Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

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

genehmigte

Dissertation

von

Stephan Seifermann

Tag der mündlichen Prüfung: 23. Mai 2022
Erster Referent: Prof. Dr. Ralf H. Reussner
Zweite Referentin: Prof. Raffaela Mirandola, Ph.D. (Politecnico di Milano)

Abstract

This thesis presents an approach to systematically consider confidentiality requirements in software architectures by representing and analyzing data flows.

The strengthening of data protection regulations such as the European General Data Protection Regulation (GDPR) and the reactions of people to data breaches such as the Cambridge Analytica scandal have shown that ensuring confidentiality in software systems is vital for organizations. To ensure confidentiality, it is necessary to consider confidentiality during the whole development process. Especially, early development phases require attention because a considerable amount of issues traces back to issues introduced in these early phases. Additionally, the effort for fixing issues originating from the software architecture increases disproportionately high in later development phases. Data-oriented representations of software systems are popular for detecting violations of confidentiality requirements in early development phases because investigating a violation often requires following the data flow.

Data Flow Diagrams (DFDs) are commonly used to reason about security in general and confidentiality in particular but plain DFDs are not sufficient to formalize and automate DFD-based analyses. Instead, DFDs and other Architectural Description Languages (ADLs) need extensions to represent the information required to reason about confidentiality. These extensions often only focus on confidentiality requirements given in terms of one particular confidentiality mechanism such as access control. The resulting single purpose approaches do not support combined mechanisms, which lowers their expressiveness. If the software architect changes the confidentiality mechanism, it is necessary to switch to an ADL supporting that mechanism, which implies a high effort for describing existing architectures in the new ADL. In addition, many analysis approaches do not provide an integration into existing ADLs and development processes, which impedes systematic application of the approaches.

Existing data-oriented approaches either considerably rely on manual activities and high expertise or do not support access control, information flow control and encryption within the same specification artifact. These three confidentiality mechanisms are the most commonly used ones, so it is likely that software architects are interested in using all of them. The manual activities include the identification of violations by inspections and the tracing of data through the system. Both require a considerable expertise in confidentiality.

In this thesis, we address the previously mentioned problems by four contributions: First, we present an extension of the DFD syntax, which represents the required information to reason about access control and information flow control combined with encryption by properties and behaviors within the same specification artifact. Second, the semantics for the extended DFD syntax formalize the behavior of DFDs by label propagation, which supports tracing data automatically. Third, a set of analysis definitions based on the DFD syntax and semantics identifies violations of confidentiality requirements given in terms of the most important variants of access control, information flow control and encryption. Fourth, integration guidelines describe how to use the data-oriented analysis framework given by the previous three contributions together with existing ADLs and their corresponding development processes.

We validated the expressiveness, result quality and modeling effort of our contributions in case studies on seventeen case study systems. The case study systems mostly stem from related work and cover five types of access control requirements, four types of information flow control requirements, two types of encryption and one combination of access control and information flow control. We validated the expressiveness of the DFD syntax as well as of two extended ADLs resulting from applying the integration guidelines and could express all but one case study system. We could also express the confidentiality requirements from sixteen case study systems by the provided analysis definitions. The DFD-based as well as the ADL-based analyses only reported expected results, so the result quality was high. We validated the modeling effort in the extended ADLs for adding and switching a confidentiality mechanism for an existing software architecture. In both validations, we could show that the ADL integrations save modeling effort by supporting the reuse of considerable parts of existing software architectures.

Software architects profit from the increased flexibility in choosing and the lowered effort in switching confidentiality mechanisms. The early detection of confidentiality violations reduces the effort of fixing the underlying issues.

Zusammenfassung

Diese Arbeit präsentiert einen Ansatz zur systematischen Berücksichtigung von Vertraulichkeitsanforderungen in Softwarearchitekturen mittels Abbildung und Analyse von Datenflüssen.

Die Stärkung von Datenschutzregularien, wie bspw. durch die europäische Datenschutzgrundverordnung (DSGVO), und die Reaktionen der Bevölkerung auf Datenskandale, wie bspw. den Skandal um Cambridge Analytica, haben gezeigt, dass die Wahrung von Vertraulichkeit für Organisationen von essentieller Bedeutung ist. Um Vertraulichkeit zu wahren, muss diese während des gesamten Softwareentwicklungsprozesses berücksichtigt werden. Frühe Entwicklungsphasen benötigen hier insbesondere große Beachtung, weil ein beträchtlicher Anteil an späteren Problemen auf Fehler in diesen frühen Entwicklungsphasen zurückzuführen ist. Hinzu kommt, dass der Aufwand zum Beseitigen von Fehlern aus der Softwarearchitektur in späteren Entwicklungsphasen überproportional steigt. Um Verletzungen von Vertraulichkeitsanforderungen zu erkennen, werden in früheren Entwicklungsphasen häufig datenorientierte Dokumentationen der Softwaresysteme verwendet. Dies kommt daher, dass die Untersuchung einer solchen Verletzung häufig erfordert, Datenflüssen zu folgen.

Datenflussdiagramme (DFDs) werden gerne genutzt, um Sicherheit im Allgemeinen und Vertraulichkeit im Speziellen zu untersuchen. Allerdings sind reine DFDs noch nicht ausreichend, um darauf aufbauende Analysen zu formalisieren und zu automatisieren. Stattdessen müssen DFDs oder auch andere Architekturbeschreibungssprachen (ADLs) erweitert werden, um die zur Untersuchung von Vertraulichkeit notwendigen Informationen repräsentieren zu können. Solche Erweiterungen unterstützen häufig nur Vertraulichkeitsanforderungen für genau einen Vertraulichkeitsmechanismus wie etwa Zugriffskontrolle. Eine Kombination von Mechanismen unterstützen solche auf einen einzigen Zweck fokussierten Erweiterungen nicht, was deren Ausdrucksmächtigkeit einschränkt. Möchte ein Softwarearchitekt oder eine Softwarearchitektin den eingesetzten Vertraulichkeitsmechanismus wechseln, muss er oder sie auch die ADL wechseln, was mit hohem Aufwand für das erneute Modellieren der Softwarearchitektur einhergeht. Darüber hinaus bieten viele Analyseansätze keine Integration in bestehende ADLs und Entwicklungsprozesse. Ein systematischer Einsatz eines solchen Ansatzes wird dadurch deutlich erschwert.

Existierende, datenorientierte Ansätze bauen entweder stark auf manuelle Aktivitäten und hohe Expertise oder unterstützen nicht die gleichzeitige Repräsentation von Zugriffs- und Informationsflusskontrolle, sowie Verschlüsselung im selben Artefakt zur Architekturspezifikation. Weil die genannten Vertraulichkeitsmechanismen am verbreitetsten sind, ist es wahrscheinlich, dass Softwarearchitekten und Softwarearchitektinnen an der Nutzung all dieser Mechanismen interessiert sind. Die erwähnten, manuellen Tätigkeiten umfassen u.a. die Identifikation von Verletzungen mittels Inspektionen und das Nachverfolgen von Daten durch das System. Beide Tätigkeiten benötigen ein beträchtliches Maß an Erfahrung im Bereich Vertraulichkeit.

Wir adressieren in dieser Arbeit die zuvor genannten Probleme mittels vier Beiträgen: Zuerst präsentieren wir eine Erweiterung der DFD-Syntax, durch die die zur Untersuchung von Zugriffs- und Informationsflusskontrolle, sowie Verschlüsselung notwendigen Informationen mittels Eigenschaften und Verhaltensbeschreibungen innerhalb des selben Artefakts zur Architekturspezifikation ausgedrückt werden können. Zweitens stellen wir eine Semantik dieser erweiterten DFD-Syntax vor, die das Verhalten von DFDs über die Ausbreitung von Attributen (engl.: label propagation) formalisiert und damit eine automatisierte Rückverfolgung von Daten ermöglicht. Drittens präsentieren wir Analysedefinitionen, die basierend auf der DFD-Syntax und -Semantik Verletzungen von Vertraulichkeitsanforderungen identifizieren kann. Die unterstützten Vertraulichkeitsanforderungen decken die wichtigsten Varianten von Zugriffs- und Informationsflusskontrolle, sowie Verschlüsselung ab. Viertens stellen wir einen Leitfaden zur Integration des Rahmenwerks für datenorientierte Analysen in bestehende ADLs und deren zugehörige Entwicklungsprozesse vor. Das Rahmenwerk besteht aus den vorherigen drei Beiträgen.

Die Validierung der Ausdrucksmächtigkeit, der Ergebnisqualität und des Modellierungsaufwands unserer Beiträge erfolgt fallstudienbasiert auf siebzehn Fallstudiensystemen. Die Fallstudiensysteme stammen größtenteils aus verwandten Arbeiten und decken fünf Arten von Zugriffskontrollanforderungen, vier Arten von Informationsflussanforderungen, zwei Arten von Verschlüsselung und Anforderungen einer Kombination beider Vertraulichkeitsmechanismen ab. Wir haben die Ausdrucksmächtigkeit der DFD-Syntax, sowie der mittels des Integrationsleitfadens erstellten ADLs validiert und konnten alle außer ein Fallstudiensystem repräsentieren. Wir konnten außerdem die Vertraulichkeitsanforderungen von sechzehn Fallstudiensystemen mittels unserer Analysedefinitionen repräsentieren. Die DFD-basierten, sowie die ADL-basierten Analysen lieferten die erwarteten Ergebnisse, was eine hohe Ergebnisqualität bedeutet. Den Modellierungsaufwand in den erweiterten ADLs validierten wir sowohl für das Hinzufügen, als auch das Wechseln eines Vertraulichkeitsmechanismus bei einer bestehenden Softwarearchitektur. In beiden Validierungen konnten wir zeigen, dass die ADL-Integrationen Modellierungsaufwand einsparen, indem beträchtliche Teile bestehender Softwarearchitekturen wiederverwendet werden können.

Von unseren Beiträgen profitieren Softwarearchitekten durch gesteigerte Flexibilität bei der Auswahl von Vertraulichkeitsmechanismen, sowie beim Wechsel zwischen diesen Mechanismen. Die frühe Identifikation von Vertraulichkeitsverletzungen verringert darüber hinaus den Aufwand zum Beheben der zugrundeliegenden Probleme.

Contents

Abstract . . . i

Zusammenfassung . . . iii

List of Figures . . . ix

List of Tables . . . xi

List of Listings . . . xiii

1. Introduction . . . **1**
- 1.1. Motivation . . . 1
- 1.2. Problem Statement . . . 3
- 1.3. Overview on State of the Art . . . 5
- 1.4. Challenges and Research Questions . . . 6
  - 1.4.1. Challenge 1: Unified Modeling Language . . . 6
  - 1.4.2. Challenge 2: Analysis Semantics . . . 7
  - 1.4.3. Challenge 3: Extension of Existing Approaches . . . 8
- 1.5. Contributions . . . 9
- 1.6. Outline . . . 10

2. Foundations . . . **11**
- 2.1. Confidentiality . . . 11
- 2.2. Model-Based Software Development . . . 13
- 2.3. Software Architecture Description . . . 15
- 2.4. Logic Programming in Prolog . . . 16

3. Running Example . . . **21**

4. Problem Analysis . . . **25**
- 4.1. Requirements . . . 25
  - 4.1.1. Syntax . . . 25
  - 4.1.2. Semantics of Syntax . . . 28
  - 4.1.3. Integration Procedure for ADLs . . . 30
- 4.2. Discussion of Possible Solutions . . . 31
  - 4.2.1. Syntax: Means for Specification . . . 31
  - 4.2.2. Syntax: System Specification . . . 31
  - 4.2.3. Syntax and Semantics: Propagation of Properties . . . 32

5. Modeling Confidentiality Characteristics of Systems by Data Flow Diagrams . . 35
5.1. Extended Data Flow Diagram Syntax 35
5.1.1. Functional Viewpoint 37
5.1.2. Confidentiality Primitives Viewpoint 39
5.1.3. Confidentiality Viewpoint 43
5.1.4. Requirements Coverage by Viewpoints 44
5.2. Extended Data Flow Diagram Semantics 44
5.2.1. Intuitive Semantics of Extended Data Flow Diagrams 45
5.2.2. Formalization of Semantics in First-Order Logic 49
5.2.3. Requirements Coverage by Semantics 63
5.3. Assumptions and Limitations 64
5.4. Summary 65

6. Confidentiality Analyses based on Label Propagation 67
6.1. Procedure for Analyses 67
6.2. Label-based Confidentiality Analysis Definitions 69
6.2.1. Information Flow Analyses 70
6.2.2. Access Control Analyses 75
6.3. Consideration of Encryption in Confidentiality Analyses 88
6.3.1. Encryption Without Keys 89
6.3.2. Encryption With Key Pairs 90
6.4. Mixing Existing Confidentiality Analyses 92
6.5. DSL for Defining Custom Analyses 93
6.5.1. Scope 94
6.5.2. Abstract Syntax 96
6.5.3. Concrete Syntax 99
6.5.4. Mapping to Logic Program 103
6.6. Requirements Coverage 109
6.7. Assumptions and Limitations 109
6.8. Summary 111

7. Integrating DFD Analyses in Architectural Description Languages 113
7.1. Integration Guidelines 113
7.2. Integrating DFD Analyses with Palladio 119
7.2.1. Call and Return Communication 120
7.2.2. Data-oriented Communication 135
7.2.3. DSL for Defining Custom Analyses 143
7.3. Requirements Coverage 153
7.4. Assumptions and Limitations 154
7.5. Summary 156

8. Validation 157
8.1. Overview 157
8.1.1. Validation Goal 1: Validate DFD Syntax 158
8.1.2. Validation Goal 2: Validate Analysis Definitions 163

8.1.3. Validation Goal 3: Validate DFD Semantics 165
8.1.4. Validation Goal 4: Validate ADL Integration Guidelines 166
8.2. Case Study Systems 172
8.2.1. Requirements on Case Study Systems 172
8.2.2. Selected Case Study Systems 176
8.3. Validation of Extended Data Flow Diagrams 194
8.3.1. Validation Design 194
8.3.2. Validation Results 195
8.3.3. Result Discussion 198
8.3.4. Threats to Validity 200
8.4. Validation of DFD Analyses 203
8.4.1. Validation Design 204
8.4.2. Validation Results 204
8.4.3. Result Discussion 205
8.4.4. Threats to Validity 205
8.5. Validation of DFD Semantics 208
8.5.1. Validation Design 208
8.5.2. Validation Results 209
8.5.3. Result Discussion 210
8.5.4. Threats to Validity 210
8.6. Validation of ADL Integration Guidelines 213
8.6.1. Validation Design 213
8.6.2. Validation Results 219
8.6.3. Result Discussion 223
8.6.4. Threats to Validity 225
8.7. Summary 232
8.7.1. Validation Goal 1: Validate DFD Syntax 233
8.7.2. Validation Goal 2: Validate Analysis Definitions 234
8.7.3. Validation Goal 3: Validate DFD Semantics 234
8.7.4. Validation Goal 4: Validate ADL Integration Guidelines 235

9. Related Work 237
9.1. Approaches to Integrate Confidentiality 238
9.1.1. Security Patterns 238
9.1.2. Generation of Development Artifacts 239
9.2. Approaches to Analyze Confidentiality 241
9.2.1. Identification of Threats 241
9.2.2. Identification of Violated Confidentiality Requirements 243
9.2.3. Calculation of Security Metrics 246
9.3. DFD Semantics 247
9.4. ADL Integrations 249

10. Conclusions 253
10.1. Summary 253
10.2. Benefits 256

10.3. Assumptions and Limitations 257
10.4. Future Work 259

A. Predicate exactCharacteristicValues/5 261

B. DFDs of Self-Created Case Study Systems 263

Bibliography 267

List of Figures

2.1. Four metalevels according to Stahl and Völter. 14
2.2. Example of graphical representation of DFDs. 15

3.1. Interactions between actor and subsystems in the TravelPlanner system. . . 21
3.2. TravelPlanner system given as DFD. 23

5.1. Overview on addressed viewpoints and view types including corresponding concerns and stakeholders. 36
5.2. Data Flow Diagram view type for the Functional viewpoint. 37
5.3. Excerpt of running example using the actor processes to describe data processing steps of actors as well as pins to describe data interface. 38
5.4. Characteristics view type for the Confidentiality Primitives viewpoint. . . . 40
5.5. Behaviors view type for the Confidentiality Primitives viewpoint. 42
5.6. Binding view type for the Confidentiality viewpoint. 44
5.7. Running example using extended data flow diagram syntax. 47
5.8. Characteristic types used in running example. 48
5.9. Flow tree excerpts for the book actor process of the running example. 55

6.1. Overview on analysis procedure. 68
6.2. Overview on analysis procedure when using the DSL. 95
6.3. Overview on metamodel of DSL. 96
6.4. Example of mapping from empty DSL constraint to logic program. 104
6.5. Example of mapping constants from DSL to logic program. 104
6.6. Example of mapping data selectors from DSL constraint to logic program. . . 105
6.7. Example of mapping node selectors from DSL constraint to logic program. . 106
6.8. Example of mapping origin node selectors from DSL constraint to logic program. 107
6.9. Example of mapping between DSL condition and Prolog. 108

7.1. Overview on integration procedure. 115
7.2. Overview on analysis procedure for extended ADL. 116
7.3. Extension of Palladio metamodel to capture properties of nodes. 124
7.4. Extension of Palladio metamodel to capture stores and properties of data. . . 125
7.5. Example of mapping usage scenarios and user actions from Palladio to a DFD. 127
7.6. Example of mapping a SEFF from Palladio to entry and exit processes in a DFD. 129
7.7. Example illustrating nested assemblies in the CreditCardCenter component. 129
7.8. Example illustrating the mapping of a data store in Palladio to a store in a DFD. 130
7.9. Example of mapping the behavior of a SetVariableAction to the behavior of a process. 131

7.10. Example of mapping a call action to DFD processes. 132
7.11. Example of mapping actions to data flows. 134
7.12. Extension of Palladio metamodel to capture the data processing effect of data channels. 139
7.13. Example of mapping data flows through sources and sinks to data flows in a DFD. 141
7.14. Example of mapping data flows between assemblies of data channels to data flows in a DFD. 142
7.15. Example of mapping between data flows from and to actions to data flows between processes in a DFD. 144
7.16. Overview on adjusted elements of metamodel of DSL. 145
7.17. Example of mapping an identity selector for a user from DSL constraint to logic program. 149
7.18. DFD elements considered by an identity selector for a user. 149
7.19. DFD elements considered by an identity selector for a user action. 150
7.20. DFD elements considered by an identity selector for a store. 150
7.21. DFD elements considered by an identity selector for a data channel. 150
7.22. DFD elements considered by an identity selector for a SEFF. 151
7.23. DFD elements considered by an identity selector for an action in a SEFF. . . 151
7.24. Example of mapping a type selector for users or user actions from DSL constraints to logic programs. 151
7.25. Example of mapping a type selector for stores from DSL constraint to the logic program. 152
7.26. Example of mapping a type selector for data channels from DSL constraint to the logic program. 152
7.27. Example of mapping a type selector for SEFFs or actions within SEFFs from DSL constraints to logic programs. 152

8.1. Overview on GQM plan to validate the contributions C1, C2 and C3. 159
8.2. Overview on GQM plan to validate the contribution C4. 160
8.3. Lattice used in the PrivateTaxi case study system. 181

B.1. DFD of ImageSharing case study system (CS13). 263
B.2. DFD of FlightControl case study system (CS14). 264
B.3. DFD of BankBranches case study system (CS16). 265

List of Tables

5.1. Overview on extended syntax elements and met requirements by DFD syntax. 45
5.2. Overview on semantics definitions and met requirements by DFD semantics. 63

