to reason about data flows. The following discussion is based on this notion, and the differentiation between black-box and white-box analysis [111, 253], presented in Section 7.2. We show different approaches to represent uncertainty in data flow analysis [101]. These approaches differ in the use of data structures to analyze uncertainty and also in the use of information available for the analysis. This addresses Problem **P1**.

We addressed the representation of uncertainty in DFD, in Section 5.5, and in architectural models, in Section 6.2. The former presented the distinction between *primary* and *secondary* uncertainty in DAGs, the latter related the five uncertainty types to elements of the unified modeling primitives [235] and to the PCM. We build on this to explain the representation

Domain	Information categories
Confidentiality	**1.** Confidentiality violation occurrence, **2.** Violated confidentiality requirements, **3.** Location within the model, **4.** Analyzed data flows, **5.** State of data at the violation
Uncertainty	**6.** Uncertainty source, **7.** Uncertainty properties and classification, **8.** Uncertainty impact within the model, **9.** Uncertainty interactions, **10.** Uncertainty mitigation

Table 7.1.: Available information categories for uncertainty-aware confidentiality analysis.

of uncertainty in architecture-based data flow analysis. As discussed previously, there are multiple approaches to data flow analysis under uncertainty which show different levels of uncertainty-awareness. In the following, we discuss which information can be used for such analyses and present five levels of data flow analysis under uncertainty.

7.3.1. Available Information in Uncertainty-Aware Data Flow Analysis

Table 7.1 shows information categories regarding confidentiality and uncertainty. The confidentiality information categories are derived from data flow-based analyses of architectural models [235]. The categories include knowledge about the existence of confidentiality violations (**1.**) and the related violated requirements (**2.**). We consider this to be the result of any usable analysis. In the running example, the confidentiality requirements of storing personal data could be violated. Additionally, such analyses can point to the location within the architectural model (**3.**) and the critical data flow (**4.**) where the violation occurred. In the running example, the violating data flow could be from the *Customer* to the *Database Service*, where the violation happens. Last, the state of the data causing the violation can be considered (**5.**). This can be realized using labels like personal data, or deployment in the cloud. The data flow analysis framework presented in Section 7.2 uses all five categories in the analysis and also when presenting identified confidentiality violations. Nevertheless, analyses with a simplified interface are imaginable, e.g., black-box analyses only report the existence of violations.

The uncertainty information categories are derived from the uncertainty management in Self-Adaptive Systems (SASs) [115] and Cyber-Physical Systems (CPSs) [2] and also based on the handling of uncertainty regarding confidentiality [104]. By investigating an uncertainty source (**6.**) in the model or its environment [2], it can be described more precisely, e.g., based on a classification (**7.**), as discussed in Chapter 5. Also, the uncertainty's impact (**8.**) on the software architecture can be derived and expressed. We discussed the distinction between uncertainty sources and their impact in Section 5.2. In a software system with multiple uncertainty sources, potential uncertainty interactions are also important to consider (**9.**). This is related to the Uncertainty Interaction Problem (UIP) and has been discussed with regard to DFDs in Section 5.5. Last, by analyzing the architectural model, mitigation techniques (**10.**) can be chosen already at design time [115]. In our running example, we identified four uncertainty sources (**U1** – **U4**), which were also classified. We analyzed the potential impact of these uncertainties using uncertainty impact analysis in Chapter 6. For instance, uncertainty **U4** causes a broad impact in the *Database Service*. We also have means to represent simple uncertainty interactions, e.g., between secondary and primary uncertainty sources in DAGs, see Section 5.5. For example, we identified an uncertainty interaction between the deployment in Uncertainty **U3** and the provider trustworthiness in Uncertainty **U4**. Last, design time approaches can also support the tool-supported mitigation uncertainty that causes confidentiality violations. We consider this to be out of scope for this work as we focus on the uncertainty-aware analysis as a required prerequisite for mitigation [272].

7.3.2. Levels of Uncertainty-Awareness in Data Flow Analysis

Based on these information categories (**1.** – **10.**), we define five levels (**L0** – **L4**) of data flow analyses under uncertainty. Regarding model-based confidentiality analysis, showing violations (**1.**) with the related reason (**2.**) to software architects is considered to be the minimum viable information [105]. The analysis results become more expressive by also showing the location (**3.**), analyzed data flows (**4.**) and the state of the data (**5.**).

L0: No uncertainty-awareness Data flow analyses that provide neither modeling nor analysis support for uncertainty show *no uncertainty-awareness*. This affects the precision and comprehensiveness [97] of confidentiality violations as the models and the analysis fall short of representing the real world, as discussed in the introduction of this chapter. The lack of awareness becomes visible in over-estimations and also in missing violations [233]. In our running example, a confidentiality analysis that ignores uncertainty would miss all violations due to the identified uncertainty sources. This level represents the state of the art before our endeavors [99, 233, 236]. Note that this does not mean that such analyses cannot be extended to consider uncertainty [99].

L1: Naive approach to uncertainty-awareness The *naive approach* to handling uncertainty is to directly model its impacts (**8.**). This means manually altering models to respect the change caused by uncertainty. In our running example, we can model different variants, e.g., with the deployment on-premise or in the cloud. The analysis rejects the software architecture if there exist confidentiality violations in at least one variant. By directly representing the uncertain effect within the model, we lose all information about the source (**6.** and **7.**). Especially when considering multiple uncertainty impacts, all variant combinations and interactions (**9.**) have to be manually modeled, which requires extensive manual effort. Additionally, changes to the model have to be done in all variants. While this yields more comprehensive results, it still highly lacks precision [29] and requires an unreasonably high manual effort.

L2: Scenario-awareness *Scenario-aware* analyses build on the idea of representing uncertainty sources (**6.**) as model variations separate from the modeled software architecture. Here, variability models, model variations, or partial models represent common ways of expressing the potential outcomes of uncertainty [255]. We also included this level of information in our classification (**7.**). The option *Scenario Uncertainty* in the category *Type* expresses a state of knowledge where the effects of an uncertainty source can be described using scenarios. As classified in Section 5.4, all uncertainty sources are describable using scenarios in our running example. The main difference to the *naive approach* (**L1**) is the automated application of these variations to the architectural model. This approach greatly reduces the manual modeling effort, as software architects only have to specify the point of variation and do not realize the variation and thus the impact of the uncertainty (**8.**) itself. The analysis rejects only such architecture variants that violate confidentiality, without considering the location (**3.**), data flows (**4.**) or the state of the data (**5.**). Realized as a black-box analysis extension, this lacks precision because it only considers the existence of violations (**1.**) of requirements (**2.**) in selected scenarios. Last, while the manual effort is reduced, the complexity of calculating variations of variations can quickly lead to a combinatorial explosion, as discussed in the context of design space exploration [143].

L3: Graph-awareness To overcome the limitations regarding scalability and expressiveness of the results, *graph-aware* analyses can be used. Realized as white-box extension [29, 35, 37, 102] of a data flow analysis, this enables the connection of uncertainty sources (**6.**) to analyzed data flows (**4.**). This is the first level that is able to represent uncertainty as a first-class concern, both within the architectural model and DFDs. It builds on the idea of representing uncertainty as a set of scenarios, introduced with the *scenario-aware* approach (**L2**). However, by relating the variation directly to the analyzed data flows, a higher expressiveness can be reached. *Graph-aware* analyses thus cover all five confidentiality related categories (**1.** – **5.**) and are able to represent Uncertainty sources (**6.**), their properties (**7.**), and their impact (**8.**) directly within the model. Additionally, they can achieve higher scalability as uncertainty sources in different data flows can be analyzed independently, which partially addresses the combinatorial explosion. Put simply, in a real-world software system with many independent data flows, the effects of uncertainties can be separate. To leverage this, the analysis requires awareness about the analyzed graph, which represents the main difference to the *scenario-aware* approach (**L2**). In our running example, the *scenario-aware* approach can only try out all variations, while the *graph-aware* approach can use the information of the different data flows from the *Customer*. However, this does not lead to more accurate results—both approaches differ in expressiveness and scalability. Note that we renamed "data flow-awareness" to "graph-awareness" compared to the original publication [101] to increase clarity[1].

[1] Like an uncertainty-aware uncertainty analysis, a data flow-aware data flow analysis does not make too much sense. However, I am not so sure about the former—what about antifragility?

L4: Impact-awareness The *graph-aware* analysis (**L3**) is able to consider the effect of single uncertainty sources, both within the architectural model and DFDs. However, the impact of uncertainties can also affect other uncertainties, i.e., uncertainty interaction. This becomes visible in uncertainty-aware data flow analysis when an uncertainty alters a data flow that contains another uncertainty. In our running example, such an interaction happens with Uncertainty **U3** about the deployment and Uncertainty **U4** about the provider's trustworthiness. The former can impact the data flow so that it does not flow to a location that is affected by the latter, as shown in Figure 5.2. By understanding such interactions and the transitive effects of propagating uncertainty, and by incrementally analyzing data flows, *impact-awareness* can be reached. This level builds on the findings of *scenario-aware* (**L2**) and *graph-aware* (**L3**) approaches and thus yields analysis results with the same accuracy regarding confidentiality violations. However, by also considering uncertainty interactions (**9.**), a higher expressiveness and scalability can be reached.

These five levels (**L0** – **L4**) of uncertainty-aware data flow analysis provide the terminology to present the different research approaches hereafter. Although the terminology is related to orders of ignorance [11] or the levels of uncertainty [41], there exists no direct mapping, as we only focus on already identified uncertainty sources, see Chapter 5. Note that these levels do not represent a comprehensive list but only a classification for the scope of this work. Further classes are imaginable, e.g., regarding the mitigation (**10.**), or by using advanced algorithms for analyzing uncertainty impacts and confidentiality violations.

Other classifications of uncertainty-aware data flow analyses are useful in addition to these levels. Regarding the wide variety of uncertainty sources, some analyses might not cover all types of uncertainty [184, 255]. Troya et al. [255] conduced a Systematic Literature Review (SLR) with 123 papers and found that the majority of approaches only consider less than three types of uncertainty, e.g., behavior uncertainty combined with design uncertainty or belief uncertainty combined with measurement uncertainty. Here, we build on the five uncertainty types of the category *Architectural Element Type* of our classification, introduced in Section 5.3. We call an uncertainty-aware data flow analysis to be *type-specific* if it supports a subset of these uncertainty types. Uncertainty *type-agnostic* approaches support all five uncertainty types.

Finding: Uncertainty-aware data flow analysis approaches can be classified depending on the supported uncertainty types and the utilization of available information within the analysis. This enables the classification of approaches that differ in accuracy, expressiveness, and scalability.

7.3.3. Examples for Uncertainty-Aware Data Flow Analysis

We illustrate the different levels (**L0** – **L4**) and types of uncertainty-aware analysis using our running example. As discussed previously, all four uncertainty sources (**U1** – **U4**) represent *Scenario* uncertainty. Based on the findings of Troya et al. [255], we can thus describe the variation scenarios of all four uncertainty sources. Table 7.2 shows the possible scenarios for each uncertainty source that has been introduced in Chapter 3.

Uncertainty source		Uncertainty scenarios
U1	User input	Valid input / Erroneous input / Malicious input
U2	Data processing	None / Encryption / Validation / Both
U3	Component deployment	On premise / Cloud service
U4	Provider trustworthiness	Trustworthy / Suspicious

Table 7.2.: Uncertainty scenarios of all uncertainty sources of the running example.

A data flow analysis that lacks uncertainty-awareness (**L0**) is not able to consider these uncertainty sources. Speaking in terms of variations, these analyses select one scenario of each uncertainty, i.e., the architectural model without modification. Only if this default scenario violates confidentiality, the violation is identified. The *naive* approach (**L1**) to uncertainty-aware analysis is to manually model all combinations of all scenarios. As this approach must assume the dependence of all uncertainty sources and their scenarios, we use the Cartesian product [76]. Thus, software architects have to manually model $3 \cdot 4 \cdot 2 \cdot 2 = 48$ software architectures in the running example, which is not feasible.

Using variation models in *scenario-aware* analysis (**L2**), software architects only model the variation points, i.e., all $3 + 4 + 2 + 2 = 11$ scenarios of the running example. The variation creation can be automatized [265], which reduces the manual modeling effort, i.e., $11 < 48$. However, the assumption on the dependence of all uncertainty sources remains and thus, all resulting 48 variants have to be analyzed. With larger amounts of uncertainty sources or scenarios, this can quickly lead to the already discussed combinatorial explosion [143].

The *graph-aware* approach (**L3**) addresses this issue. As discussed with uncertainty propagation in DFD in Section 6.4, uncertainty sources can be considered independently in all independent data flows. This reduces the number of variants that have to be analyzed in all non-trivial software systems, i.e., in software systems with more than a single data flow. Our running example comprises three independent data flows: Retrieving the support contact, querying items, and making purchases, see Figure 3.2. Using the DFD as a data structure for the analysis, the *graph-aware* approach yields more comprehensive results. In our running example, we can see that the data flow representing the support contact retrieval is neither affected by uncertainty nor violates confidentiality.

Last, the *impact-aware* approach (**L4**) builds on this and also considers uncertainty interactions. We discussed this with the interaction of the uncertainties **U3** and **U4** in Section 5.5. Note that we do not classify the uncertainty impact analysis presented in Chapter 6 as it does not represent a confidentiality analysis. However, this analysis could be considered as type-agnostic and graph-aware, as all five uncertainty types can be propagated on the DFD. This analysis does not reach *impact-awareness* because the effect of changes to the software architecture is ignored and overestimated.

In the remainder of this chapter, we present four approaches to uncertainty-aware data flow analysis. Table 7.3 classifies these approaches according to the five levels (**L0** – **L4**) and with regard to the type specificity. We also add the *uncertainty-unaware* approach of architecture-based data flow analysis for reference purposes.

Level of awareness		Uncertainty type-specific	Uncertainty type-agnostic
L0	None	Uncertainty-unaware data flow analysis [36, 233, 236]	
L1	Naive	-	-
L2	Scenario-aware	Approach for structural uncertainty, Subsection 7.4.1	-
L3	Graph-aware	Approach for environmental uncertainty, Subsection 7.4.2	Approach for tracing uncertainty, Subsection 7.5.1
L4	Impact-aware	-	Approach for impact-aware analysis, Subsection 7.5.2

Table 7.3.: Awareness levels and type specificity of uncertainty-aware data flow analysis approaches.

The data flow analysis framework [36] without explicit support for modeling or analyzing uncertainty as presented in Section 7.2 represents Level **L0**. We do not distinguish between the type specificity as the analysis does support no uncertainty at all. We also do not present naive approaches (**L1**) in this thesis, as they do not represent a noteworthy improvement on the state of the art. The first approach discussed in this thesis is the *scenario-aware* analysis (**L2**) of structural uncertainty, e.g., regarding uncertain component deployment. Due to the previously discussed limitations of scenario awareness, we continue with an approach for environmental uncertainty that is *graph-aware* (**L3**). Afterward, we present the first uncertainty type-agnostic approach for data flow analysis. This approach covers all five uncertainty types and traces confidentiality violations to variations caused by uncertainty sources. Last, we further enhance this concept and present an uncertainty *impact-aware* approach (**L4**) for data flow analysis regarding confidentiality. Generally speaking, a higher level and being type-agnostic is superior.

7.4. Uncertainty Type-Specific Data Flow Analysis

In this section, we present the first two approaches for uncertainty-aware data flow analysis. The first approach is *scenario-aware* (**L2**) and considers structural uncertainty within the software architecture. The second approach is *graph-aware* (**L3**) and deals with environmental uncertainty concerning user privileges in access control. Both approaches are *type-specific* as they are tailored to one uncertainty type. This addresses Problem **P2**.

7.4.1. Data Flow Analysis Under Structural Uncertainty

As the name suggests, structural uncertainty affects the structure of the software architecture. Examples are Architectural Design Decisions (ADDs) regarding components or their deployment. As this represents uncertainty in the software design [167], it is best represented using variation points [255]. Regarding the variation creation, there already exist approaches such as design space exploration [243]. For this analysis approach, we use the architectural optimization approach PerOpteryx [143, 144, 145, 163] and combine it with data flow analysis [236]. Software architects first model different design decisions. These design decisions are then used to automatically create different architecture variations, which are analyzed for confidentiality. Since we reuse PerOpteryx for the design space exploration, this approach can consider additional quality attributes, such as performance, or costs, and calculate the Pareto optimal candidate. This optimization enables software architects to make informed trade-off decisions between quality metrics.

PerOpteryx [143, 144] is a design space exploration and optimization approach for the PCM. It uses evolutionary search algorithms and calculates the Pareto optimal architectural variation. The different variation points of the architecture are specified in the design decision model. Here, software architects can define which entities in the software architecture can be configured. For instance, in our running example, a design decision is the deployment of the *Database Service* component to the *On Premise Server* or to the *Cloud Service*. Due to the relation of ADDs and uncertainty, explained in Section 5.2, we can use this model to express structural uncertainty sources and their scenarios. The different quality attributes are defined as quality dimensions. Each generated architectural variation that is evaluated for quality attributes is called a candidate. Besides the optimal candidates, PerOpteryx also yields all the investigated candidates. There exist various quality dimensions, such as costs or performance. However, there is no quality dimension for confidentiality, which is required for the connection to the data flow analysis. While PerOpteryx already contains a security dimension [45], we decide against it since we want to explicitly target confidentiality. The existing security dimension is focused more on the costs of introducing security measures and the costs of failure.

We add a new confidentiality violation to PerOpteryx that is modeled as real values where decreasing values are better solutions [155]. We map these to a binary classification for the confidentiality analysis results. For each quality dimension, PerOpteryx defines handlers, which can analyze a model for the given quality dimension. Therefore, we define a new confidentiality handler for the confidentiality domain, which acts as an adapter between PerOpteryx and the data flow analysis. This handler forwards the current architecture candidate models to the data flow analysis and interprets the analysis result. The interpretation of confidentiality violations is binary with the classes *confidentiality ensured* and *confidentiality violated*. Our adapter transforms *confidentiality ensured* to -1 and *confidentiality violated* to 1. This binary interpretation is required as there is currently no meaningful way of further quantifying the results of the data flow analysis [266]. The results enable the PerOpteryx optimization to rank confidential candidates higher than non-confidential ones. After the execution of PerOpteryx, software architects get a result with all tested variations and the Pareto optimal version. Figure 7.5 shows this procedure of PerOpteryx [144] using the data flow analysis. For the sake of simplicity, we only depict a single quality dimension, i.e., confidentiality. Other quality dimensions can be represented by additional swimlanes that are called during the candidate evaluation.

In our running example, only Uncertainty **U3** about the deployment represents structural uncertainty. This uncertainty can be modeled as ADD in the design decision model. PerOpteryx then creates all variations, i.e., the deployment on the *On Premise Server* and

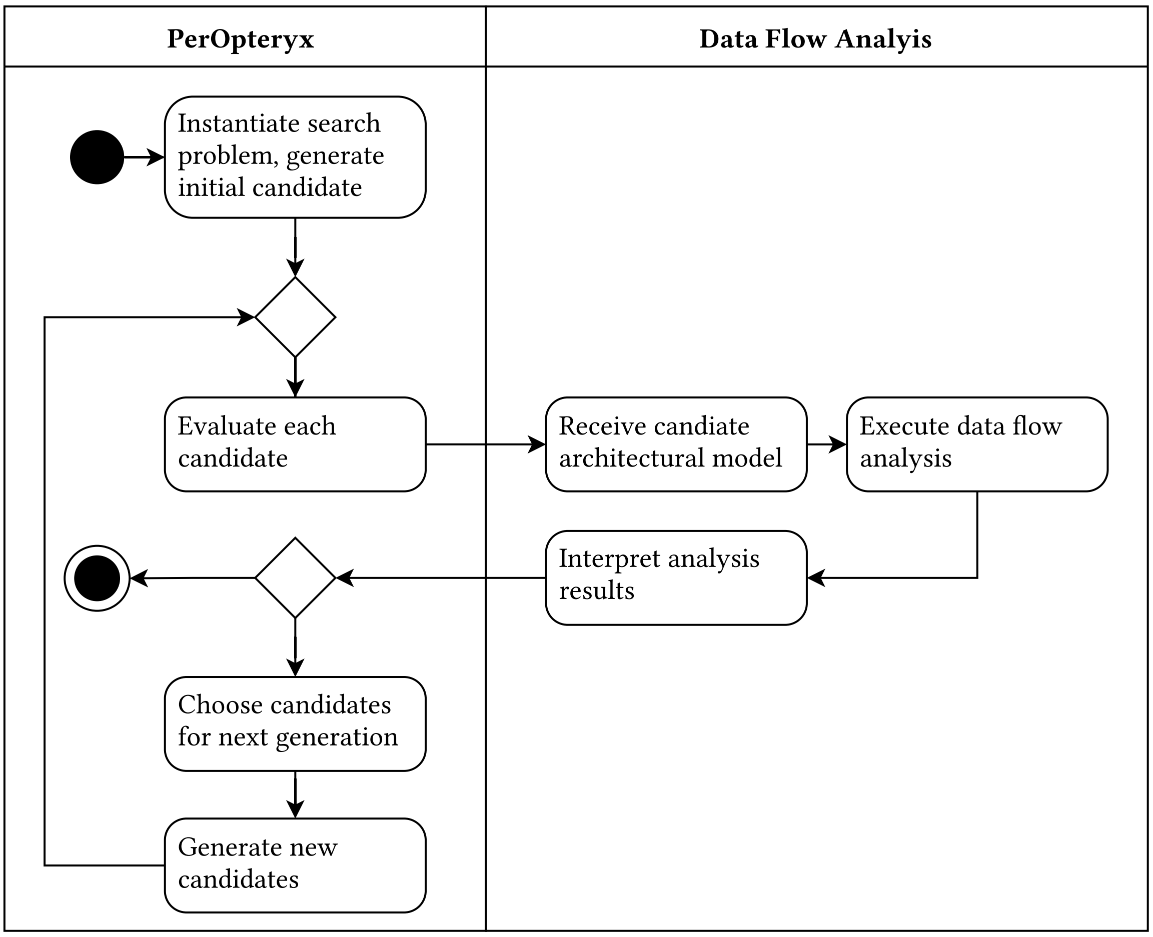

PerOpteryx
Data Flow Analyis
Instantiate search problem, generate initial candidate
Evaluate each candidate
Receive candiate architectural model
Execute data flow analysis
Interpret analysis results
Choose candidates for next generation
Generate new candidates

Figure 7.5.: Activity diagram showing the interplay of PerOpteryx and the data flow analysis using swimlanes.

on the *Cloud Service*. Both candidates are forwarded to the data flow analysis. The former ensures confidentiality while the latter causes a confidentiality violation and thus returns the value 1. Last, the deployment *on premise* is returned as the optimal version.

There are three disadvantages and two advantages of this approach. First, being type-specific to structural uncertainty limits the applicability to other uncertainty types that can affect confidentiality. Second, the black-box coupling of PerOpteryx with the data flow analysis only shows the existence of violations but no further information, e.g., on the location, or affected data flows, which impedes the interpretation by software architects. Third, confidentiality violations have no continuous occurrence, i.e., small changes to the software architecture can have a large impact. This is a known limitation of PerOpteryx and is described as a rugged search landscape with non-continuous jumps [143]. In the worst case, this can cause PerOpteryx to test out all possible variations, i.e., the Cartesian product, which resembles a brute-force approach. Nevertheless, this coupling is superior to the *naive* approach (**L1**) as it notably reduces the modeling effort. Additionally, combining confidentiality analysis with the optimization of other quality attributes represents a more realistic application scenario, where informed trade-off decisions are made by software

Algorithm 7.3 Algorithm for scenario-aware data flow analysis under uncertainty

```
1: procedure analyzeScenarioAware(model, constraint, uncertainties)
2:     violations ← ∅
3:     variations ← generateAllVariations(model, uncertainties)
4:     for variation ∈ variations do                ▷ Analyze all possible variations
5:         violations ← violations ∪ analyze(variation, constraint)
6:     end for
7:     return violations
8: end procedure
```

architects. No other approach presented in this thesis provides such functionality without requiring the combination with other architecture evaluation approaches [243].

Finding: On the one hand, the black-box coupling of data flow analysis with architectural optimization is complicated by the non-continuous occurrence of confidentiality violations, which increases the complexity of the optimization problem. On the other hand, extending an existing software architecture optimization approach simplifies the joint evaluation of multiple quality attributes.

To conclude, we present a more general approach to *scenario-aware* data flow analysis under uncertainty. Put simply, *scenario-awareness* (**L2**) is generating all possible variations of the architectural model and then analyzing them to identify confidentiality violations [265]. This is shown in Algorithm 7.3. However, such analyses are prone to the combinatorial explosion of uncertainty sources and their scenarios. Additionally, the analysis results are hard to interpret without further processing. For instance, in our running example, software architects would have to interpret the confidentiality violations of 48 architectural models to asses four uncertainty sources. We posit that *scenario-aware* analysis should only be used in specific cases, e.g., combined with design space exploration.

7.4.2. Data Flow Analysis Under Environmental Uncertainty

Environmental uncertainty arises from the system context, e.g., in the system usage, or its resource environment. Here, sensors are especially often referred to as uncertainty source [49, 50, 51, 195, 272]. Examples are sensor failure, sensor noise, or sensor inaccuracy [103]. We discussed the relation of such uncertainty and confidentiality in Chapter 5, where we classified such uncertainty as *environment* or *input* uncertainty. A particular challenge is the relation of environmental uncertainty and access control [41, 113]. Broken access control can lead to severe confidentiality violations and is listed as the TOP 1 entry in the OWASP Top 10 [192]. To address this, we present our second approach to uncertainty-aware data flow analysis: A *graph-aware* analysis (**L3**) of environmental uncertainty. The following description is based on our publication about this approach [37].

A common measure to ensure confidentiality are access control mechanisms that authorize access or processing of data. Depending on the underlying access control model, various sources of information are used to determine whether access is granted. Services use different sources, e.g., sensors, to acquire information. Services process the information and provide a resulting access control attribute, e.g., a role or a location, that can be used by the access control system. However, these services introduce a degree of uncertainty to the access control system. This uncertainty results from influencing factors of the environment when acquiring the information. The influence of the factors on the access control system can result in reduced validity of statements about confidentiality. Depending on the system design, software architects or security experts might already be able to identify potential influencing factors. For example, the ability of a GPS service to provide an accurate position is highly dependent on its surroundings, like buildings.

We extend our running example presented in Chapter 3 to illustrate this challenge. We add an *Admin* user who can directly access the *Database Service* for maintenance [264]. To enhance security, access to the database is only permitted if the accessing actor is located inside of the office. The location of each actor is monitored using a GPS location service. Here, we can choose a high-sensitivity GPS sensor, short *HSGPS*, or the normal GPS sensor in the actor's mobile phone. For this example, we focus on the differences between running the two types of GPS sensors indoors [138], where the sensor quality can degrade. Once the actor enters the building, the signal attenuation increases which results in a reduced signal-to-noise ratio. A reduced signal-to-noise ratio increases the time needed to acquire satellite data and calculate a position, and the amount of positioning errors [138]. While the HSGPS sensors are powerful enough to provide accurate positioning, the accuracy of the embedded GPS sensor of the mobile phone suffers when used inside, which leads to uncertainty regarding the actual position. This reduced accuracy introduces uncertainty about the position [51]. When this information is used within access control, it could lead to the wrong classification of the *Admin* or other actors in the building.

To address such uncertainty, we present an approach for handling known, environmental uncertainty in access control during design time. To this end, we enable the explicit representation of the known uncertainty and integrate the additional knowledge into an analysis. We propose a notion of *confidence* in the validity of access control attributes, based on three types of influencing factors of the system environment. We then show an access control analysis that integrates *confidence* into our data flow analysis framework, presented in Section 7.2. We propose the use of Fuzzy Inference Systems (FISs) [140] to combine influencing factors to a resulting confidence value.

Hu et al. [117] provide a definition and considerations regarding Attribute-Based Access Control (ABAC). They describe a trust chain, concerning the attributes used to make access control decisions. Trust chains help determine the ownership of information and services, as well as requirements for technical solutions to establish valid trust relationships. The predicate of a trust relationship revolves around the idea that the access control system can trust the validity or correctness of the information, supplied by the owner, e.g., an authorization service. Depending on the access control model, many trust relationships are required to achieve a properly working access control system. Figure 7.6 shows a

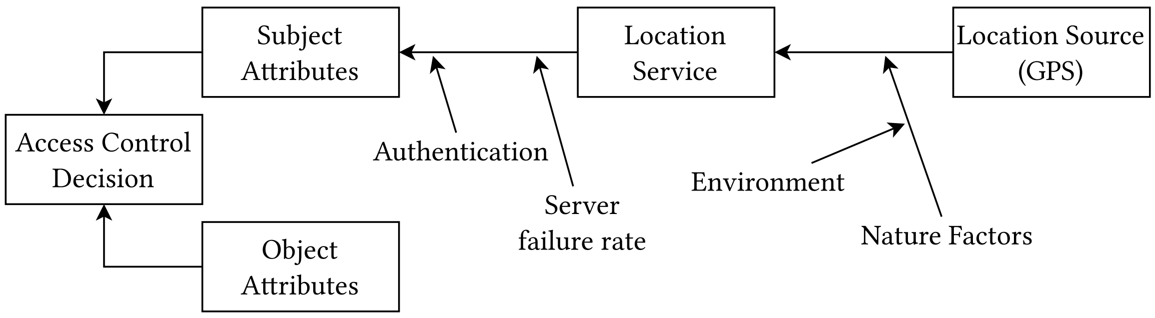


Figure 7.6.: A simplified trust chain that incorporates types of factors of the GPS location example.

trust chain of a location attribute. The *subject attributes* have a trust relationship with the *location service*, which in turn has a trust relationship with the *location source*. As indicated by the additional arrows, the trust relationships are impacted by influences of the environment of the system.

Ardagna et al. [10] define a confidence value that combines the service used for location determination and environmental conditions. Similar to the trust of Hu et al. [117], confidence describes the certainty that a location is valid, which is dependent on the environment. Based on the work of Hengartner and Zhong [114] we can identify three environmental factors that influence the uncertainty in access control. These factors can be applied to access control attributes, align well with the idea of trust relationships of Hu et al. [117] and can be represented in software architecture. We describe the three factors in the following.

First, the *source* that is used by the service to obtain information that is needed for the access control attribute. Location information, for example, might be derived from a physical access control mechanism or GPS data. Each of these sources has a different margin of error or accuracy. Second, *nature factors* that either influence the source's or the service's ability to return correct information or attributes. Depending on the sensitivity and rating, the accuracy with which GPS sensors can determine a location is heavily influenced by the physical environment, e.g., surrounding buildings [138], and weather conditions, e.g. cloudy sky [10]. Third, the *age* of an attribute, which, depending on the underlying information, might degrade validity. Depending on the attribute, age can be a combination of the time it took to gather the information from the source, the time it takes to process the information to an access control attribute and the overall time that has passed since this access control attribute has been created.

These factors and the associated uncertainties are all known, but especially when multiple factors need to be combined, their influence on the validity of an access control attribute is hard to describe. To provide a way of representing this known uncertainty, we define a notion of *confidence* in the validity of access control attributes. Confidence combines the known uncertainty of environmental factors into a single value, which represents the level of confidence that a corresponding attribute is valid. Confidence and the known uncertainty are not directly associated with an access control attribute, but rather with the service that is used by the system to derive the attribute. A service uses a source to receive

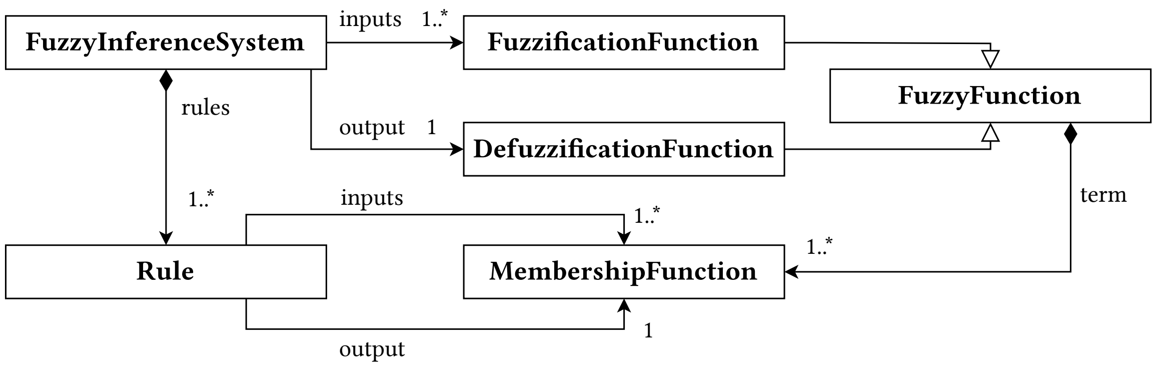


Figure 7.7.: Excerpt from the class diagram of the Fuzzy Inference System (FIS) meta model.

information. The service processes this information into an attribute and provides it to the access control system. The factors decrease the overall accuracy and thereby influence confidence. Any of the factors shown in Figure 7.6 might reduce confidence in the location attribute. For instance, this can result in *high confidence* in access control attributes, that are derived by services that use the HSGPS sensors, and *low confidence* in access control attributes derived by services using the mobile phone GPS sensor.

To combine these factors, we propose the use of fuzzy inference in the form of a FISs [140]. For over two decades, fuzzy sets and fuzzy logic have been used to describe uncertainty [140]. Fuzzy logic is also used in related work regarding design time uncertainty and uncertainty in access control [56, 70, 116, 255]. A FIS is made up of four main components [140]: A *fuzzifier* first translates the crisp values of environmental factors into fuzzy values, by applying them to a set of membership functions. The *fuzzy inference engine* uses the fuzzy input from the fuzzifier and rules to infer a fuzzy output. The *defuzzifier* translates the fuzzy output of the inference engine to a confidence value, by aggregating the fuzzy outputs. These steps are supported by a shared *knowledge base.*

We present a FIS metamodel as a way to enable software architects to create FISs. Figure 7.7 shows an excerpt of the class diagram representation of the FIS metamodel. We represent the environmental factors as *fuzzification functions*, which are made up of *membership functions.* Each membership function defines a fuzzy set and represents a *linguistic value* the environmental factor can take on. The corresponding membership functions define a degree of membership to the fuzzy set within an interval of $[0, 1]$. Using the value of the environmental factor, e.g. a 30% signal-to-noise ratio, the fuzzifier calculates a degree of membership for each fuzzy set of the associated fuzzification function. In our running example, the signal-to-noise ratio can take on the linguistic values of *low* and *high.* The linguistic values of age are *new* and *old.* As an example, the environmental factor values of a 30% signal-to-noise ratio would result in the membership degree of 0 for *high* and 0.5 for *low.* 3 minutes of age would result in the membership degree of 0 for *new* and 1 for *old.* Rules to combine the environmental factors are defined by combining a linguistic value of each environmental factor and defining a result. A rule that combines the most negative linguistic values of signal-to-noise ratio and age and consequently results in *low* confidence is defined like this: *IF signal-to-noise ratio is 'low' AND age is 'old' THEN confidence is 'low'.*

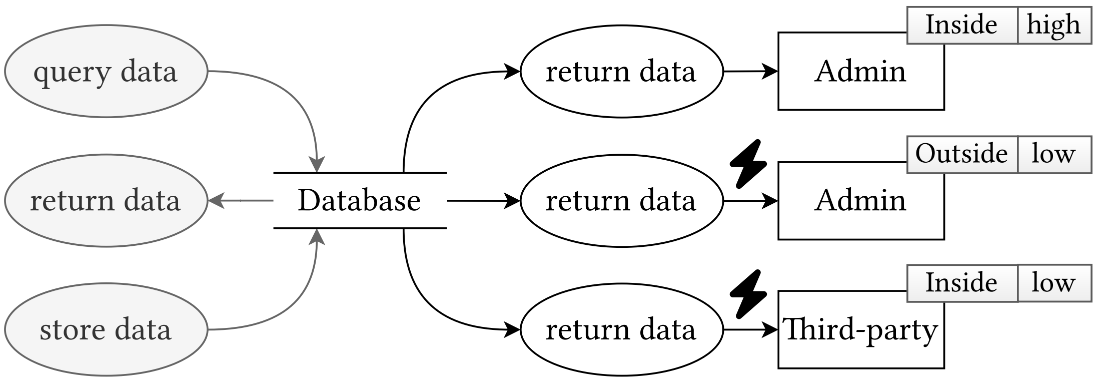


Figure 7.8.: Data Flow Diagram (DFD) of the extended running example with additional actors and their node labels. Confidentiality violations are marked with lightning bolts.

Depending on how the FIS is set up, the membership degrees of the linguistic values of a rule are combined. The result of the rule with the highest combined membership degrees is returned. Similar to how membership functions are used in the GuideArch approach [70], a software architect can initially define the environmental factor value as a fuzzy set. In subsequent stages of development, when more information becomes available, these ranges can be narrowed down or fixed to a value. For our running example, this could be done by measuring the signal-to-noise ratio in the actual physical environment.

Using FISs in the process has several benefits. Klir and Yuan [140] describe that through fuzzification an enhanced ability to model real-world problems is gained, lowering overall solution cost. The use of fuzziness also serves to better capture human common-sense reasoning and decision-making [140]. When setting up a calculation rule in general, the system's properties and environment are abstracted and simplified. As a result, information about the inputs and their influence on the result is lost. A FIS conserves more knowledge about the inputs and their influence on the calculated confidence value than a conventional mathematical function by mapping inputs to membership functions and working with natural language concepts. Additionally, a FIS is capable of capturing the meanings of sentences in natural language [140], which enables, e.g., a software architect or security expert to easily map statements or requirements about the influence of environmental factors on confidence to calculation rules.

We extend the data flow analysis presented in Section 7.2 to consider the so-calculated confidence values. Here, we build on the representation of all confidentiality-related characteristics as data labels and node labels. As confidence is directly related to such attributes, we extend these characteristics by adding confidence labels. Figure 7.8 shows a DFD of the extension of our running example with simplified data flows to the actors. The original data flows from the *Customer* on the left are marked gray. We add the node labels of the *Admin* actors a third-party actors with their corresponding *confidence* labels *low* and *heigh*. As discussed previously, these confidence labels originate from the FIS based on the input variables from the HSGPS or GPS sensors. The confidentiality requirement is defined as a data flow constraint that only allows data to flow to actors within the building with *high* confidence. Thus, the upper data flow is permitted as the *Admin* is inside the building with *high* confidence. The other data flows show confidentiality violations, as the *Admin* is either outside the building, or a *Third-Party* is considered to be inside the building with only *low* confidence. Without considering the confidence, the bottom confidentiality violation regarding the *Third-Party* would be missed.

We include all calculated confidence labels in all node and data labels and also in the formulation of data flow constraints. Although this represents a white-box extension of the data flow analysis, we do not alter the label propagation algorithm, only the textual representation of labels [37]. Thus, the calculation of the FIS can be interpreted as a preprocessing step before the data flow analysis. According to our classification, this represents a *graph-aware* data flow analysis (**L3**) under uncertainty. The analysis result contains all information from the data flow analysis, e.g., violation location, violated data flows, and the mismatching labels that cause the violation, including information about the calculated confidence. Due to the application of FISs, information about both the environmental uncertainty sources and their impact is represented within the model and the analysis, as discussed previously. Last, uncertainty can be analyzed independently on distinct data flows, which minimizes the analysis complexity. For more information on the realization, please consult our publication [37], or the data set [98].

To conclude this section, we compare and discuss both approaches of uncertainty type-specific data flow analysis. Both approaches employ already known techniques for representing and handling uncertainty [243, 255], i.e., design space exploration [143, 144] or fuzzy inference [140]. Both approaches combine these techniques with data flow analysis, either in a black-box manner as in the first approach, or a white-box manner in the second approach. This results in the first approach for structural uncertainty being *scenario-aware* and the second approach for environmental uncertainty being *graph-aware*.

Both approaches are type-specific and employ techniques that are suited for representing their targeted uncertainty type, i.e., design space exploration for structural uncertainty [243, 255], and fuzzy inference for environmental uncertainty [56, 70, 116]. Despite their differences in supported uncertainty types and realization, they show the commonality of considering uncertainty in form of scenarios. The design decision model of PerOpteryx enables software architects to describe design alternatives, i.e., different scenarios of a software architecture. The application of FISs enables software architects to calculate distinct classes of a *confidence* value, e.g., *low* or *high*, i.e., different scenarios describing the attributes in access control decisions. The suitability of scenarios or variation [265] to represent uncertainty has simultaneously been found by Troya et al. [255]. However, besides being type-specific, both approaches also share the shortcoming of requiring expert knowledge. Software architects or security experts have to instantiate the FIS meta model with accurate values or have to specify design decision models. In the following, we build on these findings to define uncertainty type-agnostic data flow analysis.

Finding: Data flow analysis can be extended by techniques like design space exploration or fuzzy inference to consider uncertainty within the analysis. Here, decomposing uncertainty sources into their scenarios is expedient.

7.5. Uncertainty Type-Agnostic Data Flow Analysis

In this section, we present the second two approaches for uncertainty-aware data flow analysis. The first approach is *graph-aware* (**L3**) and the second approach is *impact-aware* (**L4**), which represents the highest awareness level, see Section 7.3. Both approaches are type-agnostic, i.e., support all five uncertainty types introduced in Section 5.3. We also present 🔧 Abunai, which is tooling to support the modeling and analysis of the *impact-aware* approach. This addresses Problem **P3** about type-agnostic data flow analysis.

7.5.1. Tracing Uncertainty in Data Flow Analysis

Incorporating information about independent data flows—called TFGs in our terminology introduced in Section 7.2—enables the definition of *graph-aware* analysis (**L3**). We present an analysis that traces confidentiality violations back to the originating uncertainty sources using this information. This enables relating violations to concrete scenarios and simplifies the interpretation and mitigation by software architects.

In this approach, we use variation models [168, 265] to express uncertainty sources and their scenarios. Variation models are similar to the already discussed design decision models presented in Section 7.4. Software architects specify target elements and alternative elements of the software architecture. This can be applied, for instance, to deployment locations, or to actions in Service Effect Specifications (SEFFs). As discussed previously, *design uncertainty* is "normally represented in software models by variability models" [255]. Model variation can be used to describe possible outcomes of *Scenario Uncertainty*. Each uncertainty is represented by a variation point and a selection of alternative architectural elements. This description results in a variation model [168] of the software architecture. Table 7.2 shows all scenarios of all uncertainties of the running example.

Based on the variation model, architectural variants can be generated by permutation [168]. If the information about data flows were ignored, this would cause the same complexity as a *scenario-aware* analysis (**L2**). As discussed in Section 7.3, our running example requires 48 variations. However, not all uncertainties affect all TFGs. Thus, the relevant variations can be filtered in *graph-aware* analysis (**L3**). The resulting variants are then analyzed by the data flow analysis presented in Section 7.2. By relating the resulting confidentiality violations to the analyzed variation and its uncertainty, software architects can focus on critical variants, i.e., variants that violate confidentiality. The hereby defined analysis traces violations back to their originating uncertainties [29].

Table 7.4 shows the three TFGs in the running example and the uncertainty sources that can be found in these TFGs. This table can be derived based on Figure 3.2 showing the DFD of the running example and Table 7.2 showing the scenarios of all uncertainty sources. The first TFG representing the support contact request is not affected by uncertainty. The second TFG is only affected by the deployment (**U3**) and the provider trustworthiness (**U4**). The third TFG is additionally affected by the user input (**U1**) and the data processing (**U2**), i.e., by all uncertainties in the running example.

#	TFG name	Relevant uncertainty sources
1	Request support contact	-
2	View available items	**U3**, **U4**
3	Purchase items	**U1**, **U2**, **U3**, **U4**

Table 7.4.: Relation of Transpose Flow Graphs (TFGs) and uncertainty sources in the running example.

Algorithm 7.4 Algorithm for graph-aware data flow analysis under uncertainty

```
1: procedure analyzeGraphAware(model, constraint, uncertainties)
2:     violations ← ∅
3:     filtered ← ∅
4:     variations ← generateAllVariations(model, uncertainties)
5:     tfgs ← retrieveAllTFGs(model)
6:     for tfg ∈ tfgs do                                  ▷ Analyze each TFG separately
7:         for variation ∈ variations do                  ▷ Filter relevant variations
8:             filtered ← filtered ∪ filterVariation(model, tfg, variation, uncertainties)
9:         end for
10:        for variation ∈ filtered do                    ▷ Only analyze filtered variations
11:            modifiedtfg ← applyVariation(tfg, variation)
12:            violations ← violations ∪ analyze(modifiedtfg, constraint)
13:        end for
14:    end for
15:    return violations
16: end procedure
```

As the three TFGs are independent, their variations can be analyzed independently. This results in the lowered complexity of the *graph-aware* analysis (**L3**) compared to the *scenario-aware* analysis (**L2**). Instead of having to apply all possible variations to each TFG, we can first filter the relevant uncertainty sources and then only analyze all variations of these uncertainty sources. Note that this does not affect the identified confidentiality violations as the data flows represented by the TFGs are independent [233]. While the complexity is reduced, the accuracy remains the same. The more independent data flows there are, the greater this effect becomes, as we assume in real software systems [102].

Algorithm 7.4 shows the algorithm for *graph-aware* data flow analysis (**L3**). The approach is similar to *scenario-aware* data flow analysis shown in Algorithm 7.3, but comprises the aforementioned filtering. We iterate over all TFGs of a model in Line 6 and analyze each TFG separately. This is possible because *graph-aware* analysis has the information of all data flows. For each TFG, we filter the variations for relevant ones in Line 8. Afterward, we run the analysis only on these variations. This is achieved by applying the variation first in Line 11 and then forwarding the TFG to the data flow analysis.

Algorithm 7.5 Algorithm for filtering variations in graph-aware analysis

```
1: procedure filterVariation(model, tfg, variation, uncertainties)
2:     applied ← getAppliedUncertainties(variation, uncertainties)
3:     vertices ← getVertices(tfg)                          ▹ Iterate over all vertices
4:     for vertex ∈ vertices do
5:         for uncertainty ∈ applied do
6:             if affects(uncertainty, vertex, model, variation) then
7:                 applied ← applied \ {uncertainty}
8:             end if
9:         end for
10:        if applied = ∅ then                              ▹ All uncertainties are relevant
11:            return {variation}
12:        end if
13:    end for
14:    return ∅                                  ▹ At least one uncertainty is not relevant
15: end procedure
```

Algorithm 7.5 shows this filter function. A variation is relevant for a given TFG if all uncertainty sources that cause a change in the architectural model affect at least on vertex of the TFG. We start by retrieving all uncertainties that have been applied in the given variation in Line 2. This is the case, if the permutation that causes the variation [168] changes an architectural element due to said uncertainty. For instance, if one variation only changes architectural elements representing the data processing and the deployment location in our running example, this function would yield the uncertainties **U2** and **U3**. Note that this function is called with *all* possible variations calculated in Algorithm 7.4.

Afterward, we iterate over all vertices of the TFG in Line 4. For each applied uncertainty, we check whether this uncertainty affects the current vertex in Line 6. An uncertainty affects a vertex *iff* it causes a variation in an architectural element that is represented by the vertex. The underlying mapping of the data flow analysis framework presented in Section 7.2 is similar to the mapping between architectural models and DFDs discussed in Section 6.5. In our running example, uncertainty in the data processing (**U2**) affects all vertices in all TFGs representing this data processing. If we find such an uncertainty *impact*, we remove the uncertainty from the list of applied uncertainties in Line 7.

The procedure ends if, at any point in the iteration over all vertices of a TFG, the list of applied uncertainties gets empty. This is the case if all uncertainties considered in the given variation are relevant to the TFG. Thus, the variation is relevant for the *graph-aware* analysis and is returned in Line 11. If not all uncertainties that have been varied are relevant, an empty set is returned instead in Line 14.

We explain the interplay of Algorithm 7.4 and Algorithm 7.5 based on the the three TFGs of our running example, shown in Table 7.4. Here, the analyzing the first TFG representing the support contact request and the last TFG representing the item purchase are the edge

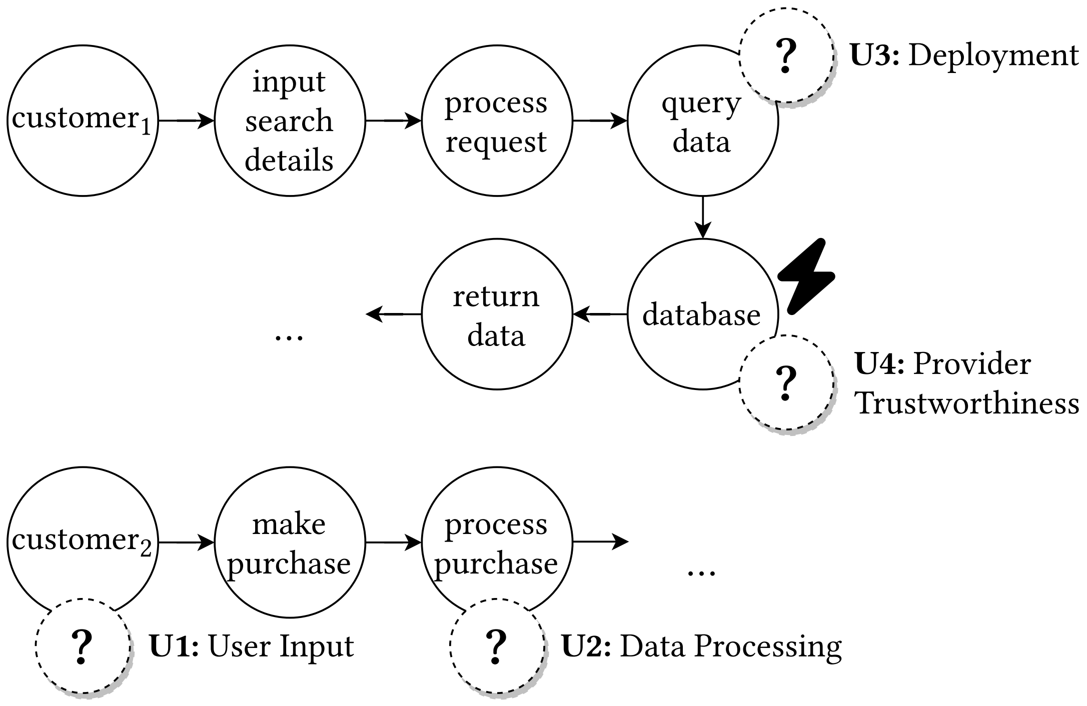


Figure 7.9.: Excerpts of two Transpose Flow Graphs (TFGs) from the running example with annotated uncertainty impacts and a confidentiality violation, marked with a lightning bolt.

cases of the filter algorithm. The first TFG has no relevant uncertainty source. Thus, Algorithm 7.5 always returns an empty list except for the single variation, where no uncertainty is applied. In this case, Line 6 trivially evaluates to *true* as the list already was empty. The third TFG is affected by all four uncertainty sources **U1** – **U4**. Thus, every permutation of the uncertainties creates a variation that is relevant for the TFG and the filter has no effect. Regarding the second TFG, only four variations are relevant, i.e., the variations that represent the scenarios of Uncertainty **U3** and Uncertainty **U4**, Table 7.4. Only these variations are returned by the filter algorithm and analyzed using the data flow analysis. This also again shows the reduced complexity of the *graph-aware* analysis (**L3**), as for most TFGs in the running example, not all variations are relevant.

The information about uncertainty sources, filtered variations, TFGs, and confidentiality violations enables the tracing approach. For each confidentiality violation, we can store the related TFG and which variation has been applied in the analysis process. The reference to the variation provides the knowledge about considered uncertainty sources and their properties. This represents the information categories **1.** – **8.** from Table 7.1. Combined, this provides software architects with comprehensive knowledge to better understand the origin of the confidentiality violations and to apply appropriate measures.

We illustrate this with one variation of the uncertainty sources of the running example. Figure 7.9 shows excerpts of the TFGs that represent the querying of items and the making of purchases. The annotated uncertainty represents the impact of the uncertainty due to changes in the architectural model. For instance, in this variation, the deployment location is the *Cloud Service* and the provider is classified as *Suspicious*, see Table 7.2. This uncertainty would cause a confidentiality violation when the data is stored in the database, as marked with the lightning bolt. As discussed with Table 7.4, this represents one of the 48 possible variations. This confidentiality violation can thus be traced back to the uncertainties **U3** and **U4**. The tracing follows the data flow in inverse direction, and could thus be interpreted as a reverse uncertainty impact analysis. The graphical representation of the TFGs in Figure 7.9 also shows the independence to the other uncertainty sources, e.g., **U1**, and **U2**. They affect other vertices of another TFG and can thus be safely ignored when interpreting the confidentiality violation in the upper TFG. This benefits the understanding of software architects and simplifies both further analyses and the mitigation.

Graph-aware data flow analysis (**L3**) minimizes the analysis complexity while enhancing the expressiveness of the analysis results. As the presented approach for tracing uncertainty builds on variation modeling [168, 265], any architectural element can be varied. This enables an uncertainty type-agnostic approach. However, this approach has two major shortcomings. First, variation modeling still requires some expert knowledge from the software architects. Although they do not have to manually model the variations or alter the DFDs by hand, they have to specify the correct variation types and variation points. This requires them to understand which variation points match the uncertainty sources they want to analyze. Failing to accurately represent the uncertainty in the variation model can impair or even void the analysis results [103]. This discussion is similar to the extension of the uncertainty impact analysis shown in Section 6.6.

Second, the approach ignores uncertainty interactions. For instance, if the *Database Service* is deployed on-premise, Uncertainty **U4** about the provider's trustworthiness has no effect. Because the uncertainty filter presented in Algorithm 7.5 does not consider this, more varied TFGs are analyzed than required. In this example, the variation of a *suspicious* cloud provider and a deployment *on premise* can be safely ignored as there would be no data flow to the cloud provider in the first place. This marks the difference between a *graph-aware* analysis (**L3**) and an *impact-aware* analysis (**L4**). The difference becomes visible when comparing Figure 5.2 with Figure 7.9. The former shows uncertainty as part of the DFD while the latter only considers changes in architectural elements that are related to DFD nodes. Generating and filtering variations cannot represent transitive effects of uncertainty impacts within a DAG. To address this, we consider uncertainty as a first-class concern in the data flow analysis, creating an *impact-aware* analysis (**L4**).

Finding: Considering Transpose Flow Graphs (TFGs) in the confidentiality analysis of uncertainty scenario combinations surpasses the analysis capabilities of analyses that only consider all possible variations. Still, the resulting analysis is limited by the expressiveness of variation modeling. To overcome this limitation regarding the transitive impact, uncertainty has to be presented as first-class concern within Data Flow Diagrams (DFDs) and the data flow analysis.

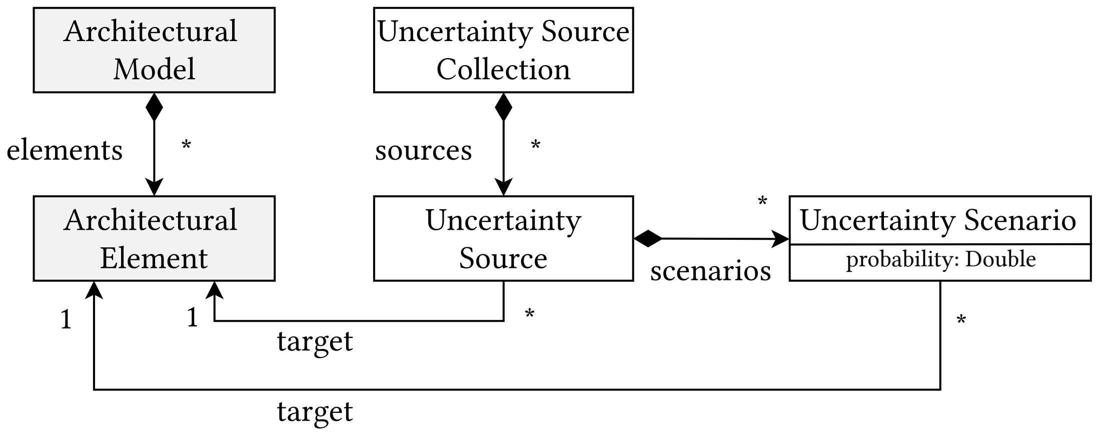


Figure 7.10.: Overview of the meta model for expressing uncertainty sources and scenarios in architectural models. Existing elements of the software architecture are highlighted gray.

7.5.2. Impact-Aware Data Flow Analysis Under Uncertainty

More than a decade ago, Garlan [80] proposed to consider uncertainty as a first-class concern in software engineering. We follow this proposal to define the last approach of this chapter, an uncertainty type-agnostic and *impact-aware* data flow analysis (**L4**). First, we present a meta model for representing uncertainty sources as part of the architectural model. This addresses the shortcoming regarding expert knowledge of variation modeling [101, 168, 265]. Afterward, we introduce the notion of a Nondeterministic Data Flow Diagram (NDFD) [104] that incorporates uncertainty as a first-class entity within the DFD. We discuss how to analyze such NDFDs and handle simple uncertainty interactions in DAGs while identifying confidentiality violations. Last, we present our tooling 🔧 ABUNAI that comprises uncertainty modeling and automated analysis support.

In presenting the previous approaches for analyzing structural uncertainty and environmental uncertainty, and tracing uncertainty, we discussed different approaches to express uncertainty as part of the architectural model. This includes design decision models [143, 144, 266], FISs [37, 140], and variation modeling [101, 168]. They share the common shortcoming of requiring expert knowledge to bridge the gap between identifying uncertainty sources and expressing them as part of the software architecture. We close this gap by providing a meta model that connects the five uncertainty types presented in Chapter 5 to the architectural model. This approach is similar to the modeling of uncertainty for impact analysis presented in Section 6.2, where we already addressed a similar issue.

Figure 7.10 shows the meta model for modeling uncertainty sources and scenarios as part of the software architecture. We use the same terminology as the uncertainty classification presented in Chapter 5, i.e., uncertainty sources and scenarios, instead of design decisions, or variations. To introduce this meta model, we only refer to the concept of an *Architectural Model*, which comprises any number of *Architectural Elements*. Software architects define an *Uncertainty Source Collection* alongside the architectural model. This collection consists of *Uncertainty Sources* that consist of *Uncertainty Scenarios*. Each source references one *Architectural Element*, which represents its default scenario. Until this point, the software

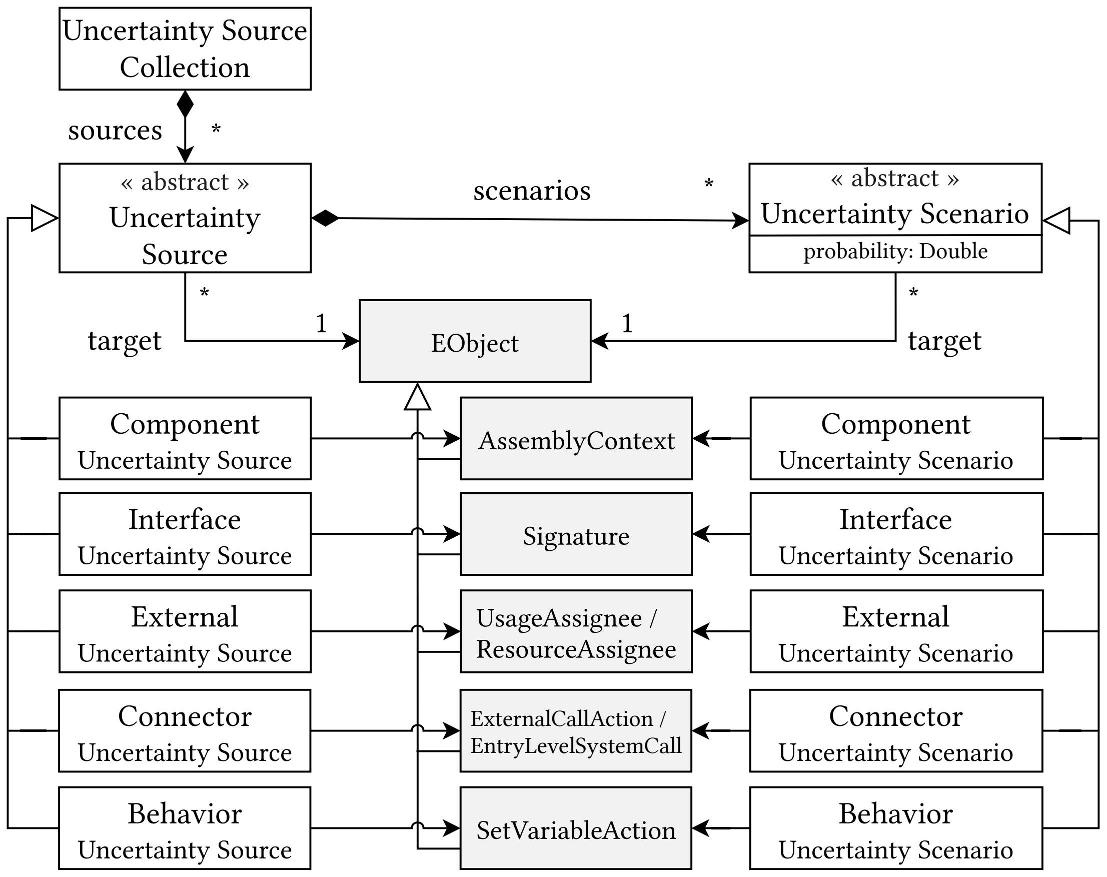

Uncertainty Source Collection
sources
*
« abstract »
Uncertainty Source
scenarios
*
« abstract »
Uncertainty Scenario
probability: Double
*
target
1
EObject
1
*
target
Component Uncertainty Source
AssemblyContext
Component Uncertainty Scenario
Interface Uncertainty Source
Signature
Interface Uncertainty Scenario
External Uncertainty Source
UsageAssignee / ResourceAssignee
External Uncertainty Scenario
Connector Uncertainty Source
ExternalCallAction / EntryLevelSystemCall
Connector Uncertainty Scenario
Behavior Uncertainty Source
SetVariableAction
Behavior Uncertainty Scenario

Figure 7.11.: Meta model of uncertainty sources and scenarios in the Palladio Component Model (PCM).

architecture is annotated with uncertainty sources like in the uncertainty impact analysis, see Chapter 6. We extend this annotation by modeling any number of *Uncertainty Scenario* that also refer to one *Architectural Element.* These scenarios represent alternative scenarios of the *Uncertainty Source* any can additionally have a probability.

Depending on the type of modeled uncertainty, see Table 5.3, software architects can refine the model. *Recognized Ignorance* is modeled as an *Uncertainty Source* that references an *Architectural Element. Scenario Uncertainty* is modeled as a *Uncertainty Source* with any number of *Uncertainty Scenarios* that also reference *Architectural Elements. Statistical Uncertainty* is modeled by enriching the *Uncertainty Scenarios* with probabilities between 0 and 1. We require that all probabilities add up to 1. The probability of the default scenario expressed by the *Uncertainty Source* is 1 minus the sum of the probabilities of all of its scenarios. However, providing probabilities is optional, the field can be left empty to describe *Scenario Uncertainty.* Depending on the available information, all options of the category *Uncertainty Type* presented in Section 5.3 can be modeled and mixed. In our running example, Uncertainty **U4** is modeled by creating an *Uncertainty Source* that refers to the *Cloud Service's* trustworthiness with the alternative *Uncertainty Scenario* of being suspicious, see Table 7.2. A high level of trust in the provider can be expressed, for example, by a probability of 0.1 for the alternative scenario of a suspicious provider.

#	Uncertainty type	Source target PCM element	Alternative scenario targets
U1	Connector	*BuyEntryLevelSystemCall*	*ErroneousEntryLevelSystemCall*
			MaliciousEntryLevelSystemCall
U2	Behavior	*UserDataProcessing*	*BrokenEncryptionProcessing*
			BrokenValidationProcessing
			EverythingBrokenProcessing
U3	Component	*DatabaseServiceOnPremise*	*DatabaseServiceInCloud*
U4	External	*ProviderTrusted*	*ProviderSuspicious*

Table 7.5.: Modeling the uncertainty sources and their scenarios in the running example.

We extend this meta model and apply it to the PCM, thereby replacing the *Architectural Elements* of Figure 7.10 with concrete PCM element types. Figure 7.11 shows the resulting meta model. We simplify some of the elements for the sake of clarity, e.g. by combining *UsageAssignees* and *ResourceAssignees*, which are modeled separately. Similarly to the overview, a *Uncertainty Source Collection* contains any number of *Uncertainty Sources* that contain any number of *Uncertainty Scenarios.* Both the *Uncertainty Source* and the *Uncertainty Scenario* target exactly one architectural element. Because the PCM is based on the Eclipse Modeling Framework (EMF), the common super type is *EObject*.

The five uncertainty types of the category *Architectural Element Type*, see Table 5.2, have their corresponding uncertainty sources and uncertainty scenarios. All sources inherit from the abstract *Uncertainty Source* and all scenarios from the abstract *Uncertainty Scenario*, respectively. For the sake of clarity, we leave out details like the inherited *probability* field and the *scenarios* relation between all sub types. All sources and scenarios target an PCM element type, e.g., *Component Uncertainty Sources* and *Component Uncertainty Scenarios* target PCM *AssemblyContexts.* This ensures valid model instances, as both sources and scenarios are required to reference the same PCM element type, e.g., different *AssemblyContexts*, or *Signatures.* Some types of *Uncertainty Sources* and *Uncertainty Scenarios* can refer to more than one PCM element, e.g., *Connector* uncertainty, and *External* uncertainty. In our implementation of this meta model [98], we realize this using sub types of *Connector* uncertainty and *External* uncertainty, respectively, to ensure type safety. Note that the referenced PCM elements are slightly different compared to the annotated elements of uncertainty impact analysis, as discussed in Section 6.2.

We apply this meta model to our running example to illustrate it with all uncertainty sources **U1** – **U4**. Table 7.5 shows the targeted elements of all uncertainty sources and also the targeted elements of alternative scenarios. Thereby it realizes the scenarios Table 7.2 and connects this collection to a concrete architectural model described based on the PCM. The restriction to concrete architectural elements simplifies the modeling of uncertainty sources and their scenarios and also minimizes the risk of an erroneous specification. Compared to, e.g., variation modeling [168, 265], this bridges the gap between the classification of software-architectural uncertainty and the software architecture.

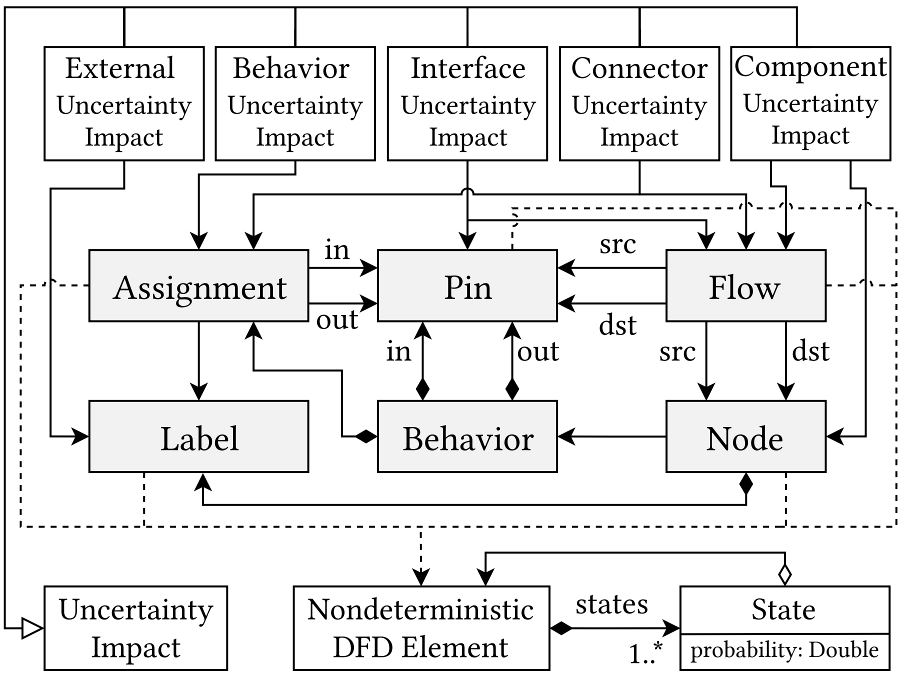

External Uncertainty Impact
Behavior Uncertainty Impact
Interface Uncertainty Impact
Connector Uncertainty Impact
Component Uncertainty Impact
Assignment
in
out
Pin
src
dst
Flow
in
out
src
dst
Label
Behavior
Node
Uncertainty Impact
Nondeterministic DFD Element
states
1..*
State
probability: Double

Figure 7.12.: Meta model of Nondeterministic Data Flow Diagrams (NDFDs).

Finding: Modeling uncertainty sources and uncertainty scenarios that reference elements of the architectural model connects the classification of software-architectural uncertainty to architectural models. This provides the foundation for analyzing uncertainty as first-class entity in software architecture.

To consider uncertainty not only as first-class entity within the software architecture but also in architecture-based data flow analysis, we need means to represent the impact in DFDs. To this end, we discussed the mapping and the representation of uncertainty to DFDs and DAGs in Section 5.5. We also introduced the notion of *primary* uncertainty that affects vertices and *secondary* uncertainty that affects edges. By connecting this notion to the previously introduced meta model for modeling uncertainty sources and scenarios, we are able to define an *impact-aware* data flow analysis (**L4**). Uncertainty is sometimes defined as "any departure from the unachievable ideal of complete determinism" [263], see Section 2.1. We build on this idea by describing the lack of determinism and define the notion of Nondeterministic Data Flow Diagrams (NDFDs). Put simply, we extend Data Flow Diagrams (DFDs) to incorporate uncertainty as first-class entity that is represented by nondeterminism. Here, we also benefit from the comprehensive foundations regarding the representation of the impact of uncertainty in DFD, provided in Section 6.2.

Figure 7.12 shows the meta model of NDFDs, thereby extending the meta model of DFDs with uncertainty impacts shown in Figure 6.3. Every DFD element originating from the unified modeling primitives [235] and highlighted in gray represents a *Nondeterministic Data Flow Diagram Element*. Such an element has one or more *States* that reference the element, or other elements. Trivially, an element with only one *State* that references the element itself represents determinism. Thus, every DFD is also a valid NDFD, which lacks nondeterminism, i.e., uncertainty. Uncertainty is represented by multiple *States* that can additionally have a *probability*. Regarding the probability, the same rules apply as for the meta model for uncertainty and scenarios in the architectural model.

This meta model enables the concise representation of uncertainty in DFDs. For instance, Uncertainty **U4** about the provider trustworthiness represents *External* uncertainty. It can be expressed by an nondeterministic *Label* with two states: *Trustworthy* and *Suspicious*. The same applies to the other uncertainty sources, e.g., Uncertainty **U2** about the processing can be represented by multiple alternative *Assignments*.

We reuse the mapping of uncertainty sources to impact locations within the DFD from Section 5.5, and the mapping of PCM models to DFDs presented in Section 7.2. Additionally, every *Uncertainty Scenario* from the meta model presented in Figure 7.11 is mapped to a *State* of the NDFD meta model. The state representing the default scenario of the *Uncertainty Source* references the DFD element itself. All alternative scenarios reference the DFD elements that represent the targeted architectural elements of these scenarios. The probability of a *State* corresponds to the probability of the *Uncertainty Scenario*.

We can apply this mapping to the modeled uncertainty sources and uncertainty scenarios of the running example, shown in Table 7.5. The *EntryLevelSystemCall* that is referenced by Uncertainty **U1** is mapped to a nondeterministic DFD element and referenced by the default state of this element. The DFD element has additional states to also represent erroneous and malicious user input. Because Uncertainty **U1** represents *Connector* uncertainty, the nondeterministic DFD elements are flows and assignments that set the labels for valid, erroneous, or malicious input. However, due to the uncertainty, we cannot know which assignment is applied, i.e., nondeterminism.

Finding: Nondeterministic Data Flow Diagrams (NDFDs) consist of all Data Flow Diagrams (DFDs) elements, which have one or multiple states. These states represent uncertainty as a first-class concern in modeling and data flow analysis.

The concept of states in NDFDs can be transferred to DAGs. This enables the application of the data flow analysis that operates on TFGs, which are DAGs. Here, we use the notion of *primary* and *secondary* uncertainty. As discussed in Section 5.5, the five uncertainty types can be represented by either *primary* or *secondary* uncertainty. *Component* uncertainty, *Connector* uncertainty, and *Interface* uncertainty affect the data flow, i.e., the edges of a DAG. *External* uncertainty and *Behavior* uncertainty affect nodes, i.e., the vertices of a DAG. We use this mapping of the elements of a DFD—that are also elements of a NDFD—to simplify the data flow analysis. In mapping PCM elements to TFGs, we map the uncertainty sources and scenarios to *primary* and *secondary* uncertainty. The resulting DAG with these uncertainties represents a NDFD and comprises uncertainty as a first-class entity. To repeat from Section 5.5, a DAG $G = (V, E)$ consists of vertices V and edges E. All *primary* uncertainty impacts form a subset of all vertices $V_u \subseteq V$. All *secondary* uncertainty impacts form a set $E_u = \{E_{u_1}, \ldots, E_{u_n}\}$, where each set comprises alternative edges.

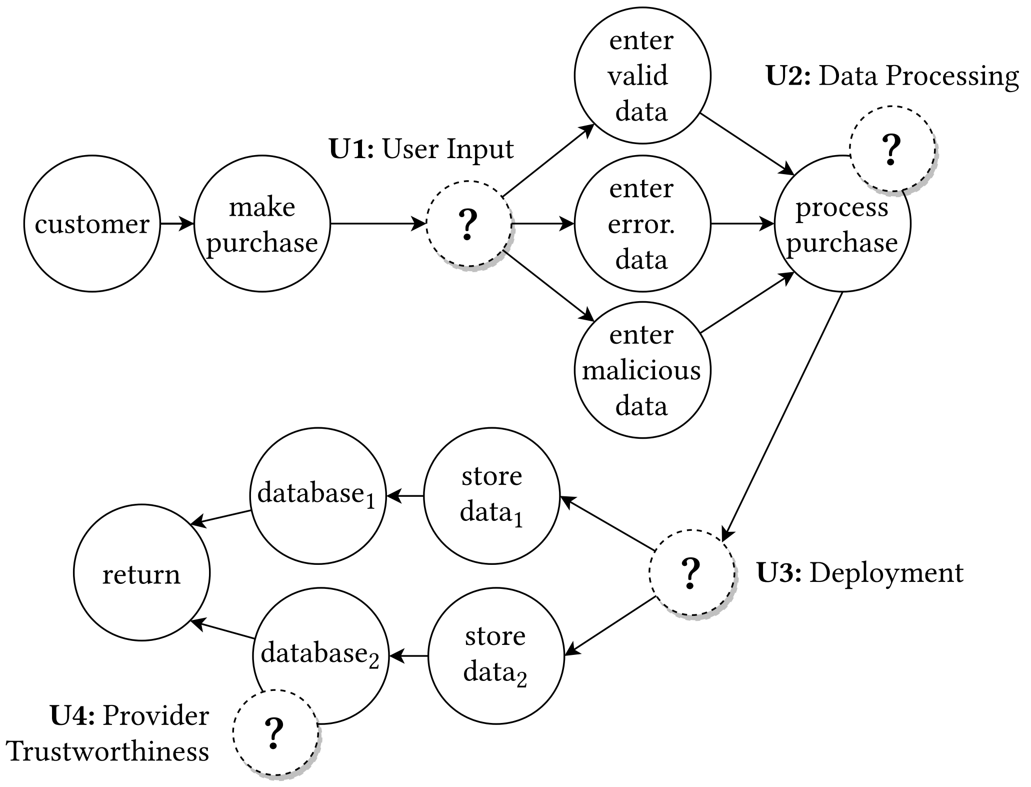


Figure 7.13.: Directed Acyclic Graph (DAG) of the Transpose Flow Graph (TFG) representing the item purchase in the running example, with primary and secondary uncertainty, denoted by question marks.

The first TFG of the support contact request contains no uncertainty and is thus not relevant, see Table 7.4. The DAG of the second TFG is shown while discussing *primary* and *secondary* uncertainty in Figure 5.2. To make the list of DAGs of the running example complete, Figure 7.13 shows the DAG of the third TFG of the running example, which represents the purchase of items. This DAG contains all four uncertainties **U1** – **U4**.

The *secondary* uncertainties are shown as alternative edges. As discussed above, the *Connector* uncertainty of the user input (**U1**) and the *Component* uncertainty of the deployment (**U3**) represent *secondary* uncertainty. The *primary* uncertainties are annotated to vertices. The *Behavior* uncertainty of the data processing (**U2**) and the *External* uncertainty of the provider trustworthiness (**U4**) represent *primary* uncertainty. The scenarios of the *secondary* uncertainty are visible in the graphical representation of the DAG, e.g., the alternative types of user input in Uncertainty **U1**. The scenarios of *primary* uncertainty are hidden in this view, because they affect DFD elements that are represented by the vertices, e.g., the *Label* of the *database$_2$* being *Trustworthy*, or *Suspicious*. Here, we refer to Section 7.2 for our introduction on TFGs and their relation to DAGs.

The TFG shown in Figure 5.2 is also suitable to illustrate the mapping from an architectural model with uncertainty sources and scenarios to NDFDs and DAGs. Table 7.5 shows three different scenarios regarding Uncertainty **U1** of the user input. As this is *Connector* uncertainty and modeled with three alternative *EntryLevelSystemCalls*, the resulting DAG also comprises the corresponding vertices, e.g., *enter valid data* and *enter malicious data*,

Algorithm 7.6 Algorithm for impact-aware data flow analysis under uncertainty

```
1: procedure analyzeImpactAware($model$, $constraint$, $uncertainties$)
2:     $violations \leftarrow \emptyset$
3:     $input \leftarrow$ retrieveAllTFGs($model$)
4:     $output \leftarrow \emptyset$
5:     for $tfg \in input$ do                                   ▷ Processing of all uncertainties
6:         $sources \leftarrow$ getSources($tfg$)               ▷ Start the evaluation at each source
7:         for $source \in sources$ do
8:             $output \leftarrow output \cup$ processVertex($source, uncertianties, \{tfg\}$)
9:         end for
10:    end for
11:    for $tfg \in output$ do                  ▷ Confidentiality analysis on the resulting TFGs
12:        $violations \leftarrow violations \cup$ analyze($tfg, constraint$)
13:    end for
14:    return $violations$
15: end procedure
```

with a *secondary* uncertainty. In the NDFD, this is represented by a nondeterministic element with three states, which translates to this part of the DAG. The same applies to Uncertainty **U3** with the two deployment options. The *primary* uncertainties **U2** and **U4** have been discussed in the previous paragraph. Here, the scenarios are not visible in the DAG, because it only represents the *Nodes* of an NDFD as vertices, and *primary* uncertainty affects these vertices. Nevertheless, the label propagation presented in Section 7.2, is presented with alternative DFD elements, e.g., alternative *Assignments* representing the data processing alternatives in Uncertainty **U2**. In sum, the notion of TFGs remains suitable to model and analyze DAG, also under uncertainty.

We build on this representation of NDFDs as *primary* and *secondary* uncertainty in DAGs to define *impact-aware* data flow analysis (**L4**). Based on this discussion, differentiating between the different analyses is straightforward. *Scenario-aware* data flow analysis (**L2**) iterates over all scenarios, without considering TFGs, as shown in Algorithm 7.3. *Graph-aware* data flow analysis (**L3**) iterates over all TFGs, without considering uncertainty affecting individual vertices, as shown in Algorithm 7.4. *Impact-aware* data flow analysis (**L4**) iterates over vertices and considers uncertainty as a first-class entity in the analysis process. This again reduces the number of TFGs that have to be analyzed, and it enables the consideration of the transitive impact of uncertainty and uncertainty interactions.

Algorithm 7.6 shows the *impact-aware* data flow analysis. It consists of two steps. First, all uncertainties in all TFG are processed. By following the flow of data starting from each source, every impact of uncertainty has to be considered in the processing in Line 8. We start at the sources because uncertainty follows the flow of data, as found in Section 6.4. Thus, processing the impact of uncertainty in this direction simplifies the analysis. The resulting TFGs are input to the confidentiality analysis in Line 12. This analysis is executed

Algorithm 7.7 Algorithm for processing vertices with primary and secondary uncertainty

```
 1: procedure processVertex(vertex, uncertainties, tfgs)
 2:     if isInPrimaryUncertaintySet(vertex, uncertainties) then
 3:         for state ∈ getStates(vertex, uncertainties) do
 4:             for tfg ∈ tfgs do
 5:                 newtfg ← applyPrimaryUncertainty(vertex, state, tfg)
 6:                 tfgs ← tfgs ∪ {newtfg}
 7:             end for
 8:         end for
 9:     end if
10:     for edge ∈ getOutgoingEdges(vertex) do
11:         if isInSecondaryUncertaintySet(edge, uncertainties) then
12:             for tfg ∈ tfgs do
13:                 newtfg ← applySecondaryUncertainty(vertex, edge, tfg)
14:                 tfgs ← tfgs ∪ {newtfg}
15:             end for
16:         end if
17:     end for
18:     for successor ∈ getSuccessors(vertex) do
19:         tfgs ← tfgs ∪ processVertex(successor, uncertainties, tfgs)
20:     end for
21:     return tfgs
22: end procedure
```

separately on every resulting, independent TFG. The collected violations are returned in Line 14. Algorithm 7.7 shows the processing of *primary* and *secondary* uncertainty in *impact-aware* data flow analysis. Put simply, instead of combining all possible scenarios of all uncertainties (**L2**), or all scenarios on one TFG (**L3**), we recursively apply only the scenarios that have an actual impact on each other (**L4**). The *processVertex* function is called with a vertex, initially the source of a TFG, all uncertainties, and a set of TFGs, initially only the TFG itself.

We first handle *primary* uncertainty, starting in Line 2. If the vertex is in the set of vertices affected by *primary* uncertainty $V_u \subseteq V$, we apply every state from the NDFD to every TFG in the set, thereby creating and adding new TFGs. Due to the set semantics, the original TFG representing the default scenario is not contained twice. For instance, the *primary* uncertainty **U2** with 4 scenarios—and thus, 4 states of the NDFD element—causes 4 new TFG. Afterward, we handle *secondary* uncertainty, starting in Line 10. Here, we filter all outgoing edges for those that are in the set of secondary uncertainty sets, i.e., edges in one $E_{u_i} \in E_u$. For every identified, outgoing edge, we apply the uncertainty to all TFGs in Line 13, thereby creating copies of all TFGs with only one of the outgoing edges in E_{u_i}. For instance, the *secondary* uncertainty **U1** with 3 states as shown in Figure 7.13 causes three copies per TFG, with only one outgoing edge from the *make purchase* vertex.

Last, we continue with all successors of the analyzed vertex until a sink is reached. Because this recursive call to the *processVertex* function in Line 19 contains the current set of TFGs, the transitive impact of uncertainty is considered. Note that the realization of this algorithm in our data set [98] comprises optimizations of this algorithm, similar to the algorithms shown in Section 6.3. In our exemplary DAG shown in Figure 7.13, this results in a lower number of resulting TFGs compared to *graph-aware* analysis (**L3**). After the processing of the vertex *process purchase*, we have 12 TFGs, caused by the 3 scenarios of Uncertainty **U1** times the 4 scenarios of Uncertainty **U4**. However, after evaluating Uncertainty **U3**, only the lower branch is affected by Uncertainty **U4**. Thus, this uncertainty is only applied to those TFGs that represent the state where the data flows to this branch instead of to all TFGs. To achieve this, we have to consider the transitive effect of uncertainty in the analysis, which requires the representation of uncertainty as first-class entity.

Finding: Representing Nondeterministic Data Flow Diagrams (NDFDs) as Directed Acyclic Graphs (DAGs) with uncertainty as a first-class entity enables the consideration of transitive effects of uncertainty and interactions of uncertainty sources. Additionally, the analysis scalability is enhanced because the number of required Transpose Flow Graphs (TFGs) is reduced.

In the introduction of this chapter, we highlighted the need for automated confidentiality analysis. We realize the concept of an *impact-aware* data flow analysis based on the framework presented in Section 7.2. Our tooling ABUNAI provides modeling support for software architects and an automated, uncertainty *impact-aware* analysis of confidentiality. The implementation is part of the data set [98].

Similarly to the tool support for uncertainty impact analysis, we build on the Palladio tooling [205, 207] that offers a meta model, the PCM, and graphical editor support. Our Java-based open-source implementation realizes the algorithms presented above. The analysis requires a PCM model and an uncertainty model that comprises uncertainty sources and scenarios as input. Our implementation is closely oriented to the meta model shown in Figure 7.11. The analysis provides a number of helpful outputs for software architects. Besides identifying confidentiality violations, combinations of uncertainty scenarios can be generated, compared, and related to the violations. This includes calculating the overall probability by multiplying the probability of analyzed uncertainty scenarios. Additionally, metrics regarding the analysis complexity are shown. It is also possible to build on the generated TFGs to create further analyses or end-to-end approaches [273]. We showcase ABUNAI based on the running example in Appendix C.

Finding: The uncertainty impact-aware analysis of confidentiality can be fully automated based on the Palladio Component Model (PCM) and the existing tooling of the Palladio approach. Uncertainty sources and scenarios are modeled as first-class entities within the software architecture and analyzed while considering the transitive impact of uncertainty and uncertainty interactions. The resulting analysis can be used for the creation of end-to-end approaches.

7.6. Complexity of Data Flow Analysis Under Uncertainty

Throughout this chapter, we discussed the complexity and scalability of the different approaches for data flow analysis under uncertainty, i.e., *scenario-aware* analysis (**L2**), *graph-aware* analysis (**L3**), and *impact-aware* analysis (**L4**). Data flow-based confidentiality analysis, such as our framework presented in Section 7.2, scale with the number of analyzed data flows, i.e., the number of TFGs [36, 231]. Additionally, the number of derived and propagated labels affects the analysis's scalability. Thus, the underlying goal is to minimize the number and size of the analyzed TFGs under uncertainty without affecting the analysis results in terms of accuracy. Furthermore, uncertainty follows the data flow, see Section 6.4, and can affect other uncertainties on this way, see Section 6.7. Encountering this challenge and analyzing combinations of uncertainty scenarios can lead to a combinatorial explosion [143]. To conclude the discussion about the awareness levels started in Section 7.3, we revisit the complexity of the approaches **L2** – **L4**. We skip the first two levels as they are not expedient to handle uncertainty in data flow analysis. This addresses Problem **P3**.

Note that we do not use the Big O notation [141] in this section. As discussed previously, the worst-case performance of all three levels (**L2** – **L4**) is shown in software systems with only one data flow and interdependent uncertainty sources. In this case, all possible variations have to be tested on all TFGs, i.e., *scenario-awareness*. This results in a worst-case asymptotic growth of $O(n^k)$, where n is the number of uncertainty scenarios and k is the number of uncertainty sources. We do not consider the number of TFGs here, as it is negligible for large k. Ultimately, this applies to all three levels and their algorithms. Thus, we focus on the actual count of TFGs to analyze in this section. Note that we also do not discuss the complexity of applying uncertainty to the TFGs in preparation for the data flow-based confidentiality analysis. All three algorithms can be implemented based on Depth-First Search (DFS) that is repeated for each uncertainty scenario, i.e., $O(n \cdot (V + E))$, where n represents the total number of scenarios, V the vertices and E the edges. Compared to the large growth of TFGs to analyze uncertainty and compared to the run time of the confidentiality analysis [231], this effect is also negligible.

Figure 7.14 shows a simplified overview of all three TFGs of the running example. We replace the vertices' names with numbers but keep the *primary* uncertainty annotated to vertices, and *secondary* uncertainty annotated to edges. Note that we use a slightly different notation in this section to simplify the mathematical notation, e.g., u_1 refers to **U1**. This figure is based on Table 7.2, showing the scenarios of uncertainties **U1** – **U4**, and Table 7.4, showing the mapping of uncertainty sources to TFGs. More detailed views of the second and third TFGs are shown in Figure 5.2 and Figure 7.13, respectively. We also include the first TFG representing the support contact request of the *Customer*. However, this TFG is not affected by uncertainty, see Chapter 3.

Let $G = \{g_1, g_2, \ldots, g_n\}$ where $n \geq 1$ be the nonempty finite set of all TFGs extracted from the architectural model, as introduced in Section 7.2, and let $|G|$ be its size. In our running example, $G = \{g_1, g_2, g_3\}$ and $|G| = 3$. Put simply, there are 3 extracted TFGs. Let $U = \{u_1, u_2, \ldots, u_m\}$ where $m \geq 1$ be the nonempty finite set of all uncertainties, both *primary* and *secondary*, and let $|U|$ be the its size. In our running example, $U = \{u_1, u_2, u_3, u_4\}$ and

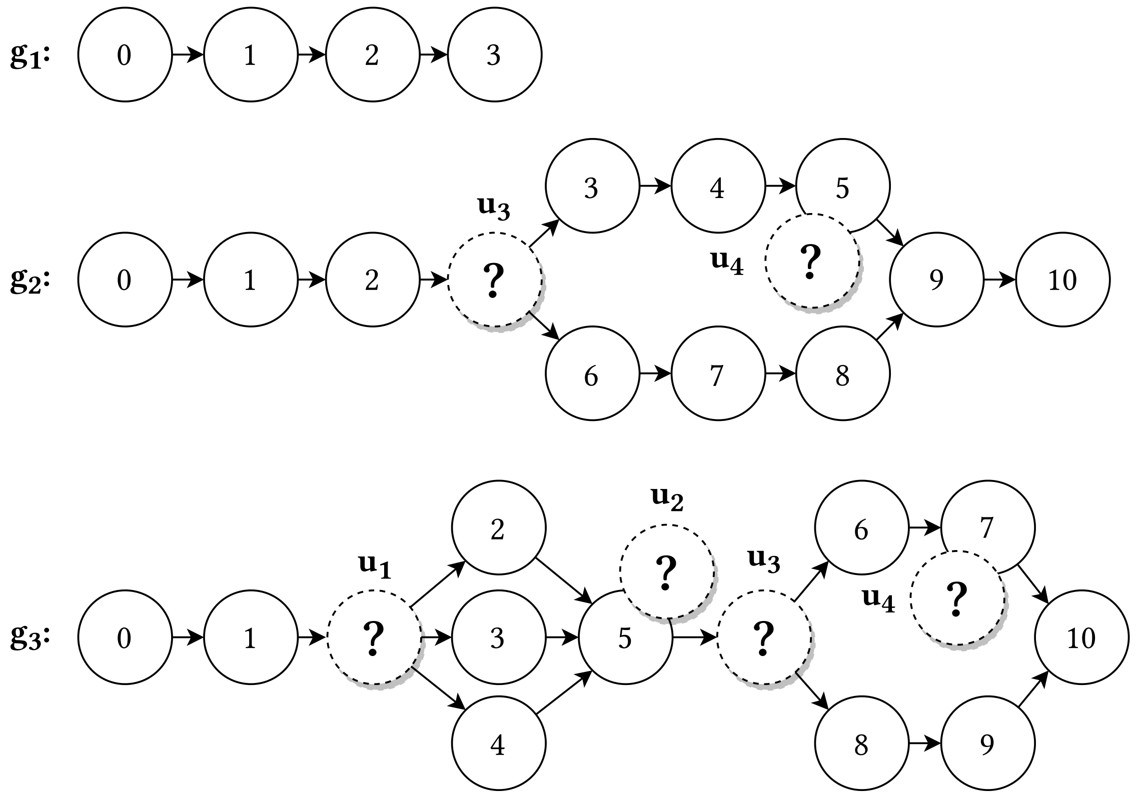


Figure 7.14.: Simplified overview of all three Transpose Flow Graphs (TFGs) of the running example with annotated primary and secondary uncertainties.

$|U| = 4$. Put simply, there are 4 uncertainties. We further split U in $U_P \subseteq U$ to represent all *primary* uncertainties and $U_S \subseteq U$ to represent all *secondary* uncertainties in U. This also means that $U = U_P \cup U_S$ and $U_P \cap U_S = \emptyset$. In our running example, $U_P = \{u_2, u_4\}$ and $U_S = \{u_1, u_3\}$. For each uncertainty source $u \in U$, let $S_u = \{s_{u_1}, s_{u_2}, \ldots, s_{u_p}\}$ where $p \geq 1$ be the nonempty finite set of all uncertainty scenarios of u and let $|S_u|$ its size. As discussed with NDFDs in Section 7.5, $p \geq 1$ is trivially satisfied, as every uncertainty source has at least one default scenario. In our running example, $S_{u_1} = \{s_1, s_2, s_3\}$ and $|S_{u_1}| = 3$. Put simply, the first uncertainty has three uncertainty scenarios, i.e., valid input, erroneous input, and malicious input, see Table 7.2.

Each uncertainty scenario brings variation into a TFG and thus has to be analyzed separately. This also applies to combinations of uncertainties, where every combination of all relevant scenarios of all relevant uncertainties has to be analyzed separately, i.e., the Cartesian product [76]. To minimize the analysis runtime and maximize its scalability, it is thus important to make a good estimation of what is *relevant*. Otherwise, the combination of scenarios can lead to the already discussed combinatorial explosion [143]. We define N as the number of TFGs that have to be analyzed in total. In the following, we define three formulas to calculate this number, based on the underlying approach. We define N_S for the *scenario-aware* analysis (**L2**), N_G for the *graph-aware* analysis (**L3**), and N_I for the *impact-aware* analysis (**L4**). Note that all three analysis approaches find the same confidentiality violations due to uncertainty, i.e., the accuracy is not affected. However, the number of TFGs that have to be analyzed can differ.

The *scenario-aware* analysis (**L2**) does not consider TFGs but only combinations of scenarios. Thus, for every possible scenario combination, every TFG has to be analyzed which represents the least optimized approach. We denote:

$$N_S = |G| \cdot \prod_{u \in U} |S_u|$$

This is equivalent to generating all possible scenario combinations, applying them to all TFGs, and analyzing them, as introduced in Algorithm 7.3. In our running example with 3 TFGs and 4 uncertainty sources, this means we have to analyze $N_S = |G| \cdot |S_{u_1}| \cdot |S_{u_2}| \cdot |S_{u_3}| \cdot |S_{u_4}| = 3 \cdot 3 \cdot 4 \cdot 2 \cdot 2 = 144$ TFGs. This is equivalent to analyzing all 3 TFGs in all 48 scenarios, as discussed in Section 7.4.

The *graph-aware* analysis (**L3**) does take TFGs into account and thus only has to consider combinations of scenarios in the same TFG. This is based on the finding that uncertainties in different flows, i.e., different TFGs, cannot affect each other, as discussed in Section 6.7. This highly reduces the amount of TFGs that have to be analyzed, especially in larger systems with many independent data flows. To count the uncertainty sources $u \in U$ and its scenarios S_u that impact a specific TFG $g \in G$, we define the *impact function*:

$$f(g,u) = \begin{cases} 1, & \text{if there is no impact of } u \text{ on } g \\ |S_u|, & \text{otherwise} \end{cases}$$

In our running example, $f(g_2, u_3) = 2$, because the uncertainty u_3 in the TFG g_2 has 2 scenarios. To recall, the second TFG represents the querying of items and Uncertainty **U3** represents the deployment with the scenarios of being deployed on-premise or in the cloud. $f(g_1, u_3) = 1$, because the uncertainty u_3 has no effect on the first TFG. Put simply, this function yields the number of scenarios if the uncertainty is relevant for a TFG; otherwise, it returns 1. Using this function, we denote:

$$N_G = \sum_{g \in G} \prod_{u \in U} f(g,u)$$

This is equivalent to iterating over all TFGs and only generating the scenario combinations relevant for the current TFG, as introduced in Algorithm 7.4. In our example, the first TFG has no uncertainty and f always returns 1. The second TFG is only impacted by u_3 and u_4 while the third TFG is impacted by all uncertainties. In sum, this means: $N_G = (1 \cdot 1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 2 \cdot 2) + (3 \cdot 4 \cdot 2 \cdot 2) = 1 + 4 + 48 = 53$ TFGs. The number of TFGs that have to be analyzed by the *graph-aware* approach is notably smaller compared to the *scenario-aware* approach, i.e., $53 < 144$. Although the third TFG represents the worst case of combining all scenarios, the other TFGs are not affected. Note that this formula only results in the same number as the *scenario-aware* approach if all TFGs are affected by all uncertainties. Trivially, this is the case if there is only one data flow within the system. However, we assume that real-world software systems comprise more than a single data flow [102]. Such a system would only provide a single functionality via a single interface with a single use case without any branching. Thus, as TFGs are independent and an uncertainty impact can always assigned to a single TFG, we find in general:

$$N_G \leq N_S$$

The *impact-aware* analysis (**L4**) extends the *graph-aware* analysis and additionally considers the transitive effect of the uncertainty scenarios. By incrementally evaluating uncertainty scenarios, the impact of previous uncertainties in the same TFG can be considered. This enables us to analyze only those scenarios that are still relevant after other uncertainty scenarios have been applied in the direction of the data flow. We consider uncertainty interactions between the *secondary* uncertainties U_S and *primary* uncertainties U_P. If an *component* uncertainty, *interface* uncertainty, or *connector* uncertainty changes the edges of a TFG, this affects all vertices—and all *primary* uncertainties that affect such vertices—in the direction of the data flow, see Section 7.5.

The easiest way to consider this in the calculation of relevant TFGs is to process all TFGs $g \in G$. Similar to Algorithm 7.7, we replace all *secondary* uncertainties $u \in U_S$ in all TFGs $g \in G$ with their scenarios. Following the notion introduced in Section 5.5, this means iterating over all $E_{u_i} \subseteq E$, i.e., all sets of alternative edges representing a *secondary* uncertainty that form a subset of all edges of a DAG. For each edge $e \in E_{u_i}$, we create a copy of the current TFG that contains the edge e but none of the other edges in E_{u_i}. The processing continues replacing *secondary* uncertainty on the resulting TFGs until all *secondary* uncertainties have been processed. In sum, we define the function $p(G, U)$ that takes a set of TFGs G and yields a set of TFGs, where all *secondary* uncertainties in U have been processed as described above. As all uncertainties have at least one default scenario, we can say: $|G| \leq |p(G, U)|$. Note that in contrast to Algorithm 7.7, we do not additionally process *primary* uncertainty for the calculation as this uncertainty cannot alter edges in the TFG. Trivially, this means $|p(G, U)| = |p(G, U_S)|$ and $|G| = |p(G, U_P)|$.

In our running example, the second TFG g_2 has one *secondary* uncertainty u_3. Applying this processing results in replacing the uncertainty u_3 with its affected edges, thereby creating 2 new TFGs, one with the edge *2* $\rightarrow$ *3* and a second one with the edge *2* $\rightarrow$ *6*, see Figure 7.14. Thus, $|p(\{g_2\}, U)| = 2$. The third TFG g_3 has two *secondary* uncertainties u_1 and u_3, with 3 and 2 scenarios, respectively. Thus, $|p(\{g_3\}, U)| = 3 \cdot 2 = 6$. We denote:

$$N_I = \sum_{g \in p(G,U)} \prod_{u \in U_P} f(g, u)$$

This is equivalent to iterating over all vertices of all TFGs and taking the transitive impact of *secondary* uncertainties into account, as introduced in Algorithm 7.6. Note that we only calculate the product of *primary* uncertainties, as we already considered the impact of the *secondary* uncertainties in the processing $p(G, U)$. In our running example, this does affect the calculation of the second and third TFG. As discussed previously, the second TFG has one *secondary* uncertainty. After the processing of this uncertainty, only one of the two resulting TFGs is affected by the remaining *primary* uncertainty. This is the TFG that represents the deployment in the cloud (**U3**) that is affected by the provider's trustworthiness (**U4**). This effect becomes even bigger in the third TFG. Here, only half of the resulting TFGs represent a deployment in the cloud (**U3**). In sum, this means: $N_I = (1 \cdot 1) + ((1 \cdot 2) + (1 \cdot 1)) + ((4 \cdot 2) + (4 \cdot 2) + (4 \cdot 2) + (4 \cdot 1) + (4 \cdot 1) + (4 \cdot 1)) = 1 + (2 + 1) + (8 + 8 + 8 + 4 + 4 + 4) = 1 + 3 + 36 = 40$ TFGs. We use parenthesis to highlight the TFGs after the processing. This represents the minimum of scenarios and TFGs that have to be analyzed to identify all potential confidentiality violations, i.e., $40 < 53 < 144$. As we replace every *secondary* uncertainty in U with additional graphs in G, we move them from the inner product to the outer sum in the formula. Thus, we find in general:

$$N_I \leq N_G \leq N_S$$

7.7. Assumptions and Limitations

In this section, we discuss the assumptions and limitations of the different approaches of data flow analysis under uncertainty presented in this chapter. This includes assumptions of single approaches and more general assumptions made in this chapter. Note that we skip the limitations of the lower awareness levels **L0** – **L3** compared to *impact-aware* analysis (**L4**) as we already comprehensively discussed their differences throughout this chapter. The same applies to the comparison of *type-specific* and *type-agnostic* approaches.

Limitations of the data flow analysis framework Our approach for data flow analysis and the data flow analysis framework [36] are based on previous work in this field [226, 227, 232, 233, 234, 235, 236, 237]. Thus, we inherit some of the limitations. First, the analysis operates on type-level, e.g., by analyzing usage scenarios and not individual users. This restricts the expressiveness of analyzable confidentiality requirements [233]. Second, all four analysis approaches use the PCM as an architectural model as previously discussed in Section 6.8. However, as the majority of the analysis is conducted on DFDs, the concepts should be generalizable. First steps in this direction have already been taken [36]. Third, the analysis does not respect state or time which limits the expressiveness but simplifies the analysis complexity [233]. Fourth, the labels used for confidentiality analysis represent discrete value sets, and there is no support for expressing continuous values other than mapping them to discrete values [233]. We refer to Seifermann [233] for more detailed explanations of the reasoning behind these limitations. Last, cycles within architectural models only have limited effects by only being traversed once. However, there exist approaches for handling cycles in the construction of DAGs [12, 148].

Independence of data flow paths Seifermann [233] discussed the assumption of independent data flow paths in data flow analysis. Our data flow analysis framework presented in Section 7.2 follows this by assuming independence of extracted TFGs regarding the propagated labels that represent confidentiality-related characteristics of the software system. Furthermore, we also assume that the impact of uncertainty in TFGs can be analyzed independently. We argue that this assumption is reasonable. The former can lead to an overestimation of confidentiality violations, which is acceptable. The latter has been thoroughly discussed in Chapter 5 and Chapter 6.

Modeling uncertainty scenarios The majority of approaches presented in this chapter are based on the assumption that modeling uncertainty scenarios is reasonable to express their effects on software systems, e.g., using design decision models [144], variation models [265], or uncertainty models [101]. The description of uncertainty using scenarios has been discussed in the literature [11, 162, 263], see Table 2.1. Troya et al. [255] conducted a Systematic Literature Review (SLR) with 123 papers and support this assumption. They found that *design uncertainty* is usually expressed using uncertainty scenarios.

Analyzing unanticipated change We assume that the information required to model and analyze the software system under uncertainty is available to software architects. A similar assumption has been made in the underlying approach for data flow analysis [233] and also in earlier discussions of assumptions in Section 5.8 and Section 6.8. Similarly, we cannot express or analyze not yet identified uncertainty sources. Nevertheless, we provide software architects with means to express uncertainty models within the architectural models and refine this model once more knowledge is gained. The connection of data flow analysis to our catalog presented in Section 5.6 further addresses this limitation.

Multiple data flows and secondary uncertainty In this chapter, we present multiple approaches for data flow analysis under uncertainty. The increased scalability of the *graph-aware* analysis (**L3**) and *impact-aware* analysis (**L4**) is based on two assumptions on the modeled software system. First, we assume that real-world software systems comprise more than a single data flow path. As discussed previously, such a system would only provide a single functionality via a single interface with a single use case without any branching. Second, we assume the existence of *secondary* uncertainty within the software system. We argue that this is reasonable as three of the five uncertainty types introduced in Table 5.2 represent *secondary* uncertainty. If this assumption does not hold, the latter analysis approaches are still applicable. However, their benefits regarding the analysis complexity are decreased.

Limited support for uncertainty interactions Based on our findings on uncertainty propagation and UFDs presented in Section 6.7, the *impact-aware* analysis (**L4**) supports simple uncertainty interactions. However, this support is limited to the impact of *secondary* uncertainty on *primary* uncertainty. Put simply, altering the edges of a DAG can affect the flow of data to vertices and thus can also affect the impact of *primary* uncertainty on these vertices. Further uncertainty interactions can happen, e.g., between two *primary* uncertainties where one uncertainty voids the impact of another uncertainty. This limitation is similar to the overestimation of the uncertainty impact discussed in Section 6.8. Addressing this requires a higher expressiveness in the modeling and analysis as more fine-grained interactions between uncertainty and confidentiality have to be considered.

Mitigation of uncertainty We consider the automated mitigation of uncertainty as out of the scope of this thesis. This includes the automated repair of uncertainty-afflicted software systems due to confidentiality violations. Additionally, we only focus on finding all potential confidentiality violations due to uncertainty. Analysis approaches that aim for fast feedback for software architects by returning partial results could thus further increase the analysis scalability. We emphasize that advanced approaches to analysis, mitigation, and repair are expedient and that our analysis approaches can provide a comprehensive foundation for such endeavors.

Uncertainty in confidentiality requirements We do not consider any uncertainty within confidentiality requirements that can occur, e.g., due to the relation to legal assessment [39]. This would require means to express uncertainty-related variation not only as part of the software system but also in the specification of data flow constraints. This could be achieved, e.g., by extending existing Domain-Specific languages (DSLs) [36, 105]. Similar to the previous limitation, we emphasize the importance of this research but consider it to be out of scope for this thesis.

7.8. Summary and Outlook

In this chapter, we presented multiple approaches to uncertainty-aware data flow analysis to identify potential confidentiality violations due to uncertainty. This represents our third Contribution **C3** and provides an answer to ❓ **Research Question 3**.

First, we discussed the foundations required to define uncertainty-aware data flow analyses, as described with Problem **P1**. We presented an extensible framework for data flow analysis [36] that enables both black-box and white-box extensions [111, 253] of confidentiality analysis. Here, we also introduced the concepts of TFGs and label propagation. Afterward, we discussed which information is available in data flow analysis under uncertainty and how this information can be used. To that end, we specified five levels (**L0** – **L4**) of uncertainty awareness: The lack of uncertainty-awareness (**L0**), naive approaches (**L1**), scenarios-awareness (**L2**), graph-awareness (**L3**), and impact-awareness (**L4**). Uncertainty-aware data flow analyses that fall into one of these categories can be further categorized into type-specific and type-agnostic approaches.

Building on these foundations, we presented two *type-specific* approaches to data flow analysis under uncertainty. This addresses Problem **P2**. First, we introduced a *scenario-aware* analysis (**L2**) of structural uncertainty. By combining the design space exploration approach PerOpteryx [143, 144] with data flow analysis, we can identify confidentiality violations in software architecture candidates. This enables the analysis of structural uncertainty, e.g., due to an uncertain deployment of a component. Second, we introduced a *graph-aware* analysis (**L3**) of environmental uncertainty. Here, we used fuzzy inference in the form of FISs to express natural factors in the environment of a software system that can affect access control decisions and, thus, confidentiality. We found that both analysis approaches share the representation of uncertainty sources as one or multiple scenarios.

Last, we moved towards *type-agnostic* approaches for uncertainty-aware data flow analysis to address Problem **P3**. First, we presented a *graph-aware* analysis (**L3**) that uses variation modeling [168, 265] to trace confidentiality violations back to uncertainty sources. Being *type-agnostic*, this approach supports all five uncertainty types introduced in Section 5.3. To also support simple forms of uncertainty interactions, we introduced an *impact-aware* analysis (**L4**) thereafter. We closed the gap between the uncertainty classification and confidentiality analysis by providing a meta model for modeling uncertainty sources and scenarios. The analysis incrementally considers the impact of uncertainty within the data flow and thus can respect relations between multiple uncertainties. We also introduced 🔧 Abunai, our PCM-based tooling for uncertainty *impact-aware* data flow analysis (**L4**). We concluded this discussion by comparing the complexity of the presented analyses.

❓ **Research Question 3** asked about the analysis of confidentiality requirements using architecture-based data flow analysis that considers uncertainty within the architectural model and thus becomes uncertainty-aware. To answer this question, we provided a discussion of available information, resulting awareness levels of data flow analyses and analysis complexity and scalability. Furthermore, we presented four analysis approaches that differ in supported uncertainty types, awareness levels (**L2** – **L4**) and type specificity. As pointed out earlier, there is no single best answer to the question of architecture-based confidentiality analysis under uncertainty. Thus, our approach to this chapter was to provide a comprehensive discussion of multiple solutions and their trade-offs.

There are multiple benefits of the variety of analysis approaches presented in this chapter with Contribution **C3**. In general, considering uncertainty within an analysis enables more precise and more comprehensive analysis results of a model and its context [194, 272, 273]. Here, all four approaches can be used standalone, thus providing means to model and analyze different types of uncertainty. An automated analysis like 🔧 Abunai supports architects by reducing the manual effort, especially in large models with many independent data flows and many uncertainty sources [101]. Furthermore, describing uncertainties as a first-class entity and part of the architectural model helps in the documentation [104]. Although we focused on confidentiality, our *impact-aware* analysis is also an approach to consider uncertainty interactions within the architecture-based analysis. Together with our finding on UFDs, this can be used to further address the UIP. Last, our data flow analysis framework can be used to define new analysis approaches, even without the need to focus on uncertainty [36].

The contribution provided in this chapter was based on the findings from ❯ **Chapter 5: Identification and Classification of Uncertainty Regarding Confidentiality** and ❯ **Chapter 6: Uncertainty Propagation to Enable Uncertainty Impact Analysis**. We used many of the concepts introduced in these earlier chapters, e.g., the five uncertainty types, or the representation of uncertainty in DAGs as *primary* and *secondary* uncertainty. Also, the foundations for uncertainty propagation and interaction were laid in these chapters. The confidentiality analysis with respect to uncertainty represents the last step of the procedure presented in ❯ **Chapter 4: Overview**. The next chapters show an overview of ❯ **Chapter 8: Evaluation Scenarios** and then present the ❯ **Chapter 9: Evaluation** of all contributions.

7.9. In Simpler Words

Confidentiality describes the property that information is not disclosed to persons or organizations that should not see this information. Put simply, if you tell me a secret, I should keep it secret—treating it confidential. The same applies to software systems and the data that is handled by these systems. If you enter your personal data, such as your name, date of birth, or credit card details, into an online shop, you assume that you can trust the shop and that your information is treated confidentially.

Architecture-based confidentiality analyses have been proposed to analyze the confidentiality of a software system at a high abstraction or early in the system design. These analyses use a model of the system and confidentiality requirements as input and identify confidentiality violations. For example, if a law requires your confidential data to be stored on a server in the same country, but the system sends it across a border, this would represent a confidentiality violation. To identify such violations, we analyze all data flows in a software architecture—this is called *data flow analysis*.

In the previous chapters, we already introduced the notion of uncertainty and also discussed that uncertainty can negatively impact a software system's confidentiality. For instance, if we do not know whether your data is actually sent to a server in another country, we cannot state that it is confidential. To identify confidentiality violations under uncertainty, we define several uncertainty-aware data flow analyses. In this chapter, we present four different analysis approaches that use different algorithms to see how uncertainty impacts confidentiality. Some of these approaches are tailored to one uncertainty type, such as uncertainty in a software system's environment. Other approaches support all uncertainty types that have been identified in Chapter 5. We call the former *type-specific* analyses and the latter *type-agnostic* analyses. Additionally, we investigate the information used by the analysis and define five levels of uncertainty awareness.