6.1. Information flow requirements for the running example given as access control matrix. 72
6.2. DSL functions yielding a boolean result. 98
6.3. DSL functions yielding a reference to a characteristic set. 98
6.4. DSL functions yielding an integer result. 99
6.5. Function precedence, associativity and notation in DSL. 103
6.6. Mapping between functions in conditions of DSL and clauses in logic program. 108
6.7. Overview on described parts and met requirements by analysis definitions. . 109

7.1. Overview on requirements on the integration guidelines and how they are met. 153
7.2. Overview on requirements on the analysis framework and how the analysis framework of the integration procedure meets them. 154

8.1. Relation between validation goals, contributions and research questions. . . 158
8.2. Overview on selected case study systems. 177
8.3. Overview on ratio of expressible systems to total amount of systems per used confidentiality mechanism. 196
8.4. Overview on categories of information required to use model elements. . . . 197
8.5. Overview on the Jaccard Coefficients for adding confidentiality mechanisms to case study systems using control flows and data flows. 221
8.6. Overview on the Jaccard Coefficients for switching confidentiality mechanisms between case study systems using control flows and data flows. 221
8.7. Overview on categories of information required to use model elements of the extended ADL. 222
8.8. Overview on values of weighted ratio metrics for expressiveness of DFDs and ADLs using control flows or data flows. 223

List of Listings

2.1. Examples of facts given in Prolog syntax. 17
2.2. Examples of simple rule given in Prolog syntax. 18
2.3. Examples of recursive rule given in Prolog syntax. 19

5.1. Example of assignments describing the behavior for joining two inputs with respect to data classification. 42
5.2. Example of assignments illustrating use of wildcards to specify the creation of data. 43
5.3. Example of assignments illustrating use of wildcards to specify forwarding of data. 43
5.4. Assignments of credit card data initially created by the user. 46
5.5. Assignments of forwarding and store behavior in running example. 46
5.6. Assignments of declassification behavior in running example. 48
5.7. Prolog facts describing nodes of a data flow diagram. 50
5.8. Prolog facts describing characteristic types and characteristics. 51
5.9. Prolog facts describing the classification characteristic type of the running example. 51
5.10. Prolog clause documenting the type of behavior of a node. 52
5.11. Prolog clauses describing the exploration of alternative inputs. 54
5.12. Prolog clauses finding all input pins for a given node. 54
5.13. Prolog clauses describing how to build a tree of data flows. 56
5.14. Simplified examples of flow trees of the running example. 57
5.15. Rule to specify the label propagation on input pins. 58
5.16. Mapping examples of terms in output characteristic rule. 60
5.17. Example of clauses determining parts of the flow tree needed for label lookup at output pins of book process of running example. 62
5.18. Characteristic rule for output pin of *process booking* process. 62

6.1. Query for detecting violations of information flow requirements given by linear ordered lattice. 74
6.2. Clauses defining the arbitrary lattice and the transitive closure. 75
6.3. Query for detecting violations of information flow requirements given by arbitrary lattice. 75
6.4. Query for detecting a violation of access control specified in DAC. 78
6.5. Rules for determining read and write access in DAC. 79
6.6. Query for detecting a violation of access control specified in Need-to-Know (MAC). 81

6.7. Query for detecting a violation of access control requirements specified in Core RBAC. 83
6.8. Example of clause defining a role hierarchy in RBAC. 84
6.9. Clauses for handling role hierarchies in RBAC. 85
6.10. Example of static constraint on role assignments in Constraint RBAC. 85
6.11. Query for detecting a violation of role assignment constraints in Constraint RBAC. 85
6.12. Examples of subject and object identifiers as well as authorizations for ABAC. 88
6.13. Query for detecting a violation of ABAC requirements. 88
6.14. Query for detecting invalid usage of private keys in decryption. 92
6.15. Query for detecting violations of RBAC requirements and requirements regarding the use of tainted data. 93
6.16. Constraint requiring data classified high to never flow to nodes cleared for low data. 100
6.17. Constraint requiring that data classifications must never be lower than node clearances. 101
6.18. Excerpt of concrete syntax demonstrating the definition of constants. 102
6.19. Excerpt of concrete syntax demonstrating the use of selectors. 102

7.1. Additional Prolog clauses to simplify accessing additional node properties. . 135
7.2. Additional Prolog clause to simplify accessing additional node properties for data channels. 143
7.3. Examples of identity selectors in constraint DSL. 148
7.4. Examples of type selector in constraint DSL. 148

A.1. Clauses providing the exactCharacteristicValues/5 predicate. 261

1. Introduction

In this thesis, we present an approach for systematically considering confidentiality requirements in the architectural design phase of software development processes. The approach supports documenting system aspects, which affect the fulfillment of confidentiality requirements, in an architectural design model as well as the automated detection of violations of these requirements. In the following, we motivate why considering confidentiality in the architectural design is crucial in Section 1.1. In Section 1.2, we identify problems in considering confidentiality in software designs, which have been reported by practitioners and tool developers. We briefly summarize, why the state of the art does not sufficiently address these problems in Section 1.3. In Section 1.4, we derive challenges, as well as research questions from the problems of the state of the art. The answers to the research questions are given in form of contributions, which we describe in Section 1.5. The outline of this thesis is covered in Section 1.6.

1.1. Motivation

The security of software systems is an important factor for organizations because the costs of a security incident can be high [GCH03]. For instance, IBM [IBM20] reported that the global average costs of a data breach of more than 50m records was $392m in 2020. Depending on the particular location and type of organization, the costs can even be significantly higher. Lost business, which can be caused by lost user trust, and the effort for attracting new customers contributes the biggest part of these costs. The Cambridge Analytica scandal [Wei18; IH18] is a good example for this negative impact on business. Another considerable factor contributing to the costs of security incidents are regulatory fines such as the ones implied by the General Data Protection Regulation (GDPR) [Eur16] of the European Union (EU). Organizations are legally obligated to protect the information about their users. If the taken measures are insufficient, which is often shown by a data breach, high fines apply. For instance, British Airways received a penalty [Den20a] of £20m and Marriott International received a £18.4m penalty [Den20b] because of such data breaches. On the other side, sufficiently addressing security is hard because software systems are highly connected, extensible and complex [McG06, pp. 26]. There is a trend of an increasing number of discovered vulnerabilities over the last years [Nat22], which implies that organizations not only have to spend effort in getting their software systems secure but also to keep them secure.

Especially, confidentiality is an important so-called *security objective*. Confidentiality requires that “information is not made available or disclosed to unauthorized individuals, entities, or processes” [Int18]. Maintaining confidentiality also means prohibiting the previously mentioned, costly data breaches. Ensuring confidentiality of personal data, which is a core concept of data protection regulations such as the GDPR [Eur16, Article 32 §1(b)], avoids many fines of such data protection regulations. Besides avoiding costs, maintaining confidentiality can also affect the relation to new and existing customers: A recent survey of Cisco [Cis19] about the importance of privacy for consumers revealed that there is a significant amount of customers (32 %) that already switched to another company for better data protection policies. Therefore, companies cannot only keep customers but can also gain new customers by appropriate data protection. Ensuring confidentiality of customer data is a large step in this direction.

Security mechanisms [Cam13, p. 32] enforce security requirements such as the confidentiality of data by restricting the access to that data according to a policy. A *policy* consists of rules that build the foundation of a decision on whether to allow or deny access to something [Ris+17, pp. 19]. The most prominent mechanisms to enforce confidentiality are information flow control, access control and encryption. Access control limits what users or processes acting on their behalf can do with data [SS94]. Many systems already contain access control mechanisms, so it is the de-facto standard for protecting confidential information [SM03]. However, access control can only restrict access to data within the system under control. Outside of the system, encryption of data can limit access [Sho14, pp. 153]. Various approaches [AT83; HJ03] use such combinations of access control and encryption. In contrast to access control, information flow control does not only limit the access to data but also the flow of information [SM03]. Simple forms of information flow control such as taint analyses are regularly used in practice but more sophisticated forms are not [Sta+19]. None of the mechanisms is clearly superior compared to the others as long as a trade-off between applicability in practice and powerful analyses is required. Therefore, all mechanisms should be taken into account to meet the confidentiality requirements and the capabilities of the system context.

Considering security already while designing the software system, which also includes creating an appropriate architecture, is beneficial for multiple reasons: First of all, it is useful to consider early development phases because historic attempts to only address security by operational means in late development stages were not sufficient [McG06, chap. 3]. Therefore, many organizations now consider security in all development phases, which explicitly includes the design phase. For instance, Microsoft gives explicit instructions on secure software design as part of their commonly known Security Development Lifecycle (SDL) [HL06, chap. 7]. The high interest in software design stems from the experience that many security violations can be traced back to design issues [KRK17] [McG06, p. 151]. In addition to that, the effort for fixing design issues affecting security is significantly lower in the design phase compared to fixing these issues in later phases [Shu+02] [Gee10]. Consequently, identifying issues already in the software architecture that preceds the more detailed software design is even more beneficial. Therefore, organizations should consider spending effort in proper system architectures and designs as well as in the early detection of issues.

Analyses of data flows are commonly used to reason about the security of software designs. This is beneficial because security issues “tend to follow the data flow, not the control flow” [Sho14, p. 44] and requirements regarding confidentiality are often formulated in terms of data and its processing [Gol+19; Kel+09]. Consequently, data-oriented modeling languages such as Data Flow Diagrams (DFDs) are commonly suggested to represent relevant aspects of the software design [Voo20; TSB19]. Therefore, design-time approaches using data flows have a high chance of being useful for designing secure systems.

The goal of this thesis is to provide an approach for detecting violations of confidentiality requirements in the early design stage of a software system by means of data flows. The approach supports describing the data processing in software architectures, determining the effect of this processing on data and detecting violations of confidentiality requirements based on exchanged data. The analyses are not limited to one particular confidentiality mechanisms but support information flow control and access control as well as encryption.

1.2. Problem Statement

Considering security in the software architecture and design is beneficial but a recent survey with practitioners [AC18] revealed that there is still a lack of considering security in software design. This certainly does not hold for all software development teams but shows that there are still problems in integrating security considerations into the software design process. We identified four major problems (P1-P4), which we will address as part of this thesis. In particular, the thesis focuses on establishing confidentiality in software architectures and designs by data-oriented modeling and analysis methods.

P1 Coupling of modeling languages and confidentiality mechanisms A recent survey on modeling languages for representing security in software designs [Ber+17] found that the identified languages often only support one particular security objective and one particular security mechanism. An example of this problem is one particular language that only supports access control based on roles but not access control based on attributes. However, such a strict coupling between security mechanism and modeling language is not feasible in practice: Security requirements might evolve while designing the software system. To meet the requirements, a new or different security mechanisms might be necessary. If a modeling language only supports one specific confidentiality mechanism, switching the confidentiality mechanism means switching the modeling language. However, switching modeling languages requires considerable effort because designers either have to remodel everything from scratch using the new modeling language, which implies high manual modeling effort, or software engineers have to define an automated mapping between the two modeling languages, which is challenging if the languages have substantial differences [TBS20]. If switching a security mechanism is not possible without considerable effort, designers most likely stick to the suboptimal solution or avoid modeling security mechanisms at all. Thus, designers need an approach that allows switching between security mechanisms without implying high manual modeling effort. A flexible modeling language that supports representing multiple security mechanisms can address this problem. With respect to confidentiality, such a flexible modeling language, which allows combining multiple confidentiality mechanisms, is necessary to support emerging approaches that combine, for instance, access control and information flow control [Wan+09; XBS06].

P2 Missing support for commonly used confidentiality mechanisms in analyses Security analyses use system models as inputs and often apply formal methods to identify violations of security requirements. Surveys on formal methods [Dav+13; GBP20] state that approaches are often not widely usable because they are fragmented, i.e. the approaches are specific for a given purpose but it is unclear how to combine these approaches to support multiple purposes. This makes analyzing combinations of confidentiality mechanisms, such as access control and information flow control [Wan+09; XBS06], hard. Additionally, using multiple formal methods requires knowledge in every used formal method, which implies high initial effort for adopting formal methods. Therefore, efficient methods for exchanging information between such approaches or consolidated approaches are necessary to lower the initial effort and increase the probability of being used in practice.

P3 Insufficient formalization of data-oriented modeling languages impedes automated analyses Reasoning about security is barely possible without explicitly considering data and its processing because security requirements are often formulated in terms of data. Automated analyses, which detect violations of security requirements, are a good way to keep the effort for analysts low in presence of complex systems. Such analyses require data-oriented models with clear semantics, which are sufficient to reason about security considerations. However, clear semantics are not naturally available for every modeling language that architects or designers use: DFDs, which use one of the simplest and most prominent data-oriented modeling language, lack clear semantics [JUN11; Sio+20]. This lack of semantics impedes the creation of analyses that yield precise results. A survey [TCS18] on threat modeling approaches, which usually use DFDs, gives a good example of the effect of this impediment: the authors could not identify an approach yielding precise results that can be used by software engineers. To summarize, it is not sufficient to provide architects and designers with data-oriented languages but these languages also have to have clear semantics that serve as foundation for data-oriented analyses. Attempts to define semantics often lead to specific semantics that only support specific analyses for specific quality properties such as performance, reliability or liveliness. However, to create flexible analyses that support multiple confidentiality mechanisms (see P2), semantics that cover the important aspects of such confidentiality mechanisms are required.

P4 Weak guidelines on consideration of confidentiality in software architecture Software development teams follow development processes to build software. Such processes provide activities and often suggest tools to use in order to create artifacts such as software architectures or designs. Thereby, the processes provide guidelines on how to create an architecture or design. One major reason for low adoption of formal methods in general and security approaches in software design in particular is the missing integration into existing development processes [Dav+13; AC19; GBP20]. Clear guidance on when and why to use security approaches is necessary to apply the approach correctly, ensure that the approach is used and that its results are available when they are required. An integration into tools can provide such guidelines at least partially, i.e. the security approach shall be used when the corresponding tool is used. Additionally, tool support eases the use of security approaches in practice: Surveys on formal methods [Dav+13; GBP20] stress that tools should be used extensively in education and that not providing mature tools is a considerable impediment for adopting formal methods. However, many researchers do not spend enough time in creating such tools. A survey on modeling languages for security in software designs [Ber+17] supports this hypothesis by stating that the majority of studied approaches do not provide tools at all. The integration into existing tools is not just a matter of pure engineering effort but also requires concepts on how to bridge potential gaps in abstractions, used description languages or paradigms. Therefore, research has to provide the concepts for integrating their approaches into existing development tools. Such an integration would already be a considerable step in the right direction [Dav+13].

1.3. Overview on State of the Art

There is a wide range of approaches for introducing security in the architecture or design phase of the software development process. In the following, we enumerate only the most important categories of approaches, give examples of particular approaches and mention their shortcomings. A detailed review of the state of the art is available in Chapter 9.

Inspection-based approaches rely on manual screening of descriptions of the system under design by humans. These approaches are flexible but the outcome heavily relies on the expertise of the person conducting the inspection, which limits its applicability. Originally, threat modeling [Sho14] relied on this. However, extensions can lift threat modeling into the categories discussed in the following.

Pattern matching is a commonly used analysis method to identify violations of security requirements. The general idea is to extend existing design documentations by security-relevant information and look for patterns that indicate a violation. Extended threat modeling approaches [Fry+14; Sio+18b; Sio+18a] have formal models that require appropriate documentation of security information. However, creating such information can be challenging. For instance, to decide if exchanged data contains critical information, it is usually necessary to consider all influencing information including their origins. In large systems, this is complex.

Propagation analyses simplify documenting security-relevant information by only requiring an initial set of information and deriving missing information. There are approaches operating on control flows [Kat+13; Jür05] as well as operating on data flows [TSB19; Ber+18]. Approaches using control flows require mapping data-oriented confidentiality policies to restrictions of control flows. With respect to approaches operating on data flows, we could not identify an approach that supports access control as well as information flow control within the same model.

Formal semantics of the system descriptions and behavior specifications are the enabler of propagation analyses. However, formal semantics of data-oriented descriptions and specifications are not always available: The most prominent notion for data-oriented system descriptions, a DFD, lacks such formal semantics in its initial definition [DeM79]. Many approaches have been made to specify such semantics [Jil+08] but there is no standard semantics yet. Semantics tailored to confidentiality are usually part of analysis approaches [TSB19; Ber+18], so the semantics share the limitations of the approaches with respect to expressiveness regarding confidentiality mechanisms. This means, we could not identify formal semantics capable of describing system behaviors relevant for access control as well as information flow control.

1.4. Challenges and Research Questions

To provide an approach for detecting violations of confidentiality requirements in software designs and architectures in a data-oriented way, we have to address three major challenges that we describe in the following. Based on these challenges, we formulate research questions to be answered in this thesis.

1.4.1. Ch1: Definition of Unified Modeling Language Supporting Various Confidentiality Mechanisms

Confidentiality requirements are often formulated in terms of established confidentiality mechanisms. For instance, the access requirements for information can be specified by roles of subjects and access rights to information, which are associated with the roles of subjects, i.e. Role-based Access Control (RBAC). However, it can also be reasonable to describe access limitations in terms of information flow policies specified by labels that classify information and that give users clearance for accessing data of a certain classification. In both scenarios, the system description has to cover all properties and behaviors that affect the decision about access on information. The properties and behaviors are usually specific to a particular confidentiality mechanism, which requires that the system description can represent these individual properties and behaviors.

The challenge is to find a modeling language supporting multiple confidentiality mechanisms without introducing dedicated model elements for each confidentiality mechanism.

The downside of dedicated model elements per confidentiality mechanism is that the resulting language only supports the considered confidentiality mechanisms and consists of many model elements that are barely used by architects or designers in most cases. Instead, a condensed set of model elements is easier to use and requires less learning effort by designers and architects. Paige, Ostroff, and Brooke refer to this language feature as *uniqueness* [POB00]. However, the modeling elements must not become as generic as code to keep the specification and learning effort within reasonable limits. A unified modeling language with respect to confidentiality mechanisms means that the language supports multiple confidentiality mechanisms. In particular, access control and information flow control are not supported together by any data-oriented modeling language as we already stated in Section 1.3. Therefore, supporting both in one unified modeling language is an open challenge.

To decide about reasonable modeling mechanisms and modeling elements, we first have to identify the information, which has to be captured in the software documentation. This is challenging because the information has to be available during the architectural design phase. For instance, behavior descriptions given as implementation code are not yet available while creating a software architecture. Because we especially focus on the confidentiality mechanisms access control and information flow control, we formulate the following two research questions:

Research Question 1: What properties and behaviors are required to reason about access control in software architectures in a data-oriented way?

Research Question 2: What properties and behaviors are required to reason about information flow control in software architectures in a data-oriented way?

The modeling language has to represent the required information. We refer to the condensed set of model elements that can represent the previously identified information as *modeling primitives*. We formulate the following research question:

Research Question 3: What modeling primitives are sufficient to describe architectural aspects affecting confidentiality in a data-oriented way?

1.4.2. Ch2: Definition of Unified Analysis Semantics Supporting Various Confidentiality Mechanisms

Automated analyses can detect violations of confidentiality requirements if the requirements as well as the system under design are specified by languages with defined semantics. This means that it is not sufficient to create a modeling language but it is also necessary to define its semantics. The challenge in defining such semantics is that they have to be capable of supporting different analyses for violations of different confidentiality mechanisms. Defining individual semantics that drive individual analyses is possible but makes understanding model elements harder for designers or architects because they always have to consider the particular analysis context and have to learn different meanings for different contexts. Because we could not identify a data-oriented approach that supports analyzing access control and information flow control within the same model using the same semantics, we see the definition of such semantics as an open challenge. The corresponding research question is as follows:

Research Question 4: What semantics of the data-oriented modeling primitives allow detecting violations of confidentiality requirements?