Although the four presented approaches differ regarding the level of uncertainty awareness, they share the representation of uncertainty in the form of scenarios. A scenario represents one possible outcome of uncertainty, e.g., whether your data flows across a country border. The uncertainty-aware analysis considers these scenarios in analyzing confidentiality and returns only those scenarios that violate confidentiality. This helps software architects to better assess whether or not an uncertainty is critical and should be mitigated. Some approaches find such critical scenarios by checking all possible variations of the software architecture. Other approaches use the available information in a more clever way and can exclude irrelevant scenarios early in the analysis.

The most advanced approaches center their analysis algorithm around uncertainty. They treat uncertainty as a so-called first-class entity or first-class concern. This means that uncertainty is of central importance and as important as every other element of the software architecture. We find that considering uncertainty in this way helps to provide more and better information to the software architects. Additionally, the analysis scales better for large software systems from the real world. We also implemented such an analysis and provided it together with this thesis. This represents the final contribution of this thesis and also concludes the procedure that has been introduced in Chapter 4.

Part III.

Validation

8. Evaluation Scenarios

In this chapter, we present the scenarios used in the evaluation of this thesis. We do not perform *case studies* as we do not study the use of our contributions in a real-life environment, e.g., by watching software architects or security experts [276]. We use the term *evaluation scenario* instead as we only investigate the intended use of our contributions based on real-world systems, similar to related work [212, 264]. Nevertheless, using case study systems in the evaluation of contributions in the software architecture community is common [46, 233, 264]. Konersmann et al. [142] conducted a Systematic Literature Review (SLR) with 153 papers and found that using case study systems represents the most common evaluation method in recent software architecture research.

In this thesis, every *evaluation scenario* is based on or related to a real software system that has been specified by others. Thus, we start each scenario with a description of its source, where it is used, and how the architectural models have been created. Afterward, we describe the software system of each scenario, which includes the software architecture and its intended use. Every scenario handles confidential data and thus has confidentiality requirements that can be specified as data flow constraints. Last, a real-world software system is subject to uncertainty. We discuss uncertainty sources and their potential impact on confidentiality based on existing collections [103, 104, 202].

This chapter introduces six evaluation scenarios. The first three, namely the *TravelPlanner*, *DistanceTracker*, and *OnlineShop* scenarios have been used in the evaluation of related approaches [147, 151, 233, 234, 236, 264, 265, 267, 269, 270]. The second three, namely the *CoronaWarnApp*, *MobilityAsAService*, and *JPlag*, represent software systems from academia [152, 199, 213, 215, 216, 217] and industry [69, 119]. In the following, we give an overview of the evaluation scenarios and then introduce them in more detail. See the data set [98] for all data flow constraints and Palladio Component Model (PCM) instances.

Literature: This chapter is based on the following (co-) authored publications: [ECSA-C 2021], [IEEE SEAA 2022], [Springer ECSA 2022], [IEEE ICSA-C 2023], [Springer ICETE 2023], [IEEE/ACM SEAMS 2023], [Springer ECSA 2024], [ACM/IEEE MODELS-C 2024]

8.1. Overview

For the evaluation of this thesis, we use software systems and scenarios of different size and from different domains. Table 8.1 shows an overview of the six evaluation scenarios,

Name	Domain	Origin	Used in
TravelPlanner	Mobility	[133, 135]	[37, 101, 105, 147, 231, 233, 234, 236, 264, 265, 266, 267, 269]
DistanceTracker	Health / Sports	[133]	[37, 101, 105, 233, 234, 236, 266]
OnlineShop	E-Commerce	[203, 234]	[36, 97, 100, 101, 102, 103, 104, 151, 266, 270]
CoronaWarnApp	Health	[69, 119]	[102, 104, 106]
MobilityAsAService	Mobility	[152]	-
JPlag	Plagiarism Detection	[199, 216]	-

Table 8.1.: Overview of all evaluation scenarios, their domain, their origin, and their usage.

Name	# Component	# SEFF	# TFG	# Vertex	# Uncertainty	# Violation
TravelPlanner	7	9	2	42	1	1
DistanceTracker	8	10	1	29	1	1
OnlineShop	2	6	3	44	4	24
CoronaWarnApp	21	58	14	687	9	163
MobilityAsAService	18	49	8	200	5	6
JPlag	3	5	3	65	4	26

Table 8.2.: Size metrics, uncertainty sources, and confidentiality violations of all evaluation scenarios.

their origin, and other evaluations where they have already been used. To encounter threats to external validity and to increase generalizability, we use scenarios from different domains, e.g., from the mobility or health domain. All evaluation scenarios originate from other work. The majority have been used in the evaluation of other publications. Only the last two scenarios, *MobilityAsAService* a service and *JPlag* are new. However, they represent well-evaluated systems with high demands on the systems' confidentiality and are thus suitable as evaluation scenarios for this thesis.

For the contributions of this thesis (**C1**, **C2**, and **C3**), a central property of a software system is the form and size of the system's Data Flow Diagram (DFD) and the scenario's uncertainty sources, as discussed in Section 5.5 and Section 7.5. Table 8.2 lists all relevant size-related metrics of the six evaluation scenarios. This includes the number of the components and Service Effect Specifications (SEFFs) of the PCM instance of each scenario, as these numbers impact the number of DFD nodes. We also show the number of Transpose Flow Graphs (TFGs) and their vertices, because these sizes are relevant both for the uncertainty impact analysis (**C2**) and for the different approaches to uncertainty-aware data flow analysis (**C3**). Last, we show the number of uncertainty sources in the evaluation scenarios and the number of potential confidentiality violations due to these uncertainties.

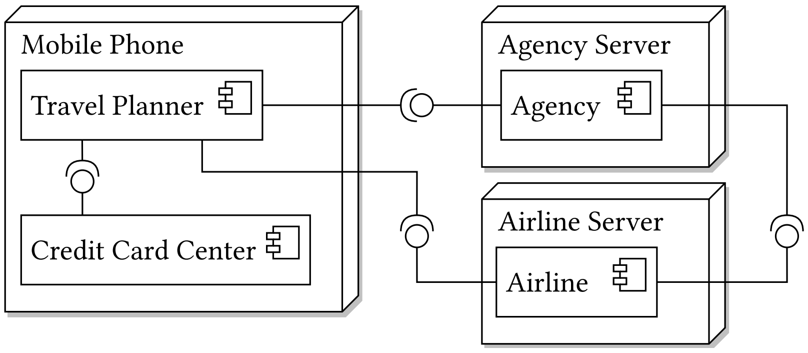


Figure 8.1.: Simplified component and deployment diagram of the TravelPlanner evaluation scenario.

The scenarios show different sizes and numbers of uncertainties and violations. The *TravelPlanner* and the *DistanceTracker* represent smaller scenarios with one uncertainty source each, and 7 and 8 components, respectively. Still, their TFGs have a total of 42 and 29 vertices, which highlights again the need for automated analysis approaches and tool support, discussed in Section 4.3. *MobilityAsAService* and *CoronaWarnApp* are the largest evaluation scenarios. They contain 5 and 9 uncertainty sources and a total of 200 and 687 TFG vertices that represent DFD nodes.

8.2. TravelPlanner

Source The *TravelPlanner* evaluation scenario originates from the iFlow project by Katkalov [133]. It originally has been used in the evaluation of information flow analysis [135]. A comprehensive description of the software architecture and a Java-based implementation are available [134]. The PCM model used in this evaluation scenario stems from related work [233, 264] and has been used in many case study-based validations [37, 101, 105, 147, 231, 234, 236, 265, 266, 267, 269].

Description Figure 8.1 shows a simplified diagram of the software architecture. Here—and also in all following figures of this chapter—we show simplified diagrams to give an overview and to increase clarity. As shown in Table 8.2, the PCM model consists of more components than illustrated. We do also not depict uncertainty sources in this diagram. We refer to the data set [98] for the complete architectural model, which includes data flow constraints and uncertainty models. The corresponding PCM model consists of 7 components with 9 SEFFs and maps to 2 TFGs with 42 vertices.

This evaluation scenario comprises three central entities: A customer, a travel agency, and an airline. The customer uses the *Travel Planner* app to search for flights. This app communicates with a travel *Agency* that queries flights from multiple *Airlines*. The aggregated results are returned to the customer. Afterward, the customer selects and books a flight using a credit card. The software architecture comprises three deployment locations that match the three entities. The customer operates the *Mobile Phone*, where

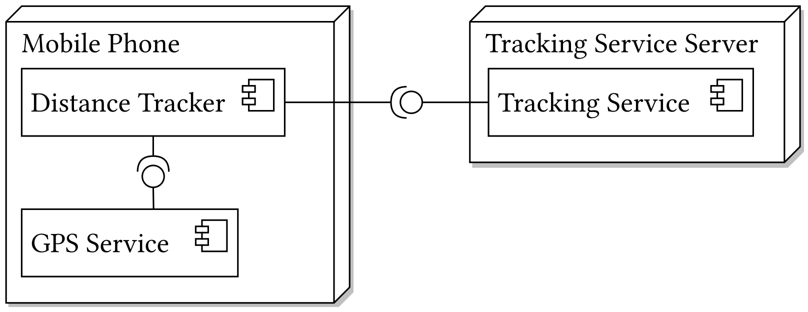


Figure 8.2.: Simplified component and deployment diagram of the DistanceTracker evaluation scenario.

the *Travel Planner* app and the *Credit Card Center* app is located. Both the *Agency* and the *Airline* operate their own servers and provide interfaces to the other components.

Confidentiality requirements In this evaluation scenario, we consider two different data types. On the one hand, the flight data represents public information without any data flow constraints. On the other hand, credit card data represents highly sensitive information with strict confidentiality requirements. This information is managed by the *Credit Card Center* and shall only leave the *Mobile Phone* in the booking process upon explicit approval by the customer. Additionally, the credit card data shall only flow directly to the *Airline* in the booking process—it shall never flow to the *Agency*.

Uncertainty sources We use this evaluation scenario as a minimal scenario for *primary* uncertainty. We add the *Behavior* uncertainty that credit card data can be used with or without user consent [101]. Here, passing the data to the *Airline* servers without consent represents a violation of the aforementioned confidentiality requirements.

8.3. DistanceTracker

Source The *DistanceTracker* evaluation scenario also originates from the iFlow project by Katkalov [133]. Similarly to the *TravelPlanner* scenario described above, it has been used in many case study-based validations [37, 101, 105, 233, 234, 236, 266].

Description Figure 8.2 shows a simplified diagram of the software architecture. As discussed above, we simplify the system structure and do not depict uncertainty sources to increase clarity and refer to the data set [98]. The corresponding PCM model consists of 8 components with 10 SEFFs and maps to 1 TFG with 29 vertices.

The evaluation scenario comprises three central entities: The user, the *Distance Tracker* app, and the online *Tracking Service*. The user shares the GPS location with the tracking app that periodically tracks the current location and calculates the distance run. This distance is transmitted to the *Tracking Service*. The software architecture comprises two

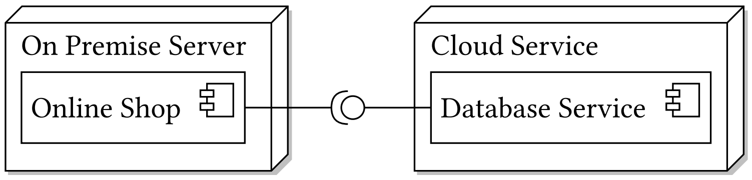


Figure 8.3.: Simplified component and deployment diagram of the OnlineShop evaluation scenario.

deployment locations. The user operates the *Mobile Phone* that hosts the *Distance Tracker* app and a *GPS service*. The *Tracking Service* is deployed on a separate server.

Confidentiality requirements The confidentiality requirement of this evaluation scenario considers the confidentiality of GPS data, i.e., precise information about the user's current location. This information is only allowed to be used locally on the *Mobile Phone*. Only in aggregated form, e.g., as calculated distance, this information is allowed to be passed to the *Tracking Service*. Thus, there shall be no flow of explicit GPS locations from the *Mobile Phone* to any other location.

Uncertainty sources We use this evaluation scenario as a minimal scenario for *secondary* uncertainty. We add the *Connector* uncertainty that GPS data can be processed on the *Tracking Service Server* [101] for additional statistics and a persisted history of runs. Passing the GPS data to the *Tracking Service* for this purpose is a confidentiality violation.

8.4. OnlineShop

Source The *OnlineShop* evaluation scenario is based on CoCoME [203] and has been adapted for architecture-based confidentiality analysis [234]. It has also been used in case study-based validations [36, 97, 100, 101, 102, 103, 104, 151, 266, 270]. We refer to this evaluation scenario throughout this thesis as the running example[1], see Chapter 3.

Description Figure 8.3 shows a simplified diagram of the software architecture. A comprehensive diagram of the software system is shown in Figure 3.1, and the architectural model is part of the data set [98]. The corresponding PCM model consists of 2 components with 6 SEFFs and maps to 3 TFGs with 44 vertices.

The evaluation scenario comprises three central entities: The customer, the *Online Shop*, and the shop's *Database Service*. We consider three usage scenarios: The user can query available items, purchase items, and request a support contact. The former two scenarios

[1] Although using an evaluation scenario as the running example represents a threat to validity, we argue that this threat is weakened by using other evaluation scenarios in addition. Furthermore, the increased understanding arising from such thorough investigation can benefit the evaluation.

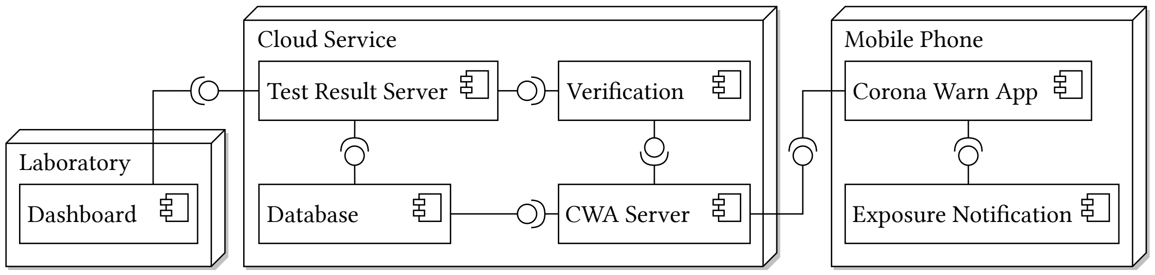


Figure 8.4.: Simplified component and deployment diagram of the CoronaWarnApp evaluation scenario.

include calls of the *Online Shop* to the *Database Service*, the latter only yields static information. The software architecture comprises two deployment locations. The *Online Shop* is deployed *On Premise* while the *Database Service* is deployed to a *Cloud Service*. We refer to Chapter 3 for a more detailed description of the software system.

Confidentiality requirements In this evaluation scenario, we consider two different data types. The public information of the online shop, like available items, and the user's private purchase details. While the former is publicly available and not sensitive, the latter is highly sensitive, comparable to the credit card details of the *TravelPlanner* evaluation scenario. The user input in the purchase process shall be validated and encrypted before flowing into the database. Additionally, depending on the deployment location and the trustworthiness of the resource provider, confidentiality can be threatened. We provide more details on the confidentiality requirements in Chapter 3.

Uncertainty sources We define four uncertainty sources in this evaluation scenario, as described in Section 3.2. Uncertainty **U1** affects the user input that can be valid, erroneous, or malicious. Uncertainty **U2** affects the data processing in the *Online Shop* component that can validate or encrypt the data. Uncertainty **U3** affects the deployment of the *Database Service* component that can be on-premise or in the cloud. Uncertainty **U4** affects the trustworthiness of the provider of the *Cloud Service* that can be trustworthy or suspicious. All uncertainty sources and their scenarios are shown in Table 7.2.

8.5. CoronaWarnApp

Source The *CoronaWarnApp* evaluation scenario is based on the German Contact Tracing App of the same name [119]. During the COVID-19 pandemic, this app was used to warn users who may have come into contact with people who had tested positive on COVID-19. The app was developed by SAP and Deutsche Telekom [69], published by the Robert Koch Institute, and downloaded more than 20 million times [119]. All source code and documentation are available open-source on GitHub [222], including a detailed description of the software architecture. We used the publicly available information to create a PCM model that was already used in multiple validations [102, 104, 106].

Description Figure 8.4 shows a simplified diagram of the software architecture. Due to the size of the evaluation scenario, we leave out many details in this diagram, e.g., the collection of statistics, or the international exchange of positive test results. The PCM component repository model is shown in Appendix D. The corresponding PCM model consists of 21 components with 58 SEFFs and maps to 14 TFGs with 687 vertices.

The evaluation scenario comprises five central entities. The user of the app, the server infrastructure, laboratories, support hotlines, and international partners. Users can voluntarily enter if they are infected to warn others and they also get warned in the case of a potential contact. The warning mechanism uses keys that are exchanged between mobile phones via Bluetooth [222]. If users are tested, they can enter the test via a code or with the help of the support hotline. The laboratories add the test results, and the server infrastructure manages both. Additionally, the keys of positive tested users can be shared with international partner servers in the EU and Switzerland. The components of the server infrastructure, e.g., the *Verification*, or the *Test Result Server*, are deployed to the *Cloud Service* of the Open Telekom Cloud. Laboratories operate their own *Dashboards* and laboratory information systems that connect to the server infrastructure. Users have the *Corona Warn App* installed on their phones that communicate with the *Exposure Notification* framework. This framework manages the key exchange and comparison.

Confidentiality requirements The Corona Warn App manages sensitive information related to health data and location data. Contact tracing apps are challenged with many risks, e.g., regarding de-anonymization or profiling [20]. We express multiple confidentiality requirements as data flow constraints [36, 105]. First, users should not be able to directly access the exchanged keys but only be warned if an exchanged key matches a key of a person who tested positive. Additionally, all keys and other credentials are considered to be sensitive information. Here, the *Verification* component plays a central role in the validation. The Corona Warn App collects statistical data that shall only be shared in aggregated form. Last, we reuse the already discussed requirements regarding secure storage, information leaks, and logging. The full list of requirements is in data set [98].

Uncertainty sources Due to the size of this evaluation scenario, we model 4 sub-scenarios with 2 to 3 uncertainty sources each, which adds up to a total of 9 uncertainty sources. In this first sub-scenario, we consider the processing of data and the interception of the communication regarding a central component in the *Cloud Service*. The second sub-scenario focuses on deployment and secure storage of test results using the *Test Result Server*. The third sub-scenario evolves around logging and validation in the *Verification* component. In the fourth sub-scenario, we focus on critical points within the system with a potentially wide impact, e.g., the *Database* and the *Exposure Notification* framework. All details about the uncertainty sources and scenarios can be found in the data set [98].

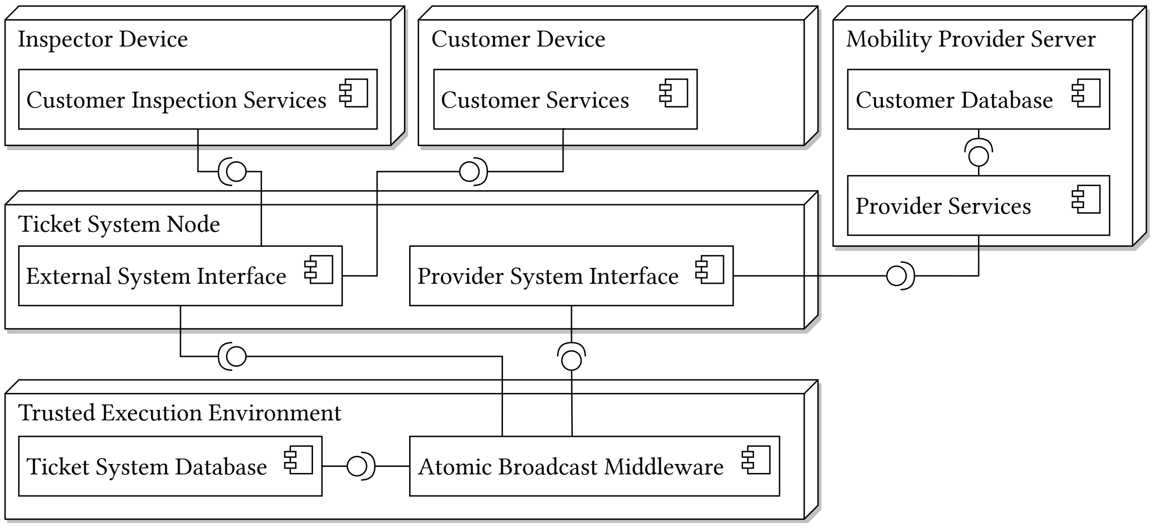


Figure 8.5.: Simplified component and deployment diagram of the MobilityAsAService evaluation scenario.

8.6. MobilityAsAService

Source The *MobilityAsAService* evaluation scenario is based on a concept for a distributed ticketing system of the same name [152]. It uses Distributed Ledger Technology (DLT) with Trusted Execution Environments (TEEs) to define a secure system with distributed governance that is scalable while ensuring confidentiality. We used the available documentation to create a PCM model that approximates the proposed concept.

Description Figure 8.5 shows a simplified diagram of the software architecture. As we use the ADL PCM to model the scenario, we can only approximate the actual behavior of a DLT, or a TEE. Additionally, we only model one mobility provider, while the underlying concept allows for many operators with decentralized governance. The corresponding PCM model consists of 18 components with 49 SEFFs and maps to 8 TFGs with 200 vertices.

The evaluation scenario comprises four central entities. The ticket system encapsulates a TEE that manages a replicated state machine and middleware to broadcast requests. We illustrate this with the *Ticket System Database* and the *Atomic Broadcast Middleware* components. The system can receive input from the three other entities: Inspectors, customers, and mobility providers. Each node provides an *External System Interface* for the former two and a *Provider System Interface* for the latter. Customers can register, check-in, check-out, and see their trip history through several components that we bundled as *Customer Services*. Inspectors can inspect customers using the *Customer Inspection Services*, i.e., check the current state of the customer. Mobility Providers have tools for billing, clearance, and analysis as part of the *Provider Services*. Additionally, they store contact and billing information of their customers in a separate *Customer Database*. We combined some of the aforementioned services, e.g., the *Customer Services*, for the sake of clarity, the full model is part of the data set [98].

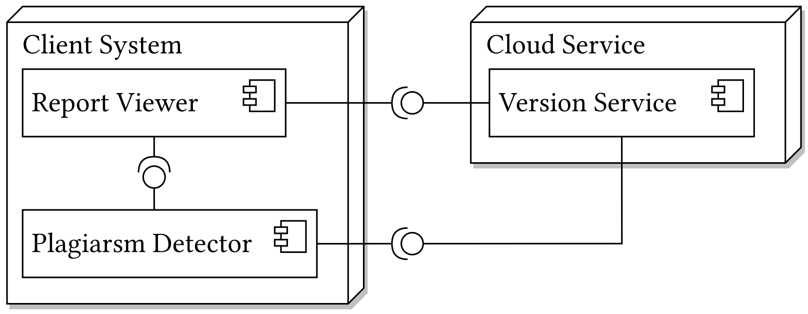


Figure 8.6.: Simplified component and deployment diagram of the JPlag evaluation scenario.

Confidentiality requirements Leinweber et al. [152] name several confidentiality requirements of the *MobilityAsAService* system. They require that only authorized entities are able to access the data of the ticket system. Additionally, providers' business secrets and customer data shall not be leaked during the replication process. Details about individual trips shall not be revealed to providers who only see the total invoice value. The system is designed to provide all entities with the minimal information required for their tasks.

Uncertainty sources We use this evaluation scenario to combine all five types of uncertainty sources defined in Table 5.2. We add an *External* uncertainty to a staff member who uses the *Provider Services* regarding the member's role and authorization. We add a *Behavior* uncertainty to the *Ticket System Database* regarding the granularity of the retrieved data. We add a *Interface* uncertainty to the trip history service that is part of the *Customer Services*, and a *Connector* uncertainty to the customer registration that is part of the *Customer Services*, both regarding the sensitivity of the retrieved data. Last, we add a *Component* uncertainty to the *Provider Services* representing a malicious mobility provider [152]. All uncertainty sources reflect changes to the *MobilityAsAService* concept that would violate confidentiality.

8.7. JPlag

Source The *JPlag* evaluation scenario is based on the plagiarism detector of the same name [199]. Plagiarism detectors compare models or code submissions of programming tasks [217], e.g., in educational context [219], to find suspicious similarities, i.e., plagiarism [216]. The *JPlag* project, started in 1996, represents one of the most widely-used plagiarism detectors that is resilient against most obfuscation attempts [213, 214, 215, 216, 218, 225]. Since 2020, the software architecture of *JPlag* has been re-engineered. Based on the available documentation of this open-source project [214], and by cooperating with the current maintainers, we created a PCM model of the core functionality.

Description Figure 8.6 shows a simplified diagram of the software architecture. The *Jplag* software architecture is designed to enable the deployment of the main components, i.e., the *Plagiarism Detector* and the *Report Viewer*, on a client system, a local machine, or a server, to simplify its integration into other products [214]. Nevertheless, we consider the illustrated deployment to be the most common use case. The corresponding PCM model consists of 3 components with 5 SEFFs and maps to 3 TFGs with 65 vertices.

The evaluation scenario comprises three entities: The *Plagiarism Detector*, a *Report Viewer* for displaying analysis results, and a *Version Service* for update notifications. The *Plagiarism Detector* receives a set of code or model submissions as input and compares them. This is done internally without any communication to other services [214]. After the comparison, all results are stored in an archive. This includes similarity scores between submissions, clusters of similar submissions, or suspiciously similar parts of the submissions. Because plagiarism detection needs human judgment [213, 216], the results are displayed in a human-readable way using the *Report Viewer.* The software architecture supports the direct communication between the *Plagiarism Detector* and the *Report Viewer*, but users can also choose to manually load a result archive, e.g., to revisit older comparisons. The *Report Viewer* is web-based but only operates within the web browser without the need for additional server infrastructure [214]. Thus, all input remains on the users' *Client System.* As *Jplag* is subject to continuous development [214], both components communicate with a *Version Service* to notify users about available updates.

Confidentiality requirements Plagiarism detectors like *JPlag* are designed for the educational context, e.g., programming tasks. Here, student data has to be treated as confidential due to administrative rules and laws like the General Data Protection Regulation (GDPR) [73]. We interpret this requirement as data flow constraint and restrict that all submissions and all plagiarism detection results shall never leave the client system. Furthermore, the authors of *JPlag* claim to never collect any usage data from running the software [214]. Thus, we additionally restrict the *Version Service* to not collect or evaluate any statistical information on the versions in use.

Uncertainty sources We use this evaluation scenario to define four uncertainty sources, of which only two negatively affect the system's confidentiality. This allows for more precise statements about potential false positives in the evaluation [198]. The first uncertainty source considers the deployment of the *Report Viewer* on the *Client System* or in the cloud. The second uncertainty source considers the usage of *JPlag* standalone or in conjunction with the *Report Viewer.* As stated previously, neither uncertainty affects the confidentiality of the software architecture. The third uncertainty source targets the deployment of the *Plagiarism Detector*, which could be deployed locally or in the cloud. This could cause sensitive student data to flow to another server, which violates confidentiality—an argument against closed-source plagiarism detection as a service [214, 216]. Last, we add a fourth uncertainty concerning the data collection behavior of the *Version Service.* As stated previously, evaluating usage data violates confidentiality.

8.8. Summary and Outlook

In this chapter, we presented six evaluation scenarios that will be used throughout the evaluation of this thesis. All evaluation scenarios are based on or related to a case study of a software system that handles confidential data. The scenarios have different sizes that range from 2 to 21 components, and also have different origins and domains. For each evaluation scenario, we described its source, software architecture, confidentiality requirements, and uncertainty sources. All scenarios are modeled using the PCM.

The first scenario, the *TravelPlanner*, represents a smartphone app to search and book flights. The second scenario, the *DistanceTracker*, is a sports or health app that tracks the distance run by a user. The third scenario, the *OnlineShop*, is an e-commerce application for purchasing items online, introduced in Chapter 3. The fourth scenario, the *CoronaWarnApp*, is a contact tracing app developed during the COVID-19 pandemic. The fifth scenario, *MobilityAsAService*, is a distributed ticketing system based on DLT and TEEs. The sixth scenario, *JPlag*, is a plagiarism detector, e.g., used for programming tasks in universities.

In sum, these scenarios provide a wide range of different software systems. Despite their differences in structure and size, they all have to ensure the confidentiality of the processed data. Furthermore, they are all subject to uncertainty. We use these scenarios throughout ❯ **Chapter 9: Evaluation**. For more details on the third scenario, the *OnlineShop*, see ❯ **Chapter 3: Running Example**. For a more detailed view of the components of the fourth scenario, the *CoronaWarnApp*, see Appendix D. Last, we again refer to our data set [98] for all uncertainty and architectural models and all data flow constraints.

8.9. In Simpler Words

In the last three chapters, we presented the contributions of this thesis. In our research area, a contribution is an enhancement of the state of the art, e.g., a new or better algorithm, a better way to collect and express knowledge, or a new method. To investigate the quality of our contributions, we perform an evaluation. In the discipline of this thesis—software architecture research—many evaluations are based on case studies. In this chapter, we refer to evaluation scenarios comparable to case studies.

We present six different evaluation scenarios of different sizes and from different domains. Examples are the health domain, the mobility domain, and e-commerce. Many of our evaluation scenarios have already been used to evaluate related approaches, e.g., the *TravelPlanner* evaluation scenario. Other scenarios, e.g., the *CoronaWarnApp* evaluation scenario, are newer but based on well-studied software systems from the real world. It is important to cover different sizes and domains because otherwise, one could say: “Your contributions only work for small software systems” or “Your contributions only work for systems in the mobility domain, but not for others”. By using various evaluation scenarios, we counteract such—otherwise justified—criticism. The following chapter will apply our contributions to these evaluation scenarios.

9. Evaluation

In this chapter, we present the evaluation of this thesis. We choose the joint presentation of the evaluation of all three contributions **C1**, **C2**, and **C3** for increased cohesiveness and clarity. The evaluation is based on the evaluation scenarios, introduced in Chapter 8.

Although all three contributions of this dissertation target confidentiality with regard to uncertainty, they differ in their nature. For instance, Contribution **C1** represents a classification, while Contribution **C3** comprises meta models and analysis approaches. Thus, we use different evaluation methods such as user studies and the aforementioned evaluation scenarios, which are comparable to case studies. This matches the findings of Konersmann et al. [142], who conducted a Systematic Literature Review (SLR) with 153 papers on the evaluation methods in software architecture research. For our research object—architecture analysis methods—case studies, motivating examples, and technical experiments represent the most common evaluation methods [142]. In this case, the most commonly investigated properties are effectiveness, functional suitability, and accuracy, which match the presented evaluation. For most of the evaluation, we focus on the presented analyses, not the presented meta models. Following the definition of Stachowiak [245], every model has pragmatism, in our case, to serve the analysis. Thus, evaluating the analysis indirectly also evaluates the underlying models [264].

The evaluation follows a comprehensive Goal Question Metric (GQM) approach [16, 17]. A *goal* is on the conceptual level and describes a quality to evaluate, i.e., the aforementioned properties to investigate. An evaluation can have multiple goals. A *question* is on the operational level and describes how the quality is measured or assessed. Each goal can have multiple questions. A *metric* is on the quantitative level and describes the data that is associated with the question. Each question can have multiple metrics. A GQM plan helps align quality, assessment, and data to minimize the risk of collecting meaningless results [17]. It also helps to structure the evaluation and increases reproducibility. We number all goals, questions, and metrics and use labels throughout this chapter.

The remainder of this chapter is structured as follows: We introduce the GQM plans for evaluating all contributions **C1** – **C3**. Then, we present the evaluation design and results for each contribution separately. We individually discuss the results and threats to validity according to Runeson and Höst [212]. Last, we give an summary of the evaluation.

Literature: This chapter is based on the following (co-) authored publications: [ECSA-C 2021], [IEEE SEAA 2022], [Springer ECSA 2022], [ACM EASE 2022], [IEEE ICSA-C 2023], [Springer ICETE 2023], [IEEE/ACM SEAMS 2023], [Springer ECSA 2024], [ACM/IEEE MODELS-C 2024]

9.1. Overview

We present the individual GQM plans of all three contributions. The first Contribution **C1** concerns the classification and identification of uncertainty. We evaluate the structure's suitability, applicability, purpose, and usability. The second Contribution **C2** focuses on architecture-based uncertainty impact analysis using uncertainty propagation. We evaluate the accuracy and effort reduction. The third Contribution **C3** comprises four approaches to uncertainty-aware data flow analysis. We evaluate the accuracy and scalability.

In total, our GQM plans comprise 8 goals, 19 questions, and 32 metrics. In the following, we show the tree of each evaluation goal and then discuss its questions and metrics.

9.1.1. Evaluation Plan for the First Contribution

The first Contribution **C1** introduces a classification of software-architectural uncertainty regarding confidentiality [104]. Additionally, it includes means to identify uncertainty sources based on an interactive catalog approach [103]. This approach is tool-supported by ARC^{3}N. The evaluation of this contribution comprises 4 goals, with a total of 12 questions and 16 metrics. The evaluation is closely aligned to the simultaneously introduced evaluation method for classifications and taxonomies by Kaplan et al. [132].

Goal G1 Figure 9.1 shows the tree of the first goal. Following the aforementioned evaluation method [132], this goal concerns the classification *structure's suitability*. This property describes whether a classification is suitable to classify the objects under study. In our case, the objects under study are sources of uncertainty with respect to confidentiality. This can be seen as a baseline check as the classification needs to have the right scope and granularity to be suited. We also motivated the classification in Chapter 5 with the lack of suitable taxonomies. We consider three evaluation questions:

Q1.1 Has the classification an appropriate level of *generality* and granularity?

Q1.2 Is the classification *appropriate*, comprising only necessary classes?

Q1.3 Is the classification *orthogonal* without overlapping classes?

The first Question **Q1.1** asks about the *generality*, where we evaluate the granularity, i.e. if the classification is not too general but also not too specific. The right level of granularity is important as a classification group objects under study into classes—a too-low number of classes reduces the usefulness of this information, while a high number too-high number of classes introduces noise and makes the classification results hard to understand. To this end, the evaluation method reuses metrics used to evaluate conceptual models with respect to tools [7, 132]. The *laconicity* (**M1.1.1**) measures the fraction of terms that are uniquely describable. Given a classification C, a finite set of objects under study $\mathcal{R}$, with $R \in \mathcal{R}$ being an object under study with relevant terms $r \in \mathcal{R}$, then $m_R^C \subseteq C \times R$ describes the relation of classes $c \in C$ to relevant terms $r \in R$. In our classification, we use the term

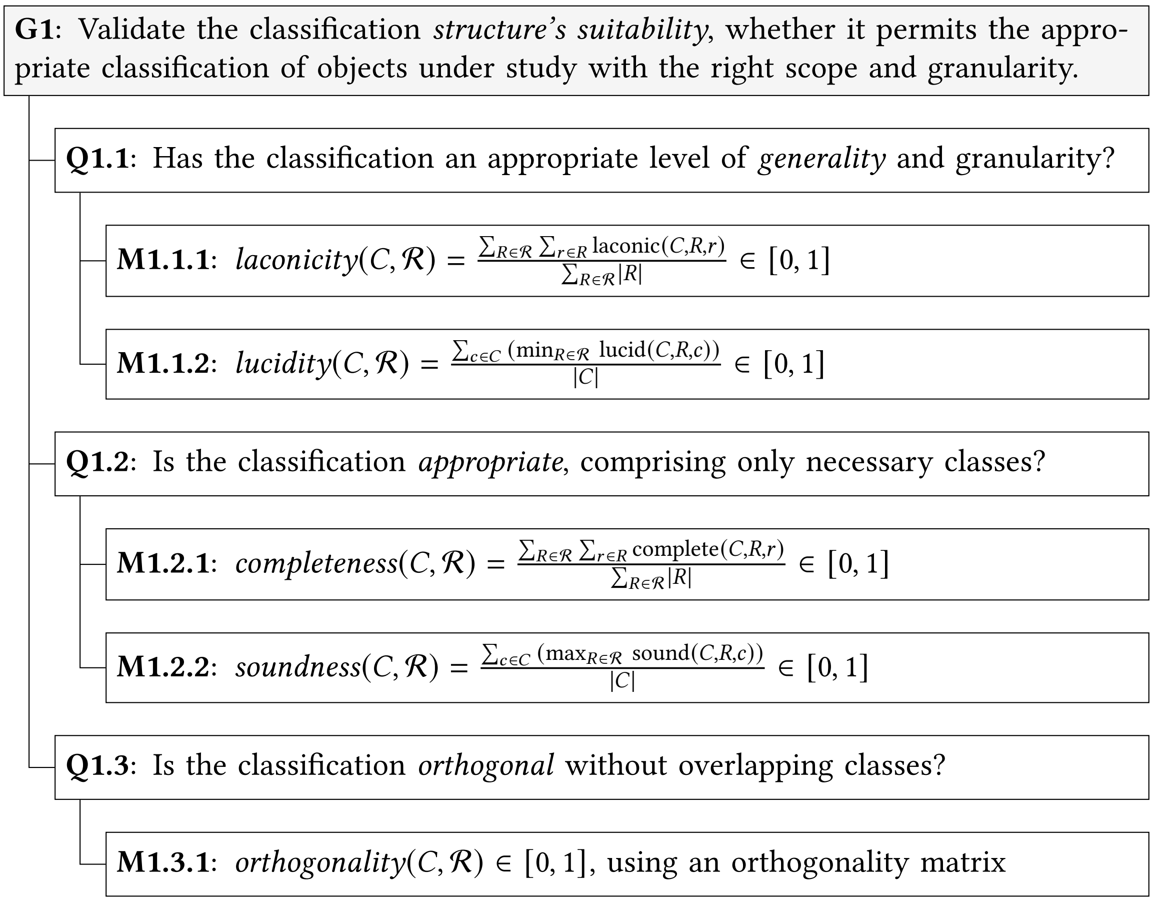

G1: Validate the classification *structure's suitability*, whether it permits the appropriate classification of objects under study with the right scope and granularity.
Q1.1: Has the classification an appropriate level of *generality* and granularity?
M1.1.1: $\mathit{laconicity}(C,\mathcal{R}) = \frac{\sum_{R\in\mathcal{R}}\sum_{r\in R}\text{laconic}(C,R,r)}{\sum_{R\in\mathcal{R}}|R|} \in [0,1]$
M1.1.2: $\mathit{lucidity}(C,\mathcal{R}) = \frac{\sum_{c\in C}(\min_{R\in\mathcal{R}}\text{lucid}(C,R,c))}{|C|} \in [0,1]$
Q1.2: Is the classification *appropriate*, comprising only necessary classes?
M1.2.1: $\mathit{completeness}(C,\mathcal{R}) = \frac{\sum_{R\in\mathcal{R}}\sum_{r\in R}\text{complete}(C,R,r)}{\sum_{R\in\mathcal{R}}|R|} \in [0,1]$
M1.2.2: $\mathit{soundness}(C,\mathcal{R}) = \frac{\sum_{c\in C}(\max_{R\in\mathcal{R}}\text{sound}(C,R,c))}{|C|} \in [0,1]$
Q1.3: Is the classification *orthogonal* without overlapping classes?
M1.3.1: $\mathit{orthogonality}(C,\mathcal{R}) \in [0,1]$, using an orthogonality matrix

Figure 9.1.: Overview of the GQM plan of the first goal regarding Contribution **C1**.

option to refer to classes, e.g., *Behavior* uncertainty and *Scenario* uncertainty are two of the 27 options of our classification. A class is laconic, if there is at most one $c \in C$ with $(c,r) \in m_R^C$. Based on this, we denote:

$$\mathit{laconicity}(C,\mathcal{R}) = \frac{\sum_{R\in\mathcal{R}}\sum_{r\in R}\text{laconic}(C,R,r)}{\sum_{R\in\mathcal{R}}|R|} \in [0,1]$$

A low laconicity indicates a too fine-grained classification with potentially too many classes to describe the object under study. Put simply, a concept of an object under study should be only describable using a single class, otherwise the classification structure's suitability could be decreased. For instance, if we have one object under study, one term that represents *Behavior* uncertainty, and one class to describe it, this class is laconic, thus $\mathit{laconicity}(C,\mathcal{R}) = \frac{1}{1} = 1.0$. If we add five additional options to describe *Behavior* uncertainty without need, this decreases the laconicity; in this case to zero, as $\frac{0}{1} = 0.0$

The *lucidity* (**M1.1.2**) measures the opposite, i.e., the fraction of classes that describe exactly one term. A class is lucid, if there is at most one $r \in R$ with $(c,r) \in m_R^C$:

$$lucidity(C,\mathcal{R}) = \frac{\sum_{c\in C}(\min_{R\in\mathcal{R}}\ \text{lucid}(C,R,c))}{|C|} \in [0,1]$$

A low lucidity indicates a too-coarse-grained classification that could overestimate classes and not distinguish enough. For instance, if we have five three classes, where two describe only one term, e.g., *Behavior* and *External* uncertainty, but the third class does describes both terms, then $lucidity(C,\mathcal{R}) = \frac{(1+1+0)}{3} = \frac{2}{3}$. Put simply, a class of classification should only describe a single concept of an object under study. A good trade-off regarding granularity is important because we want to be able to differentiate between uncertainties without assigning a separate class to every instance. Also, the granularity must fit the purpose of classifying software-architectural uncertainty regarding confidentiality.

The second Question **Q1.2** asks about the *appropriateness*, where we evaluate whether the classification has enough categories without having unnecessary categories. On the one hand, we shall be able to classify every software-architectural uncertainty that can have an impact on confidentiality. On the other hand, categories and options that are never used, should not be maintained. Similarly to the metrics used for Question **Q1.1**, we reuse metrics from the evaluation of conceptual models [7, 132]. The *completeness* (**M1.2.1**) measures the fraction of complete terms over all objects under study. Each term of all objects under study should be covered by at least one class, otherwise the completeness is reduced. A term is $r \in R$ is complete, if there is at least one $c \in C$ with $(c,r) \in m_R^C$:

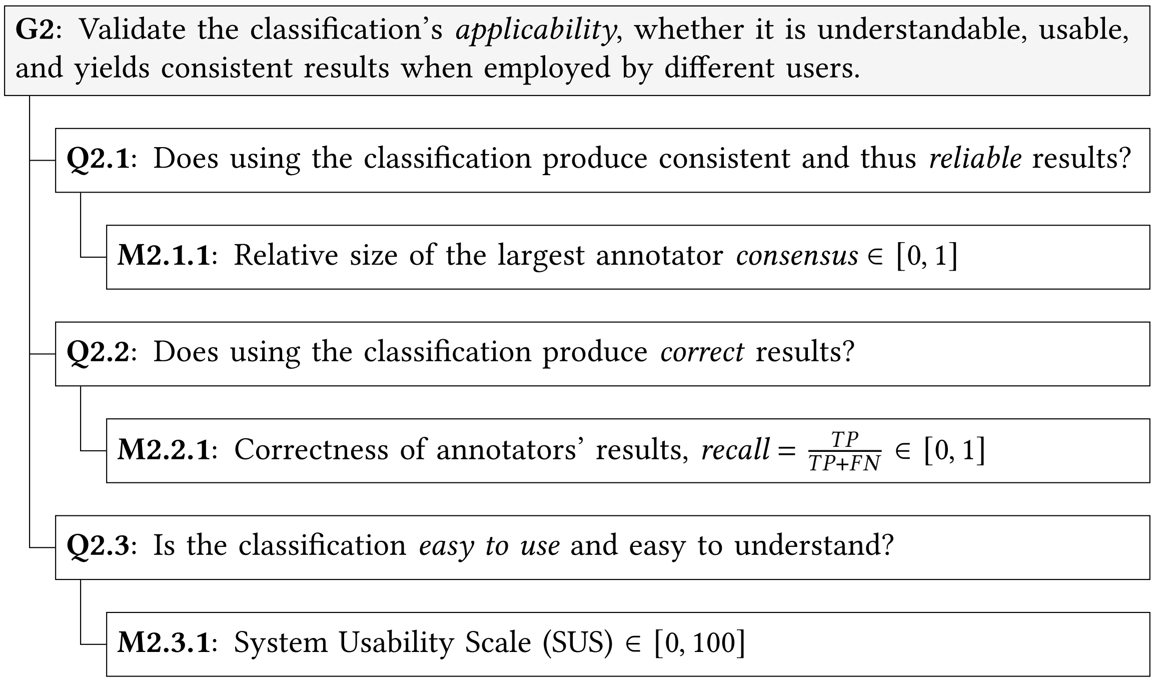
$$completeness(C,\mathcal{R}) = \frac{\sum_{R\in\mathcal{R}}\sum_{r\in R}\text{complete}(C,R,r)}{\sum_{R\in\mathcal{R}}|R|} \in [0,1]$$

A low completeness indicates missing classes as not all important properties of all objects under study can be described. Put simply, a classification that cannot be used to describe an object is not complete. For instance, if we remove the option of *Behavior* uncertainty we cannot classify many uncertainty sources appropriately. The *soundness* (**M1.2.2**) measures the fraction of sound classes in the classification. Each class should be required to describe at least one object under study, otherwise it is unnecessary can be removed. A class $c \in C$ is sound, if there is at least one $r \in R$ with $(c,r) \in m_R^C$. We denote:

$$soundness(C,\mathcal{R}) = \frac{\sum_{c\in C}(\max_{R\in\mathcal{R}}\ \text{sound}(C,R,c))}{|C|} \in [0,1]$$

A low soundness indicates unnecessary classes as not all classes are required to describe all objects under study. Put simply, a classification with low soundness introduces noise and complicates the classification process. For instance, if we add an option called *Color* uncertainty that describes the effect of the color of a button in the user interface on confidentiality, the soundness is reduced[1].

[1] Note that we do not state that there is no effect of the color of a button on confidentiality, think of, for instance, phishing attacks. However, the relevance for software-architectural uncertainty is negligible.



G2: Validate the classification's *applicability*, whether it is understandable, usable, and yields consistent results when employed by different users.

Q2.1: Does using the classification produce consistent and thus *reliable* results?

M2.1.1: Relative size of the largest annotator *consensus* $\in [0,1]$

Q2.2: Does using the classification produce *correct* results?

M2.2.1: Correctness of annotators' results, $recall = \frac{TP}{TP+FN} \in [0,1]$

Q2.3: Is the classification *easy to use* and easy to understand?

M2.3.1: System Usability Scale (SUS) $\in [0,100]$

Figure 9.2.: Overview of the GQM plan of the second goal regarding Contribution **C1**.

The third Question **Q1.3** asks about the *orthogonality*, where we evaluate whether the classification has overlapping categories. A lack of orthogonality implies that options depend on each other and can be removed to increase preciseness. To measure *orthogonality* (**M1.3.1**), we construct an orthogonality matrix. The classes of a classification denote both the columns and rows of a $n \times n$ matrix. A cell is filled with a zero if two classes are independent or a one if the class in the row implies the class in the column. The entries on the main diagonal are not considered as the dependency relation between two classes is irreflexive. The orthogonality is the fraction of $|C|^2 - |C|$ minus the number of cells filled with ones and the cell count. A low orthogonality, i.e., many dependencies between classes, indicates unclear boundaries [22]. For instance, if *Behavior* uncertainty implicates *Scenario* uncertainty, the orthogonality is decreased. Overall, a classification with bad structural quality yields ambiguous results and should be adapted.

Goal G2 Figure 9.2 shows the tree of the second goal that is also based on the aforementioned evaluation method [132]. We focus on the classification's *applicability*, i.e., whether the classification is understandable and usable, see *Usable Security* [223]. Kaplan et al. [132] propose to conduct a user study. A classification with suitable structural quality (**G1**) can still be bad if does not yield consistent results results by different users. Classifications are meant to communicate. We consider three evaluation questions:

Q2.1 Does using the classification produce consistent and thus *reliable* results?

Q2.2 Does using the classification produce *correct* results?

Q2.3 Is the classification *easy to use* and easy to understand?

The first Question **Q2.1** asks about the *reliability*, where we evaluate whether different user's results are consistent when applying the classification. An ambiguous classification with inconsistent results indicates a lack of preciseness regarding classes, categories, or their description. Here, we measure the relative size of the largest *consensus* (**M2.1.1**) among all users of the classification. A low consensus indicates a lack of comprehensibility of class names, categories, and their descriptions. For instance, if three out of five users classify an uncertainty as *Behavior* uncertainty, and the other two users classify it as *Component* uncertainty, the largest consensus is three out of five.

The second Question **Q2.2** asks about the *correctness* of the user's results by applying the classification. This can be evaluated by comparing the individual results to a predefined gold standard. As the comparison of users' and experts' results resembles a binary classification, we can apply the terminology of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) [198, 208]. We measure the true positive rate, i.e., the *recall* (**M2.2.1**), which is calculated as:

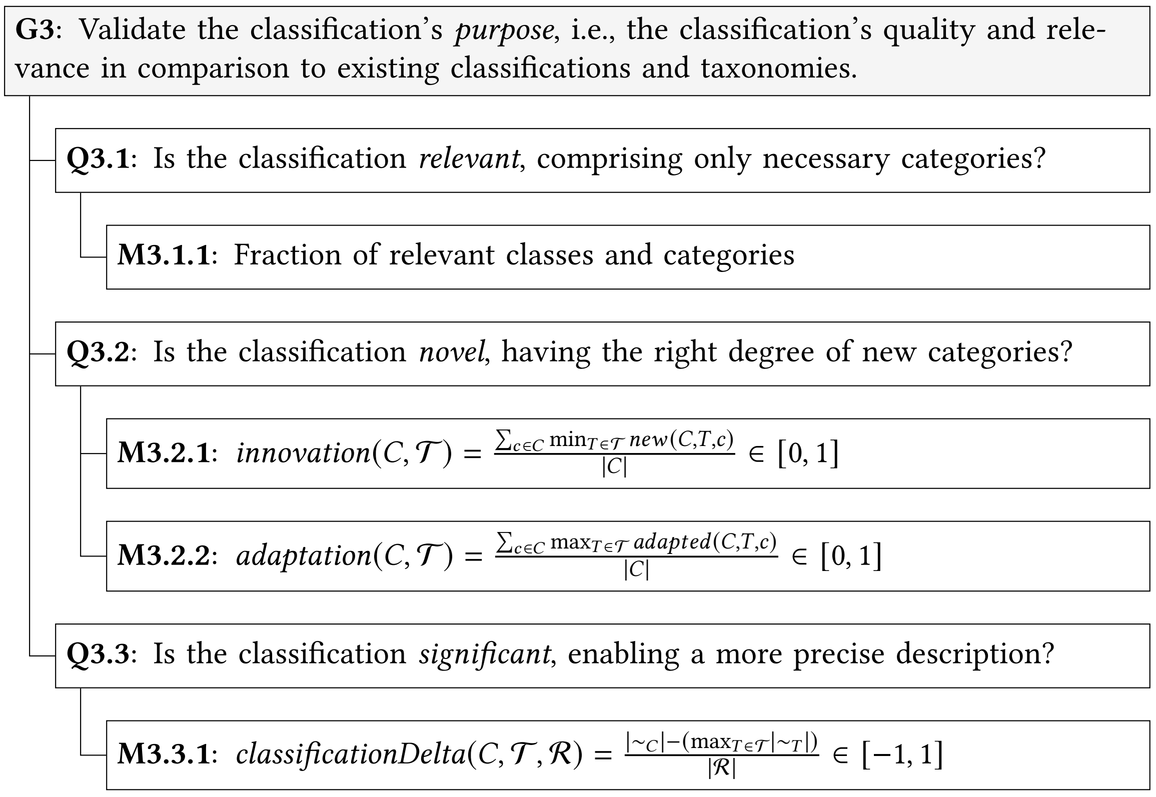
$$recall = \frac{TP}{TP + FN}$$

A low recall indicates that users could not benefit from applying the classification. For instance, if only one out of five users correctly identifies an uncertainty source as *Behavior* uncertainty, this indicates a lack of understandability of this option.

The third Question **Q2.3** asks about the *ease of use* of the classification. Besides the objective measures of concise and correct classification results, this question focuses on the perceived quality, which is subjective. Kaplan et al. [132] recommend using a standardized questionnaire, e.g., the System Usability Scale (SUS) [153] (**M2.3.1**). This questionnaire comprises ten questions regarding the usability, perceived complexity, and ease of use. Each question can be answered on a scale from 1 to 5, which is the base to calculate a score between 0 and 100 [153]. A low score indicates a lack of usability or understandability that can be addressed, e.g., by increasing the consistency of naming classes or by improving textual descriptions. Additionally, participants can be asked if they understand the categories and find them helpful and whether they experienced a knowledge gain by participating in the user study. Overall, a classification has to yield consistent and correct results without requiring too much effort in order to be usable.

Goal G3 Figure 9.3 shows the tree of the third goal. Like the previous goals, this goal follows the evaluation method for taxonomies [132]. This goal considers the classification's *purpose*, i.e., its relevance and improvement compared to the state of the art[2]. A classification with a suitable structure (**G1**) and applicability (**G2**) could still lack novelty. Then, reusing existing classifications is preferred [132]. We consider three questions:

[2] One could question whether *purpose* is a property that can be quantitatively evaluated or whether it is better to argue when discussing related work. We agree with this point of view. Nevertheless, we use the term *purpose* in this evaluation to remain consistent with the evaluation method of Kaplan et al. [132].



G3: Validate the classification's *purpose*, i.e., the classification's quality and relevance in comparison to existing classifications and taxonomies.

Q3.1: Is the classification *relevant*, comprising only necessary categories?

M3.1.1: Fraction of relevant classes and categories

Q3.2: Is the classification *novel*, having the right degree of new categories?

M3.2.1: $innovation(C, \mathcal{T}) = \frac{\sum_{c \in C} \min_{T \in \mathcal{T}} new(C,T,c)}{|C|} \in [0, 1]$

M3.2.2: $adaptation(C, \mathcal{T}) = \frac{\sum_{c \in C} \max_{T \in \mathcal{T}} adapted(C,T,c)}{|C|} \in [0, 1]$

Q3.3: Is the classification *significant*, enabling a more precise description?

M3.3.1: $classificationDelta(C, \mathcal{T}, \mathcal{R}) = \frac{|\sim_C| - (\max_{T \in \mathcal{T}} |\sim_T|)}{|\mathcal{R}|} \in [-1, 1]$

Figure 9.3.: Overview of the GQM plan of the third goal regarding Contribution **C1**.

Q3.1 Is the classification *relevant*, comprising only necessary categories?

Q3.2 Is the classification *novel*, having the right degree of new categories?

Q3.3 Is the classification *significant*, enabling a more precise description?

The first question **Q3.1** asks about the *relevance* of the classification, where we evaluate whether each category helps the purpose of the classification. In our case, the purpose is to understand the impact of uncertainty sources on confidentiality. Here, we question the relevance of all classes $c \in C$ and also of all categories. We measure the *relevance* (**M3.1.1**) as fraction of relevant classes and categories. Note that this metric is not similar to the comparison of classes and terms of Goal **G1** but considers the relevance of classes and categories as means to an end. A low relevance indicates that the classification contains irrelevant elements that should be removed.

The second question **Q3.2** asks about the *novelty* of the classification compared to previous classifications and taxonomies, i.e., the state of the art. While a research increment should contain some degree of novelty, it should also refer to existing concepts to increase validity. Thus, we measure both how many classes and categories are new and also how many of them are adapted. The trade-off between both measures depends on the purpose, e.g., a classification that combines existing classifications has a lower fraction of novel classes and categories. To this end, we measure *innovation* (**M3.2.1**) and *adaption* (**M3.2.2**). Given a classification C with classes and categories $c \in C$, a finite set of previous classifications

$T \in \mathcal{T}$ with classes and categories $d \in T$, where $\simeq \subseteq C \times T$ denotes that a class or category is adapted. Then, a class or category is new if $c \neq d$ and $c \not\simeq d$ for all $d \in T$. Otherwise, a class or category $c \in C$ is adapted if $c \simeq d$ for any $d \in T$. Based on this, we denote:

$$innovation(C, \mathcal{T}) = \frac{\sum_{c \in C} \min_{T \in \mathcal{T}} new(C, T, c)}{|C|} \in [0, 1]$$

A low innovation indicates a small difference compared to state of the art. However, this has to be interpreted with regard to the purpose of the classification, as discussed above. Thus, there is no *best* value. Nevertheless, measuring the innovation and providing arguments for the measured value helps clarify the classification's purpose. Extreme values close to zero or one are the most difficult to justify here. For instance, just copying an existing taxonomy would yield an innovation of zero. Additionally, we denote:

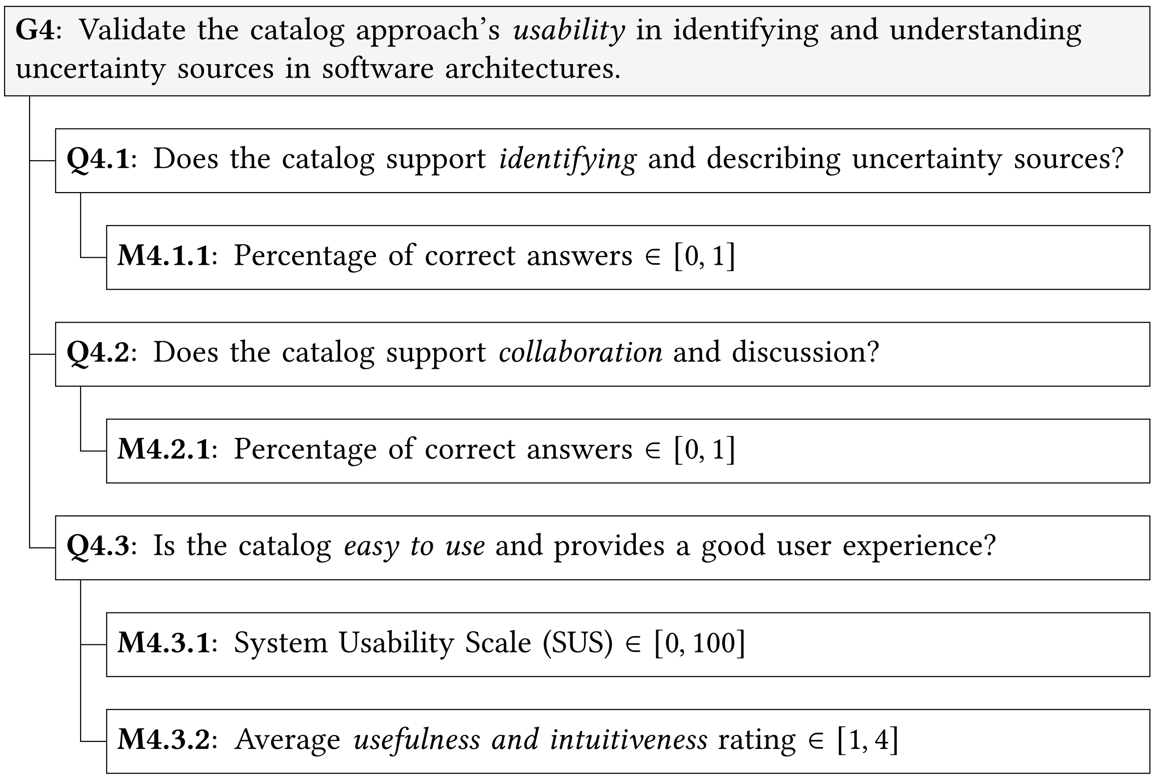
$$adaptation(C, \mathcal{T}) = \frac{\sum_{c \in C} \max_{T \in \mathcal{T}} adapted(C, T, c)}{|C|} \in [0, 1]$$

A low adaptation indicates a large difference compared to the state of the art. Similarly to the innovation, this can be desirable. Nevertheless, building on the state of the art may increase the validity. For instance, a previous taxonomy of uncertainty [195] also referred to existing classifications. Both metrics indicate the strength of the relation of the classification to other taxonomies.

The third question **Q3.3** asks about the *significance*, where we evaluate whether the classification enables a more precise description of the objects under study with respect to its purpose. Put simply, as our classification is aimed towards describing uncertainty with respect to confidentiality, it should enable a more precise description in this area than the state of the art. Given a classification C, a finite set of previous classifications $T \in \mathcal{T}$, and a finite set of objects under study $\mathcal{R}$, $\sim_T$ denotes that a pair of objects under study are classified identically with respect to the classification T, forming an equivalence class. We measure the *classificationDelta* (**M3.3.1**), that describes whether or not a classification is able to yield more and smaller equivalence class than the most precise existing classification, which indicates a higher precision. We denote:

$$classificationDelta(C, \mathcal{T}, \mathcal{R}) = \frac{|\sim_C| - (\max_{T \in \mathcal{T}} |\sim_T|)}{|\mathcal{R}|} \in [-1, 1]$$

A negative delta indicates that an already existing classification is more precise and could be used instead. A positive delta indicates an increase in preciseness, which is what we aim for. For instance, if we provide three relevant options to describe uncertainty sources where other taxonomies only provide one, the delta can be increased. However, only optimizing for the classification delta impairs other metrics, e.g., regarding the structure's suitability (**G1**), and applicability (**G2**). Thus, a good trade-off with respect to the purpose of the classification has to be achieved. If the classification fails the evaluation of purpose, it represents no significant improvement over the state of the art.



G4: Validate the catalog approach's *usability* in identifying and understanding uncertainty sources in software architectures.

Q4.1: Does the catalog support *identifying* and describing uncertainty sources?

M4.1.1: Percentage of correct answers $\in [0, 1]$

Q4.2: Does the catalog support *collaboration* and discussion?

M4.2.1: Percentage of correct answers $\in [0, 1]$

Q4.3: Is the catalog *easy to use* and provides a good user experience?

M4.3.1: System Usability Scale (SUS) $\in [0, 100]$

M4.3.2: Average *usefulness and intuitiveness* rating $\in [1, 4]$

Figure 9.4.: Overview of the GQM plan of the fourth goal regarding Contribution **C1**.

Goal G4 Figure 9.4 shows the tree of the fourth goal. This goal aims towards the uncertainty source catalog, which is realized with ARC^{3}N. We evaluate the *usability* of the catalog approach regarding identifying and understanding uncertainty sources. These are also the desired qualities of the approach, described in Chapter 5. To this end, a user study is the preferred evaluation approach. We consider three evaluation questions:

Q4.1 Does the catalog support *identifying* and describing uncertainty sources?

Q4.2 Does the catalog support *collaboration* and discussion?

Q4.3 Is the catalog *easy to use* and provides a good user experience?

The first Question **Q4.1** asks about *identifying* uncertainty and whether the catalog helps in identifying and describing uncertainty sources. This question is central as the catalog addresses the Uncertainty Awareness Problem (UAP), i.e., the problem of identifying uncertainty sources [103]. We measure the *correctness* (**M4.1.1**) as percentage of correct answers. Based on a gold standard of previously introduced uncertainty sources, we rate each answer as either correct or incorrect. A low correctness indicates that the catalog does not sufficiently support the identification of unknown uncertainty sources. For instance, if four out of five uncertainty sources in an architectural model were identified correctly, a high correctness is achieved.

The second Question **Q4.2** asks about the *collaboration* aspect of the catalog, where we evaluate whether users are able to retrieve information from online discussions about uncertainty sources. Besides supporting the identification process, the catalog has a collaborative aspect by providing the means to share knowledge about uncertainty between software architects and institutions, see Section 5.7. We measure the *correctness* (**M4.2.1**) as percentage of correct answers. This approach is the same as in answering the previous Question **Q4.1**. A low correctness indicates that the catalog does not sufficiently support collaboration between users. For instance, if only one out of five users refers to additional material like online discussions, the catalog falls short of promoting collaboration.

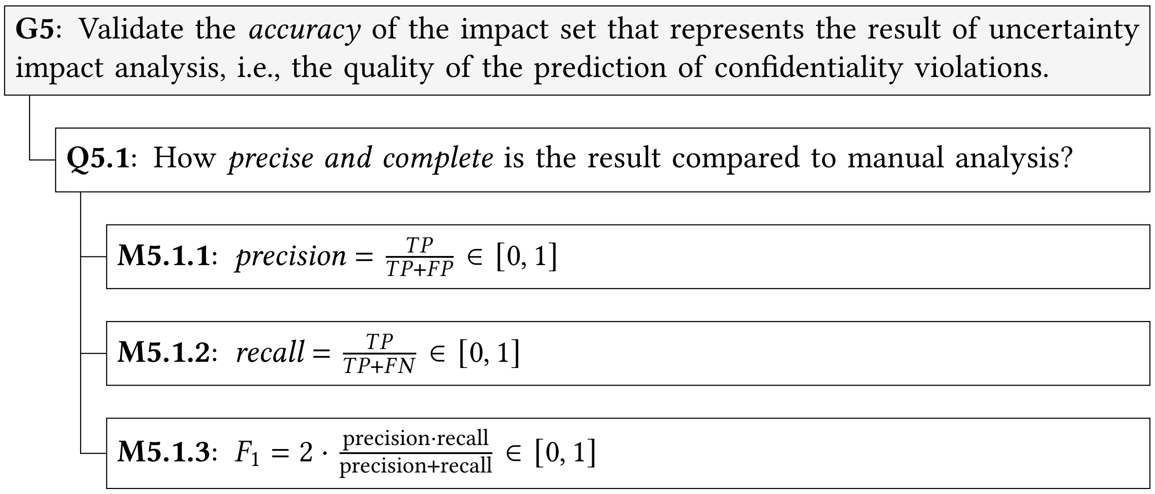
The third Question **Q4.3** asks about the *ease of use* and the user experience using the catalog and its tool support 🔧 ARC^{3}N. This question is similar to **Q2.3** of evaluating the ease of use of the classification. Furthermore, this enables the comparison of the results regarding the usability of the classification and the catalog approach. We measure the users' satisfaction using the System Usability Scale (SUS) [153] (**M4.3.1**). This questionnaire comprises ten questions and yields a score between 0 and 100, as discussed with Question **Q2.3**. A low score indicates improvement potential regarding the user experience of the catalog and its realization. Additionally, we measure the *usefulness* and *intuitiveness* (**M4.3.2**). We ask about the perceived quality of different aspects of the catalog, like examples, or explanations with respect to the underlying classification [104] on a scale from 1 to 4. We average the individual results for each question separately. A low score helps to identify aspects or features of the catalog approach that do not help in identifying uncertainty sources. This complements the SUS and enables us to derive more precise findings on the catalog approach. For instance, outliers can be used to improve the catalog by either enhancing or removing the aspect or feature. In sum, this goal comprises both objective and subjective measures to evaluate the catalog's usability.

9.1.2. Evaluation Plan for the Second Contribution

The second Contribution **C2** introduces an architecture-based uncertainty impact analysis regarding confidentiality [102]. The architectural propagation of uncertainty is related to change impact analysis [46, 210, 211]. This approach is tool-supported by 🔧 UIA. The evaluation of this contribution comprises 2 goals, with a total of 2 questions and 5 metrics. The evaluation is closely aligned with the evaluation of change impact analysis [210].

Goal G5 Figure 9.5 shows the tree of the fifth goal. This goal focuses on the *accuracy* of the uncertainty impact analysis. This analysis yields an impact set that represents an overestimation of the potential impact of uncertainty on confidentiality, see Chapter 6. Thus, the central quality is to provide an accurate impact set that predicts potential confidentiality violations with only little overestimation [33]. An impact analysis that extensively overestimates the impact is bad—however, an impact analysis that underestimates the impact is worse [102]. We consider one central evaluation question:

Q5.1 How *precise and complete* is the result compared to manual analysis?



G5: Validate the *accuracy* of the impact set that represents the result of uncertainty impact analysis, i.e., the quality of the prediction of confidentiality violations.

Q5.1: How *precise and complete* is the result compared to manual analysis?

M5.1.1: $precision = \frac{TP}{TP+FP} \in [0, 1]$

M5.1.2: $recall = \frac{TP}{TP+FN} \in [0, 1]$

M5.1.3: $F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \in [0, 1]$

Figure 9.5.: Overview of the GQM plan of the fifth goal regarding Contribution **C2**.

The Question **Q5.1** asks about the *precision* and *completeness* of the impact set as the result of the uncertainty impact analysis. We compare this result to confidentiality violations identified by architecture-based confidentiality analysis [36, 236]. We use the common terminology [210] introduced in Section 6.3 and call the *actual impact set*, the set of elements that are affected by uncertainty and would violate confidentiality in a manual confidentiality analysis. We consider the actual impact set to be the ideal result. The uncertainty impact analysis yields an *impact set of uncertainty*, i.e., the set of elements that are potentially affected. Similarly to **Q2.2**, the comparison of these sets enables the application of the terminology known from binary classification, i.e., true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) [198, 208]. An element of the impact set represents a true positive (TP) if it violates confidentiality and is thus also in the actual impact set. If the element is only in the impact set, it is a false positive (FP). If an element of the actual impact set is not found by our analysis, it represents a false negative (FN). We use this terminology in the three metrics used to answer Question **Q5.1**. First, we measure the precision (**M5.1.1**), which is calculated as:

$$precision = \frac{TP}{TP + FP}$$

A low precision indicates a large overestimation. This makes the interpretation of the impact set more difficult as software architects have to manually filter our potential false positives. In the worst case, a lack of precision can lead to an unusable impact set. For instance, if a confidentiality violation occurs in a database component but the full software system is part of the impact set, this renders the result ineffective. Second, we measure the *recall* (**M5.1.2**), which is calculated as:

$$recall = \frac{TP}{TP + FN}$$

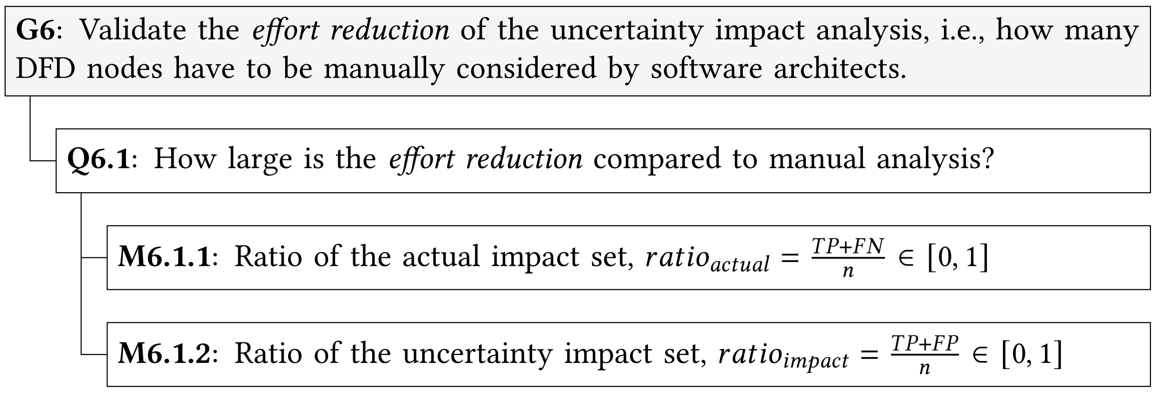

G6: Validate the effort reduction of the uncertainty impact analysis, i.e., how many DFD nodes have to be manually considered by software architects.
Q6.1: How large is the effort reduction compared to manual analysis?
M6.1.1: Ratio of the actual impact set, $ratio_{actual} = \frac{TP+FN}{n} \in [0,1]$
M6.1.2: Ratio of the uncertainty impact set, $ratio_{impact} = \frac{TP+FP}{n} \in [0,1]$

Figure 9.6.: Overview of the GQM plan of the sixth goal regarding Contribution **C2**.

A low recall indicates that the impact set misses elements from the actual impact set, i.e., underestimates potential confidentiality violations. As discussed previously, we prefer a lower precision in order to maximize the recall, as the impact analysis only represents an early estimation of the potential impact. After inspecting the impact set, software architects can identify confidentiality violations due to uncertainty using uncertainty-aware data flow analysis, see the procedure descried in Section 4.1. For instance, if the impact set overestimates the impact in a critical system part, further analysis steps might be required. Third, we measure the *F_1* score (**M5.1.3**), which is calculated as:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \in [0,1]$$

The F_1 score is the harmonic mean of precision and recall and serves as indicator on the overall accuracy. For all metrics, 0 represents the worst, and 1 is the best possible value. Put simply, this goal validates the *accuracy* by comparing the prediction of confidentiality violations to actual existing confidentiality violations.

Goal G6 Figure 9.6 shows the tree of the sixth goal. This goal considers the *effort reduction* of the uncertainty impact analysis. The impact analysis serves software architects to initially understand the impact of uncertainty to plan further analysis and mitigation steps, see Section 4.1. Especially in large software systems, manual analysis is bothersome and erroneous [234]. Thus, an impact analysis should reduce the effort of, in the worst case, manually inspecting every element of the software architecture. This approach to evaluating effort reduction stems from the evaluation of change impact analysis [46, 210] and also has been used for other propagation-based analyses in software architecture [106, 266]. As we focus on confidentiality and data flow-based analysis, we refer to every node of a Data Flow Diagram (DFD) that represents the architectural model, see Section 7.2. We consider one central evaluation question:

Q6.1 How large is the *effort reduction* compared to manual analysis?

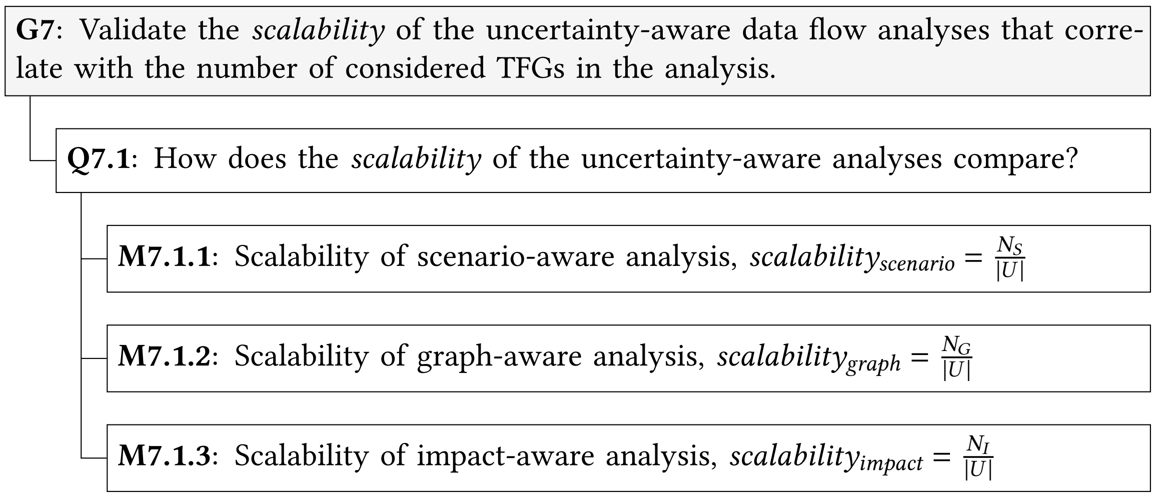
The Question **Q6.1** asks about the *effort reduction* compared to manual analysis. With manual analysis, we mean manually considering every element of the architecture, i.e., every DFD node. Here, confidentiality violations identified by architecture-based confidentiality analysis [36, 236] represent the baseline, similarly to the previous Question **Q5.1**. This enables us to reuse the terminology of binary classification [198, 208]. Elements that actually contain confidentiality violations represent true positives (TP), elements that were missed represent false negatives (FN), and overestimated elements represent false positives (FP). Independent of the use of an impact analysis, software architects have to consider at least all nodes that represent confidentiality violations. Similar to change impact analysis [210], we measure the *ratio of the actual impact set* (**M6.1.1**), which is calculated as:

$$ratio_{actual} = \frac{TP + FN}{n}$$

Here, *n* represents the total number of DFD nodes. A low ratio indicates that only a few elements of the DFD show confidentiality violations, which represents the ideal results of the impact analysis. For instance, a DFD with 10 nodes could have two nodes that violate confidentiality. In the case of a perfect recall, the set of elements containing confidentiality violations forms a lower bound to the required effort. Using the terminology introduced in Section 6.3 and also used in the previous Question **Q5.1**, the actual impact set should be a subset of the impact set. To compare this minimally required effort to the effort of the impact analysis, we additionally measure the *ratio of the uncertainty impact set* (**M6.1.2**), which is calculated as:

$$ratio_{impact} = \frac{TP + FP}{n}$$

A low ratio indicates that only a few elements are potentially impacted by uncertainty. For instance, a DFD with 10 nodes could have an uncertainty impact set comprising 3 of its nodes. The two metrics not only enable quantifying the effort reduction but also show the overestimation of the impact analysis. In the case of the DFD with 10 nodes, 2 confidentiality violations, and 3 elements in the impact set, the overestimation of 1 node becomes visible. Although we expect the ratio of the actual impact set to be lower than the ratio of the uncertainty impact set, this does not imply that uncertainty impact analysis does not reduce the effort compared to manual confidentiality analysis. As discussed in Section 4.1 and Chapter 7, uncertainty-aware confidentiality analysis requires more detailed models and, thus, modeling effort by software architects. With a reasonable overestimation, uncertainty impact analysis still reduces the overall effort with regard to a first assessment of confidentiality under uncertainty. In sum, the comparison of confidentiality analysis and uncertainty impact analysis enables a thorough evaluation of accuracy and effort reduction.



G7: Validate the *scalability* of the uncertainty-aware data flow analyses that correlate with the number of considered TFGs in the analysis.

Q7.1: How does the *scalability* of the uncertainty-aware analyses compare?

M7.1.1: Scalability of scenario-aware analysis, $scalability_{scenario} = \frac{N_S}{|U|}$

M7.1.2: Scalability of graph-aware analysis, $scalability_{graph} = \frac{N_G}{|U|}$

M7.1.3: Scalability of impact-aware analysis, $scalability_{impact} = \frac{N_I}{|U|}$

Figure 9.7.: Overview of the GQM plan of the seventh goal regarding Contribution **C3**.

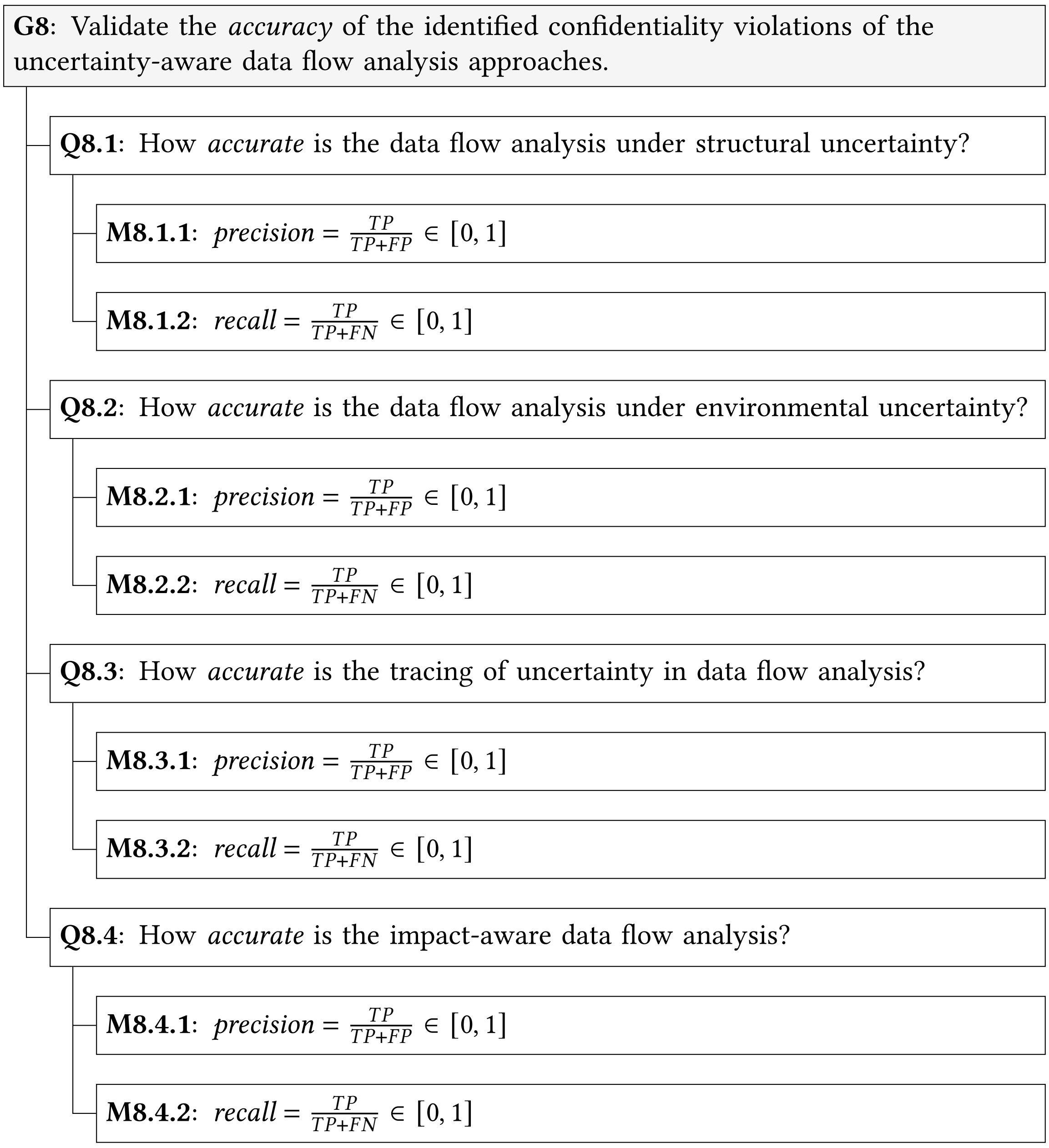
9.1.3. Evaluation Plan for the Third Contribution

The third Contribution **C3** introduces four approaches to uncertainty-aware data flow analysis to identify confidentiality violations with respect to uncertainty. We introduce different approaches that differ in modeling and analyzing uncertainty within the architectural model. The fourth approach is tool-supported by Abunai. The evaluation comprises 2 goals, with a total of 5 questions and 11 metrics. The evaluation is related to the evaluation of architecture-based confidentiality analysis [233, 264].

Goal G7 Figure 9.7 shows the tree of the seventh goal. This goal investigates the *scalability* of the different approaches to uncertainty-aware data flow analysis, introduced in Chapter 7. Although design time analyses are not as critical as runtime analyses regarding the execution time, they still can face combinatorial explosion [143]. The underlying data flow analysis framework, introduced in Section 7.2, scales with the number of analyzed Transpose Flow Graphs (TFGs) [231]. To reach high scalability, the analysis approaches must keep the number of additionally required TFGs to consider uncertainty as low as possible. We consider one central evaluation question:

Q7.1 How does the *scalability* of the uncertainty-aware analyses compare?

The Question **Q7.1** asks about the comparison of the *scalability* of the different uncertainty-aware data flow analysis approaches. We reuse the terminology introduced in Section 7.3 and distinguish between *scenario-aware*, *graph-aware*, and *impact-aware* data flow analysis under uncertainty. To this end, we introduced formulas to calculate the number of required TFGs to detect all confidentiality violations due to uncertainty. This evaluation question applies these formulas to the evaluation scenarios introduced in Chapter 8 to gain insights on the actual impact on the scalability. To compare the different analysis approaches, we calculate the ratio of the number of required TFGs to the number of uncertainty sources. We measure the scalability of *scenario-aware* analysis $scalability_{scenario} = \frac{N_S}{|U|}$ (**M7.1.1**),



G8: Validate the *accuracy* of the identified confidentiality violations of the uncertainty-aware data flow analysis approaches.

- **Q8.1**: How *accurate* is the data flow analysis under structural uncertainty?
  - **M8.1.1**: $precision = \frac{TP}{TP+FP} \in [0, 1]$
  - **M8.1.2**: $recall = \frac{TP}{TP+FN} \in [0, 1]$
- **Q8.2**: How *accurate* is the data flow analysis under environmental uncertainty?
  - **M8.2.1**: $precision = \frac{TP}{TP+FP} \in [0, 1]$
  - **M8.2.2**: $recall = \frac{TP}{TP+FN} \in [0, 1]$
- **Q8.3**: How *accurate* is the tracing of uncertainty in data flow analysis?
  - **M8.3.1**: $precision = \frac{TP}{TP+FP} \in [0, 1]$
  - **M8.3.2**: $recall = \frac{TP}{TP+FN} \in [0, 1]$
- **Q8.4**: How *accurate* is the impact-aware data flow analysis?
  - **M8.4.1**: $precision = \frac{TP}{TP+FP} \in [0, 1]$
  - **M8.4.2**: $recall = \frac{TP}{TP+FN} \in [0, 1]$

Figure 9.8.: Overview of the GQM plan of the eighth goal regarding Contribution **C3**.

the scalability of *graph-aware* analysis $scalability_{graph} = \frac{N_G}{|U|}$ (**M7.1.2**), and the scalability of *impact-aware* analysis $scalability_{impact} = \frac{N_I}{|U|}$ (**M7.1.3**). For all metrics, lower values are better. Here, the number of TFGs without considering uncertainty forms the lower bound. We present this number for all evaluation scenarios in Table 8.2. For instance, if an evaluation scenario has 3 TFGs and one analysis requires 10 TFGs to identify confidentiality violations due to uncertainty, and the second analysis only requires 7 TFGs, the second analysis scales better. While we addressed this theoretically in Section 7.6, answering this question provides us with empirical data and evidence on the actual scalability.

Goal G8 Figure 9.8 shows the tree of the eighth goal[3]. This goal validates the *accuracy* of the uncertainty-aware data flow analyses approaches. Evaluating the accuracy of architecture-based analysis is common in related approaches [142, 233, 264]. It represents the central quality property, as an analysis that misses confidentiality violations quickly looses value. The underlying data flow analysis framework [36] and the architecture-based confidentiality analysis of Seifermann [233] represents the baseline regarding accuracy. We are interested in whether we are able to additionally identify all confidentiality violations that are introduced by uncertainty sources. We consider four evaluation questions:

Q8.1 How *accurate* is the data flow analysis under structural uncertainty?

Q8.2 How *accurate* is the data flow analysis under environmental uncertainty?

Q8.3 How *accurate* is the tracing of uncertainty in data flow analysis?

Q8.4 How *accurate* is the impact-aware data flow analysis?

These four questions correspond to the four analysis approaches that were introduced in Chapter 7. Question **Q8.1** asks about the *accuracy* of data flow analysis under structural uncertainty, introduced in Subsection 7.4.1. Question **Q8.2** asks about the *accuracy* of data flow analysis under environmental uncertainty, introduced in Subsection 7.4.2. Question **Q8.3** asks about the *accuracy* of tracing uncertainty in uncertainty type-agnostic data flow analysis, introduced in Subsection 7.5.1. Question **Q8.4** asks about the *accuracy* of uncertainty impact-aware data flow analysis, introduced in Subsection 7.5.2. Similarly to the accuracy evaluation shown in **Q5.1**, we use the metrics of precision and recall to measure accuracy. We apply these metrics separately for each question, i.e., the Metrics **M8.1.1**, **M8.2.1**, **M8.3.1**, and **M8.4.1** measure precision, which is calculated as:

$$precision = \frac{TP}{TP + FP}$$

A low precision indicates a high number of additionally identified elements that do not violate confidentiality. Similarly to the discussion in **Q5.1**, we prefer recall over precision, i.e., we accept false positives to some degree if we do not miss confidentiality violations. Here, the Metrics **M8.1.2**, **M8.2.2**, **M8.3.2**, and **M8.4.2** measure recall:

$$recall = \frac{TP}{TP + FN}$$

A low recall indicates missing confidentiality violations due to uncertainty. For instance, if a scenario comprises two confidentiality violations due to uncertainty, and we only identify one, this results in a lower recall. If we identify both violations but additionally identify a third one, this results in a lower precision. We consider the *accuracy* (**G8**) to be more important than the *scalability* (**G7**). Especially at design time, a longer analysis execution time is acceptable—however, a severe lack of accuracy is not.

[3] Note that all metrics represent the same calculation of precision and recall. We opt for this representation, as they represent individual measurements, i.e., we consider the instance level, not the type level.

9.2. Evaluation of the Classification and Identification of Uncertainty

In this section, we present the evaluation of the first Contribution **C1**, introduced in Chapter 5. This contribution comprises a classification of software-architectural uncertainty regarding confidentiality and a catalog of uncertainty sources to address the UAP. The evaluation is based on the GQM plan presented in Section 9.1, and includes the Goals **G1**, **G2**, **G3**, and **G4**. We use the evaluation scenarios introduced in Chapter 8. First, we present the evaluation design. Afterward, we show and discuss the results for each goal individually. We conclude with a discussion of threats to validity.

9.2.1. Evaluation Design

The evaluation of the first Contribution **C1** starts with the investigation of the classification *structure's suitability* (**G1**). We use the evaluation scenarios of the *CoronaWarnApp*, presented in Section 8.5, to compare the *purpose* of the classification to the state of the art (**G3**). Additionally, we conduct two user studies, concerning the *applicability* of the uncertainty classification (**G2**) and the *usability* of the uncertainty catalog (**G4**).

For the evaluation of Goal **G1**, we use the metrics by Kaplan et al. [132], which also provide tool support. We measure *laconicity* (**M1.1.1**), *lucidity* (**M1.1.2**), *completeness* (**M1.2.1**), and *soundness* (**M1.2.2**). The input to this measurement is the structure of the classification, i.e., all 8 categories and 27 options. Additionally, we extract 38 terms from existing taxonomies that represent all aspects of the objects under study. Examples of such terms are "fully reducible by acquiring enough knowledge", or "uncertainty refers to user input". The full list of terms can be found in the data set [98]. Last, we provide a mapping of the identified terms to the classification options, i.e., showing how these terms would be classified. Combined, the options, the terms, and the mapping represent the input for the calculation of the metrics of Questions **Q1.1** and **Q1.2**. To measure the *orthogonality* (**M1.3.1**), we construct a self-referencing orthogonality matrix based on our classification's options. An option that is implied by another is not orthogonal and thus overlapping. Based on the 27 options of our classification, we evaluate all $27 \cdot 27 - 27 = 702$ potential combinations.

For the evaluation of Goal **G2**, we conduct a user study with 10 researchers from the domain of software architecture. First, they complete a self-assessment, where they describe their prior knowledge related to the task, e.g., uncertainty, and software architecture. Then, we provide them with a one-page summary of our classification presented in Section 5.3. We show all categories and options and also an application example that demonstrates how to use it, comparable to the running example shown in Chapter 3. All material used in the user study is part of the data set [98]. During the study, the participants have to classify two different tasks within 15 minutes time, respectively. Each task consists of an architecture diagram, a short description, and four uncertainty impacts to classify using our classification. We counterbalance the task order to mitigate learning effects and anonymize the participants' results. Last, they fill out a SUS (**M2.3.1**) to measure the *ease of use* and a questionnaire related to their understanding of our classification. No session takes longer than one hour to mitigate the effects of fatigue. After gathering all results, we measure the *reliability* (**M2.1.1**) by calculating the percentage of the agreement using the largest consensus, and the *correctness* (**M2.2.1**) by comparing the participants' result to a predefined gold standard and calculating the recall.

For the evaluation of Goal **G3**, we use the evaluation scenario of the *CoronaWarnApp*, presented in Section 8.5. This scenario represents the largest software system in our pool of evaluation scenarios, which also has comprehensive documentation. This does not only include architecture documentation but also security analysis and risk assessment. By rolling back design decisions and considering solutions for problems and risks that are related to confidentiality, realistic uncertainties can be analyzed. We create a collection of 28 uncertainty sources that are possible during the design process based on the available documentation and Architectural Design Decisions (ADDs). This collection is also part of the data set [98]. We use it as the baseline for the evaluation of the *purpose*. For each category, we investigate whether it helps to understand the impact of the uncertainty sources and is thus *relevant* (**M3.1.1**). This extends the evaluation of *generality* (**Q1.1**) and *appropriateness* (**Q1.2**) based on a real software system. We compare all categories to other taxonomies [41, 71, 162, 195, 202, 263] of uncertainty to evaluate the classification's *novelty* (**Q3.2**) and measure *innovation* (**M3.2.1**) and *adaptation* (**M3.2.2**), as proposed by Kaplan et al. [132]. We measure the *classification delta* (**M3.3.1**) by classifying all 28 uncertainty sources with our classification and with other taxonomies [41, 162, 195] with a related purpose. As our goal is a higher precision for the impact of uncertainty, we aim for a positive classification delta, i.e., a higher number of smaller equivalence classes.

For the evaluation of Goal **G4**, we conduct a user study with a total of 17 participants, including 7 students, 5 researchers, and 5 practitioners. All participants have at least basic knowledge of software architecture and design decisions. Regarding uncertainty, confidentiality, and security analysis, the expertise is more evenly spread with only very few participants considering themselves experts. We conclude that these are excellent conditions, as our approach shall not require expert knowledge but help beginners and intermediate users. To answer Questions **Q4.1** and **Q4.2**, the participants are given two assignments, which they are asked to complete with the aid of our catalog. Both assignments consist of an architecture diagram and a short scenario description, similar to our running example, see Chapter 3. In the first assignment, participants are tasked to map textual descriptions of annotated uncertainties to concrete uncertainty sources from our catalog. In the second half of the assignment, participants are asked for their opinion on the appropriateness of an uncertainty description and should provide their reasoning. While the former task provides data to evaluate the *correctness* of identifying and describing uncertainty (**M4.1.1**), the latter also considers provided context information and *collaboration* aspects (**M4.2.1**). In the second assignment, participants are tasked with identifying uncertainties present in the provided architecture diagram and giving a short reasoning for each. Here, we do not include any hint to applicable uncertainty sources, requiring the participants to identify potential uncertainties themselves by using our tool support. This also contributes to measure the *correctness* (**M4.1.1**). Similarly to

Question	Metric	Result
Generality (**Q1.1**)	Laconicity (**M1.1.1**)	0.95
	Lucidity (**M1.1.2**)	0.70
Appropriateness (**Q1.2**)	Completeness (**M1.2.1**)	0.97
	Soundness (**M1.2.2**)	1.00
Orthogonality (**Q1.3**)	Orthogonality (**M1.3.1**)	0.99

Table 9.1.: Results of evaluation Goal **G1** regarding the classification structure's suitability.

the previous goals, all materials are included in the data set [98]. To answer Question **Q4.3**, we provide a SUS (**M4.3.1**). In the subsequent feedback session, the participants are asked to rate the *usefulness and intuitiveness* (**M4.3.2**) of the different features of 🔧 ARC^{3}N, see Section 5.7. They are also asked about their learning in the different knowledge areas which they self-assessed at the beginning of the study, and they are allowed to give additional, qualitative feedback. To mitigate fatigue effects, no session took longer than one hour. After all 17 participants' results have been received, we evaluate the results by comparing them to a gold standard that has been created independently and then unified by two researchers. Also, the comparison is conducted independently by two researchers, and all inconsistencies have to be discussed and resolved.

9.2.2. Results and Discussion of the Structure's Suitability

The first Goal **G1** targets the classification structure's suitability, i.e., whether the classification's structure supports the description of uncertainty with regard to confidentiality. This goal comprises three evaluation questions with a total of five metrics. Table 9.1 shows the results that will be explained and interpreted in the following.

The *generality* (**Q1.1**) asks about an appropriate generality level. The *laconicity* (**M1.1.1**) is the fraction of terms of the classification that are uniquely describable. With our classification C and the uncertainties representing the objects under study $\mathcal{R}$, we measure $laconicity(C, \mathcal{R}) = \frac{36}{38} = 0.95$. We find two redundant terms. Uncertainty about a user's input can be classified both as *System Input* and *System Environment*, both belonging to the category *Location*, introduced in Table 5.1. Additionally, the uncertainty that does not affect confidentiality is described both with *none* regarding its impact, shown in Table 5.7 and *none* regarding its severity, shown in Table 5.8. The *lucidity* (**M1.1.2**) is the fraction of options that describe exactly one term. Using the hereabove introduced classification C and objects under study $\mathcal{R}$, we measure $lucidity(C, \mathcal{R}) = \frac{19}{27} = 0.70$. We find several terms that are described by the same option, e.g., *System Structure* describes both uncertainty in components and assembly. Another example is the *Realization Time* of the category *ResolutionTime*, shown in Table 5.5, that includes implementation and deployment as this can be simplified from a design time perspective. Furthermore, the options of the category *Manageability*, shown in Table 5.4 do not detail how to reduce uncertainty, which subsumes multiple terms. The *appropriateness* (**Q1.2**) asks whether only necessary classes

Question	Metric	Result
Reliability (**Q2.1**)	Annotator consensus (**M2.1.1**)	0.69
Correctness (**Q2.2**)	Recall (**M2.2.1**)	0.73
Ease of use (**Q2.3**)	SUS (**M2.3.1**)	68.25

Table 9.2.: Results of evaluation Goal **G2** regarding the classification's applicability.

are in the classification. The *completeness* (**M1.2.1**) is the of terms that can be described using the classification. With the given C and $\mathcal{R}$, we measure $completeness(C, \mathcal{R}) = \frac{37}{38} = 0.97$. The completeness is reduced because we do not explicitly handle known uncertainty sources that never resolve, i.e., the classification lacks an according option in the category *Resolution Time*, shown in Table 5.5. The *soundness* (**M1.2.2**) is the fraction of unnecessary options. We measure $soundness(C, \mathcal{R}) = \frac{27}{27} = 1.0$, i.e., the absence of unnecessary options. Last, the *orthogonality* (**Q1.3**) asks about overlapping classes. The *orthogonality* (**M1.3.1**) counts orthogonal, non-overlapping terms. Using an orthogonality matrix, we measure $orthogonality(C, \mathcal{R}) = \frac{695}{702} = 0.99$. We identify overlapping in 7 of 702 cases, e.g., uncertainty about the system's input implies a behavioral description and there exists the already identified relation between *none* regarding the impact and the severity of uncertainty. The full mapping and the orthogonality matrix are in the data set [98].

Regarding the *laconicity* (**M1.1.1**), we argue that the remaining two redundant terms are totally acceptable and originate due to the increased precision regarding confidentiality. Having categories to describe both the uncertainty impact type and its severity is an intentional design decision to aid software architects. Thus, we accept this small reduction in redundancy. Regarding the *lucidity* (**M1.1.2**), we could reach a higher score by adding more options to the classification. However, we argue that more fine-grained options would only harm the purpose of classifying and clustering uncertainties to understand their impact and mitigation. When evaluating *completeness* (**M1.2.1**), we only identify a lack of means to describe uncertainty that never resolves. However, from a design time point of view, it does not matter whether an uncertainty resolves at run time or never. Regarding *soundness* (**M1.2.2**), the perfect result is expected, as we intentionally build the classification to fit our purpose. Last, there are some overlaps in the *orthogonality* matrix (**M1.3.1**). However, none of the overlaps are comprehensive enough to justify the removal of a category or an option, which would negatively impact the other metrics. We conclude that these results are satisfying regarding *generality* (**Q1.1**), *appropriateness* (**Q1.2**), and *orthogonality* (**Q1.3**). Thus, the classification's structure is suitable (**G1**). We may continue with the second evaluation step as proposed by Kaplan et al. [132].

9.2.3. Results and Discussion of the Applicability

The second Goal **G2** focuses on the classification's applicability, i.e., whether the classification is understandable and usable. This goal comprises three evaluation questions with a total of three metrics. Table 9.2 shows the results. Again, we split the presentation and interpretation of the results.

To address this goal, we conducted a user study with 10 participants. The *reliability* (**Q2.1**) asks whether the classification produces consistent results when used by multiple users. We calculate the relative size of the *largest consensus* (**M2.1.1**) for each category. The average measured agreement is 69 percent. This means that on average, approximately seven out of ten participants agree on a classification. A high agreement is measured in the categories *Location*, *Impact on Confidentiality*, *Severity of the Impact*, and *Reducible by ADD*. The lowest agreement is measured in the category *Resolution Time*. The *correctness* (**Q2.2**) asks whether the user's results are correct. Using a predefined gold standard, we calculate a recall (**M2.2.1**) of 0.73. This means that on average, three out of four classifications are correct, i.e., they match with the goal standard. We find high correctness with a recall of 0.95 in the categories *Location* and *Architectural Element Type*. The lowest correctness is measured in the categories *Type* and *Resolution Time* with a recall of 0.6 and 0.65, respectively. We also find that the correctness fluctuates depending on the uncertainty impact that has to be classified by the participants. The *ease of use* (**Q2.3**) asks whether the classification is easy to understand. Using a SUS (**M2.3.1**), we calculate the result of 68.25. On the one hand, the participants valued the guidance and consistency of the classification. On the other hand, they felt not confident in using the classification and assumed that they would require additional training. In the questionnaire after the user study, most of the categories were considered understandable and helpful in describing the impact of uncertainty. The only outlier is the category *Type*. Raw results are in our data set [98].

Before discussing the results, we find that the group of participants was appropriate for this user study. The self-assessment shows that the researchers have comprehensive knowledge about software architecture but no or only little knowledge about uncertainty, or confidentiality. Architecture-related categories of the self-assessment were rated with an average knowledge of 3 out of 4. Uncertainty and security-related categories of the self-assessment were rated between 1.5 and 2 out of 4. Regarding *largest consensus* (**M2.1.1**) and the *correctness* (**M2.2.1**), one explanation of the non-perfect results is the earlier description of several categories, which were ambiguous and thus have been refined. Based on the participants' feedback, we find that the result can also be explained with the short case descriptions of about a quarter of a page and the hard timing constraints. Short descriptions of fictional architectures can leave a large room for interpretation. In view of the fact that the participants had no prior experience in classifying uncertainty, these results still are satisfying. Based on the SUS (**M2.3.1**) and the questionnaire, we find a lack in the category *Type*. However, the value of this category has already been discussed in multiple publications [162, 263] and helps in design time confidentiality analysis, see Chapter 5. Also, based on the participants' feedback, we summarize that our classification is a sufficiently useful tool for understanding the impact of uncertainty, but it requires some familiarization. Most of the participants welcomed a lively debate about their classifications after the study sessions which is what we aimed for. This finding is supported by the post-survey questionnaire, where the participants rated on average with 3.3 out of 4 that they learned something about confidentiality and the different types of uncertainty, i.e., the categories where they previously stated a lack of experience. We

Question	Metric	Result
Relevance (**Q3.1**)	Fraction of relevant classes (**M3.1.1**)	1.00
Novelty (**Q3.2**)	Innovation (**M3.2.1**)	0.49
	Adaptation (**M3.2.2**)	0.51
Significance (**Q3.3**)	Classification delta (**M3.3.1**)	0.54

Table 9.3.: Results of evaluation Goal **G3** regarding the classification's purpose.

conclude that these results are not perfect but support the *reliability* (**Q2.1**), *correctness* (**Q2.2**), and *ease of use* (**Q2.3**). Thus, the classification is applicable (**G2**). This evaluation also shows the need for appropriate documentation and assistance, which motivates our tooling 🔧 ARC^{3}N.

9.2.4. Results and Discussion of the Purpose

The third Goal **G3** targets the classification's purpose, i.e., its quality and relevance compared to existing classifications from related work. This goal comprises three evaluation questions with a total of four metrics. Table 9.3 shows the results that will be explained and interpreted in the following. Note that we combine the presentation and interpretation of this goal's results as the majority of its metrics depend on argumentation.

The *relevance* (**Q3.1**) asks about the relevance of the classification, i.e., whether it only contains elements that are required to describe the objects under study and is based on argumentation. To answer this question, the fraction of classes relevant for the purpose of the classification shall be investigated (**M3.1.1**). The purpose of our classification is to describe the impact of software-architectural uncertainty on confidentiality. The relevance of the category *Location* has already been discussed in other work [41, 195]. We use *Architectural Element Type* because this enables the connection to architectural modeling and analysis. We use this category throughout Chapter 6 and Chapter 7 as the primary dimension to distinguish the five uncertainty types. We argue that *Type* and *Manageability* are better to describe uncertainty than only to refer to its level because this helps in choosing appropriate mitigation strategies. For instance, a scenario-based, reducible uncertainty can be handled differently from a recognized, irreducible uncertainty. *Resolution Time*, *Reducible by ADD* and both impact-related categories can be used in prioritization together with connected ADDs. This prioritization and connection to ADDs is important because it helps structure the software design and also helps focus modeling and analysis capabilities [150]. *Impact on Confidentiality* and *Severity of the Impact* are reserved for an early estimation and help to streamline the subsequent analysis procedure, see Section 4.1. We close that no category can omitted without significantly reducing the expressiveness and thus the fraction of relevant classes is 1.0. The *novelty* (**Q3.2**) asks about the degree of new and adapted categories and classes compared to the state of the art. We investigate the classification and calculate *innovation* (**M3.2.1**) and *adaptation* (**M3.2.2**). Examples of adopted categories are the *Resolution Time* or *Severity of the Impact.*

Question	Metric	Result
Identification (**Q4.1**)	Correctness (**M4.1.1**)	0.88
Collaboration (**Q4.2**)	Correctness (**M4.2.1**)	0.65
Ease of use (**Q4.3**)	SUS (**M4.3.1**)	69.71
	Average rating (**M4.3.2**)	3.1/4

Table 9.4.: Results of evaluation Goal **G4** regarding the catalog's usability.

An example of a new option is the *partial reducibility*. We find the hard distinction between manageable and irreducible not precise enough for design time mitigation. Also in our running example, presented in Chapter 3, we were able to better understand and partially reduce the impact of uncertainty. This option was independently incorporated in the current working version of Precise Semantics for Uncertainty Modeling (PSUM) standard [184], which underlines its relevance. A full discussion of the relation of the classification to the state of the art can be found in Subsection 5.2.3 and in our data set [98]. We are right in the middle between *innovation* with $\frac{17}{35} = 0.49$ new categories and options, and*adaption* with $\frac{18}{35} = 0.51$ categories and options. This is expected as we build upon existing taxonomies but extend them to fit our purpose. The *significance* (**Q3.3**) asks whether the classification is able to provide a more detailed description of the objects under study. We measure the *classification delta* (**M3.3.1**) by comparing our classification to the most related taxonomies [41, 162, 195]. Here, our classification is able to distinguish the 28 uncertainties into 23 equivalence classes. The taxonomy of Bures et al. [41] yields 5 equivalence classes, the classification framework of Mahdavi-Hezavehi et al. [162] yields 8 equivalence classes and the taxonomy of Perez-Palacin and Mirandola [195] yields 4 equivalence classes. Thus, the most precise result of the state of the art is a distinction into 8 equivalence classes and the classification delta is $\frac{23-8}{28} = 0.54$. As we aimed for higher precision, a value higher than 0 is sufficient. This indicates that our classification is able to describe the relation of uncertainty and confidentiality with notably higher precision than the most precise already existing classification. We conclude that these results support the *relevance* (**Q3.1**), *novelty* (**Q3.2**), and *significance* (**Q3.3**) and thus justify the *purpose* (**G3**). This completes the evaluation method of Kaplan et al. [132].

9.2.5. Results and Discussion of the Usability

The fourth Goal **G4** revolves around the catalog's usability, i.e., whether the catalog approach helps identify and understand uncertainty sources. This goal comprises three evaluation questions with a total of four metrics. Table 9.4 shows the results. Here, we split again the presentation and interpretation of the results.

To address this goal, we conducted a user study with 17 participants. Similarly to the user study of Goal **G2**, the participants filled a self-assessment prior to the study. The correctness of the *identification* (**Q4.1**) asks whether the catalog supports the identification and description of uncertainty sources in architectural models. We calculate the percentage

Uncertainty Knowledge	Identification (**M4.1.1**)	Collaboration (**M4.2.1**)
No Prior Knowledge	0.74	0.60
Little Prior Knowledge	0.94	0.43
Good Prior Knowledge	0.94	1.00
Expert Prior Knowledge	1.00	1.00
Average of all answers	0.88	0.65

Table 9.5.: Percentage of correct answers in the user study of Goal **G4** grouped by prior knowledge.

of *correct answers* (**M4.1.1**) by comparing the participants' results to a predefined gold standard. Here, the participants reached an average correctness of 88%. This means on average, approximately nine out of ten uncertainty sources were identified and described correctly. Table 9.5 maps the percentage of correct answers onto the reported knowledge on the uncertainty of the self-assessment. Here, the prior knowledge correlates with the correctness, starting at 74% without prior knowledge, and jumping to 94% with little prior knowledge. The correctness of the *collaboration* (**Q4.2**) asks whether the catalog supports the collaboration. Again, we calculate the percentage of *correct answers* (**M4.2.1**). Table 9.5 shows a larger gap between participants with no or little knowledge and more experienced users. Nevertheless, the average correctness reaches 65%. Note that both average values take the distribution of the participants' prior knowledge into account. The *ease of use* asks about the user experience of the catalog approach and its tool support 🔧 ARC^{3}N. We use a SUS (**M4.3.1**) with a cumulative score of 69.71 and ask the participants to rate *usefulness and intuitiveness* (**M4.3.2**). The average usefulness was rated 3.2 and the average intuitiveness was 3.0 on a scale of 1 to 4, averaging to 3.1. The results regarding individual features show that the participants especially liked the explanations using example scenarios and graphics, with an average score of 3.76 out of 4 in both cases. Relating uncertainty sources to the classification and showing the inheritance relationship of uncertainties was considered useful but not intuitive, with scores of 2.1 and 2.3, respectively. The raw results of all participants and all calculations are part of the data set [98].

Similarly to the user study conducted to evaluate Goal **G2**, we find that the group of participants was appropriate for this user study. The participants had little to no prior knowledge regarding uncertainty and confidentiality but were experienced regarding software architecture and ADDs. Regarding the *correctness* of the identification (**M4.1.1**), we find that an overall correctness of 88% is satisfying. Interestingly, the correctness jumps from 74% to 94% depending on the prior knowledge, indicating that our catalog facilitates the correct identification of uncertainties even with little knowledge. We assume this group would greatly benefit from our approach, as experts already know enough about analyzing uncertainty, even without support. The correctness of 100% with expert knowledge in all questions is not surprising. Compared to the user study of Goal **G2**, which had a similar setup, we reach a notably higher correctness score in describing uncertainty, i.e., 88% compared to 73% (**M2.2.1**). This supports the need for a tool-supported catalog to complement a classification and make it useful. Regarding the *correctness* of the collaboration (**M4.2.1**), the results are acceptable but not optimal. We see a similar correlation between experience and correctness, but the results with limited prior knowledge are worse. One possible explanation could be that the earlier prototype that has been used for this user study did not highlight discussions enough. We addressed this shortcoming, enhancing both the structure and visualization quality of 🔧 ARC^{3}N, and include both variants in the data set [98]. Regarding the SUS (**M4.3.1**), the score of 69.71 indicates a good usability, comparable to the usability of the classification of 68.25 (**M2.3.1**). Regarding the participants' rating of *intuitiveness* and *usefulness* (**M4.3.2**), we also addressed the lack of intuitiveness in an enhanced version of the tool support. Overall, the majority of features were rated to be useful, which is satisfying. At the end of the user study, we asked the participants about their learnings in different categories on a scale from 1 to 4. The participants stated that they had learned the most about uncertainty and the different uncertainty types with a score of 3.4 and 3.7, respectively. This represents a promising result as it indicates that the catalog indeed complements and supports the classification and provides increased usability. We conclude that these results support the quality of the *identification* (**Q4.1**) and *collaboration* (**Q4.2**), and *ease of use* (**Q4.3**). Thus, the catalog approach is *usable* (**G4**). Combined with the evaluation results of Goal **G2** regarding the applicability of the classification, this also supports the interplay of our classification with our tool-supported catalog approach.

9.2.6. Threats to Validity

We conclude the evaluation of the first Contribution **C1** with a discussion of the threats to validity of Goal **G1** – **G4**. As the evaluation presented in this section is partially based on evaluation scenarios, we follow the guidelines of Runeson and Höst [212]. They propose to discuss internal validity, external validity, construct validity, and reliability. These categories are also suitable for other validation designs and have also been used to discuss the validity of the evaluation in related work [233, 264]. Structuring the discussion of threats to validity addresses the lack of guideline-based validity discussions of software architectural research, identified by Konersmann et al. [142].