Because there will be novel semantics, we have to define the access control and information flow control analyses in terms of the semantics. The corresponding research questions are as follows:

Research Question 5: How to formalize common access control analyses using data-oriented modeling primitives and semantics?

Research Question 6: How to formalize common information flow control analyses using data-oriented modeling primitives and semantics?

1.4.3. Ch3: Extending Existing Modeling and Analysis Approaches

The integration of a new modeling and analysis approach into existing tools and processes is beneficial because it provides good guidance to designers and architects in how to use the approach and lowers the initial learning effort for applying the approach. The integration into existing tools is challenging if they do not provide the required modeling concepts: For instance, a data-oriented modeling approach usually requires the concept of a data flow. However, existing tools often focus on control flows instead of data flows and do not provide the concept of a data flow. Bridging this gap is necessary: otherwise, architects and designers would have to remodel large parts of the system under design to incorporate the new communication paradigm. The challenge here is to require as less modifications of existing tools as possible to reduce the learning effort for architects and designers. On the other side, all required modeling concepts have to be available and the analyses have to work within the extended tooling. To achieve this, an approach to extend existing models with considerable effort is necessary. We see the following two research questions:

Research Question 7: How can formal data-oriented confidentiality analyses be integrated into existing modeling and analysis approaches for software architectures, which focus on control flows?

Research Question 8: How can formal data-oriented confidentiality analyses be integrated into existing modeling and analysis approaches for software architectures, which focus on data flows?

1.5. Contributions

We answer the previously defined research questions by the following scientific contributions:

Contribution 1: Extension of DFDs to cover confidentiality properties and behavior. The DFD syntax is one of the most prominent data-oriented modeling language to describe the architecture or design of a system and it is also widely used to reason about the security of systems. We build on the initial DFD syntax by DeMarco [DeM79] and extend it to represent properties by labels and behaviors by reusable label propagation functions. Particular labels and label propagation functions to be used within particular system architectures or designs are available via extensible catalogs. Using labels and label propagation is reasonable because the identified, relevant properties and behaviors for reasoning about access control and information flow control within software architectures, which we identified by answering RQ1 and RQ2, can be represented by discrete values and by mapping functions from discrete values to discrete values. The extended DFD syntax is capable of expressing all relevant information for the most commonly used access control and information flow control mechanisms. Therefore, the syntax presents the answer to RQ3. We evaluate the expressiveness of the syntax by a case study, in which we represent realistic systems using various types of access control and information flow control mechanisms.

Contribution 2: Data propagation semantics for DFDs. We formalize the semantics of the extended DFD from C1 by a mapping from the DFD to clauses in a first-order logic program. The resulting clauses allow to derive the labels of every data item exchanged via data flows based on the behaviors given as label propagation functions. Formalizing the propagation of data through the system is essential to derive properties of data and enable confidentiality analyses that are more powerful than analyses based on pattern-matching. The semantics are the answer to RQ4. We address four common shortcomings of existing DFD semantics such as the systematic consideration of multiple data flow paths. The evaluation of the semantics shows that analyses using these semantics can yield precise results with respect to various information flow control and access control analyses.

Contribution 3: Access control and information flow control analyses based on data propagation. The definition of particular access control and information flow control analyses provides relevant properties of data and nodes, as well as fundamental behaviors relevant for analyzing confidentiality. Especially, behaviors are often missing from descriptions of the particular confidentiality mechanism. This is sufficient for analyses based on pattern matching but not for analyses based on data propagation. We provide four access control analyses for the most common access control models Discretionary Access Control (DAC), Mandatory Access Control (MAC), RBAC and Attribute-based Access Control (ABAC), which is also the answer to RQ5. We provide three information flow analyses for common types of classification lattices in non-interference analyses. This also answers RQ6. We evaluate the precision of the analyses in a case study.

Contribution 4: Integration process for DFD-based analyses into existing ADLs. The integration process defined the required steps to integrate the previously described contributions C1 and C2 into existing Architectural Description Languages (ADLs). The process provides guidelines for ADLs focusing on control flows (RQ7) as well as for ADLs focusing on data flows (RQ8). The integration considers as much existing modeling elements as possible before suggesting extensions of the ADLs. A model transformation to be defined maps the potentially extended ADLs to a DFD, in which the analysis takes place. Another transformation maps the analysis results back into the software architecture. We evaluate the integration process by applying it to the Palladio [Reu+16] approach, which supports control flows as well as data flows. We ensure high expressiveness with respect to the systems and analyses that the extended ADL supports as well as high precision with respect to the analysis results.

1.6. Outline

The thesis is structured as follows. In Chapter 2, we introduce terminology and basic concepts, which we use throughout this thesis. Our running example for exemplifying the contributions is part of Chapter 3. In Chapter 4, we collect the requirements on a solution for addressing the problems and challenges mentioned in the introduction. Chapter 5 covers our first two contributions, which are an extended DFD syntax and corresponding semantics capable of identifying violations of confidentiality requirements in DFDs. Chapter 6 covers our third contribution, which are analysis definitions for information flow and access control analyses based on the DFD syntax and semantics. In addition, we present a Domain-specific Language (DSL) for defining custom analyses. The integration of the previously mentioned contributions into existing ADLs is the fourth contribution and subject of Chapter 7. We provide integration guidelines and describe the application of these guidelines to a particular ADL. The previously mentioned four contributions are validated in Chapter 8. We discuss related work in Chapter 9 and conclude the thesis in Chapter 10.

2. Foundations

In this thesis, we present an approach to systematically consider confidentiality in software architectures. We introduce the terminology regarding confidentiality and frequently used confidentiality mechanisms in Section 2.1. A short introduction in software architectures and the two description languages for software architectures, which we use in this thesis, is given in Section 2.3. Because the description languages are model-based, we explain the fundamentals on model-based software development in Section 2.2 before explaining the description languages. In the course of this thesis, we introduce semantics for a syntax and describe these semantics by predicates given in first-order logic. We use the notion of a logic programming language to represent these predicates and use the logic programming environment to derive and query information. Therefore, we introduce this notion as well as the method for querying the resulting logic program in Section 2.4.

We focus on short and pragmatic descriptions in this chapter. This means, we only explain as much as necessary to understand the following chapters. Giving more detailed explanations is not useful because there are many textbooks on the topics presented in this chapter that explain the topics much better than we can within reasonable page limits.

2.1. Confidentiality

This thesis is about supporting software architects in meeting confidentiality requirements while creating a software architecture. In the following, we first introduce the terminology related to confidentiality, which we use in the remainder of this thesis, and then briefly introduce the most popular mechanisms to support confidentiality. The descriptions of the mechanisms are not meant to be comprehensive. Instead, we want to introduce the terms and give an idea of the principles behind the mechanisms.

Terminology. According to ISO 27000 [Int18], *confidentiality* means that “information is not made available or disclosed to unauthorized individuals, entities, or processes”. Confidentiality is a security objective besides others such as integrity or availability [Int18]. *Security mechanisms* [Cam13, p. 32] support achieving security objectives because they enforce a security policy. In the context of confidentiality, a *policy* consists of rules that build the foundation of a decision on whether to allow or deny access to something [Ris+17, pp. 19]. We see such a policy as a specification of *confidentiality requirements*. Within this thesis, we call the security mechanisms, which support achieving confidentiality, *confidentiality mechanisms*.

Access Control limits what users or system parts acting on the behalf of users can do [SS94]. Access control is the de-facto standard for protecting confidential information in software systems [SM03]. A policy specifies the rules for legitimate or illegitimate actions. Commonly used actions in such policies are reading data or writing data. There are various ways of describing such policies. However, it is common to describe the person or system that performs an action like accessing data as the *subject* and to describe the accessed data as the *object*. Sandhu and Samarati [SS94] identified three commonly used policy types: DAC defines rules based on the identity of a subject and the identity of the object. RBAC introduces roles to decouple the rules from identities. Instead, a subject has a role and there are rules describing the allowed or denied access for a particular role. MAC defines rules based on classifications, which are assigned to subjects and objects. The rules limit the exchange of information between the classification levels. Besides these three policy types, ABAC [Hu+14] also became popular in the last years because it defines rules based on arbitrary attributes of subjects and objects. The support for arbitrary attributes increases the flexibility and even enables representing the other three types of policies within ABAC [JKS12]. Besides the term *policy type* there is also the term *access control model* [Fur08, p. 61], which essentially means the same in the context of this thesis.

Encryption transforms plaintext into ciphertext [Bau05b]. Because the information in the plaintext is no longer readable in the ciphertext, the information cannot be accessed anymore, which protects confidentiality. Cryptosystems [Bau05a] consist of an algorithm to encrypt plaintext, an algorithm to decrypt ciphertext and a defined set of inputs, which usually includes a key. Usually, cryptosystems are used instead of only the encryption operation because encrypting information without means to decrypt the ciphertext back to plaintext would provide no benefit over not sharing information at all. The most important types of cryptosystems are symmetric and asymmetric cryptosystems. In a symmetric cryptosystem [Kal05b], there is one shared key that is used for encryption and decryption. In an asymmetric cryptosystem [Kal05a], there are different keys for encryption and decryption. Usually, a so-called public key is used for encryption and a so-called private key is used for decryption. Using encryption is, especially, useful when information leaves the system under control [Sho14, pp. 153]. Outside of a system, access control cannot enforce rules of that system. In contrast, encryption can enforce rules because subjects outside of the system need the key to access the information and the system can limit the access to this key. There are also approaches [AT83; HJ03] that combine access control and encryption to combine the benefits of both mechanisms.

Information Flow Control limits the propagation of information to protect its confidentiality [SM03]. Goguen and Meseguer [GM82] define the most commonly used information flow property *non-interference* by saying that a process P_1 does not interfere with process P_2 of the same system if the input of P_1 does not affect the output of the system received by P_2. To restrict the information flow between more than two processes, lattices [Den76] are commonly used. A lattice is a directed graph of labels. An edge from label l_1 to label l_2 means that information is allowed to flow from l_1 to l_2. Within software systems, processes have such a label. Between processes with unconnected labels, non-interference has to hold. As Zdancewic [Zda04] stresses, non-interference is often not the property, which shall be achieved in real applications because such applications require information flows not covered by such strict lattices. Declassifications enable flows conflicting with such strict lattices under specific conditions. Sabelfeld and Sands describe these conditions in their work [SS09] on the dimensions of declassification.

2.2. Model-Based Software Development

In model-based software development, software engineers use models to plan and document design decisions [SV06, Sec. 1.1]. Popular examples of such models are class diagrams specified in the Unified Modeling Language (UML) [Obj17] to document the structure of a software system or UML sequence diagrams to document the interaction between parts of the software system. The contributions presented in this thesis also provide model-based means to plan, document but also analyze design decisions for software architectures. In the following, we introduce the terminology about models, which is relevant for this thesis.

Model A model is a simplified representation of the reality. Stachowiak [Sta73, pp. 131] describes a model by three characteristics: The model has to be a *representation* of something. For the definition, it does not matter, what the model actually represents. The model can even represent another model. However, the model has to be a *reduction* of the represented thing. This means, a model cannot simply be identical to the represented thing but has to omit details. The omitted details are chosen based on the served *pragmatism*. A model always has a purpose, which dictates the important aspects and the irrelevant aspects of the represented thing. Irrelevant aspects do not become part of the model. In the context of this thesis, we restrict the definition of a model further to what Stahl and Völter [SV06, Sec. 4.1.1] call a *formal model*. A formal model always adheres to a given syntax. The abstract syntax describes the structure of the formal model and defines criteria for well-formedness. The abstract syntax is given by a metamodel, which we describe in the following paragraph. The concrete syntax describes the representation of the formal model, e.g. by geometric shapes. The means for describing the concrete syntax differ depending on the particular form of representation.

Metamodel A metamodel describes the structure and rules for well-formedness of a model in an abstract way [SV06, Sec. 6.1]. The restriction of the structure and the well-formedness rules enable automated interpretations of a model because every type of content in a model is known already before a model has been created. The UML is a popular example of such a metamodel. A model is said to be an instance of a metamodel. A metamodel is specified by a meta-metamodel, which specifies the structure and well-formedness rules of the metamodel. A metamodel is said to be an instance of a meta-metamodel. In theory, this meta-relation can continue forever. To break this chain, the Object Management

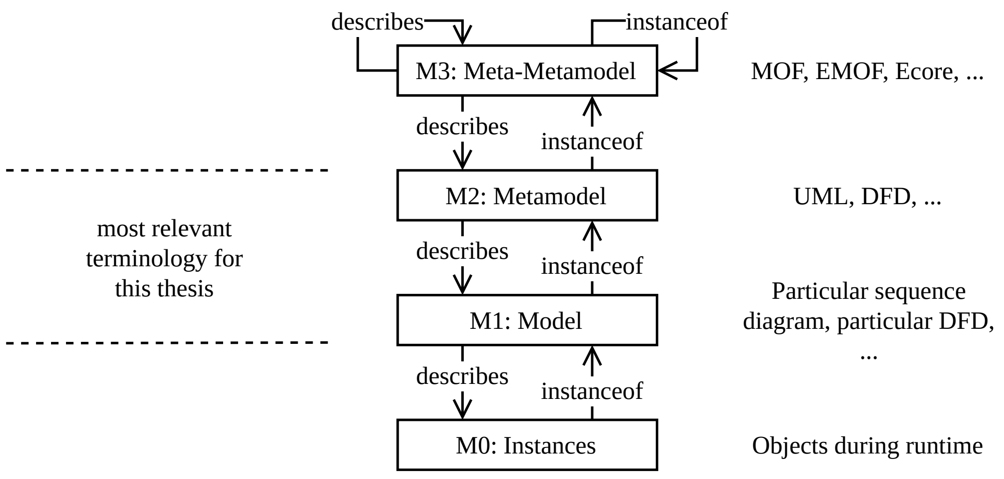


Figure 2.1.: Four metalevels according to Stahl and Völter [SV06, Sec. 6.1] including examples on the right-hand side.

Group (OMG) provides the Meta Object Facility (MOF) [Obj19], which provides a self-describing meta-metamodel to specify metamodels. The OMG does not restrict the depth of meta-relations but four levels are sufficient for many use cases [Obj19, pp. 6]. Stahl and Völter describe these four levels [SV06, chap. 6.1] as shown in Figure 2.1. The meta-metamodel is the uppermost level, describes itself and is also an instance of itself. In the context of this thesis, we use Ecore as a meta-metamodel. Ecore [Ste09, pp. 103] is an implementation of the essential concepts of MOF. Simply said, Ecore consists of classes, attributes of these classes, and references between classes. The metamodel on the second level uses the meta-metamodel to define a custom structure. Metamodels are usually visualized as class diagrams because the meta-metamodel uses the concepts known from class diagrams. Examples of metamodels are the UML metamodel or a metamodel for describing a DFD. Instances of these metamodels, i.e. the models, are on the first level. For instance, particular sequence diagrams or particular DFDs are models. On the lowest level, there are the instances of elements from the model. The objects used in an application during runtime can be such instances.

Viewpoints and Viewtypes Models can become complex if they try to capture too much information at once. This is problematic for software architectures [RW05, pp. 27] but can also happen in other complex domains. The suggested solution is to use views to tailor the representation of information to the needs of stakeholders. Goldschmidt, Becker, and Burger [GBB12] provide a taxonomy covering the most important aspects around views. A view is a selection of particular objects and relations from a model. A viewtype specifies the instructions on how to create a view. The specification is not tied to a particular model but to a metamodel. For instance, a viewtype can specify that only actors of UML use case diagrams shall be considered. The view is constructed by applying the instructions of the viewtype on a particular model. The purpose of defining viewtypes is to support stakeholders in addressing a certain concern. A viewpoint uses viewtypes to address the concerns of stakeholders. A viewpoint can use multiple view types if this is necessary.

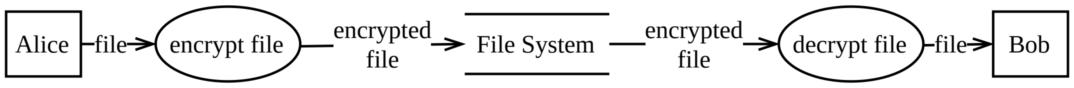


Figure 2.2.: Example of graphical representation of DFDs.

2.3. Software Architecture Description

According to Rozanski and Woods [RW05, pp. 12], “the architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them”. They further clarify that the static part of the structure covers internal design-time elements and the dynamic structure covers runtime elements and interactions. The externally visible properties includes the interaction between the system and its environment as well as quality properties such as performance or security. This means, an architecture covers considerably more than just the static structure of the system. The purpose for creating the software architecture defines the aspects to be covered in the software architecture. To document a software architecture, ADLs are used. According to ISO 42010 [Int11], an ADL can have any form as long as it can express architectural descriptions. However, we always assume an ADL to be specified in a metamodel in the context of this thesis.

In the following, we give short introductions into DFDs and Palladio. Both are ADLs, which we heavily use in the context of this thesis. We explain these ADLs on a high-level to give a basic idea of the purpose and expressiveness of the ADLs. We explain further details within the sections, in which we need these details.

Data Flow Diagrams (DFDs) DFDs as introduced by DeMarco are network representations of a system, in which all elements have a clearly specified interface [DeM79, p. 47]. The representation focuses on the flow of data rather than the flow of control because this better supports high-level discussions of functionality. Consequently, the interfaces specify the exchanged data. DFDs only consist of three types of nodes and one edge [DeM79, pp. 51]. DeMarco suggests the graphical notation shown in Figure 2.2 for representing DFDs. A *process* receives data, transforms it, and yields data. A process is visualized as an ellipse. A *file* is a temporary storage for data. Therefore, we also call a file a *store*. A store is visualized by two horizontal lines. The *source or sink* is a person or organization that is outside of the system. A source sends data to the system. A sink receives data from the system. A node can be a source and a sink at the same time. Because the distinction between these two roles is not important in this thesis, we refer to a person or organization outside of the system as *actor* or *external actor*. An actor is visualized by a rectangle. A *data flow* is the only edge. The edge is directed and connects the nodes and indicates that data is exchanged. A data flow is visualizes by a line with an arrow. DeMarco suggests to use a data dictionary [DeM79, pp. 125] to cover detailed information about data and the nodes. The data dictionary is a mix of grammar-like specifications of data types and explanations given in natural language.

Palladio Palladio is a modeling and analysis approach for predicting quality properties of component-based software architectures [Reu+16, p. 9]. According to Szyperski, a component is a reusable and composable unit of a software system with specified interfaces and defined context dependencies [Bro+98]. The idea behind Component-based Software Engineering (CBSE) [Val+16], which includes creating software architectures, is to build a system from reusable components in order to get benefits such as lower development effort or increased efficiency. Palladio supports CBSE by three viewpoints, which address the concerns of four stakeholders [Reu+16, pp. 44]:

The *structural* viewpoint supports the software architect in his/her concern to define the structure of reusable components in a component repository and to assemble instances of these components to new components or systems. The structure of a component is given by its interface, i.e. the provided services encapsulated in interfaces, as well as by the required services, i.e. the context dependencies. The *behavioral* viewpoint supports two stakeholders. The component developer uses the viewpoint for defining the behavior of a component in service effect specifications. The domain expert uses the viewpoint for defining the behavior of users of the software system in usage scenarios. The most commonly used description of the behaviors is given by a sequence of actions, which impose an effect on inputs, outputs or resources. The *deployment* viewpoint supports the concerns of the system deployer to describe the resource environment and to define the deployment of component instances to resources.