Internal validity The internal validity reflects whether the evaluation results depend only on the factors examined, without being influenced by other factors. The biggest threat to the evaluation of the structure's suitability (**G1**) is that it is only performed by a single researcher based on limited experience. Here, a different selection of terms for the calculation of metrics like laconicity (**M1.1.1**) and lucidity (**M1.1.2**) could lead to different results. We adhere to the method, metrics, and guidelines of Kaplan et al. [132] as closely as possible and also discussed the procedure with the original authors to address this threat. Another threat to internal validity arises due to the self-defined gold standard in the evaluation of applicability (**G2**). To counter this, we try to keep the derived uncertainty sources as close as possible to the documented design decision of the *CoronaWarnApp*, see Section 8.5. Other threats to the internal validity arise when conducting user studies. To counter learning effects of the participants, we counterbalance the tasks of the first user study (**G2**). Additionally, we measure the experience of all participants upfront and interpret the results of both user studies (**G2** and **G4**) with regard to the prior knowledge of the participants. Last, fatigue effects can degrade the results of a user study. We address this concern by keeping the total duration of each session lower than one hour.

External validity The external validity reflects the generalizability of the evaluation results to other cases or domains. Regarding the evaluation of the classification, the generalizability of our results is threatened by the number of participants in our user study (**G2**) and the selection of the evaluation scenario (**G3**). Still, we argue that both were large enough to identify a general trend of applicability (**G2**) and purpose (**G3**). The participating researchers deal with software architecture in their daily work. Additionally, the participants classified two architecture models with four uncertainties each, which yields a set of 80 classified uncertainties and 640 selected options. We argue that this represents a reasonable size to reason about the quality of the classification. Additionally, the *Corona Warn App* is a large open-source system that was actively observed by the community during the time of this research. Our catalog of uncertainty shows that the identified uncertainty sources are independent of the domain and can be applied to other systems, e.g., to a Cyber-Physical System (CPS) or mobility systems. Additional information on the construction of the evaluation scenario can be found in [23]. Regarding the evaluation of the catalog (**G4**), the study's limitation is the sample of 17 participants from various occupational backgrounds. While this sample size may not fully reflect the broader population of software architects, the diversity in participants' experiences and backgrounds still offers valuable insights into the utility of the approach. We do not compare the user study results with a baseline approach as we did not find a suitable one. We do not find it expedient to compare our approach to a control group, which tries to identify uncertainty sources without any tool support just by relying on classification papers [104, 202]. We also do not perform significance tests. Nevertheless, comparing the results of the user studies of the classification (**G2**) and the catalog (**G4**) implies that we outperform the accuracy of classification-based assessment without tool support.

Construct validity The construct validity reflects whether the measurements are suitable for representing the evaluation objectives. Overall, we apply a GQM-based evaluation plan [16, 17] to minimize the risk of collecting data that does not help to target the evaluation goals. Additionally, we use metrics from other work whenever possible. For evaluating the classification (**G1** – **G3**), we use the evaluation method of Kaplan et al. [132], which itself relates to other work in this area when defining goals, questions, and metrics. Here, the SUS (**G2**) provides a standardized format, which might not fit the evaluation of classifications. We mitigate this using a questionnaire that yields similar results regarding usability. The same applies to the evaluation of the catalog's usability (**G4**). Additionally, our interpretation of the results of all evaluation goals does not depend on a single metric but takes into account the bigger picture. This minimizes the risk of false conclusions due to a single defective measurement. Last, we want to state that the evaluation of our classification surpasses the comprehensiveness of other evaluations of taxonomies in related work [41, 162, 195].

Reliability The reliability reflects the repeatability and whether the evaluation results are dependent on the specific researchers. Here, the evaluation of the classification (**G1** – **G3**) is only performed by a single researcher. Different evaluators could produce different conclusions. We address this by rigorously following a simultaneously published evaluation method [132] that other researchers can also follow. The correctness results (**G4**) stem from the assessment of the participants' answers relying on individual researchers which can inherently not be standardized. The user study's execution over nine online meetings could also have influenced the consistency of the results [84]. To mitigate these risks, we use strictly structured sessions, and two researchers evaluate the results independently. To maximize the repeatability of the evaluation, we provide a data set [98] comprising all raw data, study results, calculations, questionnaires, interpretations, and different versions of the catalog's tool support 🔧 ARC^{3}N.

9.3. Evaluation of the Uncertainty Impact Analysis

In this section, we present the evaluation of the second Contribution **C2**, introduced in Chapter 6. This contribution comprises an architecture-based uncertainty impact analysis to predict confidentiality violations. The evaluation is based on the GQM plan presented in Section 9.1, and includes the Goals **G5** and **G6**. We use the evaluation scenarios introduced in Chapter 8. First, we present the evaluation design. Afterward, we show and discuss the results for each goal individually. We conclude with a discussion of threats to validity.

9.3.1. Evaluation Design

The evaluation of the second Contribution **C2** comprises investigating the *accuracy* (**G5**) and the *effort reduction* (**G6**) of uncertainty impact analysis. We use the evaluation scenarios of the *CoronaWarnApp*, presented in Section 8.5, and *MobilityAsAService*, presented in Section 8.6. These two scenarios are the largest software systems, where uncertainty impact analysis is most valuable. In order to gain insights into the actual prediction quality, we focus on these two evaluation scenarios. Both evaluation scenarios represent non-trivial software systems, with 21 and 18 components, respectively. The resulting DFD of the *CoronaWarnApp* comprises 506 nodes and the DFD of *MobilityAsAService* has 267 nodes. To gain more precise results, we split the evaluation into sub-scenarios. The *CoronaWarnApp* already consists of four sub-scenarios that comprise two uncertainty sources each. Regarding *MobilityAsAService*, we consider each of the five uncertainty sources as a sub-scenario. All models and sub-scenarios can be found in the data set [98].

Question	Metric	Result
Accuracy (**Q5.1**)	Precision (**M5.1.1**)	0.78
	Recall (**M5.1.2**)	1.00
	F_1 score (**M5.1.3**)	0.88

Table 9.6.: Results of evaluation Goal **G5** regarding the uncertainty impact analysis' accuracy.

For the evaluation of Goal **G5**, we use 🔧 UIA as automated uncertainty impact analysis to calculate the individual *impact sets of uncertainty* for all scenarios[4]. These sets consist of DFD nodes that were identified based on the uncertainty propagation algorithms described in Section 6.5. Afterward, we alter the modeled system to violate confidentiality for each annotated uncertainty source, e.g., by adding a bug in the validation of the user input or changing the server deployment locations. Here, we follow the description of the evaluation scenarios, its confidentiality requirements, and publicly available documentation. This represents the manual confidentiality analysis of what-if scenarios [233]. Manually modeling and analyzing confidentiality due to uncertainty requires more effort [101] but only yields actual violations, i.e., the *actual impact set*. Based on both sets, we evaluate the *accuracy* to answer Question **Q5.1**. We count nodes as true positive (TP) if they are contained in both sets and represent correctly predicted confidentiality violations. Elements of the impact set that do not violate confidentiality and thus overestimate the actual impact set are counted as false positives (FP). DFD nodes that neither violate confidentiality nor have been identified by our analysis are classified as true negative (TN). Elements of the actual impact set that were not identified by our analysis are counted as false negatives (FN). This enables calculating *precision* (**M5.1.1**), *recall* (**M5.1.2**), and the *F_1 score* (**M5.1.3**).

For the evaluation of Goal **G6**, we follow a similar approach. We use 🔧 UIA as automated uncertainty impact analysis to calculate the individual *impact sets of uncertainty* and manual confidentiality analysis [233] to calculate the *actual impact set*. Additionally, we use the aforementioned total count of DFD nodes to evaluate the *effort reduction* to answer Question **Q6.1**. We follow the same classification of elements in both sets as in Question **Q5.1**. This enables calculating the ratio of the actual impact set, $ratio_{actual}$ (**M6.1.1**), and the ratio of the uncertainty impact set, $ratio_{impact}$ (**M6.1.2**).

9.3.2. Results and Discussion of the Accuracy

The fifth Goal **G5** targets the accuracy of the uncertainty impact analysis, i.e., whether the resulting uncertainty impact set is both precise and complete. This goal comprises one

[4] Note that the current implementation of 🔧 UIA uses an earlier version of the analysis framework, presented in Section 7.2. Thus, the number of DFD nodes slightly differs compared to the size metrics of the evaluation scenarios, presented in Section 8.1. However, this does not negatively impact the evaluation, as we calculate all metrics in this section based on the correct reference values.

	CoronaWarnApp				*MobilityAsAService*					
	S1	*S2*	*S3*	*S4*	*S1*	*S2*	*S3*	*S4*	*S5*	∅
Precision	0.838	1.000	0.840	0.882	0.095	0.923	0.500	0.967	1.000	0.783
Recall	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
F_1 score	0.912	1.000	0.913	0.938	0.174	0.960	0.667	0.983	1.000	0.878
$ratio_{actual}$	0.155	0.080	0.105	0.300	0.010	0.062	0.016	0.150	0.140	0.113
$ratio_{impact}$	0.185	0.080	0.125	0.340	0.109	0.067	0.031	0.155	0.140	0.137

Table 9.7.: Accuracy and effort reduction of all sub-scenarios of the CoronaWarnApp and MobilityAsAService.

evaluation question and a total of three metrics. Table 9.6 shows the results. Similar to the previous sections, we split the presentation and interpretation of the results.

The *accuracy* (**Q5.1**) asks about the precision and completeness of the predicted impact set of the uncertainty impact analysis compared to manual confidentiality analysis. As described above, we use 🔧 UIA to calculate the uncertainty impact sets and measure *precision* (**M5.1.1**) and *recall* (**M5.1.2**), and calculate the F_1 score (**M5.1.3**). We repeat this for all four sub-scenarios of the *CoronaWarnApp*, introduced in Section 8.5, and all five sub-scenarios of the *MobilityAsAService* evaluation scenario, introduced in Section 8.6. Table 9.7 shows the individual results for all sub-scenarios. We see an overall high precision with an average of 0.783. In all nine sub-scenarios, every confidentiality violation was predicted by the impact analysis. Thus, the actual impact set is always a subset of the uncertainty impact set which results in the optimal recall of 1.0. We calculate a F_1 score of 0.878. In Sub-scenario *S1* of the *CoronaWarnApp*, the uncertainty focuses on one single component and is primarily propagated in the data flow diagram rather than the architectural model. Because the confidentiality violation happens near to the data sink, the overestimation results in an F_1 score of 0.912. In Sub-scenario *S2*, the uncertainty is located in the environment of the software system. The confidentiality violations are also located in the data flow diagram nodes that represent the environment, e.g., in data sinks that represent databases. Thus, this sub-scenario results in the optimal F_1 score of 1.0. Sub-scenario *S3* focuses on uncertainty in the behavior of different parts of the system. Here, the uncertain validation of exchanged keys leads to confidentiality violations when they are inserted into the database. This sub-scenario yields an F_1 score of 0.913. Sub-scenario *S4* contains only wide-spreading uncertainty sources in central components like the main server database component. Although this results in a precision of 0.882, still, all confidentiality violations are correctly predicted with a F_1 score of 0.938. Despite the lacking precision, the analysis could still correctly exclude uncertainty impacts in 7 of the 14 extracted data flows, e.g., there is no impact on the test result servers if the application server fails. In Sub-scenario *S1* of *MobilityAsAService*, the uncertainty affects the user role early in the data flow but the confidentiality violation happens near the sink. This results in the largest overestimation of all sub-scenarios and a precision of 0.095. Sub-scenario *S3* handles the leaking of user information. Due to the relatively short data flow, the precision is only 0.5, although the overestimation is comparably low. The other Sub-scenarios *S2*, *S4*, and *S5* show high precision with 0.923, 0.967, and 1.000.

Question	Metric	Result
Effort reduction (**Q6.1**)	$ratio_{actual}$ (**M6.1.1**)	0.11
	$ratio_{impact}$ (**M6.1.2**)	0.14

Table 9.8.: Results of evaluation Goal **G6** regarding the uncertainty impact analysis' effort reduction.

Regarding the *precision* (**M5.1.1**), the impact analysis provides promising results in the majority of sub-scenarios. By comparing the results of all sub-scenarios, we identify two sub-scenarios where the analysis provides larger overestimations. First, wide-spreading uncertainty sources in central elements of the software architecture naturally affect more data flows, e.g., in Sub-scenario *S4* of the *CoronaWarnApp*. Second, uncertainty sources that are located near the source of a data flow propagate until its sink, resulting in a larger impact set, e.g., in Sub-scenario *S1* of the *CoronaWarnApp* and Sub-scenario *S1* of *MobilityAsAService*. In both sub-scenarios, the precision is reduced, with values between 0.095 and 0.882. Regarding the *recall* (**M5.1.2**), we achieve the optimal result of 1.000 in all sub-scenarios. This is expected as the uncertainty impact set shall always be a superset of the actual impact set, see Section 6.3. Regarding the $F_1 score$ (**M5.1.3**) that represents the overall accuracy, we see satisfying results in the majority of sub-scenarios. As the recall stays at 1.0, the score is only lowered with suboptimal precision. Here, only Sub-scenario *S1* of *MobilityAsAService* results in a bad score of 0.174. We consider this a side effect of our optimization towards recall, see Section 9.1. Still, the analysis is able to accurately reject 7 out of 8 data flows of not being affected by uncertainty. With respect to the role of an uncertainty impact analysis as an early prediction in our procedure, see Section 4.1, we tolerate such overestimation. Here, we want to highlight the importance of optimizing for high recall, i.e., fewer false negatives rather than high precision, i.e., fewer false positives. Impact analyses, like change impact analysis, are used to make early predictions of the possible outcomes of external influences. Especially regarding security-related properties like confidentiality, missing violating elements could have severe results. Furthermore, overestimation is common among impact analyses [23, 112, 210]. We conclude that the results are satisfying regarding the *precision and completeness* (**Q5.1**) of the uncertainty impact analysis results. Thus, the uncertainty impact analysis is *accurate* (**G5**). Still, the impact analysis requires manual interpretation by software architects. We continue with the effort reduction evaluation.

9.3.3. Results and Discussion of the Effort Reduction

The sixth Goal **G6** focuses on the effort reduction of uncertainty impact analysis, i.e., whether using the analysis reduces the interpretation effort by software architects compared to manual analysis. This goal comprises one evaluation question with a total of two metrics. Table 9.8 shows the results. We present the results and interpret them afterward.

The *effort reduction* (**Q6.1**) asks about the manual interpretation effort by software architects of the impact analysis compared to manual analysis, i.e., how many DFD nodes have to be considered when investigating potential confidentiality violations. To compare the effort and to measure the reduction, we calculate the ratio of all nodes to the actual impact set $ratio_{actual}$ (**M6.1.1**) and the ratio of all nodes to the uncertainty impact set $ratio_{impact}$ (**M6.1.2**). Similar to Goal **G5**, we repeat this procedure for all nine sub-scenarios of the *CoronaWarnApp* and *MobilityAsAService* evaluation scenarios. Table 9.7 shows the results. As discussed previously, in all nine sub-scenarios, every confidentiality violation was predicted. Thus, the actual impact set is always a subset of the uncertainty impact set, which results in the optimal recall of 1.0. This also implies that the ratio of the actual impact set is a lower limit to the ratio of the uncertainty impact set, i.e., $ratio_{actual} \leq ratio_{impact} \leq 1$. A value lower than the $ratio_{actual}$ would imply that a confidentiality violation was missed, which results in a lowered recall. Trivially, if the precision reaches its optimum, both sets and also both ratios are equal. This happens in Sub-scenario *S2* of the *CoronaWarnApp* and Sub-scenario *S5* of *MobilityAsAService*. In the majority of other sub-scenarios, both ratios are relatively close, e.g., 0.155 compared to 0.185 in Sub-scenario *S1* of the *CoronaWarnApp*. The only outlier is Sub-scenario *S1* of the *MobilityAsAService* evaluation scenario. Due to the lack of precision, $ratio_{impact}$ is 0.109 instead of the optimal value of 0.010, i.e., approximately ten times more nodes have to be considered. Still, the effort is reduced compared to manual interpretation without using any analysis.

Regarding the ratio of the actual impact set $ratio_{actual}$ (**M6.1.1**), we first want to state its implications for confidentiality analysis. Although all evaluation scenarios intentionally contain multiple confidentiality violations, see Table 8.2, the amount of affected DFD nodes is relatively low. The ratio of the actual impact set ranges from 0.010 to 0.300, with an average of 0.113. This means that, on average, only 11% of all DFD nodes contain confidentiality violations. As stated previously, identifying such nodes manually without assistance is cumbersome [233, 234]. Regarding the ratio of the uncertainty impact set $ratio_{impact}$ (**M6.1.2**), the number of nodes that software architects have to consider is notably reduced by 86% to a $ratio_{impact}$ of *0.137*, which is near the optimal value of $ratio_{actual} = 0.113$. Furthermore, to reach the optimal value, software architects are required to apply uncertainty-aware confidentiality analysis, e.g., one of the approaches introduced in Chapter 7. This requires them to model uncertainty sources and scenarios as part of the software architecture, which requires more modeling effort than using an uncertainty impact analysis. With the uncertainty impact analysis, an element of the software architecture only has to be annotated with an uncertainty source without the need to define any what-if scenarios. The calculation of the impact set is fully automated. The required effort to adapt the model to reach the optimal value is arguably higher than inspecting the additional DFD nodes of the uncertainty impact set. See Chapter 4 for an explanation of the relation between impact analysis and confidentiality analysis. We conclude that results are satisfying regarding the *effort reduction* (**Q6.1**) of the uncertainty impact analysis compared to manual confidentiality analysis. Thus, the uncertainty impact analysis reduces the effort required by software architects (**G6**).

9.3.4. Threats to Validity

We conclude the evaluation of the second Contribution **C2** with a discussion of the threats to validity of Goal **G5** and **G6**. As this evaluation uses the evaluation scenarios introduced in Chapter 8, we follow the guidelines of Runeson and Höst [212]. They propose to discuss internal validity, external validity, construct validity, and reliability. This also addresses the lack of guideline-based validity discussions of software architectural research [142].

Internal validity The internal validity reflects whether the evaluation results depend only on the factors examined, without being influenced by other factors. In uncertainty impact analysis, one threat to internal validity is human error. For instance, selecting the wrong type of uncertainty source affects which propagation algorithm is chosen, ultimately affecting the result. However, we only use uncertainty types from our catalog introduced in Section 5.7 that were classified by multiple researchers. Furthermore, failing to accurately annotate uncertainty sources only marginally affects the propagation as the propagation in the DFD is type-agnostic, see Section 6.4[5]. Another threat to internal validity is accurate modeling. For instance, the position of confidentiality violations and uncertainty sources can influence the validity of the accuracy (**G5**) and effort reduction (**G6**) metrics. To mitigate these issues, we use evaluation scenarios with comprehensive documentation of the software architecture that was already independently modeled in other work [23, 36]. Furthermore, we ensure that the uncertainty sources represent a mix of different impact locations, e.g., at sources and sinks of data flows. Last, the size of the evaluation scenarios can negatively impact the expressiveness of the metrics. Here, we choose the largest available evaluation scenarios, i.e., the *CoronaWarnApp* with 21 components, resulting in a DFD with 506 nodes, and *MobilityAsAService* with 18 components, resulting in a DFD with 267 nodes. Both systems are non-trivial and appropriate for the evaluation.

External validity The external validity reflects the generalizability of the evaluation results to other cases or domains. Here, the biggest threat is the selection of evaluation scenarios for both evaluating accuracy (**G5**) and effort reduction (**G6**). To address this, we use evaluation scenarios of different size and from different domains, i.e., the health domain, and the mobility domain. Furthermore, we ensure to include all five uncertainty types introduced in Section 5.3. We argue that the quality of the uncertainty impact analysis only depends on the selection and annotation of uncertainty sources, see the discussion above, and does not depend on the software system, or its domain. The comparable results of both evaluation scenarios support this claim.

Construct validity The construct validity reflects whether the measurements are suitable for representing the evaluation objectives. Similarly to the evaluation of Contribution

[5] This concern has also been raised after presenting the uncertainty impact analysis at SEAMS '23 in Australia. Besides the type-agnostic uncertainty propagation in DFDs, the tooling UIA checks the validity of annotated uncertainty, which minimizes the impact of human error.

C1, we apply a GQM-based evaluation plan [16, 17] to minimize the risk of collecting unnecessary data. Furthermore, we use well-known metrics for binary classification [198, 208] to measure accuracy (**G5**). These metrics have also been used to evaluate the accuracy of related work and comparable analyses [233, 264]. Regarding the effort reduction (**G6**), our evaluation is closely oriented on the evaluation of change impact analysis [46, 210]. Here, we adapt counting model elements affected by a change in the software architecture and count DFD nodes that are affected by uncertainty. A similar adaptation has been proposed to evaluate the effort reduction of architecture-based attacker propagation [106, 264, 267]. However, these metrics do not represent the manual modeling effort of software architects. As discussed with the results of Goal **G6**, our analysis requires less modeling than architecture-based confidentiality analysis. Thus, reaching a comparable result in effort reduction compared to confidentiality analysis is sufficient.

Reliability The reliability reflects the repeatability and whether the evaluation results are dependent on the specific researchers. Similarly to the evaluation of Contribution **C1**, the evaluation is performed by a single researcher. Here, we ensure the validity by repeatedly having the results reviewed by other scientists [23, 200]. Furthermore, we address the repeatability by providing a data set [98] that contains all raw evaluation data, the evaluation scenarios, the analysis and interpretation of the results, and our tooling 🔧 UIA. As mentioned earlier, this includes the previous version of our data flow analysis framework used for the impact analysis, see Section 7.2.

9.4. Evaluation of the Uncertainty-Aware Data Flow Analysis

In this section, we present the evaluation of the third Contribution **C3**, introduced in Chapter 7. This contribution comprises four approaches to uncertainty-aware data flow analysis. The evaluation is based on the GQM plan presented in Section 9.1, and includes the Goals **G7** and **G8**. We use the evaluation scenarios introduced in Chapter 8. First, we present the evaluation design. Afterward, we show and discuss the results for each goal individually. We conclude with a discussion of threats to validity.

9.4.1. Evaluation Design

The evaluation of the third contribution **C3** consists of investigating the *scalability* (**G7**) and the *accuracy* (**G8**) of uncertainty-aware data flow analysis. Regarding the *scalability*, we use the evaluation scenarios of the *CoronaWarnApp*, presented in Section 8.5, and *MobilityAsAService*, presented in Section 8.6. These scenarios comprise the largest software systems, where scalability effects become more clearly visible. Using smaller software systems is possible but not expedient. Regarding the *accuracy*, we use all six evaluation scenarios. As already discussed, we consider this to be one of the most important properties of security-oriented analyses. We use different subsets of the scenarios for the first three analysis approaches presented in Chapter 7, but we combine all six scenarios to evaluate our most advanced approach, i.e., uncertainty *impact-aware* data flow analysis.

For the evaluation of Goal **G7**, we apply the complexity formulas to evaluation scenarios. These formulas describe how many TFGs an uncertainty-aware data flow analysis has to consider to identify all confidentiality violations due to uncertainty. In Section 7.6, we already discussed that the required number of TFGs relates to the type of analysis: *Scenario-aware* analysis requires equal or more TFGs than *graph-aware* analysis that requires equal or more TFGs than *impact-aware* analysis. The less TFGs an analysis requires, the better the scalability [231]. However, this theoretical discussions lacks empirical evidence, which includes a better understanding on the size of the actual differences in scalability. Put simply, designing more advanced analysis approaches regarding scalability to gain one percent better performance at design time is not expedient. To evaluate this, we introduce uncertainty sources at random locations within the evaluation scenarios and measure the required TFGs of the three analysis levels. By dividing the individual numbers of TFGs by the number of uncertainty sources, we calculate the *scalability* of *scenario-aware* analysis (**M7.1.1**), the *scalability* of *graph-aware* analysis (**M7.1.2**), and the *scalability* of *impact-aware* analysis (**M7.1.3**). We repeat this for different counts of uncertainty sources, from 1 to 25, in steps of 5. This enables us to draw conclusions on the growth of required TFGs. Additionally, we repeat the measurement for *primary* uncertainty, *secondary* uncertainty, and both, as the choice of the uncertainty types impacts the scalability, see Section 7.6. This evaluation is automated using 🔧 Abunai and part of the data set [98].

For the evaluation of Goal **G8**, we individually consider the four analysis approaches presented in Chapter 7, i.e., the *accuracy* of data flow analysis under structural uncertainty (**Q8.1**), data flow analysis under environmental uncertainty (**Q8.2**), tracing uncertainty in data flow analysis (**Q8.3**), and uncertainty impact-aware data flow analysis (**Q8.4**). For all questions, we use the metrics of *precision* (**M8.1.1**, **M8.2.1**, **M8.3.1**, **M8.4.1**) and recall (**M8.1.2**, **M8.2.2**, **M8.3.2**, **M8.4.2**). We calculate these metrics based on the provided confidentiality requirements and uncertainty sources, and the expected confidentiality violations of the evaluation scenarios, see Chapter 8 and Section 9.4. To evaluate the data flow analysis under structural uncertainty (**Q8.1**), we use the *TravelPlanner*, *DistanceTracker*, and *OnlineShop* evaluation scenarios. We model the uncertainty sources as design decisions in PerOpteryx [143, 144] and investigate whether the software architecture candidates contain confidentiality violations. To evaluate the data flow analysis under environmental uncertainty (**Q8.2**), we use the *TravelPlanner* and the *DistanceTracker* evaluation scenarios, which both use Role Based Access Control (RBAC) to ensure confidentiality. We introduce uncertainty into the access control properties and investigate the resulting confidentiality violations. To evaluate the tracing of uncertainty (**Q8.3**), we use the *TravelPlanner*, *DistanceTracker*, and *OnlineShop* evaluation scenarios. We express the uncertainty sources as variation points in the variation model and compare the analysis results to the expected results of the scenarios. Last, to evaluate the uncertainty impact-aware data flow analysis (**Q8.3**), we use all six evaluation scenarios, i.e., *TravelPlanner*, *DistanceTracker*, *OnlineShop*, *CoronaWarnApp*, *MobilityAsAService*, and *JPlag*. We can directly use the described uncertainty sources as input to the analysis, and investigate the scenario combinations that were identified by the analysis as states that violate confidentiality. Further details on the

Question	Metric	Result
Scalability (**Q7.1**)	Scenario-aware (**M7.1.1**)	2.56e+06
	Graph-aware (**M7.1.2**)	7.09
	Impact-aware (**M7.1.3**)	5.50

Table 9.9.: Results of evaluation Goal **G7** regarding the uncertainty-aware data flow analyses' scalability.

accuracy evaluation can be found in our corresponding publications [35, 101, 266]. All prototypical implementations and all evaluation data is part of the data set [98].

9.4.2. Results and Discussion of the Scalability

The seventh Goal **G7** targets the scalability of our approaches to uncertainty-aware confidentiality analysis. This goal comprises one evaluation question with a total of three metrics. Table 9.9 shows the results. Again, we split the presentation and interpretation.

The *scalability* (**Q7.1**) asks about the comparison of three levels of uncertainty-aware confidentiality analysis, as defined in Section 7.3. We randomly introduce uncertainty in the two largest evaluation scenarios, i.e., the *CoronaWarnApp* and *MobilityAsAService*. We calculate the complexity as introduced in Section 7.6 and divide the result by the number of introduced uncertainty sources. The result shows how many TFGs are required to analyze confidentiality under uncertainty using the different analysis approaches. As discussed previously, this does not effect the identified confidentiality violations but only the analysis complexity. First, we present the average number of additional TFGs per uncertainty source in both evaluation scenarios. Using uncertainty scenario-aware data flow analysis (**M7.1.1**), we measure on average $scalability_{scenario}$ = 2.56e+06, i.e., approximately 2.5 million TFGs for each modeled uncertainty source. Using graph-aware data flow analysis (**M7.1.2**), we measure on average $scalability_{graph}$ = 7.09, which is significantly lower than the complexity of the scenario-aware analysis. Last, using impact-aware data flow analysis (**M7.1.3**), we measure on average $scalability_{impact}$ = 5.50. To better understand the growth of required TFGs in the analysis, Figure 9.9 compares the number of injected uncertainty sources to the number of TFGs in the *CoronaWarnApp* evaluation scenario. Here, we differentiate between only considering *primary* uncertainty or *secondary* uncertainty, and considering both. Note that the graph uses a logarithmic scale to depict the number of TFGs. The same applies to Figure 9.10 that shows the scalability results of the *MobilityAsAService* evaluation scenario. In both evaluation scenarios, we see the rapid growth of the required TFGs for scenario-aware data flow analysis. The resulting numbers range from 16 TFGs for a single *primary* uncertainty in *MobilityAsAService* to approximately 470 million TFGs to consider 25 uncertainties in the *CoronaWarnApp*. In comparison to this, both the graph-aware and impact-aware analyses show slower growth. For 25 uncertainties, the graph-aware approach needs 469 TFGs and the impact-aware approach needs 237 TFGs in the *CoronaWarnApp*. *MobilityAsAService* shows similar results with 89 and 57 TFGs, respectively. Last, the graphs also indicate the lack of a speed up

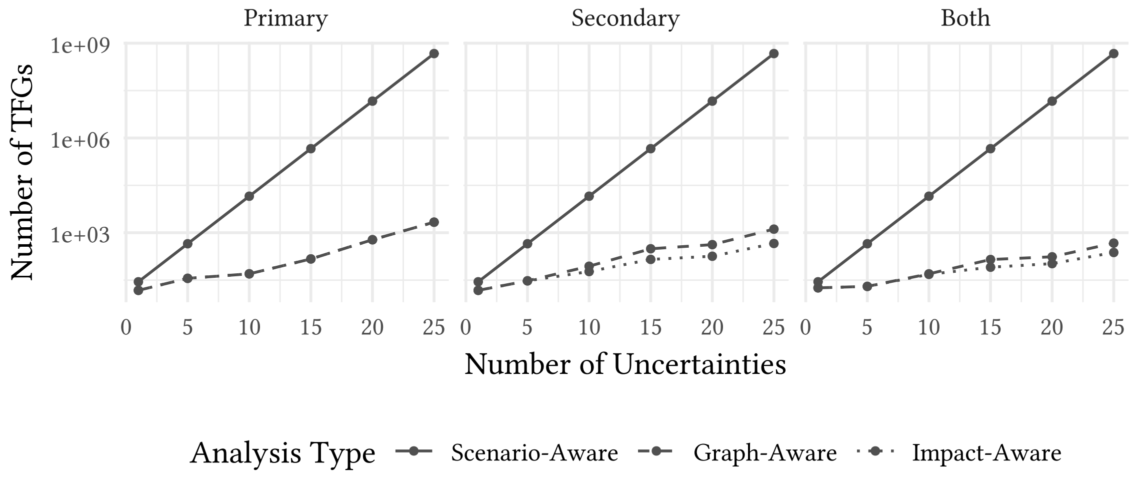


Figure 9.9.: Scalability results of the CoronaWarnApp evaluation scenario. Lower is better.

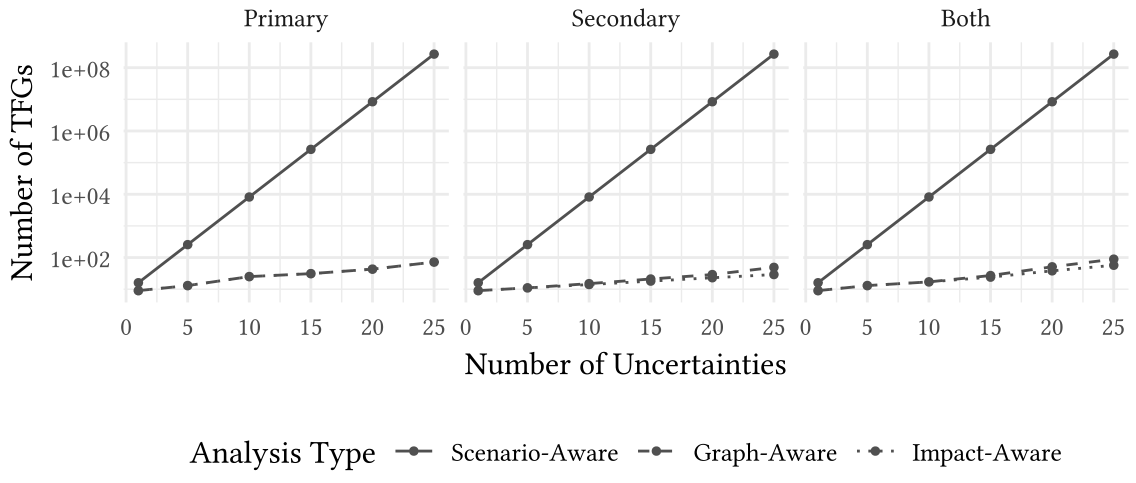


Figure 9.10.: Scalability results of the MobilityAsAService evaluation scenario. Lower is better.

regarding *primary* uncertainty of the impact-aware analysis compared to the graph-aware analysis. Here, differences only become visible when including *secondary* uncertainty, as discussed in Section 7.6. All measured complexity values are part of the data set [98].

Regarding the scalability of *scenario-aware* analysis (**M7.1.1**), the results show the combinatorial explosion already discussed in the context of design space exploration [143]. We also discussed this issue when introducing the different approaches to uncertainty-aware data flow analysis in Chapter 7. For every uncertainty source—and every scenario of every source—the scenario-aware analysis calculates the combination of all possible scenarios, i.e., the Cartesian product. Afterward, all TFGs need to be analyzed for every combination. Especially in larger software systems with many data flows, this rapidly increases the run time. However, also smaller systems are affected: The *MobilityAsAService* scenario

Question	Metric	Result
Structural uncertainty (**Q8.1**)	Precision (**M8.1.1**)	1.00
	Recall (**M8.1.2**)	1.00
Environmental uncertainty (**Q8.2**)	Precision (**M8.2.1**)	1.00
	Recall (**M8.2.2**)	1.00
Tracing uncertainty (**Q8.3**)	Precision (**M8.3.1**)	1.00
	Recall (**M8.3.2**)	1.00
Impact-aware analysis (**Q8.4**)	Precision (**M8.4.1**)	1.00
	Recall (**M8.4.2**)	1.00

Table 9.10.: Results of evaluation Goal **G8** regarding the uncertainty-aware data flow analyses' accuracy.

only has 8 TFGs compared to the 14 TFGs of the *CoronaWarnApp*. Still, for all 25 uncertainty sources, the analysis requires approximately 270 million TFGs compared to 470 million TFGs. Both numbers are high enough to make the data flow analysis framework unsuitable [231]. Regarding the scalability of *graph-aware* analysis (**M7.1.2**), the number of required TFGs is highly reduced. Using the information about depended uncertainties in the same TFG and independent uncertainties in other TFGs, the majority of scenario combinations can be safely rejected. This effect becomes visible in all non-trivial software systems that comprise more than a single TFG. Here, our measurements imply that the 8 TFG of *MobilityAsAService* are sufficient for a highly reduced complexity. Regarding the scalability of *impact-aware* analysis (**M7.1.3**), we see its relation to the occurrence of *primary* and *secondary* uncertainty. The impact-aware analysis is able to consider the uncertainty interaction between *secondary* and *primary* uncertainty, as introduced in Section 7.3. Thus, the number of TFGs is not reduced when only including *primary* uncertainty. Although the reduction when including *secondary* uncertainty is not as large compared to scenario-aware analysis, it is still notable. On average, impact-aware analysis requires only $\frac{5.50}{7.09} = 0.78$ times as many TFGs for the same result in the evaluated scenarios. Here, the results also indicate that the reduction correlates to the number of TFGs of the scenario. We conclude that the results are satisfying regarding the *scalability* comparison (**Q7.1**) of uncertainty-aware data flow analyses. Thus, we are able to define *scalable* analyses (**G7**). The results also empirically support the complexity reduction of more advanced uncertainty-aware data flow analysis, i.e., $N_I \leq N_G \leq N_S$, see Section 7.6.

9.4.3. Results and Discussion of the Accuracy

The eighth and last Goal **G8** focuses on the accuracy of the four approaches to uncertainty-aware confidentiality analysis, presented in Chapter 7. This goal comprises four evaluation questions with a total of eight metrics. Table 9.10 shows the results. In the following, we first present these results and discuss them afterward.

The *accuracy* of the data flow analysis under structural uncertainty (**Q8.1**) asks about the *precision* (**M8.1.1**) and *recall* (**M8.1.2**) of the first analysis approach, introduced in

Subsection 7.4.1. As discussed previously, we use the evaluation scenarios *TravelPlanner*, *DistanceTracker*, and *OnlineShop* together with PerOpteryx [143, 144]. Although the introducing of the confidentiality analysis requires PerOpteryx to analyze many architectural candidates, see Subsection 7.4.1, all candidates with confidentiality violations and also all candidates without violations were correctly identified. This results in a perfect score of 1.0, both regarding precision and recall. The *accuracy* of the data flow analysis under environmental uncertainty (**Q8.2**) asks about the *precision* (**M8.2.1**) and *recall* (**M8.2.2**) of the second analysis approach, introduced in Subsection 7.4.2. Here, we use the evaluation scenarios *TravelPlanner* and *DistanceTracker*. The introduction of uncertainty in RBAC produces the expected confidentiality violations, i.e., the perfect scores of 1.0 for precision and recall. The *accuracy* of the approach to trace uncertainty in data flows (**Q8.3**) asks about the *precision* (**M8.3.1**) and *recall* (**M8.3.2**) of the third analysis approach, introduced in Subsection 7.5.1. The introduction of uncertainty as variation points in the variation creation [168] of the evaluation scenarios *TravelPlanner*, *DistanceTracker*, and *OnlineShop*, produces variants of the software architecture, which partially violate confidentiality. The analysis is able to correctly identify these variations and traces identified confidentiality violations back to the originating uncertainty sources. Thus, it reaches both a precision and recall of 1.0. Last, the *accuracy* of the uncertainty-impact aware data flow analysis (**Q8.4**) asks about the *precision* (**M8.4.1**) and *recall* (**M8.4.2**) of the fourth analysis approach, introduced in Subsection 7.5.2. Here, we directly model the uncertainty sources as a first-class concern of all six evaluation scenarios, i.e., *TravelPlanner*, *DistanceTracker*, *OnlineShop*, *CoronaWarnApp*, *MobilityAsAService*, *JPlag*. As discussed previously, we provide the most comprehensive accuracy evaluation for this approach as it represents our most advanced analysis. In total, we identified 221 confidentiality violations due to uncertainty, which resulted in a precision of 1.0 and a recall of 1.0.

All four evaluation questions of Goal **G8** use the same metrics, i.e., *precision* and *recall*. Moreover, all measurements show the perfect result of 1.0, both regarding precision and recall. For the sake of simplicity, we jointly discuss all results instead of repeating the interpretation for all four questions (**Q8.1** – **Q8.4**). All presented uncertainty-aware data flow analysis approaches are based on the data flow analysis framework [36], presented in Section 7.2. The framework itself follows the data flow analysis concept of Seifermann [233] and shows equal results regarding identified confidentiality violations [231], i.e., a precision and recall of 1.0. The accuracy evaluation of Seifermann [233], which partially uses the same evaluation scenarios as used in our evaluation, results in a perfect accuracy of 1.0. Put simply, to validate the accuracy of our analysis approaches, it is sufficient to show that we did not impair the analysis quality by introducing uncertainty into the modeling and analysis procedure. The consistent results of 1.0 for both precision and recall support this. Note that this does not resemble real-world accuracy measurements using case studies, as discussed in Chapter 8. Related work uses similar approaches to accuracy evaluation showing perfect results [38, 234, 236, 264, 265, 267]. We conclude that the results are satisfying regarding the *accuracy* of data flow analyses under structural uncertainty (**Q8.1**) and environmental uncertainty (**Q8.2**), and also regarding tracing uncertainty (**Q8.3**) and the uncertainty-impact aware analysis (**Q8.4**). Thus, our approaches to uncertainty-aware data flow analysis are *accurate* (**G8**).

9.4.4. Threats to Validity

We conclude the evaluation of the third Contribution **C3** with a discussion of the threats to validity of Goal **G7** and **G8**. Similarly to the previous evaluation, we use evaluation scenarios, described in Chapter 8. We follow the guidelines of Runeson and Höst [212], who propose to discuss internal validity, external validity, construct validity, and reliability. This addresses the lack of guidelines in threats to validity discussions [142].

Internal validity The internal validity reflects whether the evaluation results depend only on the factors examined, without being influenced by other factors. Regarding Goal **G7**, the location of the injected uncertainty sources could influence the measured scalability results. However, especially for the larger measurements, we argue that using randomly chosen locations is sufficient. Additionally, the identified effect size is large, with speedups by orders of magnitude. Although we cannot eliminate this risk, we find its impact negligible. Regarding Goal **G8**, the biggest threat is the manual creation of the reference output, i.e., the gold standard of the evaluation scenarios. We apply several strategies to mitigate this threat. First, we use a mixture of evaluation scenarios from other work [133, 135, 203, 233, 234, 236], e.g., *TravelPlanner* and *DistanceTracker*, and self-defined evaluation scenarios based on real-world software systems, e.g., *CoronaWarnApp* [119], and *Jplag* [199, 214]. Second, we use publicly available documentation to ensure the validity of confidentiality violations and sources of uncertainty. Third, we continuously use the data flow analysis framework without considering uncertainty to validate identified confidentiality violations. Fourth, the simpler scenarios are small enough to manually validate identified results. Another threat to internal validity is the extension of PerOpteryx [144] using the confidentiality analysis [266], which could impact the design space exploration algorithm. We address this by only using the dedicated extension mechanism of PerOpteryx [143].

External validity The external validity reflects the generalizability of the evaluation results to other cases or domains. Regarding Goal **G7**, we only consider the biggest evaluation scenarios *CoronaWarnApp* and *MobilityAsAService*. We argue that this is sufficient, especially in large scenarios where combinatorial explosion can easily lead to scalability problems. Thus, the sufficient scalability of graph-aware and impact-aware analysis should also apply to simpler scenarios. Similarly to the evaluation of Contribution **C2**, the biggest threat for Goal **G8** is the selection of evaluation scenarios. We address this by ensuring a good mixture of minimal examples, e.g., *TravelPlanner* or *DistanceTracker*, and comprehensive scenarios, e.g., *CoronaWarnApp* and *MobilityAsAService*. All scenarios show different numbers regarding the size of the architectural model, the extracted DFDs, modeled uncertainty sources and identified confidentiality violations. Furthermore, they originate from different existing work and existing software systems from a multitude of domains, i.e., mobility, health, sports, e-commerce, and plagiarism detection. They comprise all five uncertainty types, introduced in Section 5.3. Similarly to the previous evaluation, we argue that the quality of uncertainty-aware data flow analysis does not depend on the domain and only relates to DFDs and considered uncertainty sources.

Construct validity The construct validity reflects whether the measurements are suitable for representing the evaluation objectives. Similarly to the previously presented evaluation of Contributions **C1** and **C2**, we apply a GQM-based evaluation plan [16, 17]. Regarding Goal **G7**, the metrics for scalability depend on the complexity calculation, introduced in Section 7.6. We argue that using these formulas is a valid approach as they directly represent the analysis algorithms presented in Chapter 7. However, we do not provide formal proof but empirically evaluate the complexity using the evaluation scenarios. We also do not collect runtime information as a previous study already indicated a high correlation between the number of TFGs and the analysis execution time. We argue that measuring the required number of TFGs to consider uncertainty using the evaluation scenarios is sufficient. Nevertheless, the applied metrics are not suitable for external comparisons as they lack normalization. We only focus on the comparison of the scalability results of our analysis approaches. Regarding Goal **G8**, we use precision and recall to measure accuracy. Both represent well-known metrics of binary classification [198, 208] that are commonly used to evaluate accuracy in software architectural research [142].

Reliability The reliability reflects the repeatability and whether the evaluation results are dependent on the specific researchers. In contrast to the previous parts of the evaluation, the reliability of the results of Goal **G7** and **G8** is not threatened by the fact that the evaluation is only performed by a single researcher. Here, all measurements were fully automated using the provided tooling, i.e., 🔧 UIA and 🔧 Abunai. Nevertheless, to increase repeatability, we provide all modeled uncertainties, all raw evaluation data, the prototypical implementation, and all calculations as part of our data set [98].

9.5. Summary and Outlook

In this chapter, we presented the evaluation of all three Contributions **C1**, **C2**, and **C3**. First, we introduced a comprehensive GQM-plan comprising 8 goals, 19 questions, and 32 metrics. Afterward, we explained the evaluation design, and presented and interpreted the evaluation results separately for each contribution. Table 9.11 shows an overview of all metrics, thereby combining the individually presented results of all Goals **G1** – **G8**. In the following, we summarize the results and also present key findings of the evaluation.

Our first Contribution **C1** comprises a classification of uncertainty regarding confidentiality and a catalog of uncertainty sources. We evaluated the classification structure's suitability (**G1**), its applicability (**G2**), and justified its purpose (**G3**), following the evaluation method of Kaplan et al. [132]. Additionally, we evaluated the usability (**G4**) of our catalog. With the classification, we enable a more precise description of uncertainty with regard to confidentiality. With the tool-supported catalog approach 🔧 ARC^{3}N, we intended to enhance the usability of the classification and also to address the UAP. The results of the evaluation indicate that both the classification and the catalog reach these goals. With the classification, we provide common terminology that fits the required abstraction, e.g., with a high laconicity (**M1.1.1**) of 95%, or a clearly positive classification delta (**M3.3.1**) of 0.54,

Goal	Question	Metric	Result
G1	Generality (**Q1.1**)	Laconicity (**M1.1.1**)	0.95
		Lucidity (**M1.1.2**)	0.70
	Appropriateness (**Q1.2**)	Completeness (**M1.2.1**)	0.97
		Soundness (**M1.2.2**)	1.00
	Orthogonality (**Q1.3**)	Orthogonality (**M1.3.1**)	0.99
G2	Reliability (**Q2.1**)	Annotator consensus (**M2.1.1**)	0.69
	Correctness (**Q2.2**)	Recall (**M2.2.1**)	0.73
	Ease of use (**Q2.3**)	SUS (**M2.3.1**)	68.25
G3	Relevance (**Q3.1**)	Fraction of relevant classes (**M3.1.1**)	1.00
	Novelty (**Q3.2**)	Innovation (**M3.2.1**)	0.49
		Adaptation (**M3.2.2**)	0.51
	Significance (**Q3.3**)	Classification delta (**M3.3.1**)	0.54
G4	Identification (**Q4.1**)	Correctness (**M4.1.1**)	0.88
	Collaboration (**Q4.2**)	Correctness (**M4.2.1**)	0.65
	Ease of use (**Q4.3**)	SUS (**M4.3.1**)	69.71
		Average rating (**M4.3.2**)	3.1/4
G5	Accuracy (**Q5.1**)	Precision (**M5.1.1**)	0.78
		Recall (**M5.1.2**)	1.000
		F_1 score (**M5.1.3**)	0.88
G6	Effort reduction (**Q6.1**)	$ratio_{actual}$ (**M6.1.1**)	0.11
		$ratio_{impact}$ (**M6.1.2**)	0.14
G7	Scalability (**Q7.1**)	Scenario-aware (**M7.1.1**)	2.56e+06
		Graph-aware (**M7.1.2**)	7.09
		Impact-aware (**M7.1.3**)	5.50
G8	Structural (**Q8.1**)	Precision (**M8.1.1**)	1.00
		Recall (**M8.1.2**)	1.00
	Environmental (**Q8.2**)	Precision (**M8.2.1**)	1.00
		Recall (**M8.2.2**)	1.00
	Tracing (**Q8.3**)	Precision (**M8.3.1**)	1.00
		Recall (**M8.3.2**)	1.00
	Impact-aware (**Q8.4**)	Precision (**M8.4.1**)	1.00
		Recall (**M8.4.2**)	1.00

Table 9.11.: Joint results of all evaluation goals, showing all questions, metrics, and results. This includes the evaluation Goals of structure's suitability (**G1**), applicability (**G2**), purpose (**G3**), usability (**G4**), accuracy (**G5**), effort reduction (**G6**), scalability (**G7**), and accuracy (**G8**).

and that is usable to name and describe uncertainty sources with regard to confidentiality, as seen with a recall (**M2.2.1**) of 73%. The catalog further supports the identification and description of uncertainty in architectural models with a correctness (**M4.1.1**) of 88%. Both are usable, as seen with a SUS (**M2.3.1**) of 68.25/100, or an average rating (**M4.3.2**) of 3.1/4. Based on the qualitative feedback of the participants of the two conducted user studies, combining both is expedient. We find:

ℹ **Finding:** Both the uncertainty classification and the catalog of uncertainty sources support software architects in identifying, describing, and understanding software-architectural uncertainty with regard to confidentiality. However, combining both represents the most promising approach.

Our second Contribution **C2** is an architecture-based uncertainty impact analysis to predict confidentiality violations. We evaluated the accuracy (**G5**) and effort reduction (**G6**) of the impact analysis compared to manual confidentiality analysis. With the impact analysis tool-supported with 🔧 UIA, we aim to accurately predict confidentiality violations with only a little overestimation but without any underestimation. Furthermore, we want to support software architects by directing their attention to problematic parts of the software system and thereby reducing the required effort. The evaluation results indicate that we have reached both goals. The analysis reaches the perfect recall (**M5.1.2**) of 100% while maintaining a precision (**M5.1.1**) of 78%. The required manual effort is highly reduced (**M6.1.2**), as software architects only need to consider 14% of DFD nodes on average, which is close to the optimal actual impact set ratio (**M6.1.1**) of 11%. In sum, we find:

ℹ **Finding:** The uncertainty impact analysis accurately predicts confidentiality violations due to uncertainty with only little overestimation. Although the evaluation scenarios intentionally contained a multitude of confidentiality violations, only every tenth DFD node was affected. This underlines the challenge of identifying confidentiality violations and the need for tool-supported analyses.

Our third Contribution **C3** comprises four approaches to uncertainty-aware data flow analysis to identify confidentiality violations due to uncertainty. We evaluated the scalability (**G7**) of the three central awareness levels of uncertainty in data flow analysis, and the accuracy (**G8**) of our four analysis approaches. Our goal is to identify all confidentiality violations due to uncertainty while maintaining reasonable scalability. The evaluation results indicate that this is not trivial. Our first approach, scenario-aware data flow analysis (**M7.1.1**), requires on average 2.5 million additional TFGs per uncertainty in our evaluation with 1 to 25 randomly injected uncertainty sources. This severe lack of scalability is discussed as the combinatorial explosion in the context of design space exploration [143]. However, both the graph-aware (**M7.1.2**) and the impact-aware (**M7.1.3**) data flow analysis requires significantly less TFGs per uncertainty source, averaging at 7.09, and 5.50, respectively. All four analysis approaches reach the perfect precision and recall, as expected due to the testing conditions. In sum, we find:

ℹ **Finding:** The four uncertainty-aware data flow analysis approaches accurately identify confidentiality violations due to uncertainty. However, only graph-aware and impact-aware analyses are applicable due to the severe lack of scalability of uncertainty scenario-aware data flow analysis. This underlines the need to provide scalable analysis approaches tailored to confidentiality.

Overall, we find the evaluation results satisfying. Some measurements represent the perfect, or nearly perfect results. Examples are the accuracy evaluation of uncertainty-aware data flow analysis (**G8**), the recall of uncertainty impact analysis (**G5**), or the structure's suitability (**G1**) and purpose (**G3**) of the classification. Other metrics indicate room for improvement despite being acceptable. For instance, the precision of uncertainty impact analysis **G5**, or the usability of the first Contribution **C1** could be enhanced. However, all contributions represent novel approaches to identifying, describing, and analyzing uncertainty with regard to confidentiality. Thus, we find near-optimal and partially perfect results to be sufficient and leave further optimization to future work. We find:

Finding: The evaluation provides evidence regarding the sufficient quality of this dissertation's contributions. This indicates that the classification of uncertainty, the catalog of uncertainty sources, the uncertainty impact analysis, and the uncertainty-aware data flow analyses support software architects in the architecture-based inspection and assessment of confidentiality under uncertainty.

This chapter comprises the evaluation of all three contributions, which were presented in the last three chapters. The first contribution is introduced in **Chapter 5: Identification and Classification of Uncertainty Regarding Confidentiality**. The second contribution is shown in **Chapter 6: Uncertainty Propagation to Enable Uncertainty Impact Analysis**. The third contribution is presented in **Chapter 7: Uncertainty-Aware Data Flow Analysis to Identify Confidentiality Violations**. An overview of the interplay of all contributions can be found in **Chapter 4: Overview**.

9.6. In Simpler Words

This thesis has three contributions. First, a classification and catalog approach of uncertainty sources, which help software architects identify and understand uncertainty sources. Second, an uncertainty impact analysis predicts potential confidentiality violations under uncertainty. Third, uncertainty-aware data flow analyses that identify confidentiality violations due to uncertainty. All contributions are meant to enlarge our current knowledge about the relationship between uncertainty and confidentiality and also to support software architects. In this chapter, we conduct a comprehensive evaluation to investigate whether the contributions represent an enhancement compared to the state of the art.

In an evaluation of this size, many things can go wrong. One common mistake is the collection of meaningless data to validate the contributions. To counteract this, we use a so-called Goal Question Metric (GQM)-plan. Here, goals represent the qualities, we want to show, e.g., that our analysis is accurate and usable. To investigate these goals, we answer evaluation questions, e.g., whether users were happy with our approach. Metrics quantify the answers, e.g., by counting how many users correctly describe uncertainty sources using our classification. In total, our evaluation has 8 goals, 19 questions, and 32 metrics. To gather the required data, we conducted two user studies, in which real software architects, researchers, and students used our contributions. Additionally, we use the evaluation scenarios, presented in the previous chapter.

It is very important that an evaluation is comprehensible for other researchers. Thus, this chapter describes in detail how the evaluation is designed, how the studies were conducted, and how the results were calculated. Here, we distinguish between the objective measurements and our subjective interpretation of the measurements. We also discuss potential threats to validity, i.e., what could have gone wrong during the evaluation and to which extent this affects the trustworthiness of our results. Last, we provide a data set [98] that contains all raw data, and all implementations and calculations. This enables other researchers to review our results or repeat the evaluation themselves.

Overall, the results of our evaluation are satisfying. They are not perfect—but they are good enough for us to believe that we created something that actually surpasses the state of the art. The evaluation provides evidence that our classification helps to better understand and describe uncertainty and that our catalog approach complements this very nicely. We also find that the uncertainty impact analysis does not miss confidentiality violations when predicting an uncertainty source's impact. However, it could be a little more precise, as it also yields incorrect confidentiality violations. Still, such overestimation is common among such analyses. Last, we see an excellent accuracy of our uncertainty-aware data flow analyses. All analysis approaches correctly identify all confidentiality violations due to uncertainty. We also see how important the scalability of an analysis is. Only our more advanced analysis approaches scale well enough for a higher degree of uncertainty within a software system. This so-called combinatorial explosion is a common problem of design space exploration analyses. Our evaluation indicates that our proposed solutions work very well regarding confidentiality. In sum, we are happy with these results—they took us a total of four years to collect.

Part IV.

Epilog

10. Related Work

In this chapter, we present other work that is related to our contributions[1]. As the title of this thesis implies, our contributions revolve around three central topics: Uncertainty, software architecture, and confidentiality. These topics represent the foundations of our work, presented in Chapter 2. Related work exists in all intersections of these topics. Thus, we structure related work by iteratively investigating work in these areas.

The remainder of this chapter is structured as follows: We summarize work from the software engineering and software architecture research community that also addresses uncertainty. Then, we focus on architecture-based approaches to security and confidentiality. Last, we investigate uncertainty-aware approaches to confidentiality that do not require architectural abstraction. This dissertation is at the center of these areas, combining aspects of confidentiality, software architecture, and uncertainty-related research.

Literature: This chapter is based on the following (co-) authored publications: [EMLS 2021], [ECSA-C 2021], [IEEE SEAA 2022], [Springer ECSA 2022], [Springer ICETE 2023], [IEEE ICSA-C 2023], [IEEE/ACM SEAMS 2023], [IEEE/ACM SEAMS 2024], [Springer ECSA 2024], [ACM/IEEE MODELS-C 2024]

10.1. Uncertainty and Software Architecture

The intersection of research on uncertainty and software architecture represents the largest field investigated in this chapter. This is no surprise, as also the vast majority of our publications were presented at conferences and workshops that belong to these research communities. Furthermore, uncertainty is often discussed using model-based approaches common in software architectural research [243, 255]. We start by giving an overview of surveys and research roadmaps of both communities. Afterward, we briefly summarize publications on uncertainty taxonomies and approaches related to Self-Adaptive Systems (SASs). Then, we summarize the comprehensive work that has been conducted to optimize software architectures or to use architectural knowledge in uncertainty-aware analysis. Another relation of both areas are design decisions, as already discussed with the *cone of uncertainty* [167], see Section 2.1. Last, we summarize work that focusses on uncertainty management and sharing architectural knowledge.

[1] We are well aware of the advances of generative artificial intelligence in summarizing research articles. Therefore, we assume that handwritten sections on related work—such as this one—could soon become obsolete. Thus, we focus on providing an understandable overview.

10.1.1. Surveys and Research Roadmaps

Recently, two large surveys were conducted within the Self-Adaptive System (SAS) community that are related to our work. Troya et al. [255] conducted a Systematic Literature Review (SLR) and the modeling of uncertainty. They investigate 123 primary studies to better understand the notions and formalisms used to represent uncertainty. Furthermore, they discuss when uncertainty is analyzed and whether such analysis is tool-supported. They present a comprehensive collection of formalisms used to represent uncertainty, e.g., variability models, fuzzy logic, or temporal logic. Their results indicate a lack of analysis approaches that are able to handle heterogeneous uncertainty types, such as presented in Section 6.7. They also highlight the need for consolidated modeling solutions, as currently driven with the Object Management Group (OMG) Precise Semantics for Uncertainty Modeling (PSUM) [184] standard. Furthermore, they mention the need of tool-supported processes, which we address with 🔧 Abunai, 🔧 UIA, and 🔧 ARC^{3}N. Regarding design time analyses like the one presented in this thesis, the most common approach are variability models, such as the uncertainty meta model presented in Subsection 7.5.2. We conclude that our contributions use techniques well-known from the state of the art, but surpass related work by providing comprehensive modeling support for heterogeneous uncertainty and tool-supported analyses.

Hezavehi et al. [115] present an overview of the research community based on questionnaires. In two stages, they investigate uncertainty concepts, sources, methods, and challenges. Their results motivate the design time analysis of uncertainty, which includes both uncertainty sources internal and external to the system under study. They state that "only when enough knowledge is not available at design time, [. . .], uncertainty handling should be postponed to run-time" [115]. Moreover, they find that the current state of the art lacks in including non-functional requirements both as optimization goal as well as side effects. We address this shortcoming and the problem of dealing with concurrent sources of uncertainty in Chapter 6 By providing a procedure to handle uncertainty in Section 4.1 and collecting uncertainty sources in Section 5.7, we also partially address the challenges of consolidating knowledge, providing guidelines, and dealing with unanticipated change. Last, the authors present an initial reference process that shows many activities that are also contained in the procedure presented in Section 4.1, e.g., uncertainty identification, modeling, and propagation. Calinescu et al. [47] provide the findings of another survey in the SASs community, focussed on understanding uncertainty. They also highlight the challenges of dealing with unanticipated change and enhancing the explainability of SASs. With 🔧 ARC^{3}N, we provide a step towards addressing these challenges in Section 5.6. We conclude that our approach fits current research challenges.

Research roadmaps light current research directions and provide agendas and research challenges. Here, De Lemos et al. [63] describe a general roadmap regarding the research on SASs. They especially highlight the challenge of the design space's size. We discussed this issue and how to address the combinatorial explosion regarding data flow analysis and confidentiality in Section 7.6. To also at least partially tackle the identified challenge regarding processes, we describe a procedure with activities and roles in Section 4.1.

More recently, Weyns et al. [273] describe their findings on current challenges regarding uncertainty management. They highlight the need for understanding uncertainty and providing end-to-end approaches. We present a tailored classification in Section 5.3, and a procedure that starts at the identification of uncertainty in Chapter 5 and ends after analyzing issues due to uncertainty in Chapter 7. Thereby, we address these challenges regarding confidentiality as a non-functional requirement.

10.1.2. Uncertainty Taxonomies and Classifications

To better understand uncertainty, researchers created several taxonomies [41, 48, 71, 162, 184, 195, 202, 263]. Walker et al. [263] present a taxonomy of uncertainty using three dimensions. The *location* describes where the uncertainty can be found, e.g., in the model input or context. The *nature* distinguishes between epistemic (i.e., lack of knowledge) and aleatory (i.e., natural variability) uncertainty. Last, the *level* describes how much is known about the uncertain influence. Although this taxonomy has been the baseline for many others, it does not specifically aim to describe software-related uncertainty. Perez-Palacin and Mirandola [195] build on this classification in the context of SASs. They thereby adjust the dimension *location* to better fit software models. Bures et al. [41] adapt this taxonomy again to “fit the needs of uncertainty in access control” [41]. Although this work only considers access control in Industry 4.0 scenarios, it is also a good foundation for our classification, presented in Section 5.3. Esfahani and Malek [71] describe characteristics of uncertainty and hereby focus on the variability and reducibility of different sources of uncertainty. They highlight the problem of uncertainty in the environment of a software system, e.g., due to the deployment. Mahdavi-Hezavehi et al. [162] propose a classification framework of uncertainty. They aim at architecture-based, SASs but do also not consider security, privacy, or confidentiality. Also related is the uncertainty template by Ramirez et al. [202]. They present a scheme to describe uncertainty sources for dynamically adaptive systems in requirements, design, and runtime. Due to the different scope, they describe uncertainty in software architecture as *inadequate design* which is not precise enough to identify the impact of software-architectural [104] uncertainty on confidentiality. Still, they also list *unexplored alternatives* and *misinformed trade-off analysis* which motivates our work. Armour [11] presents the order of ignorance which also influenced many of the aforementioned taxonomies, e.g., the work of Perez-Palacin and Mirandola [195].

More recently, the OMG PSUM [184] standard represents a joint effort towards the standardization of terminology to describe uncertainty. However, at the time of writing, this still represents a work in progress. A more in-depth discussion of related classifications can be found in Subsection 5.2.3. There, we not only show in detail all existing dimensions but also where we adapt or innovate on the state of the art. This question is also part of our third evaluation goal **G3**, presented in Section 9.2. We conclude that there has been a substantial effort in the last decades to classify uncertainty. Based on the comprehensive findings of related work we present a novel classification tailored to confidentiality. Revisiting related work underlines both the need for tailored classification systems and the appropriateness of our classification to confidentiality, see Subsection 5.2.3 and Section 9.2.

10.1.3. Architecting Self-Adaptive Systems

As discussed in Section 2.1 and in Section 5.2, uncertainty is central topic in the SASs research community. However, the software architectures considered in this thesis are not self-adaptive, and further research is required to understand the implications of our findings for such software systems completely. Nevertheless, we briefly summarize related work in this field.

The *Rainbow* framework by Garlan et al. [82] enables architecture-based self-adaptation. The architecture layer comprises an adaptation engine and logic for constraint evaluation to monitor the runtime system and to adapt in case of occurring issues, e.g., regarding the system performance. As this evaluation is model-based, we argue that our approach could be integrated into this framework to simplify its use at runtime. Proper modeling and analysis support—such as the one presented in this thesis—is required to identify confidentiality violations [226, 233]. In another publication, the authors present the *Znn.com* system to evaluate the Rainbow framework. We build on this software system to present our approach to addressing the UIP in Section 6.7.

Andersson et al. [8] present modeling dimensions of SASs, i.e., variation points of said systems. They distinguish four groups of dimensions: Goals, changes, mechanisms, and effects. In the scope of this work and the already discussed uncertainty classifications, this can be seen as a more general description of the underlying challenges of SASs. Using our terminology, uncertainties causing changes have sources and types, as described in Section 5.2. Here, our classification describes a small subset of these dimensions in more detail, justified by the purpose of conducting confidentiality analysis. This underlines the relation of our work to SASs, as already illustrated in recent publications [49, 273].

Whittle et al. [275] present *RELAX*, an approach targeted at considering uncertainty in requirements engineering of SASs. By weakening requirements depending on the environmental conditions monitored during runtime, self-adaptation can be achieved in a structured manner. The authors argue that, compared to ad-hoc approaches, considering uncertainty in requirements helps to describe clear adaptation boundaries and to ensure invariants. However, this approach does not consider the development of the system nor provides mechanisms to satisfy the weakened requirements. In addition, it can only cope with environmental uncertainty that is known during formulating the requirements. Nevertheless, we argue that revisiting confidentiality requirements under uncertainty represents a promising research direction for future work.

10.1.4. Architecture Evaluation Under Uncertainty

The topic of architecture evaluation under uncertainty represents the most comprehensive section of related work. Numerous approaches [4, 70, 72, 144, 159, 261] have been proposed to evaluate and optimize software architectures. They often refer to the variability or inherent uncertainty in software architecture as *design space exploration*, which can also be described as "searching better or even optimal designs" [143]. Sobhy et al. [243] conducted a SLR comprising 48 primary studies on this topic. They highlight the focus on design time, which matches our approach. We summarize the most related approaches.

PerOpteryx [143, 144, 163] is a model-based approach to optimize software architectures regarding quality properties like performance, cost, or reliability. Koziolek et al. [144] employ evolutionary optimization to find Pareto-optimal architectural candidates. To assess the quality of the candidates, they use the Palladio approach, introduced in Section 2.4. However, there is no support for confidentiality, although there is already a PerOpteryx extension for security [45]. In contrast to our approach, they modeled these by a concept of concerns. These concerns describe which design decisions are dependent on each other. In our analysis we focus on the direct impact on confidentiality. Similarly to the relation to Rainbow [82], we argue that our approach can be incorporated or combined with PerOpteryx. We demonstrated this while defining a scenario-aware data flow analysis regarding structural uncertainty in Section 7.4.

Esfahani et al. [70] present *GuideArch*, an approach to explore the architectural solution space under uncertainty. This shall enable software architects to identify critical design decisions. They apply fuzzy math to represent uncertainty and its impact on the software architecture. This enables the comparison of architectural candidates and the exploration of the solution space. Although this approach considers uncertainty and fuzziness on the architectural abstraction level, it does not consider confidentiality or other privacy-related quality properties. By applying fuzzy methods in graph-aware data flow analysis, we found that their expressiveness is limited regarding confidentiality, which is hard to quantify, see Section 7.4. Thus, we argue that the application of GuideArch to analyze confidentiality is not straightforward.

Another design space exploration tool is *ArcheOpterix* by Aleti et al. [4]. It can also optimize a given architecture for multiple criteria using evolution techniques and design constraints. However, similarly, it does not support a confidentiality analysis, and the design space modeling is more restrictive. The quality attributes of "safety, reliability, security, performance, timeliness, and resource consumption" [4] are only considered to be future work for ArcheOpterix. At the time of writing, we were unable to find analysis extensions suitable for security, privacy, or confidentiality. Vanherpen et al. [261] also present an approach to model-based design space exploration. They present a pattern catalog of techniques known from architecture optimization but do not focus on analysis support. Gerasimou et al. [85] present *EvoChecker*, a search-based approach employing evolutionary algorithms in automated model synthesis. They focus on quality of service properties like reliability, response time, and cost. In sum, design space exploration approaches are versatile, but do often not support security-related attributes but focus, e.g., on performance. While these approaches can analyze a wide variety of quality properties, they are not appropriate to consider confidentiality as they lack the required expressiveness to consider data processing and data flow constraints.

Last, there is a relation to the topics of variability, product lines, and testing, as they share similar challenges regarding the size of the design space and combinatorial explosion, as discussed in Section 7.6. Here, Abbas et al. [1] propose to model the variability in quality concerns, thereby expressing what does vary how and for which reason. This shall reduce the number of required quality attribute scenarios. Another approach is the combination of design models and test models to minimize testing efforts while facing feature interaction and ensuring a comprehensive coverage of interactions. Oster et al. [191] combine pairwise feature generation with model-based testing and Lochau et al. [156] combine feature models and state charts that represent test models. The challenge of feature interactions is comparable to dealing with uncertainty interactions, see Section 6.7. By employing the notion of independent Transpose Flow Graphs (TFGs), we can minimize the required analysis effort due to interacting uncertainties, see Section 7.2. Here, mapping the uncertainty model to the architectural model to minimize the analysis effort is comparable to the mapping applied in related work. We require full coverage of all interactions on a single TFG, as a missing variation could cause missed confidentiality violations due to uncertainty, see the discussion of recall in Section 9.1.

10.1.5. Architecture-Based Analysis

Besides automatically optimizing software architectures proposed in the previous paragraphs, architectural models are often used as a means to analyze the quality of software architectures or to help in the design process. These approaches are closely related to this thesis as we also build on the architectural model to analyze confidentiality. In the following, we summarize related architecture-based analyses.

When considering architectural models as a baseline for uncertainty analysis, we first have to address the question of incorporating uncertainty into the model. Here, Garlan [80] proposed already more than a decade ago to consider uncertainty as a first-class entity. There are numerous notations for incorporating uncertainties of different types into software models [255]. For instance, SysML [183] and the Unified Modeling Language (UML) MARTE Profile [185] provide stereotypes and properties to represent some types of uncertainty, especially measurement uncertainty. However, existing notations allow modeling mostly homogeneous uncertainties, i.e., of the same type. The aforementioned PSUM standard [184] also provides a metamodel for representing different types of uncertainty but is still a work in progress. Regarding measurement uncertainty, Bertoa et al. [28] propose its inclusion into primitive data types of UML models. Our modeling approach of uncertainty and our distinction between five central uncertainty types is also inspired by the aforementioned work, see Subsection 5.2.3. As discussed in Section 5.3 and Section 7.5, we can express all identified uncertainty sources [103] as part of the architectural model.

Famelis and Chechik [74] introduce *DeTUM*, which stands for design time uncertainty management. This tool-supported approach handles uncertainty using partial models. It introduces uncertainty in the start phase and then resolves it in later stages. They also refer to open design decisions as a source of uncertainty, similarly to the *cone of uncertainty* [167], see Section 5.2. However, the authors do not mention security-related quality attributes like confidentiality. They explicitly mention the requirement of external support to assess the impact of uncertainty—as we offer it with our work.

Goseva-Popstojanova and Kamavaram [93] present an approach for architecture-based reliability analysis under uncertainty. Here, values such as the probability that a particular component is used or the reliability of a component contribute to the analysis of uncertainty. Using Monte Carlo simulation, the authors can analyze how uncertainty propagates from model parameters into reliability estimations. Although we investigate another quality property and use data flow analysis instead Monte Carlo simulation, we argue that our method is comparable. Both approaches use appropriate architectural models and derive the impact of uncertainty. However, we argue that confidentiality and reliability as quality properties differ fundamentally. Therefore, the approach is not suitable for detecting confidentiality breaches, and we are not able to analyze reliability. In sum, there exist many approaches to model and analyze uncertainty regarding other quality properties of software architectures. However, they do not explicitly take confidentiality into account.

10.1.6. Architectural Design Decisions and Uncertainty

Architectural Design Decisions (ADDs) are related to uncertainty. As depicted with the *cone of uncertainty* [167], open design decisions introduce uncertainty about the software system to design, and making decisions reduces said uncertainty.

Kruchten [150] presents an ontology of ADDs. The author distinguishes between existence, property and executive decisions and provides an overview of ADDs attributes. This is especially relevant when considering uncertainty that can void existing decisions and require software architects to backtrack. Jansen and Bosch [124] see software architecture as a composition of ADDs. This shows how uncertainty, e.g., about the system context, can hinder good software design as the *best* decision might not be found. Although both approaches do not focus on uncertainty, they inspired our classification which is strongly coupled to architectural design, see Section 5.3.

Lytra and Zdun [159] propose the use of fuzzy logic to incorporate inherent uncertainty into reusable ADDs. This shall enable software architects to share and reuse knowledge about the impact of uncertainty on quality attributes. They employ a Fuzzy Inference System (FIS) to identify the most appropriate design decisions and also foster the importance of decision documentation. This is related to our data flow analysis under environmental uncertainty that is also based on fuzzy inference, see Section 7.4. Furthermore, we also address knowledge exchange and documentation with our uncertainty catalog, see Section 5.7. Although this approach can also handle security-related quality attributes, violations due to integration issues remain hidden. Here, the integration of a dedicated analysis—such as our confidentiality analysis—helps identifying also fine-grained problems. Future work could investigate whether defining reusable ADDs to ensure confidentiality is feasible.

Zimmermann et al. [280] present a framework to support the identification, making, and enforcement of ADDs. This is especially helpful regarding the aforementioned backtracking, i.e., reverting decisions in case of design errors. Here, awareness of dependencies minimizes the required effort and the chance of introducing new errors while backtracking.

Noppen, J.A.R. et al. [181] discuss design decisions under imperfect information by explicitly modeling uncertain aspects of the architecture based on fuzzy techniques and design trees to record the design history. Busari and Letier [43] present the *RADAR* approach [44] that comprises a modeling language and decision analysis support. They highlight dependencies between decisions and provide support to find Pareto-optimal solutions. Although these approaches help to minimize errors during the design and help to recall the decision paths, they do not support analyzing software systems regarding confidentiality. Nevertheless, we argue that such frameworks are helpful in combination with modeling and analysis support, which should be investigated in the future.

10.1.7. Uncertainty Management and Knowledge Sharing

Regarding the management of uncertainty and ADDs, the previous paragraphs introduced a multitude of approaches for modeling, analyzing, and optimizing software architectures. In the following, we focus on uncertainty management and knowledge sharing beyond the design of single software systems. Collecting and consolidating knowledge on architectural design and uncertainty supports software architects [86, 159, 273, 280]. It can help to address unanticipated change that is not completely unforeseen and can be tackled, e.g., by minimizing assumptions or creating reusable system building blocks. [81]. The need for uncertainty management, reusable methods, and end-to-end approaches has also been highlighted in the aforementioned surveys and research roadmaps [115, 255, 272, 273]. Knowledge sharing to counter uncertainty is also known from legal sciences [251].

With 🔧 ARC^{3}N, we present a catalog of uncertainty sources to overcome the limitation of knowledge being scattered among researchers and institutions. A related approach is *Decision Buddy*, a tool-supported collaborative approach regarding design decisions by Gerdes et al. [86]. By collecting and describing the effects of ADDs, the decision-making of software architects can be supported with suitable and ranked solutions. *privacypatterns.eu* is a web-based collection of patterns to enhance privacy Colesky and Caiza [59]. The collection comprises patterns like access control, single point of contact, or informed consent [58] Last, *arc42* represents an interactive collection of software qualities [247], comprising requirements, definitions, and relations. These works provide comparable approaches to our proposed solution but cannot be applied to uncertainty and confidentiality due to the domain gap. Additionally, the aforementioned tooling is partially fragile due to link decay [91] or lacks open accessible data. To counter such issues and to reach high usability and longevity, our tool support is publicly available, see Section 5.7. Ultimately, we want to stress the importance of efforts towards catalogs and guidelines, especially with regard to cooperation among researchers and science transfer[2].

Jasser and Riebisch [127] present a repository of security solutions to support ADDs. Exemplary solutions are access control, input sanitization, or the principle of the least

[2] The importance of guidelines and checklists was already emphasized by practitioners at EMLS' 21, in the very first workshop in which our approach was discussed [99]. 🔧 ARC^{3}N represents one result of these findings, which received very positive reviews and feedback from the community.

privilege. Here, the focus on security as a central quality property enables more precise classification, e.g., by relating to sub-goals like confidentiality, integrity, or availability [120]. We consider this to be highly related to our catalog approach, although we approach the challenge from the opposite direction: Instead of providing solution techniques, we focus on supporting the identification of potential problems in the form of uncertainties.

Last, Lupafya and Balasubramaniam [158] present a framework for considering uncertainty in the software design. Their conceptual model connects viewpoints to uncertainties, risks and opportunities, and their mitigation or exploitation. Furthermore, they present a classification of uncertainty based on consolidating existing work. Although the approach looks promising at first sight, it falls short of providing further management, modeling, or analysis support beyond describing uncertainty sources with attributes. Furthermore, combining existing classifications without further purpose lacks novelty [132].

10.1.8. Summary

The intersection of research on software architecture and uncertainty presented in this section represents the largest body of related work. According to the aforementioned surveys, the state of the art is still of an exploratory nature, lacks comprehensive tool support [115, 255], and does also not focus on confidentiality. Our approach addresses this gap regarding confidentiality with ready-to-use tool support that can easily be incorporated into existing approaches. Furthermore, related work often inspired us. For instance, our classification is based on existing taxonomies but tailored to confidentiality. The decision to opt for a web-based and open-source catalog approach was also influenced by the perceived link decay [91] of related work. Our approach is by far not the first to incorporate uncertainty modeling in the form of scenarios into the architectural design. However, we were unable to find related work that can identify confidentiality violations due to uncertainty. We conclude that our approach represents novel work that can be combined with existing frameworks to compose comprehensive end-to-end approaches [272].

Finding: Related work provides the foundation and inspiration for many aspects of our research. However, to the best of our knowledge, there exists no directly comparable approach. Ultimately, combining multiple analyses that differ in supported uncertainty sources and targeted quality properties is expedient.

10.2. Software Architecture and Confidentiality

The next section of related work we discuss in this thesis represents the intersection of software architecture and confidentiality. We present work from the field of model-based security analysis and existing approaches to architectural data flow analysis. Afterward, we revisit the specification of confidentiality requirements as they represent one important input to data flow analysis that can impact the accuracy of identified confidentiality violations. This includes related work on access control.

10.2.1. Model-based Security Analysis

Model-based analyses use the model representation of software systems to evaluate their quality, e.g., by using architectural models. They use techniques known from Model-Driven Software Development (MDSD), see Section 2.3. Nguyen et al. [177] present a SLR based on 108 primary studies. They find that the vast majority of papers use standardized notations like the UML, but only less than half of the investigated papers consider confidentiality. When confidentiality is discussed, it is often combined with considering authentication or authorization. Only 11% of the investigated publications focus on analyzing confidentiality. They also find a lack of analyses that simultaneously consider multiple security concerns.

Jürjens [131] present *UMLsec* that extends UML by defining a security profile [129, 130]. By incorporating security-related information into UML, software architects can reuse existing UML diagrams for security analysis. It supports different kinds of analyses, such as information flow or access control. In contrast to our confidentiality analysis approaches, it does not support access control on data, or more complex confidentiality requirements that go beyond, for instance, encrypted communication. Another UML security profile is *SecureUML* by [157]. It supports Role Based Access Control (RBAC) together with statements in the Object Constraint Language (OCL) to support dynamic properties in authorization constraints. Similarly, there is no support for more advanced constraints regarding confidentiality. Although both approaches represent relevant steps towards model-driven security, they both show the lack of comprehensive support for confidentiality beyond access control, or encryption, as already discussed in the aforementioned survey. Furthermore, they do not support uncertainty in their analyses. Last, Rønneberg [209] propose information flow analysis in CBSE, following the approach of correctness-by-construction to enable security guarantees.

Walter et al. [267] present a comprehensive approach to identifying confidentiality violations using attacker propagation [265, 267] and attack path analysis [264, 268]. They extend the Palladio Component Model (PCM) [207] to express vulnerabilities and access control with the eXtensible Access Control Markup Language (XACML) [182]. This enables the propagation of attackers within the software architecture. After each propagation step, the analysis considers which credentials or access rights the attacker might have obtained, e.g., due to vulnerabilities within the system. The approach also considers dependencies within the software architecture. For instance, if an attacker gains access to a resource container, this also affects all components deployed on this resource. The resulting attack graphs shall help software architects to estimate the impact of vulnerabilities and access control decisions. This approach is highly related to our work, as it uses a similar approach to architectural models [207] and architecture-based propagation [46, 106]. However, it has two limitations regarding our needs. First, there is no consideration of data flows, as confidentiality violations are identified by tracing attack paths. Second, there is no comprehensive support for considering uncertainty, although initial approaches exist [265]. The authors acknowledge this limitation and state that "the combination with other uncertainty mitigation approaches" [264] could be expedient.

10.2.2. Data Flow Analysis

The concept of DFDs and data flow analysis is not new [64]. Nevertheless, data flow analysis approaches are often used to assess the security of software systems. This can be achieved by building on noninterference like *JOANA* [242] or deductive verification like *KeY* [3]. Other approaches extract data flows from source code without considering the software architecture. *RogueOne* by Sofaer et al. [244] uses data flows to identify rogue updates, i.e., malicious changes to widely used libraries that aim to attack dependent software systems. More recently, GitHub [89] pushes the adoption of *CodeQL* [173] for data flow-based source code analysis and vulnerability detection. Due to the broad applicability, such analyses represent promising approaches to analyzing the confidentiality of real-world software. However, they lack the connection to the architectural abstraction, which becomes visible, e.g., when considering architecture-related information like the deployment. Here, approaches to connect source code analyses and architectural analyses exist, e.g., by Kramer et al. [147] and Reiche et al. [204]. However, they do not consider uncertainty. Combining source code analyses with architectural analyses and making uncertainty-aware represents a promising direction for future research. With appropriate tool support, DFDs represent a powerful and commonly used mechanism for threat analysis [25] that helps in identifying security-related issues [226].

The most related architectural data flow analysis is presented by Seifermann et al. [236]. Their approach considers additional context information, such as the deployment, enabling software architects to analyze confidentiality during early design phases. We introduced their underlying meta model for the confidentiality-focused analysis of DFDs in Section 2.5. Furthermore, they present an approach for label propagation to identify confidentiality violations. Software architects can specify and analyze confidentiality requirements regarding noninterference, encryption, or access control. A Domain-Specific language (DSL) [100, 105] enables the specification of data flow constraints. Our data flow analysis framework [36], introduced in Section 7.2, builds on this approach. However, the original data flow analysis is unable to consider uncertainty. The authors acknowledge this limitation and the importance of considering uncertainty in the architectural design and analysis [233].

Boltz et al. [39] focus on the collaboration of legal and software experts in the assessment of information security and data protection. They propose a model-based approach using the data flow analysis framework [36] described in Section 7.2. By using the Architectural Description Language (ADL) PCM, analyses of multiple quality properties are also possible, e.g., regarding performance and security [38]. However, their approach only considers uncertainty coming from the legal side. Here, combining our findings on modeling and analyzing uncertainty could be a promising approach. Pilipchuk [196] present an access control analysis based on business processes to align processes and access control policies [197]. By extracting access control requirements from business processes and using them in architecture-based data flow analysis, forbidden data flows can be identified. However, this approach also focuses on conformance without considering uncertainty.

Peldszus et al. [193] also present a data flow analysis approach called *SecDFD*. They check for compliance between models of the design and the implementation to identify violations.

The mapping is automated but lacks support for custom analysis definitions or data flow constraints and also does not consider uncertainty. The early detection of design flaws [256, 257] is closely related to the aforementioned data flow analysis by Seifermann et al. [236] and was also a baseline for the unified modeling primitives for DFDs used in our work, see Section 2.5. Another approach is *iFlow* by Katkalov et al. [135]. They use UML models to derive and analyze data flows. Furthermore, the generated source code can be verified. Gerking et al. [88] present a model-based information flow analysis to identify timing channels. Both approaches do not consider uncertainty, but Gerking [87] mentions the incorporation of uncertainty as potential future work.

Last, Schneider and Scandariato [228] present an automated approach to extract DFDs from the source code of microservice applications. They enrich the extracted DFDs with security-related information, e.g., regarding data storage, passwords, or logging. This enables comprehensive data flow analysis that also considers architectural information. They also published a large data set comprising the DFDs of 17 real-world microservice systems [227]. In a proof of concept, we transformed and analyzed this data set using our data flow analysis framework [36] to replicate the identified confidentiality violations. There are also other steps towards security benchmarks [14]. We state that this represents a promising first step towards comprehensive data flow analysis that considers both the information from the source code and the software architecture. However, similar to the previously discussed approaches, uncertainty is not considered.

10.2.3. Modeling Confidentiality Requirements

The data flow analysis approaches presented in this thesis use confidentiality requirements in the form of data flow constraints as input, see Section 2.5. Similarly, many related approaches discussed in the previous paragraphs use a dedicated requirement specification language. Often, confidentiality requirements are also specified using access control [177]. The specification of confidentiality requirements impacts the identified confidentiality violations, with our without considering uncertainty. We briefly summarize related work in both fields in the following.

Onabajo and Jahnke [189] investigated the properties of confidentiality requirements using grounded theory. They present a model for requirements comprising elements like data, stakeholders, statements, purpose, and temporal validity. This shall support the precise definition of requirements and enable the formal reasoning on derived rules, e.g., from legal frameworks like the General Data Protection Regulation (GDPR) [73]. To provide a foundation to describe such confidentiality requirements for data flow analysis, we presented a meta model and a DSL in previous work [105]. Data flow constraints follow the pattern of disallowing flows under certain conditions based on the characteristics of the data and involved entities or parts of the system. A generalized form of this DSL was presented with the data flow analysis framework [36]. However, these approaches are not able to express restrictions with regard to uncertainty, e.g., whether certain flows shall not be allowed under given uncertain conditions. Here, more research is required.

As discussed previously, access control is often used to express confidentiality requirements through access control policies [233]. To guide software architects from high-level requirements to low-level policies, access control policy refinement techniques have been proposed [277]. Su et al. [154] discuss the automated decomposition of policies based on the resource hierarchy in distributed applications. He and Antón [109] present an approach to define and refine access control policies by analyzing the specification of requirements and the system's database design. Furthermore, both model-based [160, 166] and verification-based [54, 169] approaches exist, e.g., using logic programming languages like Prolog [61, 279]. Unfortunately, all of these approaches do not explicitly consider any kind of uncertainty. The relation of uncertainty, i.e., the lack of knowledge and the specification and refinement of access control policies that respect uncertainty represents an interesting direction for future work.

10.2.4. Summary

The intersection of research on software architecture and confidentiality shows a multitude of modeling and analysis approaches. Following the principle of security and privacy by design [224], the early consideration and analysis of security and privacy is expedient. Moreover, fixing issues in later phases is usually more costly [32, 240]. Related work often uses a model representation to perform security analysis, e.g., based on UML, DFDs, or the PCM. Although we identified many promising approaches, we were unable to identify an architecture-based analysis that both considers confidentiality and uncertainty. Many approaches acknowledged this limitation [87, 233, 264]. Nevertheless, especially our data flow analysis framework [36] is inspired by related work regarding architecture-based [233] and code-based [228] data flow analysis. We repeat the conclusion of the previous section that these findings support the novelty of our work. Moreover, analysis combinations represent promising approaches, e.g., by closing the gap between the design and the implementation regarding confidentiality analysis under uncertainty.

ℹ **Finding:** Related work shows a multitude of security analysis approaches supporting confidentiality based on models specified using the UML, PCM, or DFDs. Despite their versatility, they lack support for considering uncertainty within the modeling and analysis of confidentiality.

10.3. Confidentiality and Uncertainty

The last section of related work discussed in this chapter is the intersection of research on confidentiality and uncertainty. This represents the smallest part of related work as we were unable to identify comprehensive approaches that consider confidentiality under uncertainty. The aforementioned surveys [243, 255] support this shortcoming, as only a few identified approaches explicitly target security; confidentiality is not mentioned at all. We split related work into two groups: uncertainty-aware confidentiality analysis and access control under uncertainty to ensure confidentiality.

10.3.1. Uncertainty-Aware Confidentiality Analysis

We identified two approaches to confidentiality analysis under uncertainty. First, work regarding uncertainty in cloud computing and, second, work regarding risk assessment of data breaches. Tchernykh et al. [254] discuss different types of uncertainty in the context of cloud computing. Moreover, they propose approaches to minimize the impact of uncertainty on the systems' reliability and privacy, e.g., data replication, error correction, and homomorphic encryption. However, the presented work is still in a preliminary state and lacks modeling or automated analysis.

Morali et al. [174] focus on the assessment of risk regarding confidentiality. By propagating data breaches through the modeled software system, the criticality can be measured. The software system is modeled as a graph showing the dependencies between infrastructure nodes regarding exchanged information. Edges of this graph are annotated with the propagation likelihood of attackers, which enables the modeling of risk. However, it is not specified how software architects should be able to annotate every edge with an appropriate propagation likelihood and the evaluation only comprises a single case study. The authors state further studies as future work. Although we assume that also this approach is still preliminary, the modeling of data dependencies and the propagation of attackers is promising. The analysis is related to the work to Walter et al. [268] but requires high modeling effort and does also not claim any automation.

10.3.2. Access Control Under Uncertainty

As discussed previously, access control represents a common approach to ensure the confidentiality of data in software systems. Here, many approaches to access control under uncertainty exist. Although these approaches are not directly comparable to our approach of architecture-based modeling and analysis of confidentiality under uncertainty, they comprise related concepts like fuzziness and taking known uncertainty into account.

Bures et al. [41] discuss the relation of uncertainty and access control in highly dynamic environments like Industry 4.0. Here, uncertainty-aware access control policies directly consider imperfect information in modeling access decisions. Hengartner and Zhong [114] present an access control model for distributed systems that incorporates trust by explicitly specifying remaining uncertainty in access decisions. Other approaches regarding uncertainty in access control also utilize fuzzy logic, e.g., to represent security patterns [116] or to create risk-adaptive access control models to cope with the uncertainty [56]. Numerous other approaches also discuss fuzzy approaches to access control [161, 165, 172, 178, 220, 221]. Only a few take the described *known uncertainty* into account when making access decisions [10, 62].

However, details about such policies are usually not specified in the architectural abstraction but are added during policy refinement. Here, the high degree of uncertainty regarding the structure, behavior, and usage of the software does not allow one to draw precise conclusions on the confidentiality of the overall system. Cheng et al. [56] explain this problem with unforeseeable tradeoffs while defining policies. Strict policies may reduce the risk of data breaches but may harm the flexibility of software systems, especially in highly dynamic environments like implied by Industry 4.0 [41]. The common gap of uncertainty-aware approaches to model access control is the lack of refinement of high-level confidentiality requirements whose abstraction is also a source of uncertainty. In sum, uncertainty-aware access control can be seen as an alternative approach to deal with uncertainty in a software system [194].

10.3.3. Summary

As discussed previously, the intersection of confidentiality and uncertainty yields the least amount of related research. One reason could be the focus of the research community on other qualities like performance and reliability [115, 243]. Nevertheless, we identified related uncertainty-aware confidentiality analyses and numerous approaches to access control under uncertainty. Both fall in one of the two categories of handling uncertainty, discussed by Perez-Palacin and Mirandola [194]. The model can be refined to incorporate the uncertainty and become more resilient, like the approach to consider uncertainty in cloud computing by Tchernykh et al. [254]. Otherwise, the uncertainty can be actively managed as part of the model, e.g., in fuzzy access control policies [56, 165, 178].

Finding: Related work comprises only a few approaches to uncertainty-aware confidentiality analysis. The identified approaches are of a preliminary nature and lack comprehensive modeling and automated analysis support.

10.4. Summary and Outlook

In this chapter, we provided an overview of the related work. First, we focused on work from the areas of software architecture and uncertainty. Here, many surveys like SLRs and roadmaps have been discussed in the last decades. We investigated taxonomies related to our classification, introduced in Chapter 5, and architecture-based analyses related to our impact analysis, introduced in Chapter 6. Furthermore, we summarized the comprehensive state of the art in architecture evaluation that is related to our data flow analyses, introduced in Chapter 7. Last, we revisited the relation of ADDs and uncertainty and existing approaches to uncertainty management.

In the second part of this chapter, we investigated work regarding software architecture and confidentiality. Here, many model-based security analyses and data flow analyses have been proposed. They share common approaches like design time modeling and analysis or using propagation to better assess the system's security. In the last part of this chapter, we discussed work that considers confidentiality under uncertainty, which yielded the smallest amount of publications. Here, we summarized existing uncertainty-aware confidentiality analyses and access control under uncertainty.

In sum, we provided a comprehensive yet comprehensible overview of the state of the art. By comparing our results from the individual sections, we can derive two initial findings on related work. First, related work often covers two, but not all three aspects of our approaches: Architectural uncertainty-aware analysis do not focus on confidentiality, architecture-based confidentiality analysis do not consider uncertainty, and uncertainty-aware confidentiality analysis lack the architectural abstraction. Second, our impression of related work supports the results from the aforementioned surveys [115, 255], e.g., the lack of supporting heterogeneous uncertainty, the lack of consolidated notions of uncertainty and providing tool support, or the challenge of end-to-end approaches [272].

However, recent advances like the OMG PSUM standardization process [184], or the research agenda to understand and manage uncertainty [273] show that the community is well aware of these challenges and actively addresses them. We hope that the contributions of this thesis, which were designed with these challenges in mind, also contribute to this research. Ultimately, please note that this related work chapter does not present a SLR. Thus, it is possible—and even probable—that we missed some related publications.

This chapter summarized work related to our contributions. These contributions are presented in ❯ **Chapter 5: Identification and Classification of Uncertainty Regarding Confidentiality**, ❯ **Chapter 6: Uncertainty Propagation to Enable Uncertainty Impact Analysis**, and ❯ **Chapter 7: Uncertainty-Aware Data Flow Analysis to Identify Confidentiality Violations**. To learn more about the foundations, see ❯ **Chapter 2: Foundations**. Next, we will conclude this thesis in ❯ **Chapter 11: Conclusion**.

10.5. In Simpler Words

Our research is part of three major research communities. First, the software architecture community researches how we can design better architectures to enhance the quality of software systems. Second, the Self-Adaptive Systems (SASs) community researches how we can build more flexible systems that adapt themselves when facing uncertainty. Third, the security community researches how to analyze and ensure security-related properties like confidentiality. In all three communities, many ideas are related to our work. In this chapter, we give an overview of the most related publications.

First, we look into architecture-based approaches that also consider uncertainty. Here, many researchers proposed methods to optimize software architectures regarding uncertainty and to help software architects make better decisions when designing software systems under uncertainty. We learned a lot from these approaches and tried to address the identified shortcomings in our contributions. Second, we look into architecture-based approaches to analyze confidentiality. Here, some of the most related work exists, e.g., the data flow analysis that inspired our data flow analysis framework, presented in Chapter 7.

However, these approaches do not consider uncertainty and thus do not solve the problem we want to address with our work. Third, we look into work that tries to understand confidentiality while considering uncertainty. Here, we only found a few related publications, some of which are in a preliminary state. In sum, we did not find a single publication that does exactly what we propose in this thesis—which is a good thing, because otherwise, the novelty of our work would be limited.

11. Conclusion

This chapter concludes this dissertation. First, we look back and give a short summary on the contributions, their validation, and the key findings of this thesis. Afterward, we present the key benefits of our work. Last, we comprehensively discuss future work and current and future research directions. Note that all previous chapters comprise detailed conclusion and outlook chapters that are summarized here. This is especially true for the contribution chapters, where we also thoroughly discussed the assumptions and limitations of our contributions. For detailed summaries, please refer to Section 5.9 regarding our classification and catalog approach (**C1**), to Section 6.9 for our uncertainty impact analysis and propagation (**C2**), to Section 7.8 for our approaches to uncertainty-aware confidentiality analysis (**C3**), and to Section 9.5 regarding the evaluation.

11.1. Summary

This thesis was located in the intersection of three topics: confidentiality, software architecture, and uncertainty. Confidentiality demands that information is not disclosed to unauthorized persons or organizations [120], which is a crucial quality property due to the high level of connection and the growing volume of data in modern software systems [187]. To address the high number of data breaches [13, 55, 83], confidentiality should be considered early in the design. However, existing software architecture-based confidentiality analyses [36, 193, 196, 233, 264] do not consider uncertainty, i.e., the lack of information or knowledge regarding the software systems or its environment [2, 120, 195]. This major challenge is also acknowledged in related work, e.g., as "lack of systematic approaches for managing uncertainty" [115] or as the need of software engineers "to identify the types of uncertainty that can affect their application domains" [255]. Put simply, software engineering is complex [272] and uncertainty is uncertain [81].

To address this, we introduced an approach that combines the analysis of uncertainty with the analysis of confidentiality. Based on the classification of uncertainty regarding confidentiality (**C1**), we presented two central analysis approaches. First, an uncertainty impact analysis (**C2**) that propagates the effects of uncertainty within the architectural model to predict the potential impact. Second, uncertainty-aware data flow analyses (**C3**) that extend the concept of data flow-based confidentiality analysis [36, 233, 234, 236] by uncertainty awareness. By embracing uncertainty as a first-class concern within software architecture [80], software architects become aware of the consequences and can identify confidentiality violations due to uncertainty. To address the lack of tool-assisted approaches to analyzing and managing uncertainty [115, 255], we provided tool-support for the identification (ARC^{3}N), propagation (UIA), and uncertainty-aware confidentiality analysis (Abunai). This addresses the challenges [99] of understanding the relation of confidentiality and uncertainty, representing uncertainty in architectural abstraction regarding confidentiality, and analyzing confidentiality under uncertainty. We used a running example throughout this thesis to exemplify the ideas and new concepts. Based on the scenario of a simplified online shop, where customers browse for available items and make purchases, we introduced uncertainty sources like user behavior, data processing, deployment, or provider trustworthiness. To deal with uncertainty, we target four key activities: identification, classification, propagation, and analysis. The resulting procedure is aligned with our three contributions and our tool support, see Chapter 4. In the following, we briefly summarize the three contributions of this dissertation:

Identification and classification of uncertainty (C1) Our first contribution comprised a classification of uncertainty regarding its impact on confidentiality [104] and a catalog approach to support the identification of new uncertainty sources. We detailed the relation of uncertainty sources and their impact and investigated existing uncertainty taxonomies [41, 162, 195, 202, 263]. Here, we also coined the term *software-architectural* uncertainty that describes uncertainty on architectural abstraction where early awareness helps in the assessment and mitigation. Afterward, we defined a classification tailored to confidentiality on architectural abstraction. The central category is the *Architectural Element Type*, with options like *External* uncertainty, or *Actor* uncertainty, that are used throughout the remaining thesis. We used this classification to represent uncertainty as a first-class concern [80] within Data Flow Diagrams (DFDs). Last, we presented an approach to the collaborative identification of uncertainty sources [103], which results in the tool-supported catalog ARC^{3}N. Combined, the classification and the catalog approach contribute to our understanding of the relation between uncertainty and confidentiality in software architecture and provide an answer to **Research Question 1**.

Finding: An uncertainty classification tailored to confidentiality provides the terminology to describe uncertainty sources and their impact. The integration of this classification into an uncertainty source catalog simplifies the identification of new uncertainty sources and improves the understanding of software architects.

Uncertainty impact analysis (C2) Our second contribution focused on the propagation of uncertainty within architectural models to define an uncertainty impact analysis. First, we discussed the representation of uncertainty in architectural models [102, 255]. Afterward, we provided algorithms for the propagation of all five uncertainty types according to our classification within architectural models and also within DFDs. Combined, this enabled the definition of an uncertainty impact analysis that predicts the potential impact of uncertainty sources and potential locations of confidentiality violations. With UIA, we also provided tool support for architecture-based uncertainty impact analysis. By combining the previously introduced catalog approach with this impact analysis, we also addressed the identification problem in this analysis. Last, we generalized our findings [49] to address the Uncertainty Interaction Problem (UIP) by defining the notion of Uncertainty Flow Diagram (UFD). To conclude, this answers ❓ **Research Question 2** about the propagation and impact assessment of uncertainty regarding confidentiality.

> ℹ **Finding:** The propagation of uncertainty within architectural models helps to estimate the potential impact and to predict confidentiality violations. Moreover, uncertainty propagation can also be used to analyze uncertainty interactions.

Uncertainty-aware confidentiality analysis (C3) As the third contribution, we presented four approaches to uncertainty-aware data flow analysis to analyze the confidentiality of architectural models. First, we introduced an extensible data flow analysis framework [36] and provided the conceptual basis of uncertainty-awareness in data flow analysis. Afterward, we presented four analysis approaches that differ in the complexity of the analysis algorithm and in supported uncertainty types. We presented two type-specific approaches, tailored to structural uncertainty, and environmental uncertainty, and two type-agnostic analyses. We also introduced 🔧 Abunai, which provides tool support for the most advanced type-agnostic approach. Last, we also discussed the complexity of uncertainty-aware data flow analysis as naive approaches suffer from the combinatorial explosion, known from design space exploration [143]. These uncertainty-aware data flow analyses represent an answer to ❓ **Research Question 3**, which asked about how to analyze confidentiality requirements while considering uncertainty.

> ℹ **Finding:** Architecture-based confidentiality analysis can be achieved by considering the impact of uncertainty on the flowing data. Here, building on the findings of uncertainty propagation and interaction helps to ensure accuracy and scalability while supporting all identified uncertainty types.

The validation of our contributions was based on a Goal Question Metric (GQM) plan [16, 17] that comprised 8 goals, 19 questions, and 32 metrics. First, we evaluated our uncertainty classification according to an taxonomy evaluation method [132], and the usability of our catalog approach. Afterward, we evaluated the uncertainty impact analysis using the same evaluation plan already used to evaluate change impact analysis [210, 211], i.e., focusing on accuracy and effort reduction. Last, we evaluated the scalability and accuracy of our uncertainty-aware data flow analyses. As part of the evaluation, we used six evaluation scenarios, see Chapter 8. This scenarios originate from related work [133, 135, 152, 203, 234] or real-world software systems [69, 119, 199, 216]. All scenarios have different confidentiality requirements, and different confidentiality violations due to uncertainty. Additionally, we conducted two user studies. Overall, we find the evaluation results satisfying. Many measurements represent perfect results, while others show room for improvement despite being acceptable. The biggest room for improvement is in the usability of the classification and catalog approaches, which is not surprising as both represent novel approaches with only prototypical tool support. However, we especially want to highlight the impressive effort reduction of 86% provided by our uncertainty impact analysis and the complexity reduction of our uncertainty impact-aware data flow analysis, see Section 9.5. In Chapter 1, we defined the research goal of this dissertation:

❶ **Research Goal:** Define a classification of uncertainty sources regarding confidentiality using the software architectural abstraction. Provide architecture-based analyses that predict the impact of uncertainty sources and assist software architects in identifying confidentiality violations with respect to uncertainty.

We conclude, that our three Contributions **C1** – **C3** address this research goal and the evaluation results show that our contributions are of high quality. Moreover, the contributions address other gaps known from recent surveys, such as the need for systematic approaches for managing uncertainty [115], the representation of uncertainty in models [255], and the need for uncertainty-aware end-to-end approaches [273]. Last, some of the concepts presented in this thesis have already been generalized, e.g., to address uncertainty interactions [49]. We hope that our findings help the research community to further advance in the areas of uncertainty, confidentiality, and software architecture.

11.2. Benefits

We see many benefits in the contributions of this dissertation. These benefits were already enumerated in the summaries of the contributions chapters, i.e., in Section 5.9, Section 6.9, and Section 7.8. In the following, we repeat the key benefits.

The explicit modeling of uncertainty and confidentiality on architectural abstraction and the incorporation of these concepts in architecture-based confidentiality analysis grants several benefits. The benefits of our classification include precise terminology to discuss and understand uncertainty regarding confidentiality. This supports both software architects and security experts in modeling and analyzing software systems. As proposed in Chapter 4, we do not require an additional *uncertainty expert* role, as the required knowledge is contained in the classification, the uncertainty source catalog, and all analyses. Here, our tooling 🔧 ARC^{3}N represents a good starting point. By identifying and assessing uncertainty sources early, the reasoning and prioritization of Architectural Design Decisions (ADDs) is simplified and costly backtracking is minimized. This is especially true regarding uncertainty interactions [50], which represent uncertainty impacts that are particularly hard to find and mitigate. Last, our classification lays the foundations for further integration of uncertainty in the architecture-based confidentiality analyses.

Propagating uncertainty helps software architects in handling uncertainty [2]. Architecture models can be annotated with uncertainty sources from existing catalogs [103, 104], which helps in the documentation and to raise awareness. The analysis helps in predicting and mitigating confidentiality violations. Using a confidentiality analysis for this purpose would require software architects to manually understand and model the impact of uncertainty, which requires more effort and expertise. As proposed in our procedure in Section 4.1, we see uncertainty propagation as the first step to analyzing an architectural model prior to any detailed confidentiality analysis. Here, our tooling 🔧 UIA presents a good starting point. The calculated models of our analysis can also be used for regression testing or to handle uncertainty at runtime [65].

Our uncertainty-aware data flow analysis approaches enable an earlier and more accurate assessment of the overall system's confidentiality. Detecting and repairing confidentiality issues earlier can reduce cost [32]. Here, we do not even require a fully defined software architecture as the modeled uncertainty can be reused compared to regression tests. The results of our uncertainty-aware confidentiality analysis utilize the available information within architectural models more efficiently and shall enable an easier interpretation by software architects. This might also minimize change efforts, subsequent faults, and costs due to confidentiality violations. This is especially true for large models with specific confidentiality violations that have to be traced through the complete system to become manageable. Here, we refer to our tooling 🔧 Abunai. Including uncertainty in the design process may increase the flexibility at runtime when facing unexpected context changes [41]. This shall help in building more resilient software systems. By including the impact of uncertainty, the quality of confidentiality analysis results increases due to higher coverage of considered problems and possible violations. This is especially true for runtime uncertainty and related to topics like Self-Adaptive System (SAS) [115, 243] and antifragility [40, 92, 201]. Lastly, our results can also be used as guidelines.

11.3. Future Work

In the following, we provide an overview of potential future work. Note that we make no claim to completeness, as there is still a lot to do in this area [115, 273]. For the sake of traceability in future publications, we enumerate all 55 ideas for future work as **FW1** – **FW55**. We divide future work into four parts: further evaluation of our approach, extension of our approach and its tool support, integration of our analyses with other approaches or frameworks, and generalization of our contributions.

Evaluation Although we already presented a comprehensive evaluation in Chapter 9, further evaluating our approach is always possible. For instance, by applying our uncertainty impact analysis and uncertainty-aware data flow analyses to more evaluation scenarios from other domains (**FW1**). Furthermore, we could repeat the user studies with the final versions of both the uncertainty classification (**FW2**) and the uncertainty catalog approach (**FW3**) to investigate changes in usability. Especially regarding the catalog approach, a user study that compares the performance to a baseline without tool support would be expedient (**FW4**). Future work could revisit the validity of identified confidentiality violations compared to previous approaches [234, 236], e.g., by utilizing formal methods (**FW5**). Here, also the formal representation of Nondeterministic Data Flow Diagrams (NDFDs) beyond Directed Acyclic Graphs (DAGs) can be expedient (**FW6**), comparable to the graph-based formalization of Alshareef et al. [6]. Last, broader studies could evaluate the usability of our analyses in real-world software architectures (**FW7**).

Extension During this dissertation, we identified several extension possibilities for all contributions. First, our catalog approach and its tool support 🔧 ARC^{3}N could be extended, e.g., by integrating recommendation techniques (**FW8**). Also, the explainability [26, 27] of uncertainty is relevant (**FW9**), as the understandability can impact the usability of software architects. This does not only include recommending and explaining uncertainty sources in isolation but also considering the software architecture under study (**FW10**). Here, the application of Large Language Modelss (LLMs) can be expedient (**FW11**).

Other extensions could target the beginning and the ending of our approach, i.e., the requirement phase and the mitigation of uncertainty [180]. First, we could (semi-) automatically derive detailed confidentiality requirements based on more abstract protection goals (**FW12**). This includes the extension of the modeling approach of confidentiality requirements as data flow constraints [105], e.g., to express multiple levels (**FW13**). An example would be higher-level requirements that originate from the law and lower-level requirements that represent, e.g., access control policies (**FW14**). This could result in approaches to uncertainty-aware access control policy refinement [97] (**FW15**). There is more research required at the intersection of legal sciences and software architecture [39], e.g., to better understand and support the legal analysis of software architecture and the collaboration of experts from both fields (**FW16**). Furthermore, defining patterns of confidentiality requirements and data flow constraints could increase the usability of architecture-based data flow analysis (**FW17**). All constraint-related future work would also benefit from a clearly defined meta model based on the new data flow analysis framework (**FW18**) that extends previous work [105]. Last, we did not consider the analysis of uncertainty within requirements (**FW19**), which can highly impact the analysis results.

Our uncertainty-aware data flow analyses yield confidentiality violations due to uncertainty. Future work could extend this to provide means to mitigate the effects of uncertainty (**FW20**). Examples are the automated repair of models to comply with confidentiality requirements (**FW21**), or constraint optimization (**FW22**). Also the holistic automated repair of confidentiality-violating models without even considering uncertainty could be possible (**FW23**). Regarding the propagation of uncertainty with 🔧 UIA, the precision of our impact analysis could be increased by further minimizing the impact set. Currently, starting from an initial uncertainty impact location, the remaining data flows are added to the impact set. However, based on guarantees or checkpoints within the system [51], e.g., guaranteed encryption, the impact set could be reused (**FW24**). Last, future work could address the 🔧 Abunai tool support. Here, the modeling of uncertainty sources could be simplified (**FW25**), e.g., by providing graphical editors that surpass the quality of the diagrams shown in Appendix A (**FW26**), or a Domain-Specific language (DSL) under uncertainty (**FW27**). Also, the integration of low-code approaches in the data flow analysis and 🔧 Abunai is possible (**FW28**). Here, the textual representation of Palladio Component Model (PCM) instances and confidentiality-related information like labels and assignments could be expedient (**FW29**). This would also address the need for end-to-end approaches [273]. Last, our work would also benefit from advances in the underlying data flow analysis framework [36, 118, 179], e.g., the graphical display of confidentiality violations (**FW30**) and uncertainty (**FW31**), support for cyclic data flow diagrams [12] (**FW32**), and graphical editors to annotate uncertainty sources to the model (**FW33**).

Integration Besides extending our approach, future work can also address the previously discussed integration with other analysis approaches or into other analysis frameworks. First, future work should investigate whether our approach can be combined with CodeQL [89, 173] to enable code-based uncertainty propagation (**FW34**). This could also enable the coupling of data flow analysis regarding architectural models and code [156, 204] (**FW35**). Another potential integration could be the extension of extracted DFDs from microservices [228] by uncertainty [179] to increase the expressiveness (**FW36**). Other viable combinations are the integration of uncertainty in attack path analysis [268] (**FW37**) or other architecture-based security analyses [131, 135, 157] (**FW38**). Here, natural language processing can be used to derive data flow constraints from requirements (**FW39**) to enable a continuous security analysis that comprises all development phases [229] (**FW40**). Furthermore, based on the software architecture, the uncertainty-aware confidentiality analysis can be combined with other quality properties (**FW41**), e.g., performance, cost, or reliability. This could be achieved with an extension of design space exploration approaches like PerOpteryx [144] (**FW42**), similar to Subsection 7.4.1. Also, the coupling of our uncertainty catalog approach 🔧 ARC^{3}N with other catalogs is possible (**FW43**), e.g., by relating uncertainty to security solutions [126]. Last, our analyses could be integrated into already existing frameworks (**FW44**), e.g., RADAR [43, 44], Rainbow [82], DeTum [74], or the approach of Lytra and Zdun [159].

Generalization Last, we present potential future work to generalize our findings. First, we only considered simple uncertainty interaction between secondary uncertainty and primary uncertainty. Here, our findings could be used to also address other types of uncertainty interactions (**FW45**), see Subsection 7.5.2. Our findings could also serve the general discussion about the UIP [50, 52, 273] (**FW46**), as initially shown by Camara et al. [49]. The same applies to the upcoming discussion about antifragility [40, 42, 92, 94, 95, 201]. Here, especially uncertainty propagation could become valuable (**FW47**). In SASs, future work could consider the application of our analyses at runtime (**FW48**). Uncertainty propagation has also been discussed in the context of coupled models of CPSs [2]. Here, existing consistency relations between models can be used for the uncertainty propagation beyond a single model. However, understanding the relation of intra-model and inter-model uncertainty propagation requires additional research (**FW49**). Furthermore, it should be researched which uncertainty types and model elements are applicable for both (**FW50**) and also regarding other quality properties than confidentiality (**FW51**). This also raises the question for future work whether the viable combination of a classification and a catalog, as presented with our first contribution, is generalizable (**FW52**). We presented approaches to represent uncertainty within architectural models, DFDs, and DAGs. Future work could define a graphical notation of uncertainty in DFDs to enhance documentation and communication (**FW53**). An initial proposal is shown in Appendix E. The easiest generalization could be the consideration of related quality properties that can also be analyzed using DFDs [36], e.g., integrity (**FW54**). Last, future work could define design patterns for uncertainty-resilient software architectures (**FW55**).