Palladio supports communication via call-and-return and by events [Reu+16, p. 102]. The communication is specified by call actions or actions to emit events. With the Palladio extension *Indirections* [WSK20], Palladio also supports communication via data flows. The communication via data flows is either given by emitting and consuming data via actions within components or by using data channels. A data channel is a special type of component that is not callable but is triggered by incoming data and that emits data to other data channels or actions consuming the data. There are dedicated interfaces for every type of communication. Call-and-return requires operational interfaces that contain callable signatures at components. Event-based communication requires sources and sinks for given event types at components. Data-oriented communication requires sources and sinks for given data types at components. Within a single system, the communication types can be mixed.

2.4. Logic Programming in Prolog

We use logic programming to describe the semantics of a DFD syntax, which we propose in this thesis. We do not intend to provide a full introduction into logic programming but only aim for explaining as much as needed to comprehend the explanations in the following chapters. For a complete introduction, we refer to comprehensive text books [Bra13; EB11], on which this section is based.

Listing 2.1: Examples of facts given in Prolog syntax.

```
1 % facts describing amount of picked apples per employee
2 pickedApples(jane, 22).
3 pickedApples(john, 20).
4
5 % query to find all people X that picked Y apples
6 ?- pickedApples(X,Y).
7 X = john, Y = 20 ;
8 X = jane, Y = 22.
```

Kowalski [Kow79] formulates the fundamental idea behind logic programming, which is that an algorithm consists of logic and control and that it is beneficial to separate these two parts. The logic part contains the problem to solve and the necessary information to solve it. The control part contains the problem solving strategies to solve the problem. In logic programming, users only specify the logic part and the control part is predefined.

The most popular language for logic programming is Prolog [FA03], which is based on first-order logic. The logic part, which is called the knowledge base, consists of a sequence of clauses. Clauses can either be facts or rules. A fact is always true and consists of a name as well as a list of arguments. Lines 2 to 3 in Listing 2.1 give examples of such facts. The name of the fact is `pickedApples`. The arguments of a fact are either numbers, lists or atoms, i.e. constants. By convention, atoms either start with a lower case letter or are escaped by quotes. The intended meaning of the facts in the example is that `jane` picked `22` apples and `john` picked `20` apples.

Prolog environments allow to query the knowledge base. A query consists of a list of goals. A goal consists of so-called compound terms. A compound term looks like a fact, i.e. it has a name and a list of arguments. In contrast to facts, the arguments are not restricted to numbers, lists and atoms but can also be variables or even compound terms. By convention, variable names start with an upper case letter. Prolog tries to find a solution for every goal by deriving a variable binding from the knowledge base. If such a variable binding is found, the goal succeeds. If every goal succeeds, the query succeeds and a solution for the query has been found. In line 6 of Listing 2.1, the query asks for a person `X`, who picked `Y` apples. Deriving a variable binding in this example simply means finding values for `X` and `Y` so that the term in the query is equal to one of the facts. The step of replacing the variables by values in order to make terms equal is called unification. By using backtracking, Prolog can identify all possible solutions. For the query presented in the example, Prolog identifies two possible solutions in lines 7 to 8.

Rules consist of a head and a body. The head is a compound term. Line 2 in Listing 2.2 gives an example of the head of a rule with name `employee` and the variable `X` as argument. The keyword `:-` separates the head and the body. The body is a conjunction of compound terms. The body in the example only consists of the compound term in line 3. The compound term has the name `pickedApples` and uses the variable `X` as first argument and the anonymous variable `_` as second argument. By convention, anonymous variables start with an underscore. The meaning of anonymous variables is that the particular value is

Listing 2.2: Examples of simple rule given in Prolog syntax.

```
1 % rule saying that someone is an employee if he/she picked apples
2 employee(X) :-
3    pickedApples(X, _).
4
5 % query for all employees X
6 ?- employee(X).
7 X = john ;
8 X = jane.
```

not important for finding a solution. The query in line 6 asks for all employees. To find variable bindings for variable `X` in the query, Prolog uses Selective Linear Definite (SLD) clause resolution. Simply said, the resolution replaces a term in a query by the terms in the body of a rule and tries to unify resulting terms with facts. This leads to a recursive evaluation. In the example, Prolog evaluates the `employee` rule by finding a variable binding for variable `X` such that the `pickedApples` term succeeds. Consequently, the query finds the two solutions shown in lines 7 to 8 when including the facts from Listing 2.1.

Recursive specifications are commonly used in Prolog. Listing 2.3 gives an example of a recursive definition of the `sumOfApples/2` predicate. Predicates are the signatures of clauses. The signature is given by a name and an arity, i.e. the number of parameters. In the given example, the fact in line 2 and the rule in line 3 both have the same signature. The fact has two arguments. The first argument is an empty list. In Prolog, lists are encapsulated by square brackets. An empty list is represented by a left and a right square bracket. The second argument is the number `0`. The intended meaning of the fact is that no employees, which is implied by the empty list of employees, picked zero apples in sum. The rule in line 3 also has two arguments. The first argument is a list containing the variables `H` and `T`. The pipe symbol is a separator between the head `H` and the tail `T` of the list. Simply said, everything before the pipe symbol is in the beginning of the list and everything after the pipe symbol is the remainder of the list. The second argument is a variable `SUM`. The intended meaning of the rule is that the variable `SUM` is the sum of the apples picked by the employees given by the list. The body of the rule consists of three clauses. The first clause queries the number of picked apples `N` for the employee `H`. The second clause queries the sum of picked apples `M` for the remaining employees `T` of the list. The third clauses sums up the numbers of picked apples and unifies the result with the variable `SUM`. The second clause makes the rule recursive. In every recursion the list of employees is shortened by one employee. Prolog tries to find a solution for the clause by either using the fact or the rule of the example. The fact can only be used if the list of employees is empty. The rule cannot be used with an empty list of employees because there would be no solution for the first clause. Therefore, the fact can be seen as terminator of the recursion. The query in line 9 uses the `sumOfApples/2` predicate to find the sum of picked apples by `jane` and `john`. Using the facts from Listing 2.1, the sum is 42.

The body of rules can contain even more complex expressions. Instead of building the conjunction of clauses by using the `,` keyword, it is also possible to build disjunctions of

Listing 2.3: Examples of recursive rule given in Prolog syntax.

```
% sumOfApples/2 finds amount of picked apples by employees
sumOfApples([], 0).        % no employees means no picked apples
sumOfApples([H|T], SUM) :-  % recursively sum up picked apples by employees
   pickedApples(H, N),
   sumOfApples(T, M),
   SUM is N+M.

% query for sum SUM of picked apples by jane and john
?- sumOfApples([jane, john], SUM).
SUM = 42.
```

clauses by connecting them with the `;` keyword. There is no logical negation. The closest match is the not-provable predicate `\+`, which succeeds if the expression following the predicate does not succeed, i.e. no solution can be found. This behavior is called *negation as failure*. The behavior is not the same as logical negation, so it has to be used with caution. Because this thesis is not about theoretical foundations of Prolog, we refer to dedicated work on issues arising from this difference [Sub99] [OKe90, p. 199].

3. Running Example

We use the TravelPlanner system [Kat+13] as a running example in this thesis. The system has already been used in the validation of the iFlow approach [Kat+13; Kat17], which aims to create software systems with secure information flows. In the following, we introduce the example.

The TravelPlanner system is meant to support a user in looking for flights and booking them. The system covers one actor and four subsystems, which are illustrated in Figure 3.1. The user uses the subsystem *TravelPlanner* on his/her smartphone to look for flights. The user enters criteria for the flight into the *TravelPlanner*, which forwards this criteria to a *TravelAgency*. The *TravelAgency* builds a query based on the given criteria and queries the *Airline* for flights. The resulting flights are sent back to the user. The user decides for a flight and loads the credit card details (*ccd*) from the *CreditCardCenter* on his/her smartphone for the payment of the flight. Before sending the payment information to the *Airline*, the user releases this payment information. This is a declassification operation, which is necessary to allow the *Airline* to access the payment information. Afterwards, the user sends the payment information together with the selected flight to the *Airline* for booking the flight. The *Airline* processes the booking and sends a commission to the *TravelAgency* for connecting the user and the *Airline*. Eventually, the user receives a confirmation.

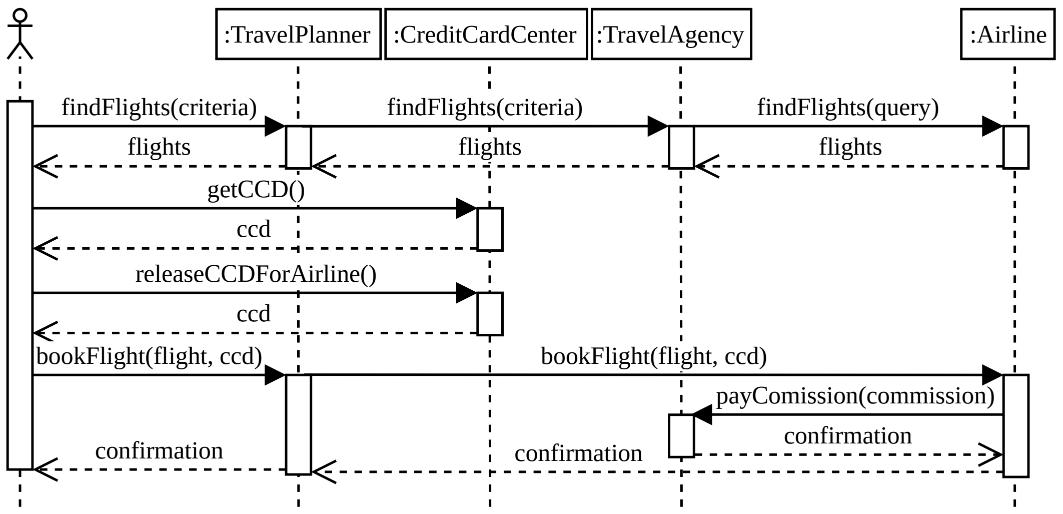


Figure 3.1.: Interactions between actor and subsystems in the TravelPlanner system given as UML sequence diagram (visualization based on previous publication [Sei+22]).

The confidentiality requirement for the system is that the user and the subsystems must only access information, for which they have been cleared. The user and the subsystems are cleared for information if their clearance level is greater or equal to the classification level of information. For simplicity, we just use numbers to represent these levels. The *TravelAgency* is cleared for level 1. The information about flights and commissions is classified for level 1. The *Airline* is cleared for level 2. The confirmation is classified for level 2. The user including his/her apps is cleared for level 3. The payment information is classified for level 3. This means that the *Airline* must not access the payment information. To avoid violating the confidentiality requirement, the declassification operation takes the payment information and explicitly reclassifies it to level 2, which means it is accessible by the *Airline*.

Amongst others, we use DFDs to represent systems and analyze them for violated confidentiality requirements. Originally, the TravelPlanner system has been specified in UML using call-and-return communication. To support the explanations in this thesis, we also need a version of the TravelPlanner system using data flows. Therefore, we created a DFD, which represents the TravelPlanner system. We already presented this version in a previous publication [Sei+22].

The DFD representing the TravelPlanner system is illustrated in Figure 3.2. For the sake of better comparison with the original version, we indicate the subsystems, to which the DFD elements belong, in the top of the diagram within gray boxes. In the following description, we use temporal relations to explain the DFD in an intuitive way. However, the DFD does not imply such a temporal relation but only defines data dependencies.

The user is represented as actor. An outgoing data flow transports the criteria to look for flights to a process of the *TravelPlanner*. The *TravelPlanner* delegates the data to the *TravelAgency*, which builds a query and sends it to the *Airline*. The *Airline* loads flights from a storage, uses the query to filter the flights and passes these flights back to the *TravelAgency*. Eventually, the filtered flights are propagated back to the *User*. Afterwards, the user passes a consent to the process *declassify CCD* and receives declassified credit card details from it. The user is ready to book the flight and sends the selected flight as well as the declassified credit card data to the process *TravelPlanner*, which delegates the data to the *Airline*. The *Airline* creates and stores a booking and sends a commission to the *TravelAgency*. Afterwards the *Airline* sends a confirmation to the user.

The DFD contains some differences compared to the version using calls. First of all, there are data flows to put data into data stores. The *User* initially sends credit card details to the store and a *FlightPlanner* sends flights into a flight store. Initializing stores is also necessary in the version using calls but this step has been omitted for the sake of simplicity. The second notable change is the addition of a *ccd* data flow from the *CCD Storage* to the *User*. The data flow is colored in gray. This data flow introduces an issue, which leads to a violation of the confidentiality requirements, because the *User* can use the payment information, which has not been classified, in the booking. We detail this issue in later descriptions.

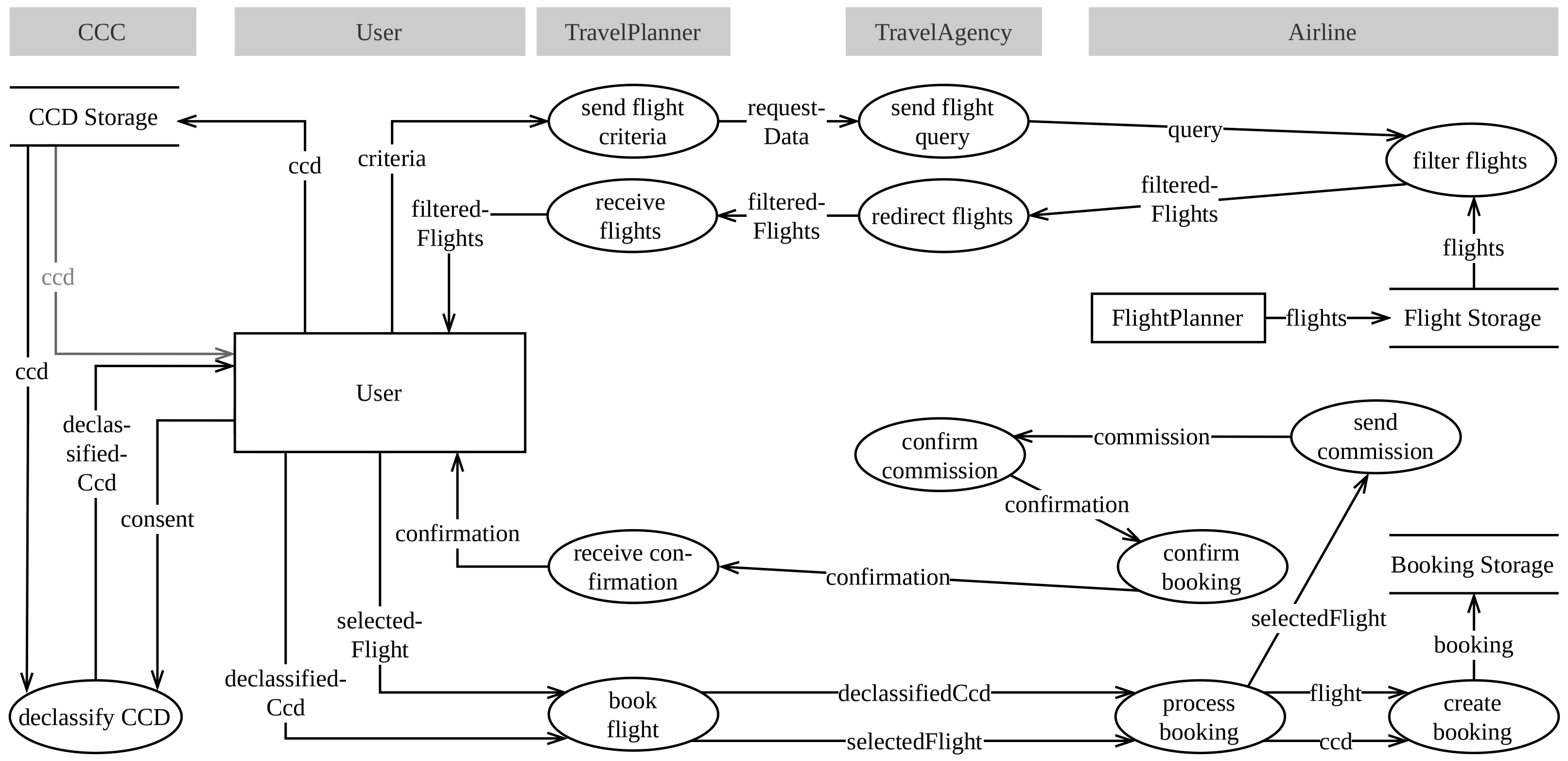


Figure 3.2.: TravelPlanner system given as DFD (gray boxes in top indicate subsystems of original system and gray data flow introduces issue).

4. Problem Analysis

The introduction in Chapter 1 stated the high-level goal of the thesis, which is providing an approach for detecting violations of confidentiality requirements in the early design stage of a software system by means of data flows. To achieve this high-level goal, we have to address the challenges (Ch*n*) described in the introduction. We address the challenges by providing the contributions, which are an extended syntax for DFDs, corresponding semantics, access control and information flow analyses as well as an integration procedure of DFD analyses in existing ADLs. To make sure that we address the challenges appropriately, we formulate requirements regarding the contributions in Section 4.1. In Section 4.2, we discuss possible approaches to meet these requirements.

4.1. Requirements

The contributions mentioned in the introduction imply the need for three artifacts to be developed: 1) a syntax for describing the system including aspects relevant for confidentiality, 2) semantics for that syntax and 3) an integration procedure for analyses based on the previous artifacts into existing ADLs. The particular access control and information flow analyses make use of these three developed artifacts. Formulating requirements for particular analyses is not necessary because the only requirement is to detect violations of confidentiality requirements. The definition of a violation is already given by literature. The main user of the artifacts is the software architect. Because we cannot expect that software architects have detailed security expertise, it is reasonable to also consider a security expert, who can support the software architect.

In the following, we will collect functional and non-functional requirements for the previously mentioned artifacts. Throughout this section, we use the requirements templates of Pohl and Rupp [PR15, pp. 53] to formulate requirements in natural language. We refer to requirements by the identifier R*n*.*m*, where *n* is the number of the artifact, to which the requirement belongs to, and *m* is a continuously incremented identifier. By meeting the requirements, the resulting artifacts will also address the corresponding challenge. We structure the presentation of the requirements by the corresponding artifacts.

4.1.1. Syntax

The syntax provides means to structure the information about systems and confidentiality aspects. Stahl and Völter [SV06, sec. 4.1.1] distinguish between concrete and abstract syntaxes. A concrete syntax specifies the representation of information, e.g. by a sequence of tokens in a text. In contrast, an abstract syntax only specifies the structure of the input but not its particular representation. For instance, an abstract syntax would specify that a named class exists and a concrete syntax would specify that a class is represented by a rectangle with the name of the class in it. In this thesis, we only specify requirements for the abstract syntax to focus on the required concepts. We also introduce examples of concrete syntaxes for the abstract syntax but we do not prescribe a particular concrete syntax. We do not restrict the concrete syntax except from the concepts to be represented in order to allow considering the preferences of architects and the organization, in which they work. Finding an appropriate and usable concrete syntax is a research topic on its own, which requires different approaches to construct and evaluate the concrete syntax than the ones we apply in this thesis. Therefore, presented concrete syntaxes are only examples and no contributions of this thesis. The following requirements cover the needs of software architects and security experts in using such a syntax.