In sum, future work has many directions, from revisiting the evaluation, to integrating, extending and generalizing the contributions of this dissertation. All of this future work will build on our research, which was the investigation of architecture-based confidentiality analysis under uncertainty. We look forward to future insights and research results!

11.4. In Simpler Words

This chapter concludes this dissertation. First, we summarize the content of this thesis, thereby focusing on our three contributions. The first contribution was a classification of uncertainty regarding confidentiality. Here, we provided the foundations to reason about uncertainty. For instance, we discussed which types of uncertainty are relevant regarding confidentiality, and which are not. Furthermore, we presented a catalog of uncertainty sources that helps software architects identify new uncertainty sources in the software architecture they try to design or analyze. An example of such an uncertainty source is user behavior, which is always unpredictable to some degree. The second contribution was an uncertainty impact analysis that propagates uncertainty within the software architecture to predict confidentiality violations. This helps software architects to quickly assess the potential uncertainty impact without much additional effort. For instance, they could get the result that an uncertainty only affects non-critical parts of the software, which does not violate confidentiality. The third contribution comprised four uncertainty-aware data flow analyses to identify confidentiality violations due to uncertainty. This helps software architects to build more resilient software systems that ensure confidentiality even when facing uncertainty. Using our approach requires less expertise as the knowledge is encoded as part of the analyses. Due to the automation of our analysis, software architects also require less effort when analyzing an architectural model. Last, our approach helps in the documentation and communication of uncertainty in architectural models and diagrams.

We conclude this chapter by presenting 55 ideas for future work. Of course, there is much more research to conduct in this area, but our ideas can serve as a starting point. We discuss future work regarding further evaluation of our contributions, and many analysis extensions that also add features to our tooling. Afterward, we show how our concepts can be integrated into other work, e.g., existing frameworks for architecture-based analysis. Overall, such future work would provide better and more comprehensive tools for software architects and further simplify their work—besides helping to answers current open research questions. Last, we discuss the generalization of our findings. Here, especially uncertainty propagation represents a promising concept for other applications. For instance, it already has been discussed in the context of uncertainty interactions and in the propagation between different models of the same software system. After four years of research, this concludes our work. We hope that our results will contribute to advancing research in the area of software architecture, confidentiality, and uncertainty[1].

[1] For you, dear reader, who certainly likes numbers as much as I do: In this dissertation, the word *uncertainty* was used 3393 times. You are welcome.

Bibliography

[1] N. Abbas, J. Andersson, and D. Weyns. “Modeling variability in product lines using domain quality attribute scenarios”. In: *Proceedings of the WICSA/ECSA 2012 Companion Volume*. WICSA/ECSA ’12. Association for Computing Machinery, 20, 2012, pp. 135–142.

[2] M. Acosta, S. Hahner, A. Koziolek, T. Kühn, R. Mirandola, and R. Reussner. “Uncertainty in coupled models of cyber-physical systems”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 25th International Conference on Model Driven Engineering Languages and Systems (MODELS). MODELS ’22. ACM, 2022, pp. 569–578.

[3] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich. *Deductive software verification–the key book*. Springer, 2016.

[4] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. “ArcheOpterix: An extendable tool for architecture optimization of AADL models”. In: *2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software. 2009, pp. 61–71.

[5] Z. Alexeeva, D. Perez-Palacin, and R. Mirandola. “Design Decision Documentation: A Literature Overview”. In: *Software Architecture*. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 84–101.

[6] H. Alshareef, K. Tuma, S. Stucki, G. Schneider, and R. Scandariato. “Precise Analysis of Purpose Limitation in Data Flow Diagrams”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ARES ’22. Association for Computing Machinery, 23, 2022, pp. 1–11.

[7] S. Ananieva, S. Greiner, T. Kühn, J. Krüger, L. Linsbauer, S. Grüner, T. Kehrer, H. Klare, A. Koziolek, H. Lönn, S. Krieter, C. Seidl, S. Ramesh, R. Reussner, and B. Westfechtel. “A conceptual model for unifying variability in space and time”. In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC ’20. Association for Computing Machinery, 2020, pp. 1–12.

[8] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. “Modeling Dimensions of Self-Adaptive Software Systems”. In: *Software Engineering for Self-Adaptive Systems*. Springer, 2009, pp. 27–47.

[9] F. Arat and S. Akleylek. “Attack Path Detection for IIoT Enabled Cyber Physical Systems: Revisited”. In: *Computers & Security* 128 (1, 2023), p. 103174.

[10] C. A. Ardagna, M. Cremonini, E. Damiani, S. D. C. di Vimercati, and P. Samarati. “Supporting location-based conditions in access control policies”. In: *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. ASIACCS ’06. Association for Computing Machinery, 21, 2006, pp. 212–222.

[11] P. G. Armour. “The Five Orders of Ignorance”. In: *Communications of the ACM* 43.10 (2000), p. 4.

[12] B. Arp, N. Niehues, T. Hüller, F. Schwickerath, N. Boltz, and S. Hahner. “Analyzing Cyclic Data Flow Diagrams Regarding Information Security”. In: Softwaretechnik-Trends Band 44, Heft 4. Gesellschaft für Informatik e.V., 2024.

[13] R. Ayyagari. “An Exploratory Analysis of Data Breaches from 2005-2011: Trends and Insights”. In: *Journal of Information Privacy and Security* 8.2 (1, 2012), pp. 33–56.

[14] A. Bambhore Tukaram, S. Schneider, N. E. Díaz Ferreyra, G. Simhandl, U. Zdun, and R. Scandariato. “Towards a Security Benchmark for the Architectural Design of Microservice Applications”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ARES 2022: The 17th International Conference on Availability, Reliability and Security. ACM, 23, 2022, pp. 1–7.

[15] J. Bang-Jensen and G. Z. Gutin. *Digraphs*. Springer Monographs in Mathematics. Springer London, 2009.

[16] V. R. Basili, G. Caldiera, and H. D. Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of software engineering* (1994), p. 10.

[17] V. R. Basili and D. M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738.

[18] F. L. Bauer. “Encryption”. In: *Encyclopedia of Cryptography and Security*. Springer US, 2005, pp. 202–202.

[19] L. Bauer, L. F. Cranor, R. W. Reeder, M. K. Reiter, and K. Vaniea. “Real life challenges in access-control management”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’09: CHI Conference on Human Factors in Computing Systems. ACM, 4, 2009, pp. 899–908.

[20] L. Baumgärtner, A. Dmitrienko, B. Freisleben, A. Gruler, J. Höchst, J. Kühlberg, M. Mezini, R. Mitev, M. Miettinen, A. Muhamedagic, T. D. Nguyen, A. Penning, D. Pustelnik, F. Roos, A.-R. Sadeghi, M. Schwarz, and C. Uhl. “Mind the GAP: Security & Privacy Risks of Contact Tracing Apps”. In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). 2020, pp. 458–467.

[21] S. Becker, H. Koziolek, and R. Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software*. Special Issue: Software Performance - Modeling and Analysis 82.1 (1, 2009), pp. 3–22.

[22] D. Bedford. “Evaluating classification schema and classification decisions”. In: *Bulletin of the American Society for Information Science and Technology* 39.2 (2013), pp. 13–21.

[23] N. Benkler. “Architecture-based Uncertainty Impact Analysis for Confidentiality”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2022. 169 pp.

[24] B. J. Berger, K. Sohr, and R. Koschke. “Automatically Extracting Threats from Extended Data Flow Diagrams”. In: *Engineering Secure Software and Systems*. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 56–71.

[25] K. Bernsmed, D. S. Cruzes, M. G. Jaatun, and M. Iovan. “Adopting threat modelling in agile software development projects”. In: *Journal of Systems and Software* 183 (1, 2022), p. 111090.

[26] M. M. Bersani, M. Camilli, L. Lestingi, R. Mirandola, M. Rossi, and P. Scandurra. “A Conceptual Framework for Explainability Requirements in Software-Intensive Systems”. In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. 2023 IEEE 31st International Requirements Engineering Conference Workshops (REW). IEEE, 2023, pp. 309–315.

[27] M. M. Bersani, M. Camilli, L. Lestingi, R. Mirandola, M. Rossi, and P. Scandurra. “Architecting Explainable Service Robots”. In: *Software Architecture*. Vol. 14212. Springer Nature Switzerland, 2023, pp. 153–169.

[28] M. F. Bertoa, L. Burgueño, N. Moreno, and A. Vallecillo. “Incorporating measurement uncertainty into OCL/UML primitive datatypes”. In: *Software and Systems Modeling* 19.5 (1, 2020), pp. 1163–1189.

[29] T. Bitschi. “Uncertainty-aware Confidentiality Analysis Using Architectural Variations”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2022.

[30] BMDV. *BMDV - The Federal Ministry for Digital and Transport launches process for a Mobility Data Act*. 2022. URL: `https://bmdv.bund.de/SharedDocs/EN/PressRelease/2022/081-wissing-data-is-the-basis-of-progress.html` (visited on 09/23/2024).

[31] E. D. P. Board. *Hamburg Commissioner Fines H&M 35.3 Million Euro for Data Protection Violations in Service Centre | European Data Protection Board*. 2020. URL: `https://www.edpb.europa.eu/news/national-news/2020/hamburg-commissioner-fines-hm-353-million-euro-data-protection-violations_en` (visited on 09/23/2024).

[32] B. Boehm and V. Basili. “Software defect reduction top 10 list”. In: *Computer* 34.1 (2001). Conference Name: Computer, pp. 135–137.

[33] S. Bohner. “Software change impacts-an evolving perspective”. In: *International Conference on Software Maintenance, 2002. Proceedings.* International Conference on Software Maintenance, 2002. Proceedings. 2002, pp. 263–272.

[34] N. Boltz, M. Walter, and R. Heinrich. “Context-Based Confidentiality Analysis for Industrial IoT”. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). 2020, pp. 589–596.

[35] N. Boltz. “Architectural Uncertainty Analysis for Access Control Scenarios in Industry 4.0”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[36] N. Boltz, S. Hahner, C. Gerking, and R. Heinrich. “An Extensible Framework for Architecture-Based Data Flow Analysis for Information Security”. In: *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium.* Springer Nature Switzerland, 2024, pp. 342–358.

[37] N. Boltz, S. Hahner, M. Walter, S. Seifermann, R. Heinrich, T. Bures, and P. Hnetynka. “Handling Environmental Uncertainty in Design Time Access Control Analysis”. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2022, pp. 382–389.

[38] N. Boltz, L. Schmid, B. Taghavi, C. Gerking, and R. Heinrich. “Modeling and Analyzing Zero Trust Architectures Regarding Performance and Security”. In: *Software Architecture.* Springer Nature Switzerland, 2024, pp. 253–269.

[39] N. Boltz, L. Sterz, C. Gerking, and O. Raabe. “A Model-Based Framework for Simplified Collaboration of Legal and Software Experts in Data Protection Assessments”. In: INFORMATIK 2022. Gesellschaft für Informatik, Bonn, 2022, pp. 521–532.

[40] H. de Bruijn, A. Größler, and N. Videira. “Antifragility as a design criterion for modelling dynamic systems”. In: *Systems Research and Behavioral Science* 37.1 (2020), pp. 23–37.

[41] T. Bures, P. Hnetynka, R. Heinrich, S. Seifermann, and M. Walter. “Capturing Dynamicity and Uncertainty in Security and Trust via Situational Patterns”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles.* Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 295–310.

[42] S. Burton, R. Calinescu, and R. Mirandola. “Resilience and Antifragility of Autonomous Systems (Dagstuhl Seminar 24182)”. In: *Dagstuhl Reports* 14.4 (2024). Place: Dagstuhl, Germany Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 142–163.

[43] S. A. Busari and E. Letier. “RADAR: A Lightweight Tool for Requirements and Architecture Decision Analysis”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). 2017, pp. 552–562.

[44] S. A. Busari. “Modelling and Analysing Software Requirements and Architecture Decisions under Uncertainty”. Doctoral. UCL (University College London), 28, 2019. 364 pp.

[45] A. Busch, Y. Schneider, A. Koziolek, K. Rostami, and J. Kienzle. “Modelling the Structure of Reusable Solutions for Architecture-Based Quality Evaluation”. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).* 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 2016, pp. 521–526.

[46] K. Busch. *An Architecture-based Approach for Change Impact Analysis of Software-intensive Systems.* Publication Title: KIT Scientific Publishing. KIT Scientific Publishing, 19, 2020.

[47] R. Calinescu, R. Mirandola, D. Perez-Palacin, and D. Weyns. “Understanding Uncertainty in Self-adaptive Systems”. In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS).* 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). 2020, pp. 242–251.

[48] J. Camara, D. Garlan, W. G. Kang, W. Peng, and B. Schmerl. *Uncertainty in Self-Adaptive Systems: Categories, Management, and Perspectives.* 1, 2017.

[49] J. Camara, S. Hahner, D. Perez-Palacin, A. Vallecillo, M. Acosta, N. Bencomo, R. Calinescu, and S. Gerasimou. “Uncertainty Flow Diagrams: Towards a Systematic Representation of Uncertainty Propagation and Interaction in Adaptive Systems”. In: *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* SEAMS ’24. Association for Computing Machinery, 7, 2024, pp. 37–43.

[50] J. Cámara, R. Calinescu, B. H. C. Cheng, D. Garlan, B. Schmerl, J. Troya, and A. Vallecillo. “Addressing the uncertainty interaction problem in software-intensive systems: challenges and desiderata”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems.* MODELS ’22: ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems. ACM, 23, 2022, pp. 24–30.

[51] J. Cámara, B. Schmerl, and D. Garlan. “Software architecture and task plan co-adaptation for mobile service robots”. In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* SEAMS ’20. Association for Computing Machinery, 18, 2020, pp. 125–136.

[52] J. Cámara, J. Troya, A. Vallecillo, N. Bencomo, R. Calinescu, B. H. C. Cheng, D. Garlan, and B. Schmerl. “The uncertainty interaction problem in self-adaptive systems”. In: *Software and Systems Modeling* 21.4 (1, 2022), pp. 1277–1294.

[53] G. Canfora, L. Sansone, and G. Visaggio. “Data flow diagrams: reverse engineering production and animation”. In: *Proceedings Conference on Software Maintenance 1992.* Proceedings Conference on Software Maintenance 1992. 1992, pp. 366–375.

[54] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano. “A comprehensive approach to the automatic refinement and verification of access control policies”. In: *Computers & Security* 80 (1, 2019), pp. 186–199.

[55] L. Cheng, F. Liu, and D. Yao. “Enterprise data breach: causes, challenges, prevention, and future directions”. In: *WIREs Data Mining and Knowledge Discovery* 7.5 (2017), e1211.

[56] P.-C. Cheng, P. Rohatgi, C. Keser, P. A. Karger, G. M. Wagner, and A. S. Reninger. “Fuzzy Multi-Level Security: An Experiment on Quantified Risk-Adaptive Access Control”. In: *2007 IEEE Symposium on Security and Privacy (SP ’07)*. 2007, pp. 222–230.

[57] S.-W. Cheng, D. Garlan, and B. Schmerl. “Evaluating the effectiveness of the Rainbow self-adaptive system”. In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. 2009, pp. 132–141.

[58] M. Colesky et al. “A system of privacy patterns for user control”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC ’18. 2018, pp. 1150–1156.

[59] M. Colesky and J. C. Caiza. “A System of Privacy Patterns for Informing Users: Creating a Pattern System”. In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. EuroPLoP ’18. 2018.

[60] E. Costante, F. Paci, and N. Zannone. “Privacy-Aware Web Service Composition and Ranking”. In: *2013 IEEE 20th International Conference on Web Services*. 2013 IEEE 20th International Conference on Web Services. 2013, pp. 131–138.

[61] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman. “Policy refinement: Decomposition and operationalization for dynamic domains”. In: *2011 7th International Conference on Network and Service Management*. 2011 7th International Conference on Network and Service Management. ISSN: 2165-963X. 2011, pp. 1–9.

[62] F. Cuppens and A. Miege. “Modelling contexts in the Or-BAC model”. In: *19th Annual Computer Security Applications Conference, 2003. Proceedings*. 19th Annual Computer Security Applications Conference, 2003. Proceedings. 2003, pp. 416–425.

[63] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. “Software engineering for self-adaptive systems: A second research roadmap”. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Springer, 2013, pp. 1–32.

[64] T. DeMarco. “Structure analysis and system specification”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, 1979, pp. 255–288.

[65] M. Derakhshanmanesh, J. Ebert, M. Grieger, and G. Engels. “Model-integrating development of software systems: a flexible component-based approach”. In: *Software & Systems Modeling* 18.4 (1, 2019), pp. 2557–2586.

[66] R. Diestel. *Graph Theory*. Vol. 173. Graduate Texts in Mathematics. Springer Berlin Heidelberg, 2017.

[67] E. W. Dijkstra. *Notes on structured programming*. Second edition. T.H. - Report 70-WSK-03. 1970.

[68] C. Durugbo, J. Erkoyuncu, A. Tiwari, J. Alcock, R. Roy, and E. Shehab. “Data uncertainty assessment and information flow analysis for product-service systems in a library case study”. In: *International Journal of Services Operations and Informatics* 5 (1, 2010), pp. 330–350.

[69] M. F. Enaya, T. Klingbeil, J. Krüger, D. Broneske, F. Feinbube, and G. Saake. “A case study on the development of the German Corona-Warn-App”. In: *Journal of Systems and Software* 213 (1, 2024), p. 112020.

[70] N. Esfahani, S. Malek, and K. Razavi. “GuideArch: Guiding the exploration of architectural solution space under uncertainty”. In: *2013 35th International Conference on Software Engineering (ICSE).* 2013 35th International Conference on Software Engineering (ICSE). 2013, pp. 43–52.

[71] N. Esfahani and S. Malek. “Uncertainty in Self-Adaptive Software Systems”. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers.* Lecture Notes in Computer Science. Springer, 2013, pp. 214–238.

[72] N. Esfahani, K. Razavi, and S. Malek. “Dealing with uncertainty in early software architecture”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* FSE ’12. Association for Computing Machinery, 11, 2012, pp. 1–4.

[73] C. of European Union. *REGULATION (EU) 2016/679 (General Data Protection Regulation).* 2016. URL: `https://eur-lex.europa.eu/eli/reg/2016/679/2016-05-04` (visited on 01/19/2021).

[74] M. Famelis and M. Chechik. “Managing design-time uncertainty”. In: *Software & Systems Modeling* 18.2 (1, 2019), pp. 1249–1284.

[75] E. B. Fernandez. “A methodology for secure software design”. In: *Procs. of the 2004 Int. Conf. on Software Engineering Research and Practice (SERP’04.* 2004, pp. 21–24.

[76] B. de Finetti. *Theory of Probability: A critical introductory treatment.* John Wiley & Sons, 2017.

[77] D. Firesmith. “Specifying Reusable Security Requirements.” In: *The Journal of Object Technology* 3.1 (2004), p. 61.

[78] FIRST. *CVSS v4.0 Specification Document.* accessed 07/03/2024.

[79] O. Foundation. *OWASP Top 10:2021.* 2021. URL: `https://owasp.org/Top10/` (visited on 04/30/2022).

[80] D. Garlan. “Software engineering in an uncertain world”. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER ’10.* the FSE/SDP workshop. ACM Press, 2010, p. 125.

[81] D. Garlan. “The Unknown Unknowns Are Not Totally Unknown”. In: *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).* 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). ISSN: 2157-2321. 2021, pp. 264–265.

[82] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. “Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”. In: *Computer* 37.10 (2004), pp. 46–54.

[83] K. M. Gatzlaff and K. A. McCullough. “The Effect of Data Breaches on Shareholder Wealth”. In: *Risk Management and Insurance Review* 13.1 (2010), pp. 61–83.

[84] G. Gehrig. “Enabling the Collaborative Collection of Uncertainty Sources Regarding Confidentiality”. Bachelor’s Thesis. Karlsruher Institut für Technologie (KIT), 2023. 56 pp.

[85] S. Gerasimou, R. Calinescu, and G. Tamburrelli. “Synthesis of probabilistic models for quality-of-service software engineering”. In: *Automated Software Engineering* 25.4 (1, 2018), pp. 785–831.

[86] S. Gerdes, M. Soliman, and M. Riebisch. “Decision buddy: tool support for constraint-based design decisions during system evolution”. In: *2015 1st International Workshop on Future of Software Architecture Design Assistants (FoSADA).* 2015 1st International Workshop on Future of Software Architecture Design Assistants (FoSADA). 2015, pp. 1–6.

[87] C. Gerking. “Model-Driven Information Flow Security Engineering for Cyber-Physical Systems”. PhD thesis. 2020.

[88] C. Gerking, D. Schubert, and E. Bodden. “Model Checking the Information Flow Security of Real-Time Systems”. In: *Engineering Secure Software and Systems.* Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 27–43.

[89] GitHub. *CodeQL.* 2021. URL: `https://codeql.github.com/` (visited on 09/21/2024).

[90] GitHub. *GitHub.com Documentation.* accessed 04/29/2024. 2024.

[91] D. H.-L. Goh and P. K. Ng. “Link decay in leading information science journals”. In: *Journal of the American Society for Information Science and Technology* 58.1 (2007), pp. 15–24.

[92] A. Gorgeon. “Anti-Fragile Information Systems”. In: *Proceedings of the International Conference on Information Systems - Exploring the Information Frontier, ICIS 2015, Fort Worth, Texas, USA, December 13-16, 2015.* Association for Information Systems, 2015, p. 19.

[93] K. Goseva-Popstojanova and S. Kamavaram. “Assessing uncertainty in reliability of component-based software systems”. In: *ISSRE.* 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003. 2003, pp. 307–320.

[94] V. Grassi and R. Mirandola. “The Tao way to anti-fragile software architectures: the case of mobile applications”. In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C).* 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C). IEEE, 2021, pp. 86–89.

[95] V. Grassi, R. Mirandola, and D. Perez-Palacin. “Towards a Conceptual Characterization of Antifragile Systems”. In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C).* 2023.

[96] S. Gürses, J. H. Jahnke, C. Obry, A. Onabajo, T. Santen, and M. Price. "Eliciting confidentiality requirements in practice". In: *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. 2005, pp. 101–116.

[97] S. Hahner. "Architectural Access Control Policy Refinement and Verification under Uncertainty". In: *Companion Proceedings of the 15th European Conference on Software Architecture (ECSA-C)*. 15th European Conference on Software Architecture (ECSA 2021). CEUR Workshop Proceedings, 2021, pp. 1–5.

[98] S. Hahner. *Data Set: Architecture-Based and Uncertainty-Aware Confidentiality Analysis*. doi: 10.5281/zenodo.14796551. Zenodo, 2024.

[99] S. Hahner. "Dealing with Uncertainty in Architectural Confidentiality Analysis". In: *Proceedings of the Software Engineering 2021 Satellite Events*. 8th Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems (EMLS). Gesellschaft für Informatik, 2021, pp. 1–6.

[100] S. Hahner. "Domain-specific Language for Data-driven Design Time Analyses and Result Mappings for Logic Programs". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020. 138 pp.

[101] S. Hahner, T. Bitschi, M. Walter, T. Bureš, P. Hnětynka, and R. Heinrich. "Model-based Confidentiality Analysis under Uncertainty". In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C). IEEE, 2023, pp. 256–263.

[102] S. Hahner, R. Heinrich, and R. Reussner. "Architecture-Based Uncertainty Impact Analysis to Ensure Confidentiality". In: *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). IEEE, 2023, pp. 126–132.

[103] S. Hahner, N. Niehues, N. Boltz, M. Fuksa, and R. Heinrich. "ARC^{3}N: A Collaborative Uncertainty Catalog to Address the Awareness Problem of Model-Based Confidentiality Analysis". In: *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24)*. ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems. ACM, 2024.

[104] S. Hahner, S. Seifermann, R. Heinrich, and R. Reussner. "A Classification of Software-Architectural Uncertainty Regarding Confidentiality". In: *E-Business and Telecommunications*. Communications in Computer and Information Science. Springer Nature Switzerland, 2023, pp. 139–160.

[105] S. Hahner, S. Seifermann, R. Heinrich, M. Walter, T. Bureš, and P. Hnětynka. "Modeling Data Flow Constraints for Design-Time Confidentiality Analyses". In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 2021 IEEE 18th International Conference on Software Architecture (ICSA). IEEE, 2021, pp. 15–21.

[106] S. Hahner, M. Walter, R. Heinrich, and R. Reussner. "Architecture-based Propagation Analyses Regarding Security". In: *Software Engineering 2024.* Software Engineering 2024 (SE 2024). Gesellschaft für Informatik e.V., 2024, pp. 121–122.

[107] N. B. Harrison and A. Aguiar. "The Nature of Questions that Arise During Software Architecture Design". In: *Software Architecture.* Springer Nature Switzerland, 2024, pp. 37–52.

[108] T. Haselton. *Yahoo just said every single account was affected by 2013 attack - 3 billion in all.* CNBC. Section: Technology. 2017. URL: `https://www.cnbc.com/2017/10/03/yahoo-every-single-account-3-billion-people-affected-in-2013-attack.html` (visited on 09/23/2024).

[109] Q. He and A. I. Antón. "Requirements-based Access Control Analysis and Policy Specification (ReCAPS)". In: *Information and Software Technology* 51.6 (1, 2009), pp. 993–1009.

[110] R. Heinrich, K. Busch, and S. Koch. "A Methodology for Domain-Spanning Change Impact Analysis". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). 2018, pp. 326–330.

[111] R. Heinrich, F. Durán, C. Talcott, and S. Zschaler. *Composing Model-Based Analysis Tools.* Springer International Publishing, 2021.

[112] R. Heinrich, S. Koch, S. Cha, K. Busch, R. Reussner, and B. Vogel-Heuser. "Architecture-based change impact analysis in cross-disciplinary automated production systems". In: *Journal of Systems and Software* 146 (2018), pp. 167–185.

[113] R. Heinrich, S. Seifermann, M. Walter, S. Hahner, R. Reussner, T. Bureš, P. Hnětynka, and J. Pacovský. "Dynamic Access Control in Industry 4.0 Systems". In: *Digital Transformation: Core Technologies and Emerging Topics from a Computer Science Perspective.* Springer, 2023, pp. 143–170.

[114] U. Hengartner and G. Zhong. "Distributed, Uncertainty-Aware Access Control for Pervasive Computing". In: *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07).* Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07). 2007, pp. 241–246.

[115] S. M. Hezavehi, D. Weyns, P. Avgeriou, R. Calinescu, R. Mirandola, and D. Perez-Palacin. "Uncertainty in Self-adaptive Systems: A Research Community Perspective". In: *ACM Transactions on Autonomous and Adaptive Systems* 15.4 (2021), 10:1–10:36.

[116] H. H. Hosmer. "Using fuzzy logic to represent security policies in the multipolicy paradigm". In: *ACM SIGSAC Review* 10.4 (1992), pp. 12–21.

[117] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* NIST SP 800-162. National Institute of Standards and Technology, 2014, NIST SP 800–162.

[118] T. Hüller, F. Schwickerath, B. Arp, N. Niehues, N. Boltz, and S. Hahner. "Towards a Data Flow Diagram-Centric Confidentiality Analysis in Palladio". In: Softwaretechnik-Trends Band 44, Heft 4. Gesellschaft für Informatik e.V., 2024.

[119] R. K. Institute. *Open-Source Project Corona-Warn-App*. 2020. URL: `https://www.coronawarn.app/en/` (visited on 07/29/2021).

[120] International Organization for Standardization. *ISO/IEC 27000:2018(E) Information technology – Security techniques – Information security management systems – Overview and vocabulary*. Standard. 2018.

[121] International Organization for Standardization. *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise - Architecture description*. Standard. 2022.

[122] J. Isaak and M. J. Hanna. "User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection". In: *Computer* 51.8 (2018), pp. 56–59.

[123] M. Jackson. "The World and the Machine". In: *1995 17th International Conference on Software Engineering*. 1995 17th International Conference on Software Engineering. ISSN: 0270-5257. 1995, pp. 283–283.

[124] A. Jansen and J. Bosch. "Software Architecture as a Set of Architectural Design Decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). 2005, pp. 109–120.

[125] A. Jansen. "Architectural design decisions". PhD thesis. 2008.

[126] S. Jasser. "Constraining the Implementation Through Architectural Security Rules: An Expert Study". In: *Product-Focused Software Process Improvement*. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 203–219.

[127] S. Jasser and M. Riebisch. "Reusing security solutions: a repository for architectural decision support". In: *Proccedings of the 10th European Conference on Software Architecture Workshops*. ECSAW '16. Association for Computing Machinery, 28, 2016, pp. 1–7.

[128] JCGM 100:2008. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement (GUM)*. ISO Joint Com. for Guides in Metrology, 2008.

[129] J. Juerjens. "Principles for secure systems design". PhD thesis. University of Oxford, 2002.

[130] J. Jürjens. "Towards Development of Secure Systems Using UMLsec". In: *Fundamental Approaches to Software Engineering*. Red. by G. Goos, J. Hartmanis, and J. van Leeuwen. Springer Berlin Heidelberg, 2001, pp. 187–200.

[131] J. Jürjens. "UMLsec: Extending UML for Secure Systems Development". In: *UML 2002 — The Unified Modeling Language*. Red. by G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 2460. Springer Berlin Heidelberg, 2002, pp. 412–425.

[132] A. Kaplan, T. Kühn, S. Hahner, N. Benkler, J. Keim, D. Fuchß, S. Corallo, and R. Heinrich. "Introducing an Evaluation Method for Taxonomies". In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*. EASE '22. ACM, 2022, pp. 311–316.

[133] K. Katkalov. "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme". PhD thesis. University of Augsburg, 2017.

[134] K. Katkalov. *Modeling the Travel Planner Application with IFlow*. 2013. URL: `https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html` (visited on 08/15/2024).

[135] K. Katkalov, K. Stenzel, M. Borek, and W. Reif. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *2013 International Conference on Social Computing*. IEEE, 2013, pp. 51–56.

[136] J. Kephart and D. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003). Conference Name: Computer, pp. 41–50.

[137] A. D. Kiureghian and O. Ditlevsen. "Aleatory or epistemic? Does it matter?" In: *Structural Safety*. Risk Acceptance and Risk Communication 31.2 (1, 2009), pp. 105–112.

[138] M. B. Kjærgaard, H. Blunck, T. Godsk, T. Toftkjær, D. L. Christensen, and K. Grønbæk. "Indoor Positioning Using GPS Revisited". In: *Pervasive Computing*. Springer, 2010, pp. 38–56.

[139] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. "Enabling consistency in view-based system development — The Vitruvius approach". In: *Journal of Systems and Software* 171 (1, 2021), p. 110815.

[140] G. Klir and B. Yuan. *Fuzzy sets and fuzzy logic*. Vol. 4. Prentice hall New Jersey, 1995.

[141] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[142] M. Konersmann, A. Kaplan, T. Kühn, R. Heinrich, A. Koziolek, R. Reussner, J. Jürjens, M. al-Doori, N. Boltz, M. Ehl, D. Fuchs, K. Groser, S. Hahner, J. Keim, M. Lohr, T. Sağlam, S. Schulz, and J.-P. Töberg. "Evaluation Methods and Replicability of Software Architecture Research Objects". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022 IEEE 19th International Conference on Software Architecture (ICSA). IEEE, 2022, pp. 157–168.

[143] A. Koziolek. "Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes". PhD thesis. 2011.

[144] A. Koziolek, H. Koziolek, and R. Reussner. "PerOpteryx: automated application of tactics in multi-objective software architecture optimization". In: *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. QoSA-ISARCS '11. Association for Computing Machinery, 20, 2011, pp. 33–42.

[145] H. Koziolek. "Performance evaluation of component-based software systems: A survey". In: *Performance Evaluation*. Special Issue on Software and Performance 67.8 (1, 2010), pp. 634–658.

[146] H. Koziolek. *Tracing the Practical Impact of Software Architecture Research.* Medium. 2022. URL: `https://medium.com/@heiko.koziolek/tracing-the-practical-impact-of-software-architecture-research-a2b91136455` (visited on 10/07/2024).

[147] M. E. Kramer, M. Hecker, S. Greiner, K. Bao, and K. Yurchenko. *Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems.* 2017.

[148] R. Kramer, R. Gupta, and M. Soffa. "The combining DAG: a technique for parallel data flow analysis". In: *IEEE Transactions on Parallel and Distributed Systems* 5.8 (1994). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 805–813.

[149] P. Kruchten. "The 4+1 View Model of architecture". In: *IEEE Software* 12.6 (1995). Conference Name: IEEE Software, pp. 42–50.

[150] P. Kruchten. "An Ontology of Architectural Design Decisions in Software-Intensive Systems". In: *Proceedings of the 2nd Groningen Workshop on Software Variability Management* (2004).

[151] J. Kunz. "Efficient Data Flow Constraint Analysis". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2018.

[152] M. Leinweber, N. Kannengießer, H. Hartenstein, and A. Sunyaev. "Leveraging Distributed Ledger Technology for Decentralized Mobility-as-a-Service Ticket Systems". In: *Towards the New Normal in Mobility: Technische und betriebswirtschaftliche Aspekte.* Springer Fachmedien, 2023, pp. 547–567.

[153] J. R. Lewis. "The system usability scale: past, present, and future". In: *International Journal of Human–Computer Interaction* 34.7 (2018). Publisher: Taylor & Francis, pp. 577–590.

[154] Linying Su, D. W. Chadwick, A. Basden, and J. A. Cunningham. "Automated decomposition of access control policies". In: *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05).* Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05). 2005, pp. 3–13.

[155] O. Liu. "Design Space Evaluation for Confidentiality under Architectural Uncertainty". Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[156] M. Lochau, S. Oster, U. Goltz, and A. Schürr. "Model-based pairwise testing for feature interaction coverage in software product line engineering". In: *Software Quality Journal* 20.3 (1, 2012), pp. 567–604.

[157] T. Lodderstedt, D. Basin, and J. Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security". In: *UML 2002 — The Unified Modeling Language.* Lecture Notes in Computer Science. Springer, 2002, pp. 426–441.

[158] C. Lupafya and D. Balasubramaniam. "A Framework for Considering Uncertainty in Software Systems". In: *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC).* 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC). 2022, pp. 1519–1524.

[159] I. Lytra and U. Zdun. “Supporting architectural decision making for systems-of-systems design under uncertainty”. In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. SESoS ’13. Association for Computing Machinery, 2, 2013, pp. 43–46.

[160] C. Maeder, K. Sohr, R. Wete Nguempnang, N. Meyer-Larsen, and R. Müller. “Modeling and Validating Role-Based Authorization Policies for a Port Communication System with UML and OCL.” In: *The Journal of Object Technology* 19.3 (2020), 3:1.

[161] P. N. Mahalle, P. A. Thakre, N. R. Prasad, and R. Prasad. “A fuzzy approach to trust based access control in internet of things”. In: *Wireless VITAE 2013*. Wireless VITAE 2013. 2013, pp. 1–5.

[162] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns. “A Classification Framework of Uncertainty in Architecture-Based Self- Adaptive Systems with Multiple Quality Requirements”. In: *Managing Trade-Offs in Adaptable Software Architectures* (2017), p. 33.

[163] A. Martens, H. Koziolek, S. Becker, and R. Reussner. “Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms”. In: *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. WOSP/SIPEW ’10. Association for Computing Machinery, 28, 2010, pp. 105–116.

[164] R. C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, 2017.

[165] C. Martínez-García, G. Navarro-Arribas, and J. Borrell. “Fuzzy Role-Based Access Control”. In: *Information Processing Letters* 111.10 (2011), pp. 483–487.

[166] F. Massacci and N. Zannone. “A Model-Driven Approach for the Specification and Analysis of Access Control Policies”. In: *On the Move to Meaningful Internet Systems: OTM 2008*. Lecture Notes in Computer Science. Springer, 2008, pp. 1087–1103.

[167] S. McConnell. *Software project survival guide*. Microsoft Press, 1998.

[168] P. Mehl and M. Walter. *Palladio Addon: Uncertainty-VariationCreation*. 2022. URL: `https://github.com/FluidTrust/Palladio-Addons-Uncertainty-VariationCreation` (visited on 11/28/2022).

[169] D. Mery and S. Merz. “Specification and Refinement of Access Control”. In: (2007), p. 21.

[170] Meta. *Introducing Meta Llama 3: The most capable openly available LLM to date*. Meta AI. 2024. URL: `https://ai.meta.com/blog/meta-llama-3/` (visited on 09/23/2024).

[171] S. Migliorini, R. Verdecchia, I. Malavolta, P. Lago, and E. Vicario. “Architectural Views: The State of Practice in Open-Source Software Projects”. In: *Software Architecture*. Springer Nature Switzerland, 2024, pp. 396–415.

[172] I. Molloy, L. Dickens, C. Morisset, P.-.-c. Cheng, J. Lobo, and A. Russo. “Risk-based access control decisions under uncertainty”. In: Computer Science. 1, 2012.

[173] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. “.QL: Object-Oriented Queries Made Easy”. In: *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers.* Lecture Notes in Computer Science. Springer, 2008, pp. 78–133.

[174] A. Morali, E. Zambon, S. Etalle, and P. Overbeek. “IT confidentiality risk assessment for an architecture-based approach”. In: *2008 3rd IEEE/IFIP International Workshop on Business-driven IT Management.* 2008 3rd IEEE/IFIP International Workshop on Business-driven IT Management. 2008, pp. 31–40.

[175] C. Morris. *Massive data leak exposes 700 million LinkedIn users’ information.* Fortune. 2021. URL: `https://fortune.com/2021/06/30/linkedin-data-theft-700-million-users-personal-information-cybersecurity/` (visited on 09/23/2024).

[176] J. von Neumann and O. Morgenstern. “Theory of Games and Economic Behavior”. In: *Theory of Games and Economic Behavior.* 1944.

[177] P. H. Nguyen, M. Kramer, J. Klein, and Y. L. Traon. “An extensive systematic review on the Model-Driven Development of secure systems”. In: *Information and Software Technology* 68 (1, 2015), pp. 62–81.

[178] Q. Ni, E. Bertino, and J. Lobo. “Risk-based access control systems built on fuzzy inferences”. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security.* ASIACCS ’10. Association for Computing Machinery, 2010, pp. 250–260.

[179] N. Niehues, B. Arp, T. Hüller, F. Schwickerath, N. Boltz, and S. Hahner. “Integrating Security-Enriched Data Flow Diagrams Into Architecture-Based Confidentiality Analysis”. In: Softwaretechnik-Trends Band 44, Heft 4. Gesellschaft für Informatik e.V., 2024.

[180] N. Niehues, S. Hahner, and R. Heinrich. “An Architecture-Based Approach to Mitigate Confidentiality Violations Using Machine Learning”. In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA).* 2025 IEEE 22nd International Conference on Software Architecture (ICSA). accepted, to appear. IEEE, 2025.

[181] Noppen, J.A.R., van den Broek, P.M., and Aksit, Mehmet. “Software development with imperfect information”. In: *Soft computing* 12.1 (2008). Publisher: Springer, pp. 3–28.

[182] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0.* 2013. URL: `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html` (visited on 12/22/2021).

[183] Object Management Group. *OMG Systems Modeling Language (SysML), version 2.0.* 2023.

[184] Object Management Group. *Precise Semantics for Uncertainty Modeling (PSUM), Version 1.0 Beta 2.* 2024.

[185] Object Management Group. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1.* 2011.

[186] Object Management Group. *Unified Modeling Language (UML) Specification. Version 2.5.* 2015.

[187] M. A. Olivero, A. Bertolino, F. J. Dominguez-Mayo, M. J. Escalona, and I. Matteucci. "Security Assessment of Systems of Systems". In: *2019 IEEE/ACM 7th International Workshop on Software Engineering for Systems-of-Systems (SESoS) and 13th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES).* 2019 IEEE/ACM 7th International Workshop on Software Engineering for Systems-of-Systems (SESoS) and 13th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES). 2019, pp. 62–65.

[188] OMG. *About the Meta Object Facility Specification Version 2.5.1.* 2016.

[189] A. Onabajo and J. Jahnke. "Properties of Confidentiality Requirements". In: *19th IEEE Symposium on Computer-Based Medical Systems (CBMS'06).* 19th IEEE Symposium on Computer-Based Medical Systems (CBMS'06). 2006, pp. 841–846.

[190] F. Oquendo. "Coping with Uncertainty in Systems-of-Systems Architecture Modeling on the IoT with SosADL". In: *2019 14th Annual Conference System of Systems Engineering (SoSE).* 2019 14th Annual Conference System of Systems Engineering (SoSE). 2019, pp. 131–136.

[191] S. Oster, M. Zink, M. Lochau, and M. Grechanik. "Pairwise feature-interaction testing for SPLs: potentials and limitations". In: *Proceedings of the 15th International Software Product Line Conference, Volume 2.* SPLC '11. Association for Computing Machinery, 21, 2011, pp. 1–8.

[192] OWASP Foundation. *OWASP Top 10:2021.* accessed 05/11/2022. 2021.

[193] S. Peldszus, K. Tuma, D. Struber, J. Jurjens, and R. Scandariato. "Secure Data-Flow Compliance Checks between Models and Code Based on Automated Mappings". In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE, 2019, pp. 23–33.

[194] D. Perez-Palacin and R. Mirandola. "Dealing with uncertainties in the performance modelling of software systems". In: *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures.* QoSA '14. Association for Computing Machinery, 2014, pp. 33–42.

[195] D. Perez-Palacin and R. Mirandola. "Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation". In: *Proceedings of the 5th ACM/SPEC international conference on Performance engineering.* ICPE '14. Association for Computing Machinery, 2014, pp. 3–14.

[196] R. Pilipchuk. "Architectural Alignment of Access Control Requirements Extracted from Business Processes". Dissertation. Karlsruhe Institute of Technology (KIT), 2021.

[197] R. Pilipchuk, S. Seifermann, and R. Heinrich. “Aligning Business Process Access Control Policies with Enterprise Architecture”. In: *Proceedings of the Central European Cybersecurity Conference 2018*. CECC 2018. Association for Computing Machinery, 15, 2018, pp. 1–4.

[198] D. M. W. Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *CoRR* abs/2010.16061 (2011).

[199] L. Prechelt, G. Malpohl, M. Philippsen, et al. “Finding plagiarisms among a set of programs with JPlag”. In: *Journal of Universal Computer Science* 8.11 (2002).

[200] D. Priss. “A Mobility Case Study Framework for Validating Uncertainty Impact Analyses regarding Confidentiality”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2022.

[201] J. Ramezani and L. M. Camarinha-Matos. “Approaches for resilience and antifragility in collaborative business ecosystems”. In: *Technological Forecasting and Social Change* 151 (2020), p. 119846.

[202] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng. “A taxonomy of uncertainty for dynamically adaptive systems”. In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). 2012, pp. 99–108.

[203] *The Common Component Modeling Example: Comparing Software Component Models*. Red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum. Vol. 5153. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008.

[204] F. Reiche, J. Schiffl, B. Beckert, R. Heinrich, and R. Reussner. *Modeling and Verifying Access Control for Ethereum Smart Contracts*. 2021.

[205] R. Reussner. *The Palladio Approach*. Modeling and Simulating Software Architectures. 2024. URL: `https://www.palladio-simulator.com/` (visited on 07/11/2024).

[206] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg. *The Palladio Component Model*. 2011. URL: `https://publikationen.bibliothek.kit.edu/1000022503` (visited on 07/01/2021).

[207] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016.

[208] C. J. van Rijsbergen. *Information Retrieval*. 2nd ed. Butterworths, 1979.

[209] R. C. Rønneberg. *Quantitative Information Flow Control by Construction for Component-Based Systems*. 15, 2024. arXiv: `2401.07677[cs]`.

[210] K. Rostami, R. Heinrich, A. Busch, and R. Reussner. “Architecture-Based Change Impact Analysis in Information Systems and Business Processes”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017 IEEE International Conference on Software Architecture (ICSA). 2017, pp. 179–188.

[211] K. Rostami, J. Stammel, R. Heinrich, and R. Reussner. “Architecture-based Assessment and Planning of Change Requests”. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures.* CompArch ’15: Federated Events on Component-Based Software Engineering and Software Architecture. ACM, 4, 2015, pp. 21–30.

[212] P. Runeson and M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical software engineering* 14.2 (2009), p. 131.

[213] T. Sağlam, M. Brödel, L. Schmid, and S. Hahner. “Detecting Automatic Software Plagiarism via Token Sequence Normalization”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* ICSE ’24. Association for Computing Machinery, 12, 2024, pp. 1–13.

[214] T. Sağlam, S. Hahner, D. Fuchß, and L. Schmid. *JPlag: State-of-the-Art Software Plagiarism & Collusion Detection.* 2024. URL: `https://github.com/jplag/jplag` (visited on 08/14/2024).

[215] T. Sağlam, S. Hahner, L. Schmid, and E. Burger. “Automated Detection of AI-Obfuscated Plagiarism in Modeling Assignments”. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training.* ICSE-SEET ’24. Association for Computing Machinery, 2024, pp. 297–308.

[216] T. Sağlam, S. Hahner, L. Schmid, and E. Burger. “Obfuscation-Resilient Software Plagiarism Detection with JPlag”. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings.* ICSE-Companion ’24. Association for Computing Machinery, 2024, pp. 264–265.

[217] T. Sağlam, S. Hahner, J. W. Wittler, and T. Kühn. “Token-based plagiarism detection for metamodels”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* 25th International Conference on Model Driven Engineering Languages and Systems (MODELS). MODELS ’22. ACM, 2022, pp. 138–141.

[218] T. Sağlam, N. Niehues, S. Hahner, and L. Schmid. “Mitigating Obfuscation Attacks on Software Plagiarism Detectors via Subsequence Merging”. In: IEEE Conference on Software Engineering Education and Training (CSEE&T). accepted, to appear. IEEE, 2025.

[219] T. Sağlam, L. Schmid, S. Hahner, and E. Burger. “How Students Plagiarize Modeling Assignments”. In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C).* 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). ACM, 2023, pp. 98–101.

[220] F. Salim, J. Reid, E. Dawson, and U. Dulleck. “An Approach to Access Control under Uncertainty”. In: *2011 Sixth International Conference on Availability, Reliability and Security.* 2011 Sixth International Conference on Availability, Reliability and Security. 2011, pp. 1–8.

[221] D. R. d. Santos, C. M. Westphall, and C. B. Westphall. "A dynamic risk-based access control architecture for cloud computing". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. 2014 IEEE Network Operations and Management Symposium (NOMS). 2014, pp. 1–9.

[222] SAP and D. Telekom. *Corona-Warn-App*. GitHub. 2023. URL: `https://github.com/corona-warn-app` (visited on 01/29/2023).

[223] M. A. Sasse and I. Flechais. *Usable Security: Why Do We Need It? How Do We Get It?* O'Reilly, 2005.

[224] P. Schaar. "Privacy by design". In: *Identity in the Information Society* 3.2 (2010). Publisher: Springer, pp. 267–274.

[225] L. Schmid, S. Hahner, and T. Sağlam. "JPlag: Detecting Obfuscated Software Plagiarism using Token Normalization Graphs". In: *Software Engineering 2025*. accepted, to appear. Gesellschaft für Informatik (GI), 2025.

[226] S. Schneider, N. E. D. Ferreyra, P.-J. Quéval, G. Simhandl, U. Zdun, and R. Scandariato. "How Dataflow Diagrams Impact Software Security Analysis: an Empirical Experiment". In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE Computer Society, 1, 2024, pp. 952–963.

[227] S. Schneider, T. Özen, M. Chen, and R. Scandariato. "microSecEnD: A Dataset of Security-Enriched Dataflow Diagrams for Microservice Applications". In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR). 2023, pp. 125–129.

[228] S. Schneider and R. Scandariato. "Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java". In: *Journal of Systems and Software* 202 (2023), p. 111722.

[229] S. Schulz, F. Reiche, S. Hahner, and J. Schiffl. "Continuous Secure Software Development and Analysis". In: *Proceedings of Symposium on Software Performance 2021*. Symposium on Software Performance. CEUR Workshop Proceedings, 2021, pp. 1–6.

[230] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[231] F. Schwickerath, N. Boltz, S. Hahner, M. Walter, C. Gerking, and R. Heinrich. *Tool-Supported Architecture-Based Data Flow Analysis for Confidentiality*. 2023. arXiv: `2308.01645[cs]`.

[232] S. Seifermann. "Architectural Data Flow Analysis". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE, 2016, pp. 270–271.

[233] S. Seifermann. "Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements". Dissertation. Karlsruhe Institute of Technology (KIT), 2022.

[234] S. Seifermann, R. Heinrich, and R. Reussner. “Data-Driven Software Architecture for Analyzing Confidentiality”. In: *2019 IEEE International Conference on Software Architecture (ICSA).* IEEE, 2019, pp. 1–10.

[235] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner. “A Unified Model to Detect Information Flow and Access Control Violations in Software Architectures”. In: *Proceedings of the 18th International Conference on Security and Cryptography.* 18th International Conference on Security and Cryptography. SCITEPRESS - Science and Technology Publications, 2021, pp. 26–37.

[236] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner. “Detecting violations of access control and information flow policies in data flow diagrams”. In: *Journal of Systems and Software* 184 (1, 2022), p. 111138.

[237] S. Seifermann, M. Walter, S. Hahner, R. Heinrich, and R. Reussner. “Identifying Confidentiality Violations in Architectural Design Using Palladio”. In: *Companion Proceedings of the 15th European Conference on Software Architecture (ECSA-C).* 15th European Conference on Software Architecture (ECSA 2021). CEUR Workshop Proceedings, 2021, pp. 1–4.

[238] M. Shaw. “What makes good research in software engineering?” In: *International Journal on Software Tools for Technology Transfer* 4.1 (1, 2002), pp. 1–7.

[239] A. Shostack. *Threat Modeling: Designing for Security.* John Wiley & Sons, 12, 2014. 624 pp.

[240] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. “What we have learned about fighting defects”. In: *METRICS.* 2002, pp. 249–258.

[241] L. Sion, K. Yskout, D. Van Landuyt, A. van den Berghe, and W. Joosen. “Security Threat Modeling: Are Data Flow Diagrams Enough?” In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops.* ICSEW’20. Association for Computing Machinery, 27, 2020, pp. 254–257.

[242] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab. “Checking probabilistic noninterference using JOANA”. In: *it - Information Technology* 56.6 (2014), pp. 280–287.

[243] D. Sobhy, R. Bahsoon, L. Minku, and R. Kazman. “Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review”. In: *ACM Transactions on Software Engineering and Methodology* (2021), p. 50.

[244] R. J. Sofaer, Y. David, M. Kang, J. Yu, Y. Cao, J. Yang, and J. Nieh. “RogueOne: Detecting Rogue Updates via Differential Data-flow Analysis Using Trust Domains”. In: *ICSE ’24.* ICSE ’24. 2024.

[245] H. Stachowiak. *Allgemeine Modelltheorie.* Springer, 1973.

[246] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, Inc., 2006.

[247] G. Starke. *arc42 Quality Model.* 2024.

[248] Statista. *Number of internet users worldwide 2005-2023*. Statista. 2004. URL: `https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/` (visited on 09/23/2024).

[249] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[250] D. Stengel. "Verfeinerung von Zugriffskontrollrichtlinien unter Berücksichtigung von Ungewissheit in der Entwurfszeit". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2021.

[251] L. Sterz, C. Werner, and O. Raabe. "Intelligente Verkehrssysteme – IT-Sicherheit in offenen Infrastrukturen". In: *Recht der Datenverarbeitung* 6 (2022).

[252] M. Sulochana and O. Dubey. "Preserving Data Confidentiality Using Multi-cloud Architecture". In: *Procedia Computer Science*. Big Data, Cloud and Computing Challenges 50 (1, 2015), pp. 357–362.

[253] B. Taghavi and S. Weber. "A Survey of Analysis Composition Operators in the Context of Palladio". In: Softwaretechnik-Trends Band 43, Heft 4. Gesellschaft für Informatik e.V., 2023.

[254] A. Tchernykh, U. Schwiegelsohn, E.-g. Talbi, and M. Babenko. "Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability". In: *Journal of Computational Science* 36 (1, 2019), p. 100581.

[255] J. Troya, N. Moreno, M. F. Bertoa, and A. Vallecillo. "Uncertainty representation in software models: a survey". In: *Software and Systems Modeling* 20.4 (1, 2021), pp. 1183–1213.

[256] K. Tuma, R. Scandariato, and M. Balliu. "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019 IEEE International Conference on Software Architecture (ICSA). IEEE, 2019, pp. 191–200.

[257] K. Tuma, L. Sion, R. Scandariato, and K. Yskout. "Automating the early detection of security design flaws". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. Association for Computing Machinery, 2020, pp. 332–342.

[258] N. Ubayashi, Y. Kamei, and R. Sato. "When and Why Do Software Developers Face Uncertainty?" In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). 2019, pp. 288–299.

[259] UpStream. *2022 Global Automotive Cybersecurity Report*. 2022.

[260] UpStream. *2024 Global Automotive Cybersecurity Report*. 2024.

[261] K. Vanherpen, J. Denil, P. D. Meulenaere, and H. Vangheluwe. "Design-Space Exploration in Model Driven Engineering". In: *Proceedings of the First International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA 2014)* (2014).

[262] P. Villalobos, A. Ho, J. Sevilla, T. Besiroglu, L. Heim, and M. Hobbhahn. “Will we run out of data? Limits of LLM scaling based on human-generated data”. In: *Proceedings of the 41st International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 2640-3498. PMLR, 8, 2024, pp. 49523–49544.

[263] W. E. Walker, P. Harremoës, J. Rotmans, J. P. Van Der Sluijs, M. B. Van Asselt, P. Janssen, and M. P. Krayer von Krauss. “Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support”. In: *Integrated assessment* 4.1 (2003). Publisher: Taylor & Francis, pp. 5–17.

[264] M. Walter. “Context-based Access Control and Attack Modelling and Analysis”. Dissertation. 2023.

[265] M. Walter, S. Hahner, T. Bures, P. Hnetynka, R. Heinrich, and R. Reussner. “Architecture-based attack propagation and variation analysis for identifying confidentiality issues in Industry 4.0”. In: *at - Automatisierungstechnik* 71.6 (2023), pp. 443–452.

[266] M. Walter, S. Hahner, S. Seifermann, T. Bures, P. Hnetynka, J. Pacovsky, and R. Heinrich. “Architectural Optimization for Confidentiality Under Structural Uncertainty”. In: *Software Architecture.* Lecture Notes in Computer Science. Springer International Publishing, 2022, pp. 309–332.

[267] M. Walter, R. Heinrich, and R. Reussner. “Architectural Attack Propagation Analysis for Identifying Confidentiality Issues”. In: *2022 IEEE 19th International Conference on Software Architecture (ICSA).* 19th International Conference on Software Architecture. Institute of Electrical and Electronics Engineers (IEEE), 2022, 12 S.

[268] M. Walter, R. Heinrich, and R. Reussner. “Architecture-Based Attack Path Analysis for Identifying Potential Security Incidents”. In: *Software Architecture.* Lecture Notes in Computer Science. Springer Nature Switzerland, 2023, pp. 37–53.

[269] M. Walter and R. Reussner. “Tool-Based Attack Graph Estimation and Scenario Analysis for Software Architectures”. In: *Software Architecture. ECSA 2022 Tracks and Workshops.* Springer International Publishing, 2023, pp. 45–61.

[270] B. P. Weimann. “Automated Cloud-to-Cloud Migration of Distributed So ware Systems for Privacy Compliance”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2017.

[271] H. Weisbaum. *Trust in Facebook has dropped by 66 percent since the Cambridge Analytica scandal.* 2018. URL: `https://www.nbcnews.com/business/consumer/trust-facebook-has-dropped-51-percent-cambridge-analytica-scandal-n867011` (visited on 01/19/2023).

[272] D. Weyns. *An introduction to self-adaptive systems: A contemporary software engineering perspective.* John Wiley & Sons, 2020.

[273] D. Weyns, R. Calinescu, R. Mirandola, K. Tei, M. Acosta, A. Bennaceur, N. Boltz, T. Bures, J. Camara, A. Diaconescu, G. Engels, S. Gerasimou, I. Gerostathopoulos, S. Getir Yaman, V. Grassi, S. Hahner, E. Letier, M. Litoiu, L. Marsso, A. Musil, J. Musil, G. Nunes Rodrigues, D. Perez-Palacin, F. Quin, P. Scandurra, A. Vallecillo, and A. Zisman. “Towards a Research Agenda for Understanding and Managing Uncertainty in Self-Adaptive Systems”. In: *ACM SIGSOFT Software Engineering Notes* 48.4 (2023), pp. 20–36.

[274] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. “On Patterns for Decentralized Control in Self-Adaptive Systems”. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Springer, 2013, pp. 76–107.

[275] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. Bruel. “RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems”. In: *2009 17th IEEE International Requirements Engineering Conference*. 2009 17th IEEE International Requirements Engineering Conference. 2009, pp. 79–88.

[276] C. Wohlin. “Case Study Research in Software Engineering—It is a Case, and it is a Study, but is it a Case Study?” In: *Information and Software Technology* 133 (1, 2021), p. 106514.

[277] X. Yang and J. Alves-Foss. “Security Policy Refinement: High-Level Specification to Low-Level Implementation”. In: *2013 International Conference on Social Computing*. 2013 International Conference on Social Computing. 2013, pp. 502–511.

[278] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren. “Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model”. In: *Modelling Foundations and Applications*. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 247–264.

[279] H. Zhao, J. Lobo, A. Roy, and S. M. Bellovin. “Policy refinement of network services for MANETs”. In: *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*. 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops. 2011, pp. 113–120.

[280] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. “Reusable architectural decision models for enterprise application development”. In: *Software Architectures, Components, and Applications: Third International Conference on Quality of Software Architectures, QoSA 2007, Medford, MA, USA, July 11-23, 2007, Revised Selected Papers 3*. Springer, 2007, pp. 15–32.

Part V.

Appendix

A. Running Example in the Palladio Component Model

Our running example is shown in Chapter 3. To illustrate the architectural model of the running example using the Palladio Component Model (PCM), we show two diagrams. First, Figure A.1 shows a graphical representation of the Palladio *component repository model*, comprising the components and their interfaces. Second, Figure A.2 shows the Service Effect Specification (SEFF) of *purchaseItem*, comprising the default purchasing behavior and alternatives due to uncertainty, e.g., the processing of purchase data with erroneous encryption, or lacking validation. The behavior is represented using assignments, e.g., by assigning the *Validated* and *Encrypted* labels in the upper left, as defined by Seifermann et al. [237].

Both diagrams are exported from the Palladio Bench [205], which represents the foundation of our tool support. The diagrams can be created and altered using Eclipse's inline diagram editors. Furthermore, Figure A.3 shows an excerpt from our tool support to create uncertainty models, following the meta model presented in Figure 7.11. Here, the uncertainty scenarios of the *Behavior* uncertainty **U2** reference the alternative actions shown in Figure A.2. Using these models and previously specified data flow constraints [36, 105] as an input, 🔧 Abunai can identify confidentiality violations due to uncertainty, see Subsection 7.5.2. The remaining PCM models and the required tooling to display them can be found in the data set [98].

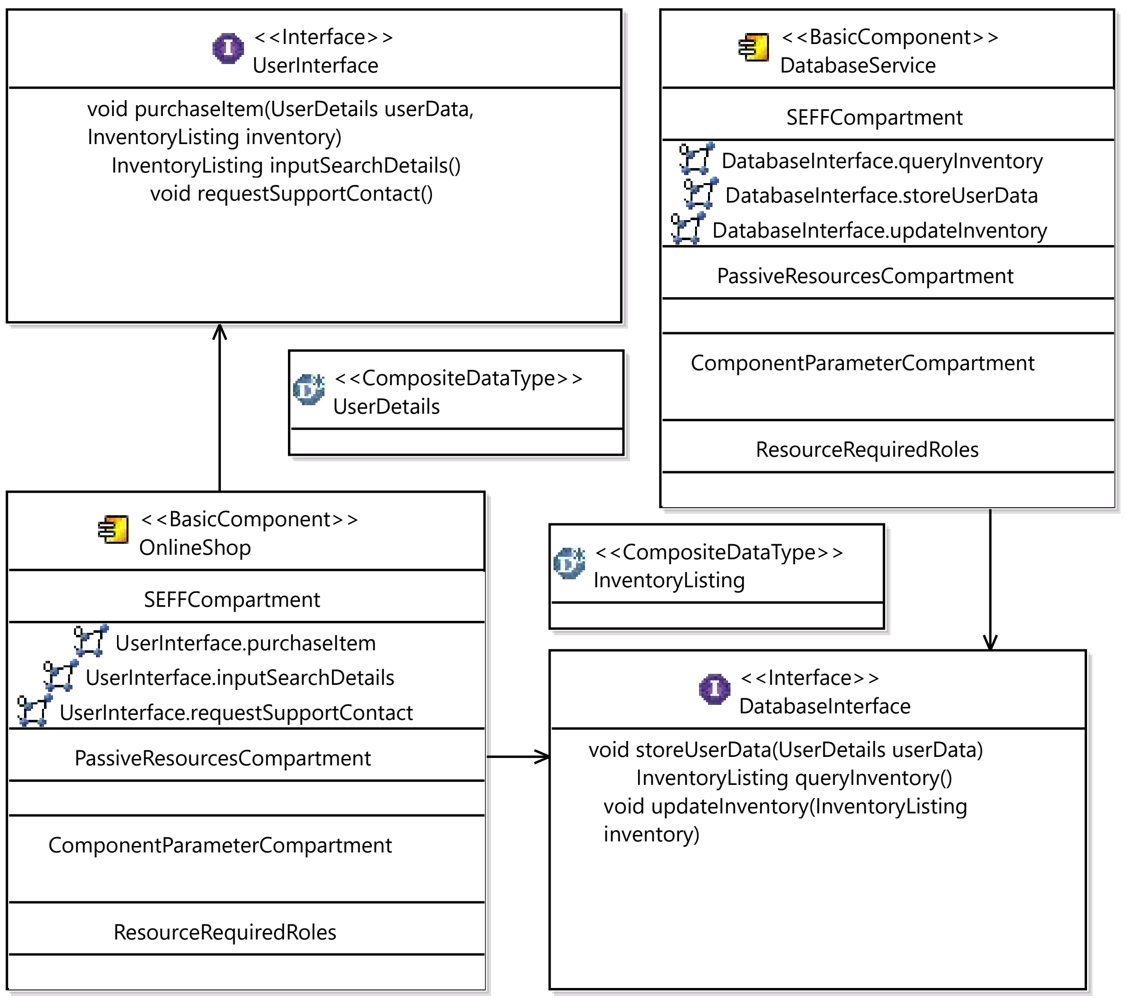

<<Interface>>
UserInterface
void purchaseItem(UserDetails userData, InventoryListing inventory)
InventoryListing inputSearchDetails()
void requestSupportContact()
<<BasicComponent>>
DatabaseService
SEFFCompartment
DatabaseInterface.queryInventory
DatabaseInterface.storeUserData
DatabaseInterface.updateInventory
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<CompositeDataType>>
UserDetails
<<BasicComponent>>
OnlineShop
SEFFCompartment
UserInterface.purchaseItem
UserInterface.inputSearchDetails
UserInterface.requestSupportContact
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<CompositeDataType>>
InventoryListing
<<Interface>>
DatabaseInterface
void storeUserData(UserDetails userData)
InventoryListing queryInventory()
void updateInventory(InventoryListing inventory)

Figure A.1.: Palladio repository model of the running example, exported from the Palladio-Bench [205].

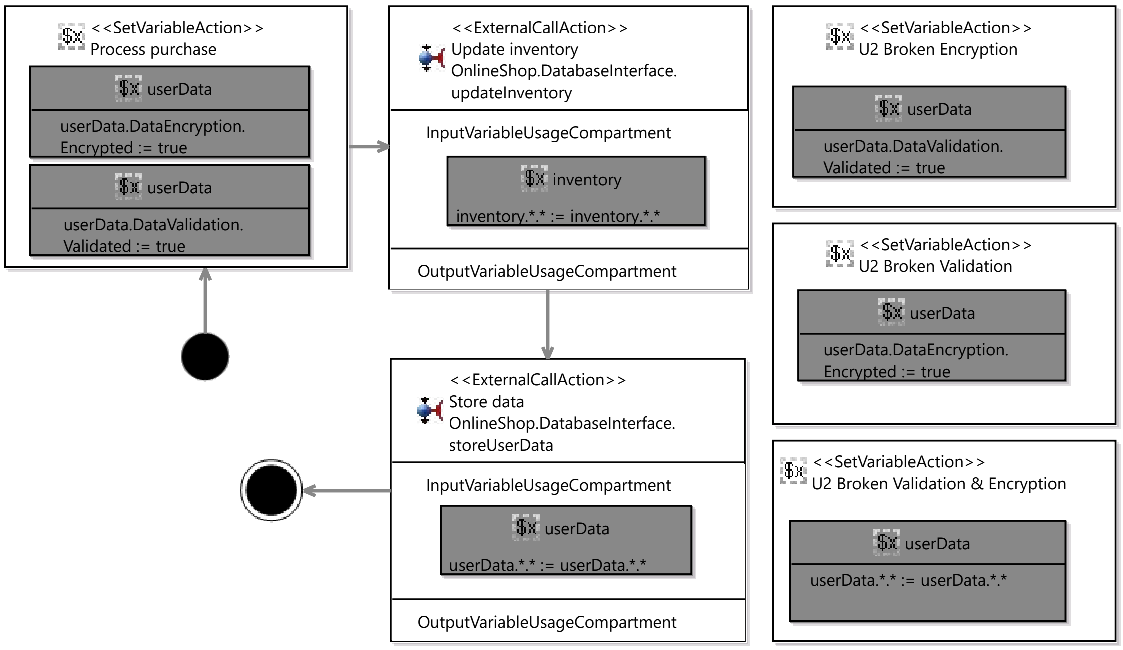
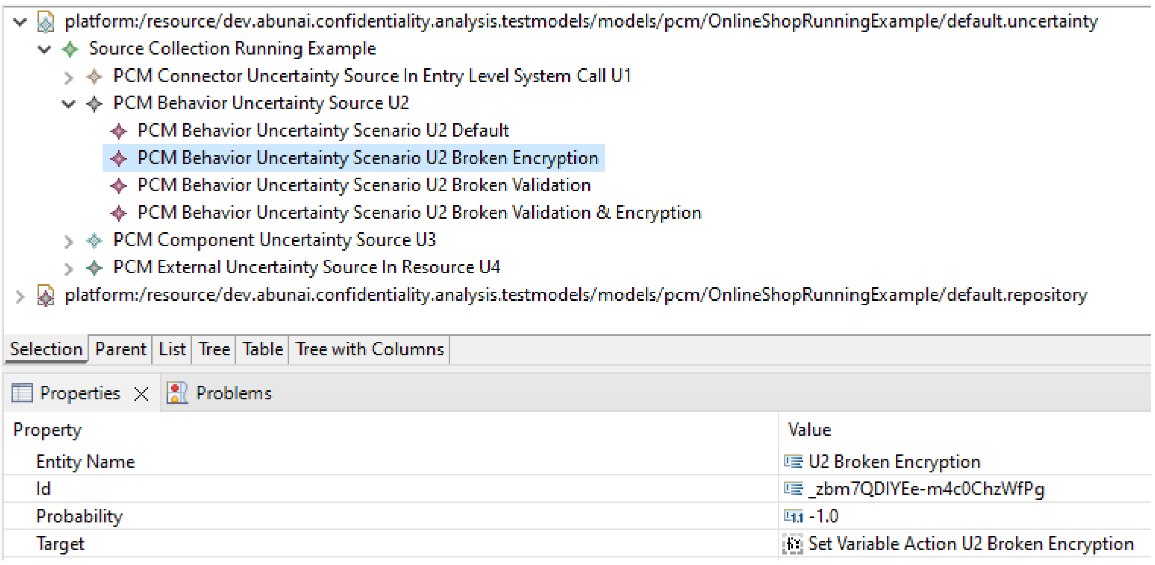


Figure A.2.: Exemplary Palladio Service Effect Specification (SEFF) of the running example, exported from the Palladio-Bench [205].


platform:/resource/dev.abunai.confidentiality.analysis.testmodels/models/pcm/OnlineShopRunningExample/default.uncertainty
- Source Collection Running Example
  - PCM Connector Uncertainty Source In Entry Level System Call U1
  - PCM Behavior Uncertainty Source U2
    - PCM Behavior Uncertainty Scenario U2 Default
    - PCM Behavior Uncertainty Scenario U2 Broken Encryption
    - PCM Behavior Uncertainty Scenario U2 Broken Validation
    - PCM Behavior Uncertainty Scenario U2 Broken Validation & Encryption
  - PCM Component Uncertainty Source U3
  - PCM External Uncertainty Source In Resource U4

platform:/resource/dev.abunai.confidentiality.analysis.testmodels/models/pcm/OnlineShopRunningExample/default.repository

Selection | Parent | List | Tree | Table | Tree with Columns
Properties
Problems

Property	Value
Entity Name	U2 Broken Encryption
Id	_zbm7QDIYEe-m4c0ChzWfPg
Probability	-1.0
Target	Set Variable Action U2 Broken Encryption

Figure A.3.: Uncertainty model of the running example, showing four uncertainty sources and their scenarios. Screenshot from the Palladio-Bench [205].

B. Impact Sets of the Running Example

Our uncertainty impact analysis has been introduced in Chapter 6. We show the resulting impact sets after the architectural propagation and the propagation in the Data Flow Diagram (DFD) of the running example, see Chapter 3. These are the results of the automated uncertainty impact analysis 🔧 UIA. Listing B.1 shows the impact set of Uncertainty **U1**, Listing B.2 shows the impact set of Uncertainty **U2**, Listing B.3 shows the impact set of Uncertainty **U3**, and Listing B.4 shows the impact set of Uncertainty **U4**. We removed the element IDs from the analysis output for the sake of brevity. The comprehensive analysis output can be found in the data set [98].

```
Result of the PCM propagation:
SEFFActionSequenceElement (Beginning view)
SEFFActionSequenceElement (Beginning buy)
SEFFActionSequenceElement (Beginning requestSupport)
SEFFActionSequenceElement (Beginning view)
CallingUserActionSequenceElement / calling (ViewEntryLevelSystemCall)
CallingUserActionSequenceElement / returning (ViewEntryLevelSystemCall)
CallingUserActionSequenceElement / calling (BuyEntryLevelSystemCall)
CallingUserActionSequenceElement / returning (BuyEntryLevelSystemCall)
CallingUserActionSequenceElement / calling (RequestSupportCall)
CallingUserActionSequenceElement / returning (RequestSupportCall)
CallingUserActionSequenceElement / calling (ViewShopCall)
CallingUserActionSequenceElement / returning (ViewShopCall)
Result of the DFD propagation:
CallingUserActionSequenceElement / calling (ViewShopCall)
SEFFActionSequenceElement (Beginning view)
CallingSEFFActionSequenceElement / calling (DatabaseLoadInventory)
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
CallingSEFFActionSequenceElement / returning (DatabaseLoadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending view)
CallingUserActionSequenceElement / returning (ViewShopCall)
UserActionSequenceElement (Stopping usage: View Shop)
CallingUserActionSequenceElement / calling (ViewEntryLevelSystemCall)
SEFFActionSequenceElement (Beginning view)
CallingSEFFActionSequenceElement / calling (DatabaseLoadInventory)
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
CallingSEFFActionSequenceElement / returning (DatabaseLoadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending view)
CallingUserActionSequenceElement / returning (ViewEntryLevelSystemCall)
CallingUserActionSequenceElement / calling (BuyEntryLevelSystemCall)
SEFFActionSequenceElement (Beginning buy)
SEFFActionSequenceElement (UserDataProcessing)
CallingSEFFActionSequenceElement / calling (DatabaseStoreInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Ending updateInventory)
CallingSEFFActionSequenceElement / returning (DatabaseStoreInventory)
CallingSEFFActionSequenceElement / calling (DatabaseStoreUserData)
SEFFActionSequenceElement (Beginning storeUserData)
SEFFActionSequenceElement (Ending storeUserData)
CallingSEFFActionSequenceElement / returning (DatabaseStoreUserData)
SEFFActionSequenceElement (Ending buy)
CallingUserActionSequenceElement / returning (BuyEntryLevelSystemCall)
UserActionSequenceElement (Stopping usage: Buy from Shop)
CallingUserActionSequenceElement / calling (RequestSupportCall)
SEFFActionSequenceElement (Beginning requestSupport)
SEFFActionSequenceElement (Ending requestSupport)
CallingUserActionSequenceElement / returning (RequestSupportCall)
UserActionSequenceElement (Stopping usage: Request Support)
```

Listing B.1: Result of the uncertainty impact analysis of Uncertainty **U1** in the running example.

```
Result of the PCM propagation:
SEFFActionSequenceElement (UserDataProcessing)

Result of the DFD propagation:
SEFFActionSequenceElement (UserDataProcessing)
CallingSEFFActionSequenceElement / calling (DatabaseStoreInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Ending updateInventory)
CallingSEFFActionSequenceElement / returning (DatabaseStoreInventory)
CallingSEFFActionSequenceElement / calling (DatabaseStoreUserData)
SEFFActionSequenceElement (Beginning storeUserData)
SEFFActionSequenceElement (Ending storeUserData)
CallingSEFFActionSequenceElement / returning (DatabaseStoreUserData)
SEFFActionSequenceElement (Ending buy)
CallingUserActionSequenceElement / returning (BuyEntryLevelSystemCall)
UserActionSequenceElement (Stopping usage: Buy from Shop)
```

Listing B.2: Result of the uncertainty impact analysis of Uncertainty **U2** in the running example.

```
Result of the PCM propagation:
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Beginning storeUserData)
SEFFActionSequenceElement (Beginning loadInventory)

Result of the DFD propagation:
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
CallingSEFFActionSequenceElement / returning (DatabaseLoadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending view)
CallingUserActionSequenceElement / returning (ViewShopCall)
UserActionSequenceElement (Stopping usage: View Shop)
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
CallingSEFFActionSequenceElement / returning (DatabaseLoadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending view)
CallingUserActionSequenceElement / returning (ViewEntryLevelSystemCall)
CallingUserActionSequenceElement / calling (BuyEntryLevelSystemCall)
SEFFActionSequenceElement (Beginning buy)
SEFFActionSequenceElement (UserDataProcessing)
CallingSEFFActionSequenceElement / calling (DatabaseStoreInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Ending updateInventory)
CallingSEFFActionSequenceElement / returning (DatabaseStoreInventory)
CallingSEFFActionSequenceElement / calling (DatabaseStoreUserData)
SEFFActionSequenceElement (Beginning storeUserData)
SEFFActionSequenceElement (Ending storeUserData)
CallingSEFFActionSequenceElement / returning (DatabaseStoreUserData)
SEFFActionSequenceElement (Ending buy)
CallingUserActionSequenceElement / returning (BuyEntryLevelSystemCall)
UserActionSequenceElement (Stopping usage: Buy from Shop)
```

Listing B.3: Result of the uncertainty impact analysis of Uncertainty **U3** in the running example.

```
Result of the PCM propagation:
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Ending updateInventory)
SEFFActionSequenceElement (Beginning storeUserData)
SEFFActionSequenceElement (Ending storeUserData)
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)

Result of the DFD propagation:
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
CallingSEFFActionSequenceElement / returning (DatabaseLoadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending view)
CallingUserActionSequenceElement / returning (ViewEntryLevelSystemCall)
CallingUserActionSequenceElement / calling (BuyEntryLevelSystemCall)
SEFFActionSequenceElement (Beginning buy)
SEFFActionSequenceElement (UserDataProcessing)
CallingSEFFActionSequenceElement / calling (DatabaseStoreInventory)
SEFFActionSequenceElement (Beginning updateInventory)
SEFFActionSequenceElement (Ending updateInventory)
CallingSEFFActionSequenceElement / returning (DatabaseStoreInventory)
CallingSEFFActionSequenceElement / calling (DatabaseStoreUserData)
SEFFActionSequenceElement (Beginning storeUserData)
SEFFActionSequenceElement (Ending storeUserData)
CallingSEFFActionSequenceElement / returning (DatabaseStoreUserData)
SEFFActionSequenceElement (Ending buy)
CallingUserActionSequenceElement / returning (BuyEntryLevelSystemCall)
UserActionSequenceElement (Stopping usage: Buy from Shop)
SEFFActionSequenceElement (Beginning loadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending loadInventory)
CallingSEFFActionSequenceElement / returning (DatabaseLoadInventory)
SEFFActionSequenceElement (RETURN)
SEFFActionSequenceElement (Ending view)
CallingUserActionSequenceElement / returning (ViewShopCall)
UserActionSequenceElement (Stopping usage: View Shop)
```

Listing B.4: Result of the uncertainty impact analysis of Uncertainty **U4** in the running example.

C. All Confidentiality Violations in the Running Example

We give an overview of all confidentiality violations identified in the running example by Abunai. First, Listing C.1 shows all combinations of uncertainty scenarios that cause the violation of at least one confidentiality requirement. Second, Listing C.2 exemplifies the result of the first scenario combination by showing all confidentiality-violating vertices. The full analysis results can be found in the data set [98].

```
[U1 Invalid, U2-Default, U3 Alternative]
[U1 Invalid, U2 Broken Encryption, U3 Alternative]
[U1 Invalid, U2 Broken Validation, U3 Alternative]
[U1 Invalid, U2 Broken Validation & Encryption, U3 Alternative]
[U1 Invalid, U2-Default, U3-Default, U4-Default]
[U1 Invalid, U2-Default, U3-Default, U4 Alternative]
[U1 Invalid, U2 Broken Encryption, U3-Default, U4-Default]
[U1 Invalid, U2 Broken Encryption, U3-Default, U4 Alternative]
[U1 Invalid, U2 Broken Validation, U3-Default, U4-Default]
[U1 Invalid, U2 Broken Validation, U3-Default, U4 Alternative]
[U1 Invalid, U2 Broken Validation & Encryption, U3-Default, U4-Default]
[U1 Invalid, U2 Broken Validation & Encryption, U3-Default, U4 Alternative]
[U1-Default, U2 Broken Encryption, U3-Default, U4-Default]
[U1-Default, U2 Broken Encryption, U3-Default, U4 Alternative]
[U1-Default, U2 Broken Validation & Encryption, U3-Default, U4-Default]
[U1-Default, U2 Broken Validation & Encryption, U3-Default, U4 Alternative]
[U1 Invalid, U2 Broken Encryption, U3-Default, U4-Default]
[U1 Invalid, U2 Broken Encryption, U3-Default, U4 Alternative]
[U1 Invalid, U2 Broken Validation & Encryption, U3-Default, U4-Default]
[U1 Invalid, U2 Broken Validation & Encryption, U3-Default, U4 Alternative]
[U1 Malicious, U2 Broken Encryption, U3-Default, U4-Default]
[U1 Malicious, U2 Broken Encryption, U3-Default, U4 Alternative]
[U1 Malicious, U2 Broken Validation & Encryption, U3-Default, U4-Default]
[U1 Malicious, U2 Broken Validation & Encryption, U3-Default, U4 Alternative]
```

Listing C.1: All uncertainty scenarios of the running example that cause confidentiality violations.

```
[U1 Invalid, U2-Default, U3 Alternative] ->
{SEFFPCMVertex (Beginning purchaseItem,
_nGp9cITjEeywmO_IpTxeAg), SEFFPCMVertex (Process purchase, _oEBNYDIXEe-m4c0ChzWfPg),
CallingSEFFPCMVertex / calling (Update inventory, _yVL18ITkEeywmO_IpTxeAg),
CallingSEFFPCMVertex / returning (Update inventory, _yVL18ITkEeywmO_IpTxeAg),
CallingSEFFPCMVertex / calling (Store data, _11NAEITkEeywmO_IpTxeAg), SEFFPCMVertex
(Beginning storeUserData, _oGmXgYTjEeywmO_IpTxeAg), SEFFPCMVertex (Ending
storeUserData, _oGmXgoTjEeywmO_IpTxeAg), CallingSEFFPCMVertex / returning (Store
data, _11NAEITkEeywmO_IpTxeAg), SEFFPCMVertex (Ending purchaseItem,
_nGp9cYTjEeywmO_IpTxeAg), CallingUserPCMVertex / returning (U1 Purchase item
Invalid, _UQNBoC1WEe-lUc5YrocPyg), UserPCMVertex (Stopping usage: Buy from Shop,
_LPwS4iHdEd6lJo4DCALHMw)},
```

Listing C.2: Detailed result of the first confidentiality violation of the running example.

D. Palladio Repository Model of the Corona Warn App

In the following, we show the full Palladio *component repository model* of the *Corona Warn App* evaluation scenario, see Section 8.5. Due to the large size of this diagram, we split it into four parts, shown in Figure D.2, Figure D.3, Figure D.4, and Figure D.5. Figure D.1 gives an overview of the structure. The original full-sized diagram and all PCM model files can be found in the data set [98].

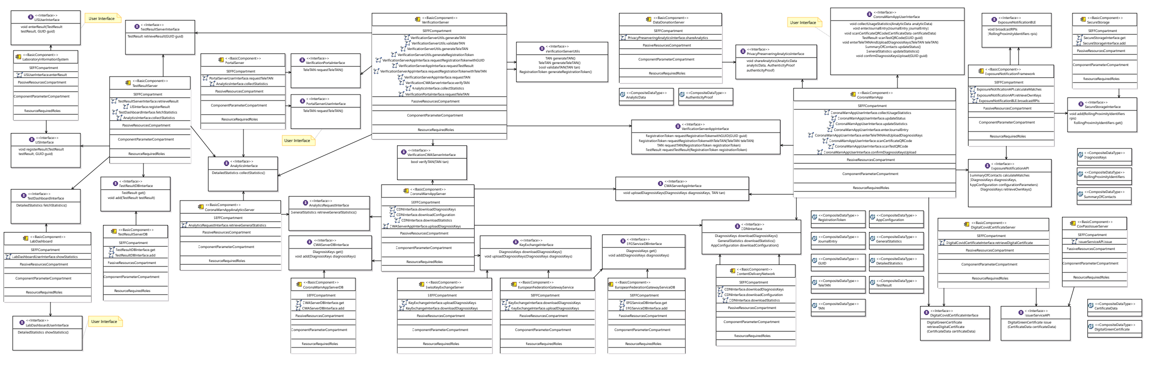


Figure D.1.: Palladio component repository model of the Corona Warn App evaluation scenario (overview), exported from the Palladio-Bench [205].

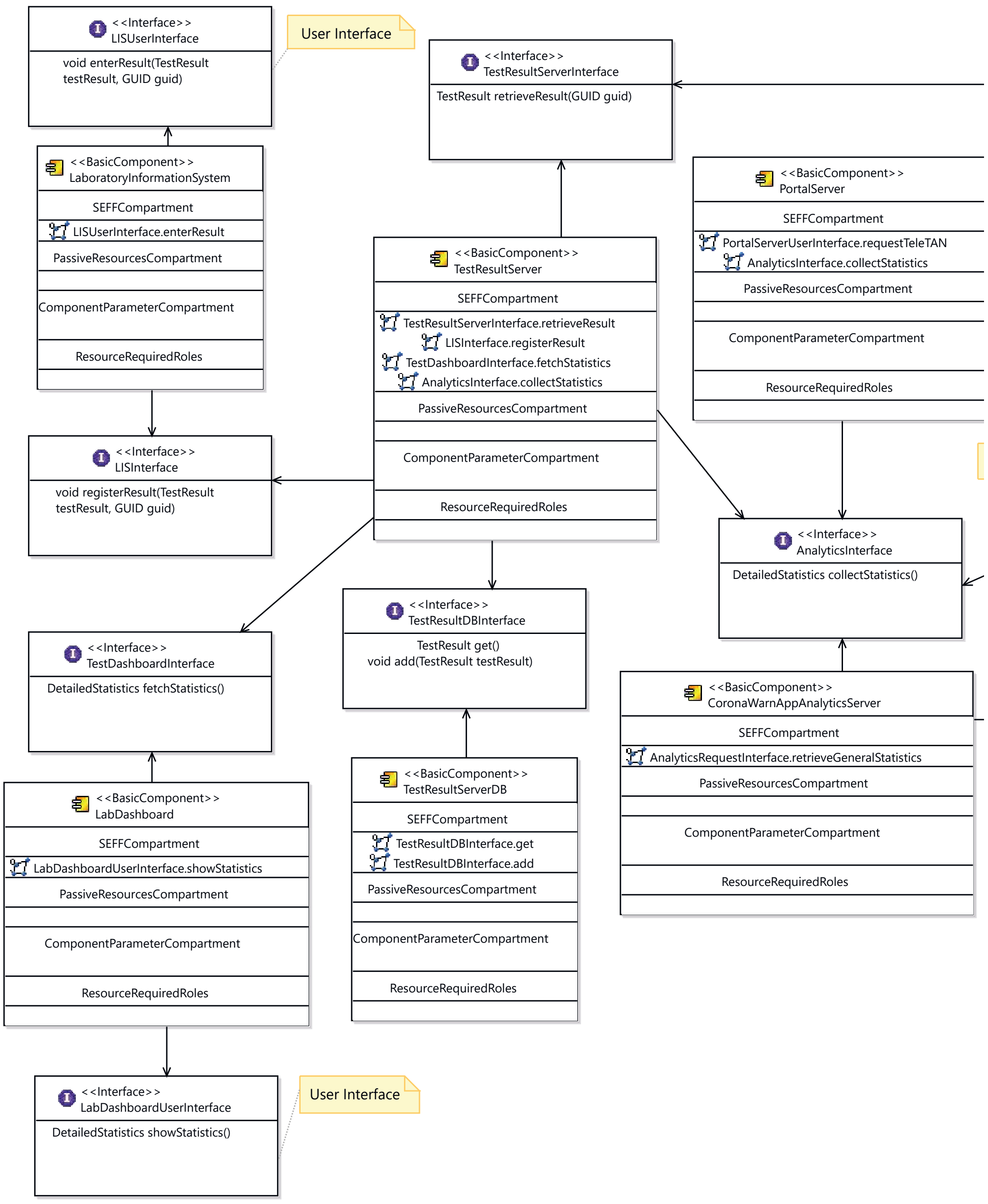

<<Interface>> LISUserInterface
void enterResult(TestResult testResult, GUID guid)
User Interface
<<Interface>> TestResultServerInterface
TestResult retrieveResult(GUID guid)
<<BasicComponent>> LaboratoryInformationSystem
SEFFCompartment
LISUserInterface.enterResult
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>> PortalServer
SEFFCompartment
PortalServerUserInterface.requestTeleTAN
AnalyticsInterface.collectStatistics
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>> TestResultServer
SEFFCompartment
TestResultServerInterface.retrieveResult
LISInterface.registerResult
TestDashboardInterface.fetchStatistics
AnalyticsInterface.collectStatistics
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<Interface>> LISInterface
void registerResult(TestResult testResult, GUID guid)
<<Interface>> AnalyticsInterface
DetailedStatistics collectStatistics()
<<Interface>> TestResultDBInterface
TestResult get()
void add(TestResult testResult)
<<Interface>> TestDashboardInterface
DetailedStatistics fetchStatistics()
<<BasicComponent>> CoronaWarnAppAnalyticsServer
SEFFCompartment
AnalyticsRequestInterface.retrieveGeneralStatistics
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>> LabDashboard
SEFFCompartment
LabDashboardUserInterface.showStatistics
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>> TestResultServerDB
SEFFCompartment
TestResultDBInterface.get
TestResultDBInterface.add
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<Interface>> LabDashboardUserInterface
DetailedStatistics showStatistics()
User Interface

Figure D.2.: Palladio component repository model of the Corona Warn App evaluation scenario (part 1 of 4), exported from the Palladio-Bench [205].

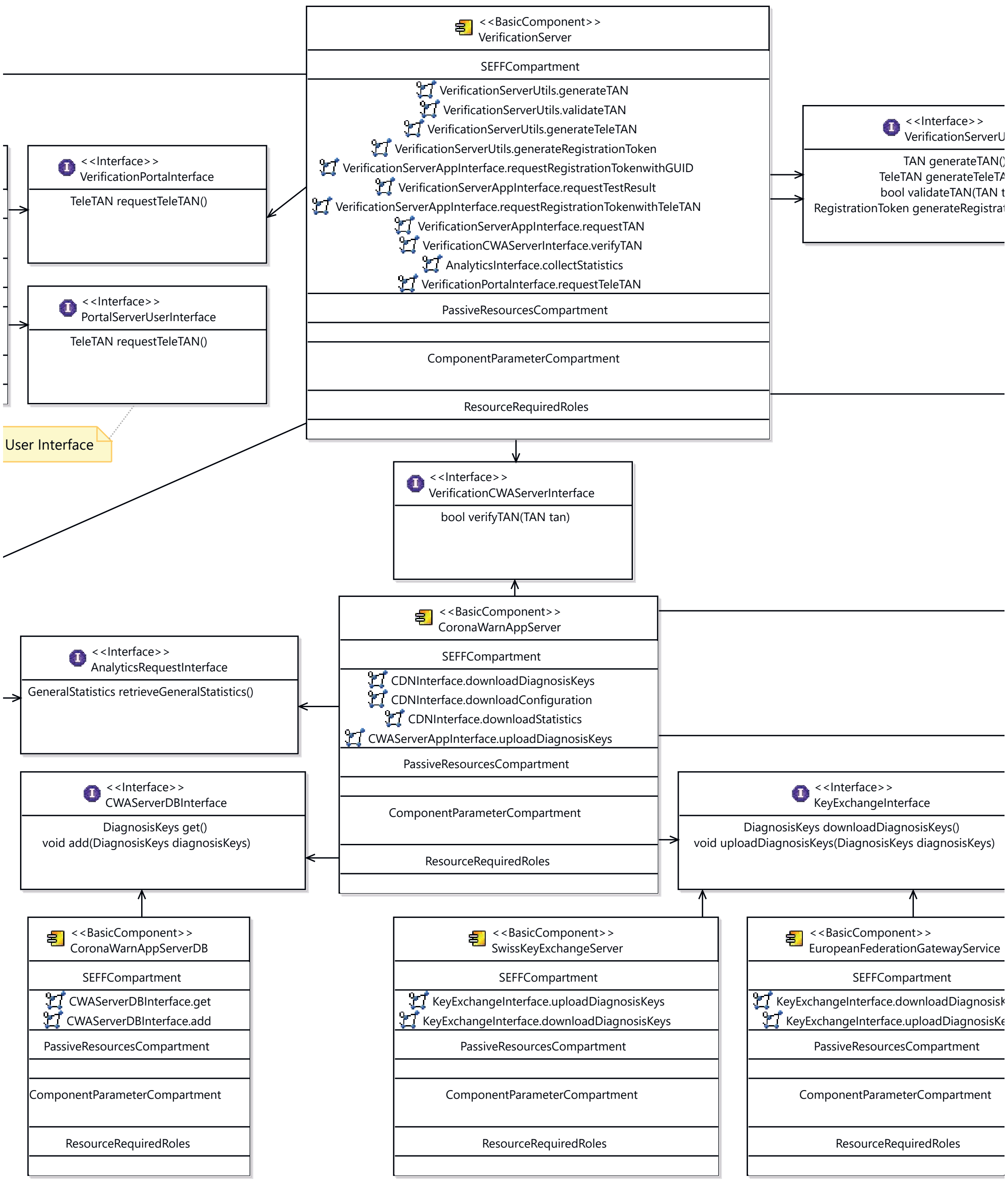
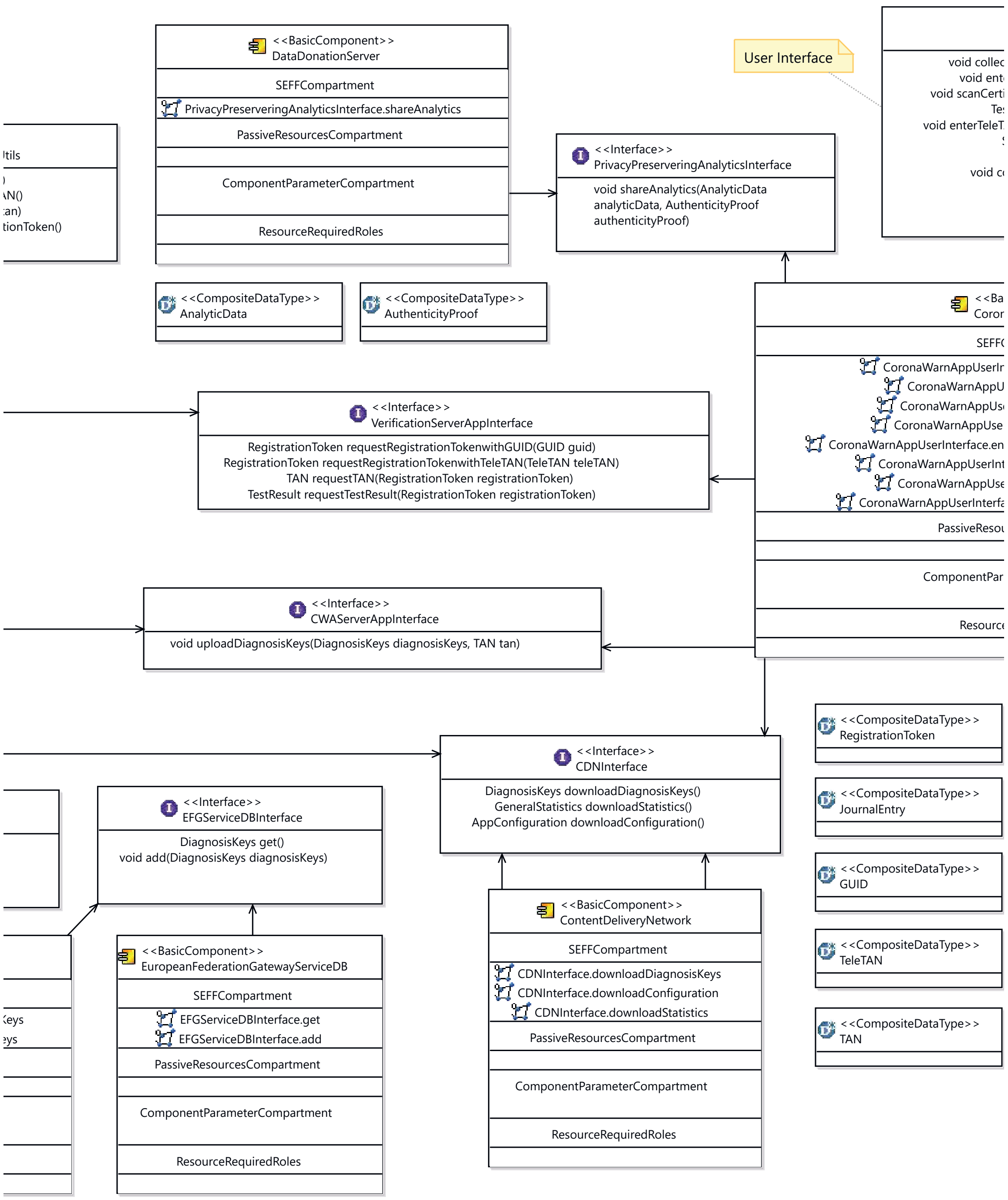
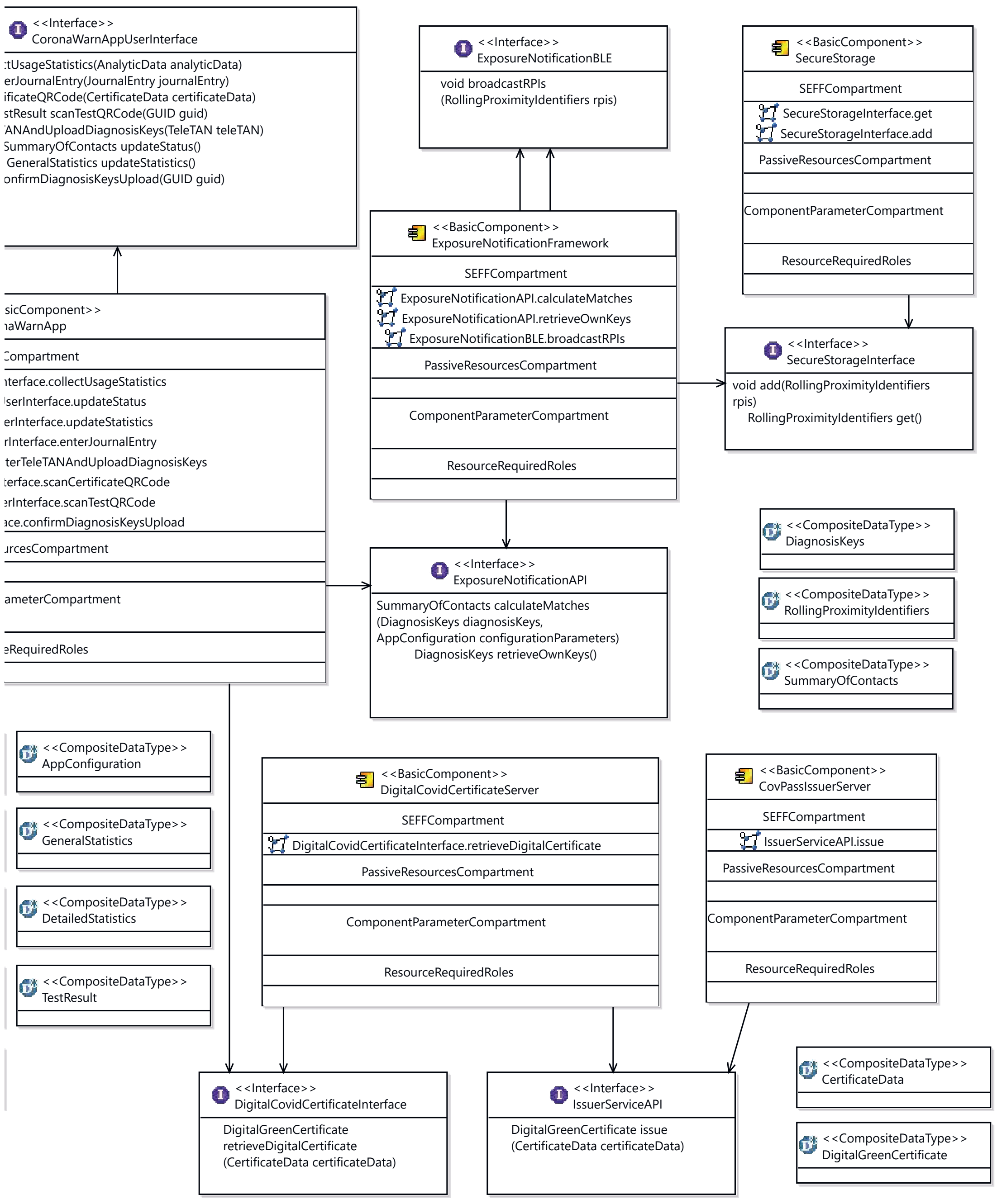

<<BasicComponent>>
VerificationServer
SEFFCompartment
VerificationServerUtils.generateTAN
VerificationServerUtils.validateTAN
VerificationServerUtils.generateTeleTAN
VerificationServerUtils.generateRegistrationToken
VerificationServerAppInterface.requestRegistrationTokenwithGUID
VerificationServerAppInterface.requestTestResult
VerificationServerAppInterface.requestRegistrationTokenwithTeleTAN
VerificationServerAppInterface.requestTAN
VerificationCWAServerInterface.verifyTAN
AnalyticsInterface.collectStatistics
VerificationPortalInterface.requestTeleTAN
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<Interface>>
VerificationServerU
TAN generateTAN(
TeleTAN generateTeleTA
bool validateTAN(TAN t
RegistrationToken generateRegistrat
<<Interface>>
VerificationPortalInterface
TeleTAN requestTeleTAN()
<<Interface>>
PortalServerUserInterface
TeleTAN requestTeleTAN()
User Interface
<<Interface>>
VerificationCWAServerInterface
bool verifyTAN(TAN tan)
<<BasicComponent>>
CoronaWarnAppServer
SEFFCompartment
CDNInterface.downloadDiagnosisKeys
CDNInterface.downloadConfiguration
CDNInterface.downloadStatistics
CWAServerAppInterface.uploadDiagnosisKeys
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<Interface>>
AnalyticsRequestInterface
GeneralStatistics retrieveGeneralStatistics()
<<Interface>>
CWAServerDBInterface
DiagnosisKeys get()
void add(DiagnosisKeys diagnosisKeys)
<<Interface>>
KeyExchangeInterface
DiagnosisKeys downloadDiagnosisKeys()
void uploadDiagnosisKeys(DiagnosisKeys diagnosisKeys)
<<BasicComponent>>
CoronaWarnAppServerDB
SEFFCompartment
CWAServerDBInterface.get
CWAServerDBInterface.add
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>>
SwissKeyExchangeServer
SEFFCompartment
KeyExchangeInterface.uploadDiagnosisKeys
KeyExchangeInterface.downloadDiagnosisKeys
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>>
EuropeanFederationGatewayService
SEFFCompartment
KeyExchangeInterface.downloadDiagnosisK
KeyExchangeInterface.uploadDiagnosisKe
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles

Figure D.3.: Palladio component repository model of the Corona Warn App evaluation scenario (part 2 of 4), exported from the Palladio-Bench [205].



User Interface
<<BasicComponent>>
DataDonationServer
SEFFCompartment
PrivacyPreserveringAnalyticsInterface.shareAnalytics
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<Interface>>
PrivacyPreserveringAnalyticsInterface
void shareAnalytics(AnalyticData analyticData, AuthenticityProof authenticityProof)
<<CompositeDataType>>
AnalyticData
<<CompositeDataType>>
AuthenticityProof
<<Interface>>
VerificationServerAppInterface
RegistrationToken requestRegistrationTokenwithGUID(GUID guid)
RegistrationToken requestRegistrationTokenwithTeleTAN(TeleTAN teleTAN)
TAN requestTAN(RegistrationToken registrationToken)
TestResult requestTestResult(RegistrationToken registrationToken)
<<Interface>>
CWAServerAppInterface
void uploadDiagnosisKeys(DiagnosisKeys diagnosisKeys, TAN tan)
<<Interface>>
CDNInterface
DiagnosisKeys downloadDiagnosisKeys()
GeneralStatistics downloadStatistics()
AppConfiguration downloadConfiguration()
<<Interface>>
EFGServiceDBInterface
DiagnosisKeys get()
void add(DiagnosisKeys diagnosisKeys)
<<BasicComponent>>
EuropeanFederationGatewayServiceDB
SEFFCompartment
EFGServiceDBInterface.get
EFGServiceDBInterface.add
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>>
ContentDeliveryNetwork
SEFFCompartment
CDNInterface.downloadDiagnosisKeys
CDNInterface.downloadConfiguration
CDNInterface.downloadStatistics
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<CompositeDataType>>
RegistrationToken
<<CompositeDataType>>
JournalEntry
<<CompositeDataType>>
GUID
<<CompositeDataType>>
TeleTAN
<<CompositeDataType>>
TAN

Figure D.4.: Palladio component repository model of the Corona Warn App evaluation scenario (part 3 of 4), exported from the Palladio-Bench [205].



<<Interface>> CoronaWarnAppUserInterface
:tUsageStatistics(AnalyticData analyticData)
erJournalEntry(JournalEntry journalEntry)
ificateQRCode(CertificateData certificateData)
stResult scanTestQRCode(GUID guid)
ANAndUploadDiagnosisKeys(TeleTAN teleTAN)
SummaryOfContacts updateStatus()
GeneralStatistics updateStatistics()
onfirmDiagnosisKeysUpload(GUID guid)
<<Interface>> ExposureNotificationBLE
void broadcastRPIs (RollingProximityIdentifiers rpis)
<<BasicComponent>> SecureStorage
SEFFCompartment
SecureStorageInterface.get
SecureStorageInterface.add
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>> ExposureNotificationFramework
SEFFCompartment
ExposureNotificationAPI.calculateMatches
ExposureNotificationAPI.retrieveOwnKeys
ExposureNotificationBLE.broadcastRPIs
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
sicComponent>> iaWarnApp
Compartment
iterface.collectUsageStatistics
serInterface.updateStatus
erInterface.updateStatistics
rInterface.enterJournalEntry
terTeleTANAndUploadDiagnosisKeys
terface.scanCertificateQRCode
erInterface.scanTestQRCode
ice.confirmDiagnosisKeysUpload
urcesCompartment
ameterCompartment
eRequiredRoles
<<Interface>> SecureStorageInterface
void add(RollingProximityIdentifiers rpis)
RollingProximityIdentifiers get()
<<CompositeDataType>> DiagnosisKeys
<<CompositeDataType>> RollingProximityIdentifiers
<<CompositeDataType>> SummaryOfContacts
<<Interface>> ExposureNotificationAPI
SummaryOfContacts calculateMatches (DiagnosisKeys diagnosisKeys, AppConfiguration configurationParameters)
DiagnosisKeys retrieveOwnKeys()
<<CompositeDataType>> AppConfiguration
<<CompositeDataType>> GeneralStatistics
<<CompositeDataType>> DetailedStatistics
<<CompositeDataType>> TestResult
<<BasicComponent>> DigitalCovidCertificateServer
SEFFCompartment
DigitalCovidCertificateInterface.retrieveDigitalCertificate
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<BasicComponent>> CovPassIssuerServer
SEFFCompartment
IssuerServiceAPI.issue
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles
<<Interface>> DigitalCovidCertificateInterface
DigitalGreenCertificate retrieveDigitalCertificate (CertificateData certificateData)
<<Interface>> IssuerServiceAPI
DigitalGreenCertificate issue (CertificateData certificateData)
<<CompositeDataType>> CertificateData
<<CompositeDataType>> DigitalGreenCertificate

Figure D.5.: Palladio component repository model of the Corona Warn App evaluation scenario (part 4 of 4), exported from the Palladio-Bench [205].

E. Towards a Graphical Notation for Uncertainty in Data Flow Diagrams

As addressed previously in this dissertation, there is an ongoing discussion on standardizing the representation of uncertainty in software models. Troya et al. [255] conducted a Systematic Literature Review (SLR) on the wide variety of existing proposals on the representation of uncertainty and find that it is a "good time for the modeling community to try to organize their efforts" [255]. This includes a consensus on the types of uncertainty, but also appropriate notations. In this dissertation, we presented five types of uncertainty sources that are relevant regarding confidentiality and we argued why there should not be less or more than these five types, see Chapter 5. Furthermore, we proposed notations for uncertainty in Data Flow Diagrams (DFDs) and Directed Acyclic Graphs (DAGs), see Section 5.5 and Subsection 7.5.2. However, we did intentionally not define a precise graphical notation, as such an endeavor requires a larger consensus like the ongoing effort to define the Precise Semantics for Uncertainty Modeling (PSUM) standard [184]. Nevertheless, we want to propose an initial graphical notation for uncertainty in DFDs that might help the ongoing discussion.

Figure E.1 shows our proposal. Regarding the abstract syntax, we use the elements of our meta model for Nondeterministic Data Flow Diagrams (NDFDs), see Figure 7.12. The graphical representation of DFDs follows the concrete syntax of the unified modeling

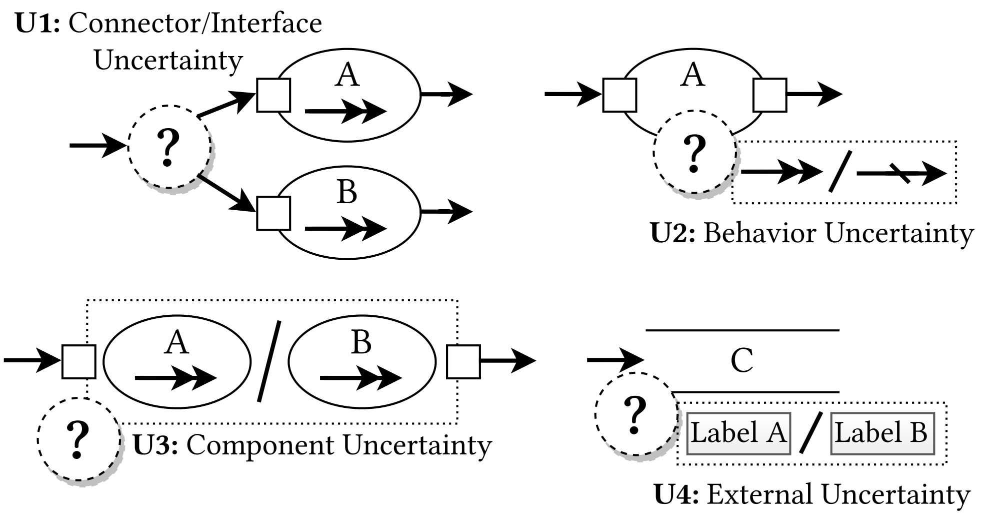


Figure E.1.: Initial proposal for graphically representing uncertainty in Data Flow Diagrams (DFDs).

primitives by Seifermann et al. [235]. As specified by DeMarco [64], nodes are denoted by circles or lines, depending on their type, and flows are denoted by arrows. In our example, *A* and *B* represent processes and *C* represents a file. Pins are denoted by rectangles that decouple flows and nodes, and the behavior specified by assignments is annotated to the nodes. In our example, we show two different behaviors, denoted with different arrow types, e.g., the two-headed arrow $\twoheadrightarrow$. Last, node labels are also annotated, e.g., *Label A*, and *Label B*. For an introduction, see Section 2.5.

Our NDFD meta model shows the relation of the five uncertainty types to the DFD element types. *Connector* and *Interface* uncertainty are *secondary* uncertainty types, that affect the flow between nodes, see Section 5.5. We denote this using the question mark syntax introduced in Chapter 3, by annotating the question mark to the alternative flow. In Figure E.1, this is shown as an alternative flow to either node *A* or node *B* due to Uncertainty **U1**. *Behavior* uncertainty is *primary* uncertainty, directly affecting a node. We denote this using the question mark and by showing alternative arrows that represent the behavior. For instance, the behavior of node *A* could be the forwarding or the encryption of data due to Uncertainty **U2**. *Component* uncertainty affects a complete node and thus subsumes *Behavior* and *External* uncertainty. We denote this again using the question mark syntax and show the alternative nodes, e.g., node *A* and node *B* due to Uncertainty **U3**. A benefit of this graphical notation is that it shows that both nodes require the same interface, i.e., the same pins. Last, *External* uncertainty affects node labels. We denote this using the question mark and by enumerating alternative labels, see Uncertainty **U4**.

Although this graphical notation has many benefits, e.g., directly representing the different scenarios, it is not perfect yet. For instance, we cannot represent the difference between *Connector* and *Interface* uncertainty. Additionally, we cannot represent the impact of *Connector* uncertainty on the assignments of the outgoing node. Last, to avoid ambiguity, we always have to specify the variation due to the uncertainty, e.g., the different labels, or different behaviors. Otherwise, the graphical syntax would, e.g., not differentiate between *Behavior* and *External* uncertainty. Due to these shortcomings, we felt not confident to claim this graphical syntax as part of our contributions. Instead, we focused on the representation of uncertainty in DAGs that is more straightforward by only differentiating between *primary* and *secondary* uncertainty. However, in the light of the recent standardization efforts [184] and to address the need for discussing uncertainty in DFDs based on drawing diagrams, we conclude this dissertation with this short proposal[1].

[1] **Acknowledgement**: Not a single sentence of this dissertation was generated by ChatGPT, Gemini, LLaMA, or a comparable tool. We only used the language-related tools DeepL (https://www.deepl.com/) and Grammarly (https://grammarly.com/) to correct spelling and grammatical mistakes.