First of all, the syntax has to be capable of describing commonly used aspects of systems. Rozanski and Woods [RW05, p. 36] present six viewpoints to describe systems by software architectures. The functional viewpoint essentially describes the system structure, i.e. system parts, their interfaces and their connections. The information viewpoint describes the flow and manipulation of data. To get a complete view, the description has to cover data manipulation by the system as well as by the user. For the system, this means that the view describes its behavior. For users, this means that the view describes their behavior, which essentially describes their usage of the system. The deployment viewpoint describes the hardware as well as the assignment of system parts to hardware. The remaining three viewpoints either target different phases, i.e. the development or operation phase, or focus on concurrency. Because we focus on the system information required to identify confidentiality violations in the early design phase, the viewpoints about late phases are out of scope. The concurrency of data-oriented systems is often derived from data dependencies and the system structure, so we do not have to explicitly consider this. Eventually, this brings us to the following requirement:

R1.1) The syntax shall provide the software architect with the ability to describe the structure, behavior, deployment and usage of the system.

Besides the previously mentioned generic system descriptions, the syntax also has to cover specific aspects only relevant for confidentiality. In data-oriented system descriptions, it is necessary to know the properties of data, which are required to reason about meeting confidentiality requirements. A first step in this direction is to define the available types of properties. For instance, it is reasonable to define a property type *Role* with a value range consisting of all particular roles in a system. A particular property, i.e. an instance of a property type, can then hold a subset of the particular roles in a system. The property types usually depend on the confidentiality mechanism, so security expertise can be required to identify all relevant property types. Therefore, the security expert will most likely define the property types. This brings us to the following requirement:

R1.2) The syntax shall provide the security expert with the ability to describe property types.

The behavior of the system and the usage of the system can change the properties of data. If the way of describing behavior or usage is not sufficient to determine such changes, it is necessary to provide means for making such changes explicit. Security experts have the expertise to identify types of behaviors that are relevant for reasoning about confidentiality requirements. Additionally, they can specify the changes of data properties, which these behavior types imply. However, security experts cannot decide on where these behavior types are actually used in the system. In contrast, software architects have the expertise to decide on which system part behaves in a way that matches the defined behavior types. This brings us to the following requirements.

R1.3) The syntax shall provide the security expert with the ability to specify types of behaviors, which describe the effect of system behavior or system usage on data properties.

R1.4) The syntax shall provide the software architect with the ability to assign behavior types to system parts or usage descriptions.

Besides data, system parts can also have properties that are required to reason about meeting confidentiality requirements. Again, security expertise is required to identify required properties of system or usage parts as well as reasonable combinations of multiple properties. A security expert can specify these combinations depending on the confidentiality requirements. Software architects have the expertise to identify system or usage parts that should have the predefined combinations of properties. This brings us to the following requirements.

R1.5) The syntax shall provide the security expert with the ability to specify combinations of properties for system parts or usage descriptions.

R1.6) The syntax shall provide the software architect with the ability to assign combinations of properties to system parts or usage descriptions.

The overall goal is to provide a unified approach, which supports analyzing multiple confidentiality mechanisms as well as analyzing combinations of multiple confidentiality mechanisms. Using the same language constructs for representing various confidentiality mechanisms helps to reduce the learning effort for software architects as well as security experts. In addition, such general applicable language constructs have the potential to support more confidentiality mechanisms than specific language constructs. All language constructs necessary for the previously described specification tasks of software architects and security experts shall be general applicable to avoid limiting the resulting syntax to a few confidentiality mechanisms. Additionally, mixing behaviors and properties of different confidentiality mechanisms should be possible to combine the benefits of various mechanisms. This means that software architects and security experts shall be able to describe system aspects relevant for multiple confidentiality mechanisms within the same input artifact. Besides combining confidentiality mechanisms, replacing a confidentiality mechanism with another one can be reasonable. For instance, a simple mechanism can be feasible for a limited set of requirements but as soon as more requirements arise, a more powerful mechanism might be necessary. To increase the probability that a mechanism is replaced if necessary, the effort for doing so should be as low as possible. At least, the effort for switching confidentiality mechanisms should be low compared to the overall effort for representing the system and another confidentiality mechanism. This brings us to the following requirements.

R1.7) The syntax shall provide the security expert with the ability to specify property types, behaviors and properties for various confidentiality mechanisms by the same language constructs.

R1.8) The syntax shall provide the software architect with the ability to assign behaviors or properties for various confidentiality mechanisms by the same language constructs.

R1.9) The syntax shall provide the security expert with the ability to specify property types, behaviors and properties for various confidentiality mechanisms within the same input artifact.

R1.10) The syntax shall provide the software architect with the ability to assign behaviors or properties for various confidentiality mechanisms within the same input artifact.

R1.11) The syntax shall provide the software architect with the ability to switch confidentiality mechanisms with low effort.

4.1.2. Semantics of Syntax

The semantics of the syntax assign a meaning to the syntactical elements, which drives analyses for violations of confidentiality requirements. If an element is not needed during the analysis, the semantics should clearly state that the particular element has no meaning to avoid ambiguities. This brings us to the first requirement for the semantics:

R2.1) The semantics shall specify the meaning of every construct in the syntax.

As motivated before, analyses have to determine properties of data and system parts to identify violated confidentiality requirements in data-oriented system descriptions. Properties can be static but they can also emerge from processing various data items. To improve the comprehensibility of analysis results, it is not only necessary to determine properties but the semantics also have to be capable of explaining the origin of a property. For instance, a data property might emerge from data processing, so the origin should contain the processing steps. This brings us to the following requirements:

R2.2) The semantics shall provide analyses with the ability to determine properties of data and system parts.

R2.3) The semantics shall provide analyses with the ability to determine the origin of properties.

Security experts and software architects mainly interact with the analyses built on the semantics. The software architect uses these automated analyses to identify design decisions in the analyzed architecture, which violate confidentiality requirements. To keep the definition of an analysis simple, it should be enough for security experts and software architects to specify an analysis goal but not the procedure to meet the analysis goal. An analysis framework, which only requires an analysis goal to define an analysis, is necessary to achieve such simple analysis definitions. The analysis not only has to yield an analysis result but also means to identify the faulty design decision, which lead to the confidentiality violation. A software architect needs information about the origin of the data item, which violates a confidentiality requirement. It might be necessary to have a look at multiple previous data processing steps and data items to identify the design decision leading to the violation. Therefore, a software architect needs a trace, i.e. the processing steps including the transmitted data. To keep the effort low for conducting an analysis, e.g. after adjusting the software architecture, the analyses should be automated. This way, the software architect can run an analysis frequently and gets fast feedback about design decisions. This brings us to the following requirements for an analysis framework, which can also affect the semantics that build the foundation of the analysis framework:

R2.4) The analysis framework shall provide the security expert with the ability to define an analysis based on an analysis goal.

R2.5) The analysis framework should provide the software architect with the ability to define an analysis based on an analysis goal.

R2.6) The analysis framework shall provide the software architect with trace information about properties.

R2.7) The analysis framework shall provide the software architect with automated analyses for violations of confidentiality requirements.

The most prominent confidentiality mechanisms are information flow control and access control. The analysis framework as well as the semantics have to support analyses of the most common types of confidentiality requirements given in terms of information flow control and access control. The analyses should not be part of the analysis framework but shall imply requirements regarding the expressive power. This brings us to the following requirements:

R2.8) The analysis framework shall provide the security experts with the ability to define information flow control analyses.

R2.9) The analysis framework shall provide the security experts with the ability to define access control analyses.

4.1.3. Integration Procedure for ADLs

The integration of the analysis approach into existing ADLs and their tooling is necessary to simplify adoption of the approach by lowering the initial effort for grasping the approach and by providing guidance on when to use the approach.

To make the integration procedure widely applicable, the procedure should work for existing ADLs that use the control flow paradigm, the data flow paradigm or both. The resulting extended ADL might not be data-oriented but the underlying analysis can still operate on data flows that stem from control flows. Apart from that, the resulting extended ADLs shall meet all requirements already listed for the syntax. Focusing only on ADLs operating on data flows is not feasible because many modeled architectures only provide control flows and switching the paradigm would require considerable effort. This brings us to the following requirements:

R3.1) The integration procedure shall cover ADLs using control flows.

R3.2) The integration procedure shall cover ADLs using data flows.

One essential goal of the integration into existing ADLs is to lower the learning and migration effort for software architects. To achieve that, the integration procedure has to aim for as less modifications of existing ADLs as possible. To give a counter example, assume we just merge the syntax with an existing ADL. Certainly, analyses could be conducted in this extended ADL but software architects would still have to get to know the new elements and would have to remodel existing architectures using the new model elements. This brings us to the following requirement:

R3.3) If an ADL provides a required concept, the integration procedure shall reuse the corresponding model elements.

The integration procedure will also have to yield an analysis framework that security experts and software architects will use. All requirements for the previously mentioned analysis framework that operates on the data-oriented input apply to the ADL analysis framework too. The integration procedure should not yield an integration that requires knowledge about the syntax, semantics or analysis framework mentioned in the sections before. For instance, an integration shall not require knowledge about the concept of a data processing chain if the ADL only uses control flows. However, essential concepts such as characteristics, which describe properties of architectural elements or data, or behaviors, which describe how system activities affect the characteristics mentioned before, have to be available. Therefore, extending the ADL can be necessary, if there are no counterparts of the new concepts but the extensions must use the terminology used in the ADL and not the terminology used in the DFD. This brings us to the following requirements:

R3.4) The integration procedure shall yield an analysis framework that meets all requirements of the analysis framework for DFDs.

R3.5) The integration procedure shall yield an integration that only uses concepts from the architectural domain.

4.2. Discussion of Possible Solutions

There are multiple possible approaches to meet the previously described requirements. In this section, we would like to discuss alternative solutions on a coarse-grained level and then decide for one solution that we will use to realize our contributions.

4.2.1. Syntax: Means for Specification

As explained in 4.1.1, we focus on the required concepts, which are represented in the syntax, rather than the concrete representation of these concepts. Therefore, we specify an abstract syntax. This means that typical approaches to specify concrete syntaxes such as syntactic metalanguages, which includes Extended Backus–Naur form (EBNF) [Int96], are no appropriate means for specifying the abstract syntax. Metamodeling is a commonly accepted approach to specify abstract syntaxes. The MOF [Obj19] provides standardized means to specify the abstract syntax by a metamodel. To visualize the metamodel, we will use class and object diagrams of the UML, which is a commonly accepted way of visualizing metamodels.

4.2.2. Syntax: System Specification

The syntax has to describe the software system as well as its behavior and usage in a data-oriented way. It is possible to create a syntax from scratch but this increases the required initial knowledge for grasping the introduced concepts and makes migrating existing system descriptions into the new syntax hard. Instead, it is beneficial to reuse existing system descriptions and extend it by the missing features.

There are many different ways to describe systems in a data-oriented way. Prominent examples that are often used to reason about security are Petri nets and DFDs. Activity diagrams given in UML are also popular.

Petri nets [Pet62] use places, transitions and edges to describe distributed systems. Places hold tokens, which can be transformed in transitions. The distribution of tokens describes a system state. Petri nets can represent control flows and data flows [Liu+20]. They often serve as formal foundation for analyzing information systems and workflows. There are various extensions [JR91] available that improve the capabilities of Petri nets. Such extensions enable the detection of access control violations [Kno00] as well as of information flow violations [AH97].

DFDs [DeM79] focus on data flows and data processing. The diagrams consist of only few types of model elements and are considered intuitively comprehensible. They are the most used system descriptions as part of threat modeling [Sho14]. The popular threat modeling tool of Microsoft [Mic21] is also based on DFDs. Various extensions of DFDs exist to support threat modeling regarding various security objectives [TCS18]. The discussion of the state of the art in Chapter 9 gives more examples on DFD-based analyses.

UML activity diagrams can represent data flows by so-called *object flows*. The UML is commonly used to describe software architectures but many users do not strictly adhere to the standard according to a survey with 80 architects [LCM06]. In addition, the UML provides many types of model element but many of them are rarely used or even unknown according to a literature study [Reg+15]. This informal use and the amount of model elements to consider are challenging for automated analyses. As a consequence, we could only find one approach [HSS14] that consequently uses object flows to identify violations of confidentiality requirements.

System descriptions based on Petri nets as well as on DFDs are eligible foundations of the syntax. We decide to use DFDs instead of Petri nets because DFDs are already used in established approaches to identify violations of security requirements, i.e. threat modeling, and the provided types of model elements match domain concepts known by architects better than the types of model elements of Petri nets.

4.2.3. Syntax and Semantics: Propagation of Properties

To detect violations of confidentiality requirements, it is necessary to know, which data is available in which part of the system. More precisely, it is necessary to know the properties of data. It is possible to determine these properties manually by reasoning about data processing and assigning the properties directly to data flows. However, this requires high initial effort as well as high effort if the system changes. Additionally, it is a repetitive and error-prone activity. Therefore, we aim for an automated propagation of data properties as part of the semantics.

The syntax as well as the semantics have to cover data processing to determine their effect on data properties, which is the core of the logic to propagate data properties. To capture the processing effect, it is either possible to assign processing effects to elements based on their type or assign processing effects to individual elements. For instance, Hoisl, Sobernig, and Strembeck [HSS14] assign a propagation behavior to fork or condition nodes in UML activity diagrams. The downside of this approach is that either the propagation rules are limited because of the limited amount of available types of model elements or dedicated models elements for specific purposes have to be introduced into the language. This approach lowers the uniqueness [POB00], which aims for providing a small set of powerful features in contrast to a large set of specific features. Additionally, it hinders extensibility. In contrast, separating the specification of the processing effect and the nodes improves uniqueness and also enables extensibility to effects not considered while developing the syntax and the semantics. Therefore, we plan to assign the processing effect to individual elements.

Especially for the semantics, the types of properties are relevant. The values of a property can range from a set of discrete values, such as boolean variables or enumeration literals, to continuous values, such as integer or real numbers. Continuous values are more expressive but also require more complex data processing specifications and semantics. Discrete values are simpler to handle and are also quite common in predicting confidentiality: Information flow analyses usually use a lattice of security labels [Smi+15]. Access control often decides about access based on discrete values such as roles, security levels or individual access rights [Fur08, pp. 61]. Because most established information flow and access control mechanisms restrict themselves to discrete values, it is reasonable to adopt this limitations to simplify modeling and analyzing systems.

The concept to propagate discrete values is already used in machine learning. In machine learning, these discrete values are called *labels* and the propagation is called *Label propagation*. According to Zhu and Ghahramani [ZG02], label propagation means that a set of unlabeled data is incrementally labeled by deriving new labels from a small initial set of labeled data. This definition matches our understanding of label propagation. However, we will not derive the new labels by similarity, which is the approach initially suggested by Zhu and Ghahramani [ZG02], but we will derive the labels from the effect of data processing. We will use the term *label propagation* in the following to refer to our propagation algorithm.

5. Modeling Confidentiality Characteristics of Systems by Data Flow Diagrams

DFDs are established models for reasoning about the security of systems. However, DFDs are often used in an informal way because they lack means for representing properties and behavior relevant for confidentiality in a precise way. Consequently, there are no formalized semantics supporting automated analyses. Nevertheless, DFDs provide a good foundation for developing our approach as motivated in Section 4.2. This chapter presents solutions on how to bridge the existing gaps and how to meet the requirements given in Section 4.1. All explanations in this chapter are based on a previous publication [Sei+22].

The solution involves defining an extended syntax as described in Section 5.1 as well as corresponding semantics as described in Section 5.2. Together, syntax and semantics provide means for defining automated confidentiality analyses. The implemented syntax as well as the mapping for assigning the semantics to the syntax are available in our data set [Sei22]. The limitations of the proposed solution as well as assumptions regarding its usage are covered in Section 5.3. Eventually, Section 5.4 gives a short summary of the whole chapter.

5.1. Extended Data Flow Diagram Syntax

The simple DFD syntax as introduced by DeMarco [DeM79, pp. 51] is not sufficient for automated detection of confidentiality violations. An extension of the DFD syntax is necessary to meet the requirements for the syntax described in Section 4.1.1. The identified need for an extension is in line with recent security research using DFDs [TCS18; Sio+20].

To structure the extended syntax and illustrate its usage by stakeholders, we define architectural viewpoints, which the ISO/IEC 42010 standard [Int11] suggested to reason about architectural description languages. According to Goldschmidt, Becker, and Burger [GBB12], a viewpoint addresses exactly one concern by potentially multiple view types. A view type is an abstract syntax describing the part of the software architecture that is necessary to address the concern of the viewpoint. Stakeholders are interested in concerns and define, i.e. provide the requirements for, a viewpoint. The requirements described in Section 4.1 address the three major concerns illustrated in Figure 5.1.

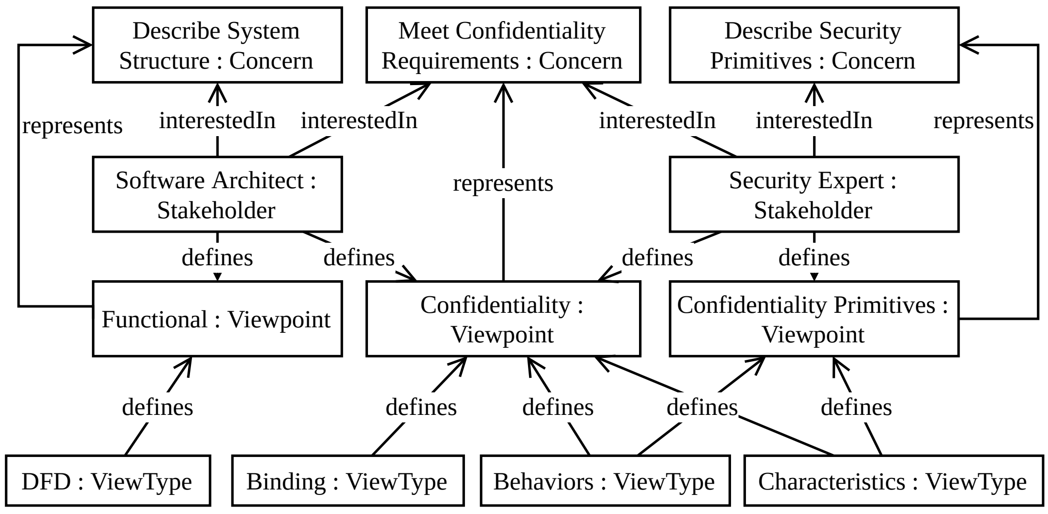

Describe System Structure : Concern
Meet Confidentiality Requirements : Concern
Describe Security Primitives : Concern
represents
interestedIn
interestedIn
interestedIn
interestedIn
represents
Software Architect : Stakeholder
represents
Security Expert : Stakeholder
defines
defines
defines
defines
Functional : Viewpoint
Confidentiality : Viewpoint
Confidentiality Primitives : Viewpoint
defines
defines
defines
defines
defines
DFD : ViewType
Binding : ViewType
Behaviors : ViewType
Characteristics : ViewType

Figure 5.1.: Overview on addressed viewpoints and view types including corresponding concerns and stakeholders given as UML object diagram.

The most fundamental concern of a software architect is to describe the structure of the system. This covers the used components, which can be system parts or actors, including their interfaces as well as the wiring of components. DeMarco calls this the *Functional* viewpoint [DeM79, p. 47] in the context of DFDs, so we stick to this term. Consequently, the DFD is the view type used by architects to address the functional view point. The extensions to this view type that address data processing of system parts and actors (R1.1) as well as the definition of data interfaces are covered in Section 5.1.1.

The security expert is mainly concerned with describing security primitives. In the context of this thesis, these primitives address confidentiality aspects only, i.e. confidentiality properties (R1.2, R1.5) and fundamental behaviors affecting these properties (R1.3). In the running example, the classification level applied to data is such a property and the declassification is a behavior that changes the particular classification of a data item. Consequently, we call the corresponding viewpoint *Confidentiality Primitives*. The *Characteristics* view type covers the confidentiality properties and the *Behaviors* view type covers the behavior types. We describe both view types and how they address many of the requirements on modeling confidentiality aspects in Section 5.1.2.

Both, the software architect and security expert, are interested in meeting the confidentiality requirements in the system under design. The *Confidentiality* viewpoint represents this concern by bridging the gap between pure system structure and pure confidentiality primitives. The software architect can bridge this gap by binding confidentiality primitives of the security expert to the system structure in the *Binding* view type. Additionally, it might be necessary to define confidentiality properties and behaviors that are specific for the particular system. In the running example, the particular clearance and classification levels depend on the system, so they are no generic confidentiality primitives. The software architect and the security expert work together in defining these system-specific properties and behaviors. Additionally, the security expert might assist in binding the confidentiality

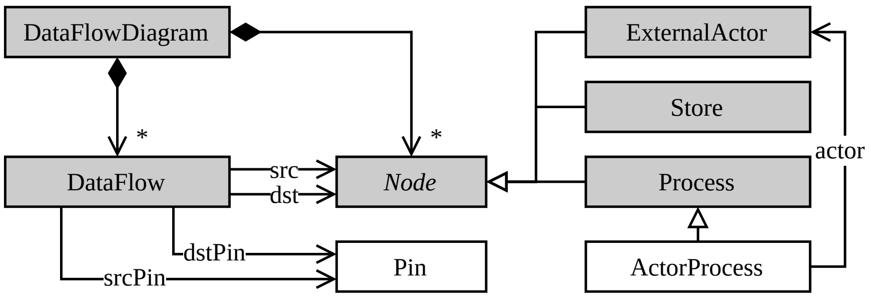


Figure 5.2.: Data Flow Diagram view type for the Functional viewpoint given as UML class diagram.

primitives to elements of the software system. Section 5.1.3 focuses on the *Binding* view type but also discusses interactions between security experts and software architects.

A summarizing overview on how the extended syntax presented within the aforementioned viewpoints meets the requirements presented in Section 4.1 is given in Section 5.1.4.

5.1.1. Functional Viewpoint

The Functional viewpoint represents the concern of the software architect to describe the system structure. The view type that the architect uses within the viewpoint is an extended version of the DFD as defined by DeMarco. The shaded elements in Figure 5.2 represent the elements in the original definition, which already meet R1.1 to a large scale. In the following, we discuss how the extensions illustrated by non-filled elements in Figure 5.2 meet the remaining aspects of the requirement.

In realistic systems, a system function is often used by various other system functions. In control flow models, this would mean calls from multiple locations. In data flow models, the equivalent to calls is sending data to a node. Therefore, using the very same node means that there are multiple incoming data flows for the same type of required information, i.e. there are alternative flows to choose from. However, there are no means for distinguishing such alternative flows from mandatory flows. A clear interface definition can solve this by defining types of mandatory incoming data. If more than one data flow for the same type of data arrives, these are alternative flows. To realize this, we introduce the concept of a `Pin`. A `Pin` describes a required input or provided output of a `Node`. All data flows between nodes have to use these pins to avoid ambiguities. This means that nodes have to send all data through pins. In Figure 5.3, we visualize these pins by empty squares at the edges of nodes. Pins, which are the target of a data flow are input pins. Pins, which are the source of a data flow are output pins. Because incoming or outgoing data is the only way to communicate between nodes, the set of pins of a node defines its interface. Multiple data flows arriving at a pin mean alternative sources of equivalent data. This means that more than one node can provide equivalent data to another node, which is roughly the same as calling the same operation from different sources in the control flow terminology. Pins and the definition of interfaces are a commonly known and established concept, which is also part of the UML [Obj17, p. 515]. The benefit of using a pin over

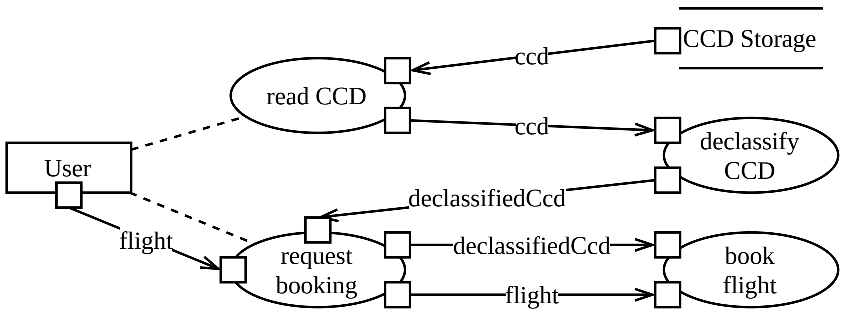


Figure 5.3.: Excerpt of running example using the actor processes to describe data processing steps of actors as well as pins to describe data interface.

other concepts such constraints for selecting flows or defining sets of valid flows is the low effort for adding a new, mandatory input. To do so, it is sufficient to just add a new pin. When using constraints, it would be necessary to adjust every constraint to include the new mandatory input. The same holds for set definitions.

Besides the data processing of the system, R1.1 explicitly mentions the data processing done by actors. To consider their data processing steps, we added the additional node type ActorProcess. The new node type is essentially a process just like the ones used to describe the data processing of the system but holds an additional reference to an actor. The intended meaning of this reference is that the process belongs to the data processing activities carried out by the referenced actor. The benefit of this modeling approach is that software architects can use the already known modeling elements for describing data processing steps for the system and actors. Nevertheless, the new ActorProcess maintains a clear distinction between processes of systems and processes of actors. To give an example, we demonstrate the usage of the actor processes in an excerpt of the running example shown in Figure 5.3. The dashed line visualizes the reference of an ActorProcess to the corresponding actor. The processes *read CCD* and *request booking* belong to the *User* and describe the corresponding data processing steps: First, the user reads the credit card data and sends it to the declassification. Next, the received declassified data is passed to the *book flight* process together with the flight to be booked. Without these additional processes, the user would directly receive and emit credit card data, which requires distinguishing the declassified and regular credit card data by the name of the data flow. In contrast, the additional processes allow a clear, structural separation between regular and declassified credit card data because the data processing done by the user is clear now.

The requirement to represent deployment information (R1.1) will be addressed by the mechanism to express properties of nodes. Deployment information is not different to other properties relevant for confidentiality as long as the sole purpose of representing it is considering it in finding violations of confidentiality requirements.

5.1.2. Confidentiality Primitives Viewpoint

The Confidentiality Primitives viewpoint represents the concern of the security expert to describe security primitives. Thereto, the security expert uses the *Characteristics* and *Behaviors* view type. The following paragraphs explain how these view types support the security expert and how they address many requirements regarding the definition of confidentiality properties (R1.2 and R1.5) and behaviors (R1.3). Both view types do not introduce concepts, which are specific for a particular confidentiality mechanism. Instead, a security expert uses the generic concepts to introduce specific information (R1.7), which also means that various confidentiality mechanisms can be mixed within one model (R1.9).

5.1.2.1. Characteristics View Type

The characteristics view type provides means to describe available property types (R1.2) and particular properties (R1.5). In the following, we use the term *characteristic* to refer to properties of nodes and data to distinguish arbitrary, untyped properties describing aspects of nodes and data from strongly typed characteristics that are relevant for confidentiality. Using an unstructured set of properties would be possible but increases the specification effort when propagating and comparing properties. The increased effort stems from the missing ability to refer to groups and thereby treat a set of properties in the same way. In unstructured properties, there has to be a dedicated rule for handling every property. In our running example, comparing the classification and clearance properties would require dedicated logic for every element of the cross product of these properties. Instead, types and value ranges within properties can ease such comparisons. In our running example, it would be sufficient to define the levels as ordered value range and compare the index of the classification property with the index of the clearance property.

As Figure 5.4 illustrates, an Enumeration defines the range of values. The enumeration holds a set of Literals. In the running example, the enumeration would be called *Levels* and the literals would be the particular levels such as *User* or *User,Airline*. A CharacteristicType is the type of a characteristic (R1.2). It uses an enumeration to define its range of values. Separating the characteristic type and its used value range is beneficial because this allows to reuse the value range in multiple characteristic types. In the running example, there are two characteristic types: *Clearance* and *Classification*. Both characteristic types share the same range of values, which are the *Levels*.

A Characteristic defines an instance of a characteristic type (R1.5). It refers to a particular type and selects a set of applied literals. Selecting a set instead of a single literal is a convenient way to avoid the need for defining multiple characteristics for the same type. For instance, the selection of multiple literals is useful for assigning roles to a node. The actual binding of such a characteristic to a node is covered by the *Confidentiality* viewpoint (see Section 5.1.3).

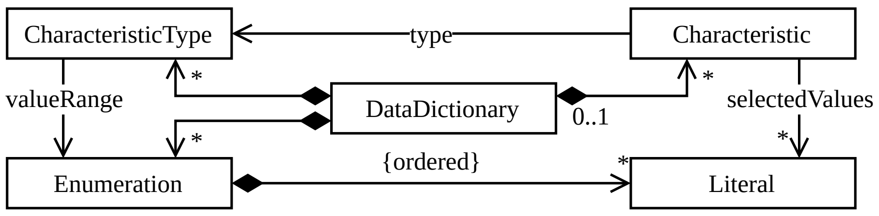


Figure 5.4.: Characteristics view type for the Confidentiality Primitives viewpoint given as UML class diagram.

With respect to the planned label propagation approach mentioned in Section 4.2, a label is the tuple of characteristic type and a literal. This is crucial because the same literal can have different meanings depending on the corresponding characteristic type. In the running example, the *User* label on data means that the data is classified for users. The *User* label on a node means that the node is cleared for data classified by level *User* at most. A characteristic referring to multiple literals implies a label for each literal, i.e. a tuple of characteristic type and selected literal. Making type and value explicit, makes comparisons type-safe and definitions of label propagation functions simpler. We will illustrate the latter later by a so-called *wildcard* mechanism.

Labels, i.e. characteristic types and literals, can be specific to a particular system or generic. The labels in the running example are specific to the system because the underlying levels are named based on the involved nodes, which also applies to the labels. Therefore, these labels are not reusable in other systems. In contrast, generic labels named *high*, *medium* or *low* could express the confidentiality requirements in the same way but would support reuse in further systems. The DataDictionary builds a catalog of characteristic types, enumerations and characteristics. This catalog can contain reusable or specific elements. This lowers the specification effort because confidentiality primitives only have to be defined once but can be reused in various systems. In the running example, there is also a catalog but it will not be used by other systems because the contained elements are not reusable. However, changing the literals from system-specific to generic labels would enable reuse.

5.1.2.2. Behaviors View Type

The behaviors view type covers behavior descriptions of nodes by means of label propagation functions (R1.3). A label propagation function describes how outgoing labels can be derived from incoming labels.

Before describing the modeling approach for such functions, we describe the underlying idea in an informal way: The fundamental design decisions influencing our solutions are that i) a label is a tuple of a particular characteristic type and particular literal and ii) all data flows have to go through pins. Consequently, labels leave and enter a node through pins, so it is reasonable to always refer to pins when defining outgoing labels or using incoming labels. Because there is a fixed set of available, discrete labels, the availability of a particular label can be seen as a particular boolean variable with the value *true*. If the label is not available, the value would be *false*. Consequently, the set of available labels at a particular pin is just a set of boolean variables describing the availability for this particular pin. A label propagation function is essentially a sequence of assignments of truth values to the boolean variables representing labels on output pins. The truth value of the assignment can depend on boolean variables of incoming pins. The benefit of assignments and truth values is that even complex logic can be realized by boolean logic in an intuitive way.

The view type shown in Figure 5.5 realizes the idea of describing the label propagation by a sequence of assignments and encapsulates it in a reusable BehaviorDefinition. This definition consists of the set of input pins and output pins as well as the assignments. This is reasonable because a label propagation function specifies labels for an output and relies on incoming labels, so it assumes that a defined set of inputs and a defined set of outputs is available. Both are essentially sets of pins. The list of assignments is ordered, which means that an assignment to a variable in position n can override an assignment to the same variable in position m if $n > m$. An assignment always consists of a left-hand side (lhs) and right-hand side (rhs). The left-hand side is the boolean variable that shall be set. This variable is given by a DataCharacteristicReference, which is a triple of pin, characteristic type and literal. On the left-hand side, the pin must be an output pin because a node can only affect outputs by its data processing. To simplify descriptions, we use the concrete syntax `pin.ct.l` to refer to the boolean variable specified by the pin `pin`, characteristic type `ct` and literal `l`. The right-hand side of the assignment is a Term. It is reasonable to allow operations and truth values of boolean algebra to be terms. The supported operations are Not, And and Or. The supported truth values are True and False. Additionally, referring to boolean variables, which means labels on input pins, is necessary. Labels from input pins can be referenced by a DataCharacteristicReference. On the right-hand side, such a reference must refer to an input pin because labels on output pins are not available while determining the very same labels on the output pins. In addition, labels on the node, which is using the BehaviorDefinition, can be referenced by the ContainerCharacteristicReference that requires a characteristic type and a literal to be given.

In the running example, the nodes *filter flights* and *process booking* have a label propagation function that applies the highest received classification label to the output. This function breaks down to three assignments for the three possible classification levels on the output as shown in Listing 5.1. Line 1 means that the highest classification level *User* can be assigned if one of the inputs (`in1` or `in2`) has this classification level. Line 8 means that the lowest classification level *UserAirlineTA* can be assigned if both inputs have this label. The remaining lines mean that the medium classification level *UserAirline* can be assigned if one input has this medium level and the other input has the medium or the lower level.

A wildcard mechanism allows omitting characteristic types and literals to safe specification effort in some cases. If an element is omitted, we write $*$ instead of the element in the examples. We demonstrate the benefit in a moment after explaining the general idea. The mechanism requires the characteristic types and literals to be omitted on the left-hand side and the right-hand side. The meaning of omitted information is as follows: A missing literal means that there is virtually one assignment for each literal of the corresponding

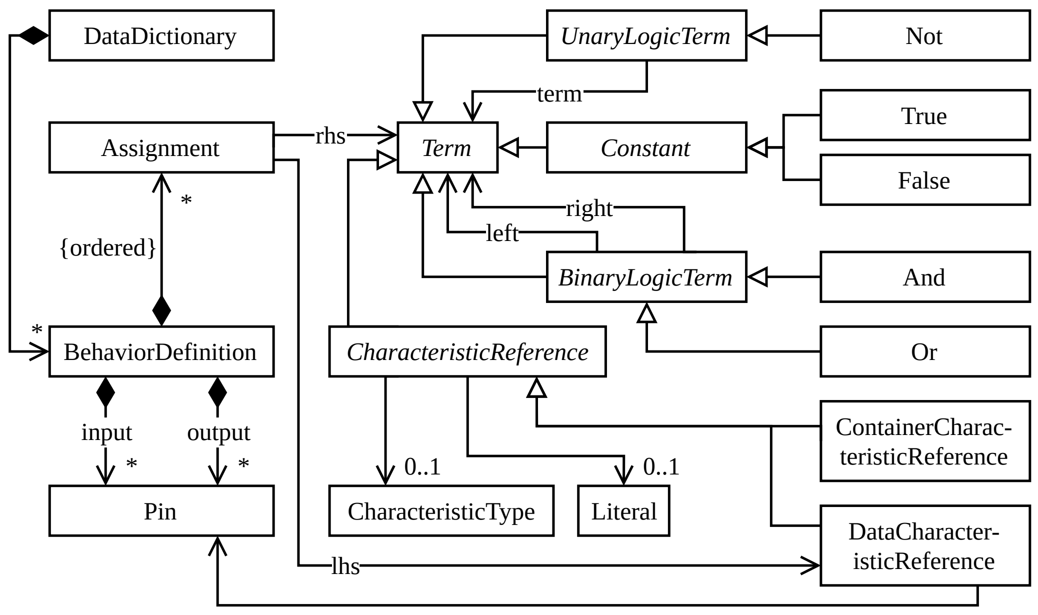


Figure 5.5.: Behaviors view type for the Confidentiality Primitives viewpoint given as UML class diagram.

Listing 5.1: Example of assignments describing the behavior for joining two inputs with respect to data classification.

```
out.class.User := in1.class.User OR in2.class.User
out.class.UserAirline :=
   (in1.class.UserAirline AND
      (in2.class.UserAirline OR in2.class.UserAirlineTA))
   OR
   (in2.class.UserAirline AND
      (in1.class.UserAirline OR in1.class.UserAirlineTA))
out.class.UserAirlineTA := in1.class.UserAirlineTA AND in2.class.UserAirlineTA
```

characteristic type. In each of these assignments, the particular literal is inserted in all empty literal places. In our running example, a missing literal in a reference to the classification characteristic would virtually result in three assignments: one assignment for the *User* level, one for the *UserAirline* level and one for the *UserAirlineTA* level. The precondition to use literal wildcards is that all references omitting the literal reference a characteristic type with the same enumeration, i.e. the characteristic types have the same set of available literals. This is a reasonable precondition because otherwise characteristic references could become invalid, i.e. the literal would not match the characteristic type. The example shown in Listing 5.2 demonstrates the use of literal wildcards. The assignment in line 1 specifies that the output `out` should have the classification literals enabled that match the enabled clearance literals of the container. Such an assignment might be useful when data is created because it is reasonable to assume that the created data has the same classification level as the clearance of the creating node. Lines 2 to 4 are equivalent to line 1. As this example demonstrates, only one instead of three assignments have to be specified, which saves effort.

Listing 5.2: Example of assignments illustrating use of wildcards to specify the creation of data.

```
out.class.* := container.clear.*
out.class.User := container.clear.User
out.class.UserAirline := container.clear.UserAirline
out.class.UserAirlineTA := container.clear.UserAirlineTA
```

Listing 5.3: Example of assignments illustrating use of wildcards to specify forwarding of data.

```
1 out.*.* := in.*.*
2 out.class.* := in.class.*
3 out.clear.* := out.clear.*
```

Assignments can also leave out characteristic types together with leaving out literals. A missing characteristic type means that there is virtually one assignment for each characteristic type, in which the particular characteristic type has been inserted. After that, the logic for handling omitted literals described before applies. The example shown in Listing 5.3 illustrates such an assignment. The assignment in line 1 specifies that exactly all labels of the input apply to the output. The lines 2 and 3 are equivalent to line 1, assuming that there are two characteristic types to describe classifications *class* and to describe clearance *clear*. The amount of saved assignments increases if the amount of available characteristic types increases.

5.1.3. Confidentiality Viewpoint

The Confidentiality viewpoint addresses the concern of both, software architect and security expert, to meet confidentiality requirements in the system under design. Three view types support the work of both stakeholders: The Characteristics and Behaviors view types have already been introduced previously. In the context of the Confidentiality viewpoint, these view types allow software architects to support security experts in defining system-specific behaviors and characteristics, which includes deployment information represented as characteristics on nodes (R1.1). The *Binding* view type is the third and new view type in the Confidentiality viewpoint, which we describe in the following.

The *Binding* view type shown in Figure 5.6 allows the software architect to bind behaviors and characteristics to nodes in the architecture. Thereto, all DFD nodes presented in the Functional viewpoint must reference exactly one BehaviorDefinition and can reference multiple Characteristics. All involved elements have already been created in other view types like explained before. To bind labels, i.e. characteristics, to a node (R1.6), the architect adds the corresponding characteristic to the list of referenced characteristics. To bind a label propagation function, i.e. a behavior definition, to a node (R1.4), the architect sets the reference to the behavior to the particular behavior definition. This means that architects use the same means for assigning characteristics and behaviors to nodes no matter what confidentiality mechanism is used (R1.8). Mixing confidentiality mechanism within one model is also possible (R1.10).

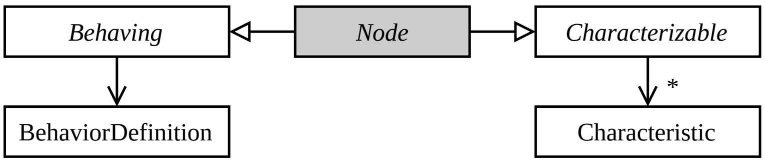


Figure 5.6.: Binding view type for the Confidentiality viewpoint given as UML class diagram.

There is one constraint for binding a behavior to a node: If the node is an actor, the behavior must not include assignments that use *DataCharacteristicReferences* on the right-hand side of an assignment. Doing so would imply that the labels on output pins depend on labels on input pins. However, this would mean that an actor is no longer a source or sink of data flows as initially defined by DeMarco. In our running example, this situation appears when the user sends credit card data to the flight booking process. Certainly, the credit card data sent to the flight booking process is the same as the data received from the credit card center, so the labels should be the same. In such cases, *ActorProcesses* can establish this relation without violating the source or sink role of an actor.

5.1.4. Requirements Coverage by Viewpoints

The previously described viewpoints and the view types supporting them address most of the requirements regarding the syntax from Section 4.1.1. Table 5.1 gives a summarizing overview on how the syntax meets the requirements.

The requirements R1.1 to R1.6 demand syntax extensions to represent information. As the table illustrates, the extended DFD can represent the requested information. The requirements R1.7 to R1.11 define how the extended DFD shall represent the information or how the architect and expert use the extensions. The DFD meets the requirements about a unique representation of information (R1.7 and R1.8) and about representing information specific for particular confidentiality within a single artifact (R1.9 and R1.10) by generic modeling concepts for representing information specific for particular confidentiality mechanisms. The last requirement about a low effort for switching confidentiality mechanisms (R1.11) is met by the separation of the system structure and confidentiality-specific information. It is possible to remove or replace the confidentiality-specific information without the need for remodeling the whole system. This lowers the effort compared to approaches that only support one particular confidentiality mechanism. However, this aspect is also part of the validation described later.

5.2. Extended Data Flow Diagram Semantics

Clear semantics enable automated reasoning about properties of a model. DFDs as defined by DeMarco do not have such clear semantics but an intuitive definition. As a consequence, various approaches [Jil+08] to formalize DFD semantics have been made. None of these

ID	Description	User	Syntax Extension
R1.1	structure, usage, deployment, behavior	architect	actor processes, node characteristics
R1.2	property types	expert	characteristic types
R1.3	behavior types	expert	behavior definition
R1.4	bind behavior	architect	behavior reference for nodes
R1.5	define properties	expert	characteristics
R1.6	bind properties to nodes	architect	characteristics reference for nodes
R1.7	behavior/property types by same concept	expert	characteristic/behavior types
R1.8	bind behaviors/properties by same concept	architect	references to characteristic/behavior types
R1.9	multiple confidentiality mechanisms in same artifact	expert	characteristic/behavior types
R1.10	multiple confidentiality mechanisms in same artifact	architect	references to characteristic/behavior types
R1.11	low effort switching confidentiality mechanisms	architect	references to characteristic/behavior types

Table 5.1.: Overview on extended syntax elements and met requirements by DFD syntax.

semantics provide commonly agreed universal semantics for DFDs but provides semantics tailored to particular use cases. We also do not attempt to define such universal semantics because this would not provide any benefit compared to tailored and concise semantics. The semantics for the extended DFDs provide means for describing the label propagation mechanism, which we motivated in Section 4.2.3. Security experts can use the analysis framework built on the semantics to formulate an analysis goal (R2.4) that reveals violations but we will describe building analyses as well as particular analyses for information flow (R2.8) and access control (R2.9) in Chapter 6. The chapter will also describe ways of how the software architect can formulate an analysis goal (R2.5).

We first explain the semantics in an intuitive way by demonstrating its meaning for the running example in Section 5.2.1. Afterwards, we formalize the semantics in first-order logic and describe the mapping between the extended DFD syntax and the semantics in Section 5.2.2. We explain how the semantics meet the requirements regarding the semantics in Section 5.2.3.

5.2.1. Intuitive Semantics of Extended Data Flow Diagrams

We already explained the meaning of the language constructs of the extended DFD syntax as part of the introduction of the syntax in Section 5.1 to foster comprehensibility. In this subsection, we focus on explaining the label propagation mechanism and detection goal definition in an intuitive way by example.

Listing 5.4: Assignments of credit card data initially created by the user.

```
ccd.class.User := true
```

Listing 5.5: Assignments of forwarding and store behavior in running example.

```
out.*.* := in.*.*
```

We use the version of the running example shown in Figure 5.7 that adopts the extended DFD syntax. The difference to the plain DFD is the usage of pins, actor processes and behaviors. A square at the border of a node represents a pin. Input pins only have incoming edges and output pins only have outgoing edges. The actor processes are shown within the corresponding actors. The behavior is shown by a symbol ($\twoheadrightarrow$, $\rightarrowtail$, $\rightsquigarrow$) within the node. For a sake of simplicity, the figure does not include node characteristics because the clearance of a node is the only applied characteristic. The clearance is already visible by the gray filled headings in the top of the figure: Nodes lying under the *CCC*, *User* or *TravelPlanner* heading have a clearance for the *User* level. Nodes under the *TravelAgency* heading have clearance for the *UserAirlineTA* level. Nodes under the *Airline* heading have clearance for the *UserAirline* level.

The violation to detect in the example is that a node with clearance level n accesses data with classification level m, where $n < m$. The levels are the already known levels *User*, *UserAirline* and *UserAirlineTA*. The order of these levels is given by their order of definition in the corresponding enumeration. Figure 5.8 illustrates this order by an annotated index. To give a concrete example, a node with clearance for *UserAirline* ($n = 1$) accessing data classified for *User* ($m = 2$) means a violation ($1 < 2$).

In order to detect such violations in the DFD, we have to know all labels of all nodes and all labels of all data. The label propagation mechanism calculates all of these labels by propagating a set of initial labels through the network of nodes. We cover the exact procedure by the formalization in Section 5.2.2. In the following, we only describe the underlying principle by example.

Assume, the propagation starts at the pin of the *User* yielding *ccd.* Because the user creates and enters credit card data, there is no other source for deriving labels. Therefore, the data needs an initial classification label that is specified by the excerpt of the behavior of the user in Listing 5.4. The assignment explicitly sets the classification to the *User* level by applying the corresponding label to the *ccd* output. Consequently, the data arriving at the *CCD Storage* is classified for the user. The behavior of storages is essentially a forwarding behavior ($\twoheadrightarrow$). As shown in Listing 5.5, this behavior takes the input labels and applies them to the output. This means that the data yielded by the *CCD Storage* is also classified for the *User* level.

The semantics of multiple data flows starting at an output pin is that the labels are propagated to all destinations. Consequently, the *User* label from the *CCD Storage* is now propagated to the *declassify CCD* process as well as the *User.* The *declassify CCD* process has the declassification behavior ($\rightsquigarrow$) shown in Listing 5.6. All assignments are applied in

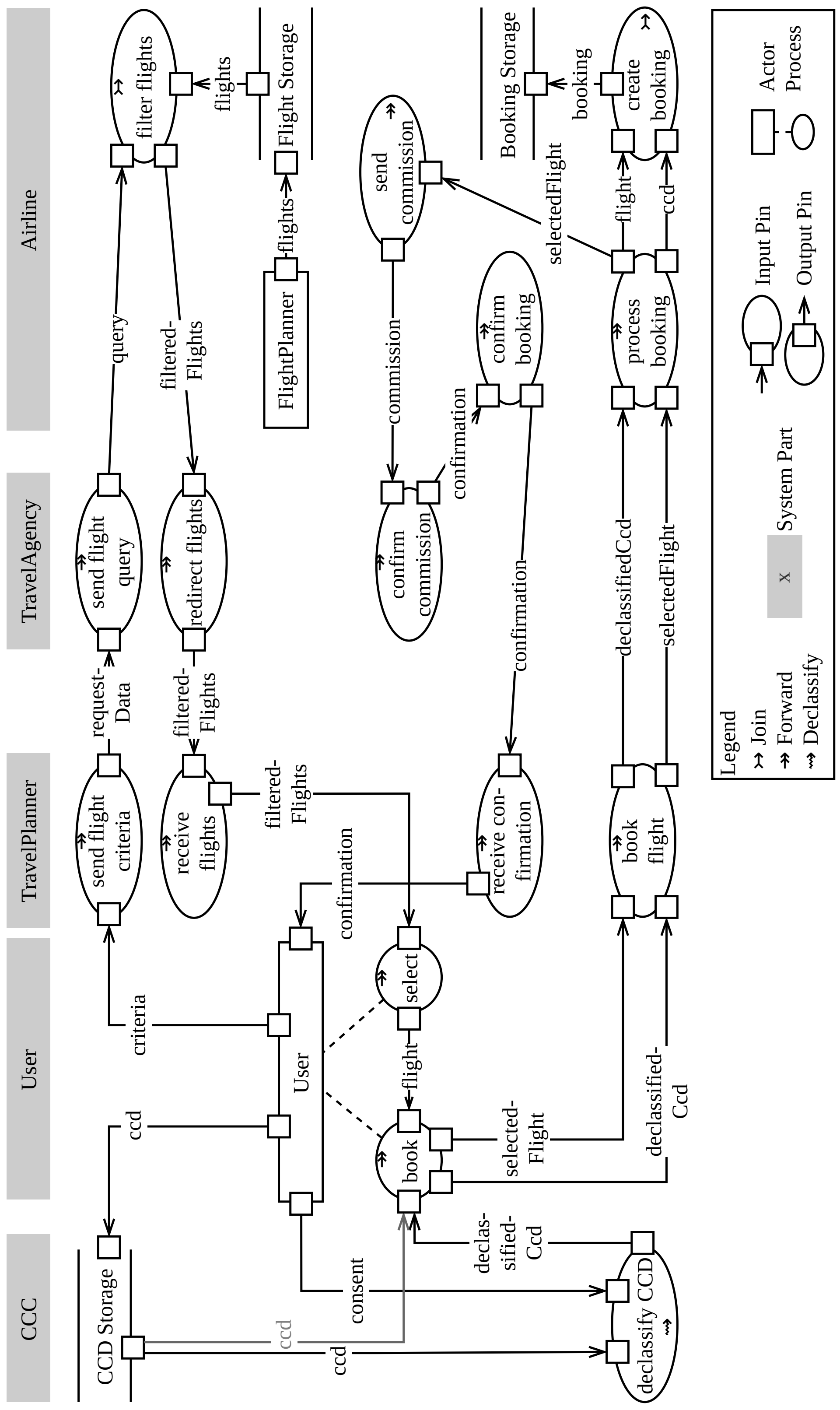


Figure 5.7.: Running example using extended data flow diagram syntax.

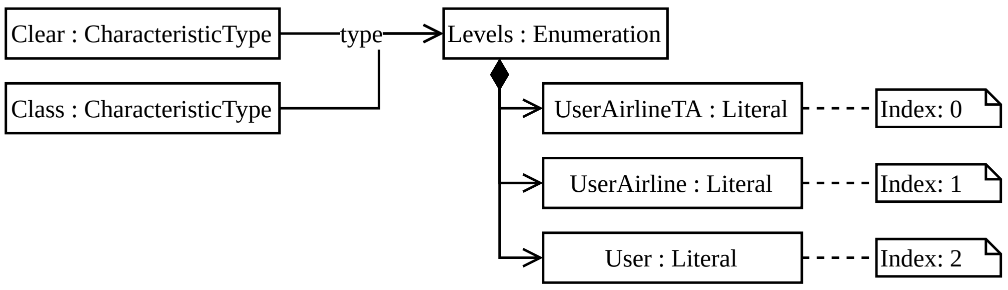


Figure 5.8.: Characteristic types used in running example given as UML object diagram.

Listing 5.6: Assignments of declassification behavior in running example.

```
out.*.* := in.*.*
out.class.* := false
out.class.UserAirline := true
```

the given order before labels are emitted. First, all labels are copied from the input to the output. This is essentially the forwarding behavior presented before. Next, the behavior has to change the classification level. Thereto, it first deletes all classification labels and applies the *UserAirline* label. The effect of this behavior is that all labels are copied unchanged but the classification labels are replaced by a new label. At the *declassify CCD* process, the incoming credit card data classified for *User* will be classified for *UserAirline* after the processing. Labels of the incoming *consent* data are not important for the resulting labels, so it is not considered in the assignments.

Next, the credit card data arrives at the input pin for credit card data at the *User*. Multiple incoming data flows on the same pin mean that these data flows are alternatives. Because we are interested in all potential violations, the label propagation mechanism has to consider all possible choices. For the sake of simplicity, we assume that the mechanism creates *n* DFDs for *n* possible input flows. Actually, the mechanism is more efficient, which the formalization in the next section will show. In the particular example, the mechanism creates two DFDs: one DFD selects the credit card data from the storage and the other one selects the data from the declassification process. In these two DFDs, there are different labels available at the pin: The DFD considering the data flow from the storage has the *User* label available at the pin. The other DFD has the *UserAirline* label available at the pin. The approach of creating a DFD for every alternative yields multiple deterministic DFDs, i.e. DFDs without alternative data flows. The detection of violations has to consider all of these DFDs.

For the sake of brevity, we do not discuss all propagations in detail but assume that all propagations took place. The result are two fully labeled DFDs. In the first DFD, the credit card data received by the user is classified by the *User* label. Consequently, the credit card data arriving at the *process booking* process is classified the same. This implies a violation because the *process booking* process has only a clearance label for *UserAirline* but the arriving data is classified with the higher *User* label. In addition, a further violation appears on the *create booking* process and the *Booking Storage* because of the same reason:

The *create booking* process joins (⤚) the flight and the credit card data as specified by the join behavior in Listing 5.1 on page 42, i.e. the highest incoming classification (*User*) is applied to the outgoing classification. The *Booking Storage* receives the booking classified by the *User* label but the storage only has a clearance label for *UserAirline*. In the second DFD, there is no violation because the credit card data has been declassified and the classification of the credit card data leaving the user is *UserAirline*. This is exactly the same level as the clearance of the following nodes.

To summarize, a set of initial labels is given by behaviors assigning these labels explicitly. The label propagation starts from these initial labels and sequentially applies the label propagation functions. Whenever there are multiple incoming data flows for a pin, there will be multiple DFDs, which only have exactly one incoming data flow for the pin. The detection of violations takes place on each resulting DFD.

5.2.2. Formalization of Semantics in First-Order Logic

We formalize the previously introduced informal semantics by first-order logic. To do so, we describe the meaning of every element of the extended DFD metamodel by a set of clauses (R2.1). This description also serves as specification of the mapping between syntax and semantics. We use the notion of Prolog as syntax for describing these clauses because it is concise and yields ready to use logic programs. We use these logic programs to execute the label propagation and label comparison in order to build automated analyses (R2.7) as we show in Chapter 6.

In the following, we structure the description of the semantics by the view types introduced in the syntax description in Section 5.1. This is reasonable because view types describe subsets of the extended DFD metamodel and the formalization is about specifying the meaning of metamodel elements. Section 5.2.2.1 describes the DFD view type, which is about structural elements of DFDs. The characteristics view type mainly concerned with the definition of characteristic types and instances of them is covered in Section 5.2.2.2. The semantics of the behaviors defined in the behaviors view type is covered in Section 5.2.2.3. The formalization of behaviors is also the formalization of the label propagation (R2.2). We do not cover the binding view type explicitly because it is mainly concerned with binding behaviors and characteristics to structural elements. Instead, we assume that this binding already exists in the descriptions of the remaining view types. We also do not cover the goal definition (R2.4 and R2.5) and do not discuss how to build particular information flow and access control analyses (R2.8 and R2.9) in this chapter but discuss all of them as part of the definition of particular analyses in Chapter 6.

5.2.2.1. Functional View Type

The functional view type covers the structure of the DFD. The only semantic aspect to cover for the structure is to formalize that a structural DFD element of a certain type exists.

Listing 5.7: Prolog facts describing nodes of a data flow diagram.

```
1 % all variables (sequence of capital letters) become constants
2 actor(N).                                % ExternalActor
3 store(N).                                % Store
4 process(N).                              % Process
5 actorProcess(N, A).                      % ActorProcess
6 inputPin(N, PIN).                        % Pin (in input reference)
7 outputPin(N, PIN).                       % Pin (in output reference)
8 dataflow(F, NSRC, PINSRC, NDST, PINDST). % DataFlow
```

Establishing the existence of structural elements is beneficial because analysis goals can consider the various types of elements e.g. by only looking for violations at stores.

We use the facts as shown in Listing 5.7 to do so. The comments after facts describe the corresponding class of the DFD metamodel. For every instance of such a class, we create one particular fact that replaces the variables with particular identifiers. In our running example, we create two facts describing the actors *User* and *FlightPlanner* by replacing `N` in the fact in line 2 with a unique identifier for the particular external actor. We do the same for all elements shown in listing 5.7. `N` is always replaced with the unique identifier of the corresponding node. In line 5, we replace `A` with the identifier for the actor to which the actor process belongs to.

For input and output pins, we create one fact for every node that uses a behavior containing the pin. For instance, if n nodes refer to the same behavior containing i input pins and o output pins, we create $n * i$ facts for input pins (see line 6) and $n * o$ facts for output pins (see line 7). The effect of the label propagation does not depend on whether a behavior is reused or not, so we do not represent reuse information in the semantics for a sake of simplicity. For input pins and output pins as shown in lines 6 and 7, we replace `N` with the identifier of the node that holds the behavior that contains the pin and replace `PIN` with a unique identifier of the pin, i.e. the identifier uniquely identifies a particular pin at a particular node without the need to know `N`.

For every data flow, we create a fact as shown in line 8. The fact represents all information from the metamodel. We replace `F` with a unique identifier of the corresponding data flow, `NSRC` with the identifier of the source node, `PINSRC` with the identifier of the source pin, `NDST` with the identifier of the destination node and `PINDST` with the identifier of the destination pin. The meaning of the data flow fact is that there exists a data flow `F` from the source pin `PINSRC` of node `NSRC` to the destination pin `PINDST` of the destination node `NDST`.

5.2.2.2. Characteristics View Type

The characteristics view type introduces characteristic types and characteristics. Along the lines of DFD nodes and edges, we have to define the existence of characteristic types

Listing 5.8: Prolog facts describing characteristic types and characteristics.

```
1 % all variables (sequence of capital letters) become constants
2 characteristicType(CT).            % CharacteristicType
3 characteristicTypeValue(CT, V, I).  % Literal (transitively referenced by CT)
4 nodeCharacteristic(N, CT, V).       % Characteristic (referenced by Node)
```

Listing 5.9: Prolog facts describing the classification characteristic type of the running example.

```
characteristicType('class').
characteristicTypeValue('class', 'UserAirlineTA', 0).
characteristicTypeValue('class', 'UserAirline', 1).
characteristicTypeValue('class', 'User', 2).
```

and characteristics existence. We do so by the facts shown in Listing 5.8. The comment after the fact describes the corresponding class of the DFD metamodel.

The facts in lines 2 and 3 of Listing 5.8 define the existence of a characteristic type. For every instance of such a characteristic type, we create one particular fact that replaces the variable `CT` with a particular identifier for that characteristic type. In our running example, we create two facts defining the existence of the characteristic types representing the clearance and classification by replacing `CT` in line 2 with a unique identifier for the corresponding characteristic type. The meaning of the `characteristicType` fact is that there exists a characteristic type with an identifier `CT`.

The fact in line 3 represents the literals available for a characteristic type. We do not represent enumerations but directly relate the characteristic type to the literals of the enumeration referenced by the characteristic type. Representing the enumeration is not required for the label propagation, so we omit it to simplify the label propagation logic. In the fact shown in line 3, we replace `CT` by the identifier of the characteristic type, `V` by the identifier of the literal and `I` by the index of the literal in the enumeration referenced by the characteristic type. This means that there are $n * l$ facts representing literals if there are n characteristic types referring to the same enumeration that holds l literals. For instance, the characteristic type describing the classification of data in our running example yields the facts shown in Listing 5.9. The meaning of the `characteristicTypeValue` fact is that there is a literal `V` with index `I` that belongs to a characteristic type `CT`. Representing the index is beneficial because analyses can use the order of literals. In our running example, the literals are ordered in a way that literals with higher indexes are semantically bigger than literals with lower indexes. Therefore, an analysis can use this order to compare literals with each other.

The fact in line 4 represents characteristics bound to nodes (R2.2). We only represent bound characteristics because unbound characteristics do not impact the label propagation. For every binding of a characteristic to a node, we create a fact that replaces `N` with the identifier of the node to which the characteristic is bound, `CT` with the identifier of a characteristic type and `V` with the identifier of a literal. This means that we create $l * n$ facts if a characteristic containing l literals of a characteristic type is bound to n nodes. The

Listing 5.10: Prolog clause documenting the type of behavior of a node.

```
% all variables (sequence of capital letters) become constants
behavior(N, B).
```

meaning of `nodeCharacteristic` is that the literal `V` of characteristic type `CT` is available on node `N`.

5.2.2.3. Behaviors View Type

The behaviors view type represents the label propagation functions by sequences of assignments. Therefore, the semantics described in the following define the behavior of nodes rather than their pure existence. First, we explain and formalize the handling of alternative data flow paths. As motivated by the running example, it is necessary to consider all possible combinations of alternative data flow paths in order to identify possible violations. We explain what an alternative flow path is and define the term *flow tree* as representation of alternative data flow paths originating from a particular node. The *flow tree* provides means to identify the source of data and its properties (R2.3 and R2.6). Afterwards, we specify how the label propagation works for input pins and output pins. To increase efficiency in presence of alternative flow paths, we replace forward label propagation by backward *label lookup*. As part of the semantics definition for output pins, we also cover how the assignments of the behavior definitions map to Prolog clauses. Together, these semantics provide means to determine the properties of data based on label propagation (R2.2).

Definition of Behavior The behavior definition covers the pins and the assignments. To make the used behavior explicit in the semantics, we introduce a predicate `behavior/2`. For every node, we add a fact as shown in Listing 5.10 for every node. The fact uses the identifier of the node as a first argument and the identifier of the behavior, which the node uses, as second argument. The main purpose of these facts is to be able to lookup nodes by their used behavior. The effect of data processing on data and required foundations to formalize this effect are discussed in the following paragraphs.

Alternative Data Flow Paths A data flow path is a sequence of data flows that a data item traverses to reach a certain pin of a node. Multiple data flows targeting the same pin on the same node mean that they all provide the same required type of data, i.e. they are alternative inputs. Assignments can refer to data on every input pin, so we require that a node selects at least one data flow for every input pin to guarantee that the assignments can be made. In order to detect all possible violations, we have to consider all possible combinations of alternative input flows for all pins on a node. This means $\prod_{i=1}^{n} f_i$ combinations if n is the amount of input pins of a node and f_i is the amount of alternative flows for pin i with $i \in \mathbb{N}^*$. In the running example extended by pins in Figure 5.7 on page 47, the pin of the *User* actor that receives credit card data has two alternative inputs, so both flows have to be considered. Because data travels along a path of data flows through the system, all possible combinations of alternative flows on every node on this path have to be considered as well.

Alternative Data Flows We formalize the exploration of all possible input combinations by the clauses shown in Listing 5.11. The goal of the rule in line 1 is twofold: First, it finds a set of input flows `FS` that contains exactly one data flow for every input pin of node `N`. Second, it finds the input flow `FLOW` from this set that targets the given pin `PIN`. The rule is capable of finding all possible sets of input flows, which are available when reevaluating the rule. After evaluating the rule, the variables `FS` and `FLOW` are bound to a list of flows and a flow, respectively. To give an example, this rule can yield two sets of variable bindings for the actor process *book* in our running example shown in Figure 5.7 on page 47: One binding set binds `FLOW` to the *ccd* data flow originating from the *CCD Storage* and binds `FS` to the set of this flow and the data flow coming from the actor process *select*. The other binding binds `FLOW` to the *declassifiedCcd* data flow originating from the *declassify CCD* process and binds `FS` to the set of this flow and the data flow coming from the actor process *select*.

To achieve this, the rule requires a set of clauses to be fulfilled: Line 2 ensures that the given pin `PIN` of node `N` is an input pin. The clause in line 3 binds the variable `FS` to one particular set of input flows for node `N`. This set has to contain one data flow for every input pin of node `N`. The clause can find all possible combinations of input flows for the node, which ensures that all possible combinations are considered. Line 4 now binds the `FLOW` variable to a flow from the flow set `FS` that is an input flow for pin `PIN`. Finally, the clause in line 5 ensures that the selected flow `FLOW` is not in the set of already visited flows `VISITED` by ensuring that the intersection between the visited set and the set consisting only of the selected flow is empty. This avoids evaluation cycles in DFDs containing loops. If the flow has already been visited, the flow and the set of input flows cannot be used anymore, so another set of flows has to be found. The resolution mechanism of Prolog does this automatically. In the following, we explain the used clauses.

The `inputFlowsSelection` rule in line 6 finds a set of input flows `FS` for a given node `N`. This rule finds all possible combination of input flows when reevaluating it. The rule first identifies the set of input pins `PINS` for the node `N`. The clauses representing the `findAllInputPins` predicate used in line 7 are given in Listing 5.12. Essentially, the predicate yields a sorted list of all input pins of node `N`, so the predicate always only has one valid variable binding. The predicate `inputPinsFlowSelection` finds a flow for every pin in the set of input pins. The resulting set is bound to `FS`.

The `inputPinsFlowSelection` predicate finds a set of data flows for a set of pins so that the set of data flows contains exactly one data flow to every pin. The corresponding clauses do this by recursion. In the same way, the `inputFlowSelection` rules starting in line 13 find a data flow `F` for a given pin `PIN` by recursively searching the given list of flows for a data flow targeting the requested pin.

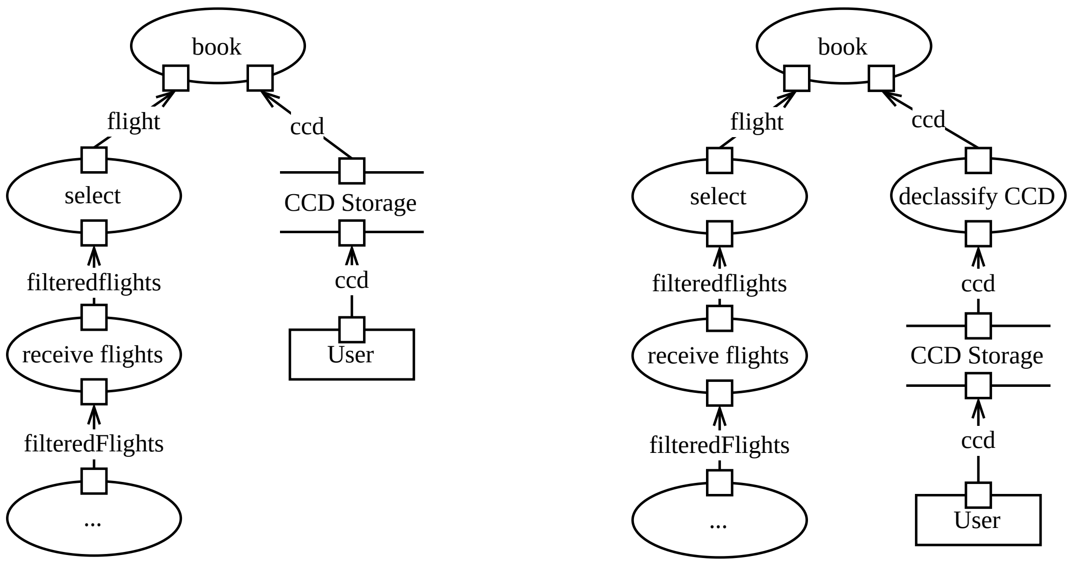
Listing 5.11: Prolog clauses describing the exploration of alternative inputs.

```
inputFlow(N, PIN, FS, FLOW, VISITED) :-
  inputPin(N, PIN),                       % ensure pin is input pin
  inputFlowsSelection(N, FS),             % find set of input flows
  inputFlowSelection(PIN, FS, FLOW),      % find flow to pin from set
  intersection(VISITED, [FLOW], []).      % avoid data flow cycles
inputFlowsSelection(N, FS) :-             % find input flows set for node
  findAllInputPins(N, PINS),
  inputPinsFlowSelection(PINS, FS).
inputPinsFlowSelection([], []).           % end recursion (no pin left)
inputPinsFlowSelection([PIN|T], [F|FT]) :-   % recursive: find flows to pins
  dataflow(F, _, _, _, PIN),
  inputPinsFlowSelection(T, FT).
inputFlowSelection(PIN, [F|_], F) :-      % end recursion (found flow)
  dataflow(F, _, _, _, PIN).
inputFlowSelection(PIN, [H|T], F) :-      % recursive: find flow to pin
  dataflow(H, _, _, _, PIN2),
  PIN \= PIN2,
  inputFlowSelection(PIN, T, F).
```

Listing 5.12: Prolog clauses finding all input pins for a given node.

```
findAllInputPins(N, PINS) :-
  findAllInputPins(N, [], PINS),
  sort(PINS, PINS).
findAllInputPins(N, PINS, RESULT) :-
  inputPin(N, PIN),
  intersection(PINS, [PIN], []),
  findAllInputPins(N, [PIN | PINS], RESULT).
findAllInputPins(N, PINS, PINS) :-
  \+ (
    inputPin(N, PIN),
    intersection(PINS, [PIN], [])
  ).
```

Flow Trees To detect all possible confidentiality violations, it is not sufficient to only consider alternative input flows at one node but also consider combinations of these alternatives in a sequence of visited nodes. We call such a combination a *flow tree.* A flow tree is a tree of reversed data flows. The root of such a tree is a node. In the second level, there are nodes that send data to the root node. Data flows connect the nodes on the second level to the root node. The third level is given by applying the previous construction rules to all nodes on the second level. Because there can be multiple alternative flows, as we explained in the paragraphs before, there can also be multiple flow trees for the same root node. Figure 5.9 presents the excerpts of the two flow trees for the *book* actor process. In Figure 5.9a, the process uses the credit card details from the storage. The right branch ends when it reaches the user. The left branch continues until all branches end with an actor or a process without inputs. In contrast, Figure 5.9b visualizes the selection of declassified credit card data from the *declassify CCD* process. Again, the right branch ends with the user.



(a) Usage of plain credit card data. **(b)** Usage of declassified credit card data.

Figure 5.9.: Flow tree excerpts for the book actor process of the running example.

Simply said, the flow tree can be seen as a subgraph of the whole DFD that eliminates all alternative data flows by selecting exactly one particular data flow whenever there is a choice. The flow tree enables label propagation because it removes ambiguities, i.e. which flow to use to collect labels to be used in the label propagation function. Additionally, the flow tree reduces the amount of nodes and flows to be considered to the nodes that actually can have an effect on the label propagation. For instance, nodes after the *book* actor process cannot influence the labels available to the *book* process and do not have to be considered therefore. To detect all possible violations, it is necessary to consider all possible flow trees.

A flow tree can be found for every pin of every node by the clauses shown in Listing 5.13. The predicate `flowTree/3` shown in line 1 yields a flow tree `S` for the pin `PIN` of a node `N`. The flow tree is a nested list of data flow identifiers. We omit the nodes in this list because they can be easily derived from the data flows. The clauses for realizing the predicate `flowTree/3` have to consider three types of pins: i) output pins of actors, ii) output pins of (actor) processes or stores and iii) input pins. This covers all possible pin types at all possible node types. In the following, we explain the rules supporting these cases. The `flowTree/3` rule evaluates these specific rules with the same parameters and an additional empty list of already visited flows that is used to break data flow cycles as already described in the paragraphs before.

Flow trees for output pins of actors (i) are always empty because they are the start of a data flow and therefore cannot depend on inputs. As the rule in line 3 shows, the flow tree for an output pin `PIN` of actor `N` is always the empty list `[]`. The already visited flows passed in the fourth argument are not considered because no flow is to be selected by this rule, so no flow cycle can be produced.

Listing 5.13: Prolog clauses describing how to build a tree of data flows.

```
 1 flowTree(N, PIN, S) :-                       % rule to be used by other rules
 2   flowTree(N, PIN, S, []).
 3 flowTree(N, PIN, [], _) :-                   % outputs of actors
 4   outputPin(N, PIN),
 5   actor(N).
 6 flowTree(N, PIN, S, VISITED) :-              % inputs of nodes
 7   inputPin(N, PIN),
 8   dataflow(F, NSRC, PINSRC, N, PIN),
 9   flowTree(NSRC, PINSRC, TMP, [F|VISITED]),
10   S = [F|TMP].
11 flowTree(N, PIN, S, VISITED) :-              % outputs of processes or stores
12   outputPin(N, PIN),
13   (process(N);store(N)),
14   inputFlowsSelection(N, FLOWS),
15   flowTreeForFlows(N, S, FLOWS, VISITED).
16 flowTreeForFlows(_, [], [], _).              % end of recursion
17 flowTreeForFlows(N, S, [F|T], VISITED) :-    % recursive sub tree derivation
18   intersection([F], VISITED, []),
19   flowTreeForFlows(N, STAIL, T, VISITED),
20   dataflow(F, NSRC, PINSRC, _, _),
21   flowTree(NSRC, PINSRC, TMP, [F|VISITED]),
22   SHEAD = [F|TMP],
23   S = [SHEAD|STAIL].
```

The flow tree of input pins of nodes (iii) always starts with a flow to this input pin and continues with the flow tree for the output pin on the other end of the data flow. Therefore, the rule in line 6 first finds a data flow `F` arriving at input pin `PIN` of node `N`. The `dataflow` clause can find all possible data flows arriving at the requested pin. The solving algorithm of Prolog automatically considers such other flows when reevaluating the rule, which means that all flow trees starting with these flows are considered. Next, the rule finds the flow tree `TMP` of the output pin `PINSRC` of node `NSRC`, which is the pin on the other side of the selected data flow `F`. To avoid data flow cycles, the selected flow is added to the list of already visited flows `VISITED` when evaluating the `flowTree` rule for the output pin in line 9. The final flow tree `S` is given by concatenating the selected flow `F` with the flow tree `TMP` of the output pin.

Output pins of processes or stores (ii) are more complicated because a set of input flows has to be considered for these node types instead of only a single input flow. The rule in line 11 does so. First, it ensures that the given node `N` is a process or store and that the pin `PIN` is an output pin. Afterwards it finds a valid selection of incoming data flows by evaluating the clause `inputFlowsSelection` as described in the paragraphs before. Again, this clause can yield all possible combinations of input flows for the given node, so this ensures that all possible flow trees are considered. Finally, the `flowTreeForFlows` clause finds the flow tree `S` for the input flow selection `FLOWS` while considering the already visited data flows `VISITED`. The `flowTreeForFlows` rule shown in line 17 takes a set of data flows `[F|T]` and recursively determines the flow trees for all output pins that the given data flows use. The full flow tree `S` is given by concatenating all flow trees for the output pins.

Listing 5.14: Simplified examples of flow trees of the running example.

```
 1 ?- flowTree('User', 'ccd', S).        % flow tree for output pin of actor
 2 S = [].
 3 ?- flowTree('book', 'ccd', S).        % flow trees for input pin of process
 4 S = ['ccd', ['ccd']] .
 5 S = ['declassifiedCCD', ['ccd', ['ccd']]] ;
 6 ?- flowTree('book', 'output', S).     % flow trees for output pin of process
 7 S = [
 8       ['flight', ['filteredFlights', ['filteredFlights', [...|...]]]],
 9       ['ccd', ['ccd']]
10 ] ;
11 S = [
12       ['flight', ['filteredFlights', ['filteredFlights', [...|...]]]],
13       ['declassifiedCCD', ['ccd', ['ccd']]]
14 ] .
```

For every flow in the flow set, the rule ensures that the flow has not already been visited and recursively evaluates itself with the remaining set of flows `T`.

The flow trees for the three cases explained before look like shown in Listing 5.14. The flow tree for output pins of actors is always empty as shown in line 1. As shown in line 3, the flow trees for input pins always start with the flow to this pin. In the example, there are two flow trees for the input pin receiving credit card information at the *book* process: one tree uses the credit card data from the store and the other use uses the declassified credit card data. Please note that the identifiers of the data flows have been shortened for a sake of simplicity. In the full logic program, this identifier would be unique. The flow trees for output pins of processes look basically the same but contain the flow trees for all input pins of the node. In line 6, there are two flow trees for the output pin of the *book* process. The flow tree is a list of two lists. The first list is the same in both examples because there is only one data flow path that a flight can take to the *book* process. The second list is essentially the flow tree for the input pin that we described before.

Label Lookup We formalize the label propagation as backward lookup instead of forward propagation. Label lookup means that a label at a certain pin can be determined by looking up labels on previous pins on a data flow path. This means, the lookup starts at the root of the flow tree and traverses the branches. The benefit of using a backward lookup instead of a forward propagation is that the lookup can stop if a label is found. For instance, there is no need to continue looking up a label on previous nodes if a node explicitly sets a label, which means that the labels of previous nodes are not necessary anymore to determine whether the label shall be available at the pin. In contrast, a propagation starts at the leaves of a flow tree and therefore requires the propagation to walk through the whole tree to reach the root even if we are only interested in the labels available at the root node. Because both methods yield the same labels, we choose the potentially more efficient lookup approach. We introduce the predicate `characteristic/6` that evaluates to true if a label (literal of a characteristic type) is available at the pin of a certain node. In the following, we describe how this predicate is defined for input and output pins.

Listing 5.15: Rule to specify the label propagation on input pins.

```
characteristic(N, PIN, CT, V, [F|S], VISITED) :-
  inputPin(N, PIN),
  dataflow(F, NSRC, PINSRC, N, PIN),
  intersection([F], VISITED, []),
  characteristic(NSRC, PINSRC, CT, V, S, [F|VISITED]).
```

Label Lookup on Input Pins The labels available on an input pin solely depend on the labels available at the output pin, from which a data flow is coming from. Therefore, the `characteristic` rule as shown in Listing 5.15 only selects a matching incoming data flow, ensures that this flow has not already been visited and determines if the characteristic value `V` of characteristic type `CT` is available at the source of the data flow, i.e. the output pin of the source node. By reevaluating the rule, all matching data flows are evaluated. The flow tree for the input pin is given by the concatenation of the selected data flow `F` and the flow tree `S` of the output pin, from which data flow `F` originates. Because the label lookup at input pins does not depend on the assignments of the corresponding node, this rule is sufficient to handle the lookup at all input pins, i.e. no specific rules are required.

Label Lookup on Output Pins The labels available on an output pin depend on the assignments of the corresponding behavior and on the labels on the input pins. We already formalized the lookup of labels on input pins in the previous paragraph. Therefore, we focus on the formalization of assignments in the following. Because assignments are meant to be executed in the given order, later assignments can override effects of previous assignments. We first describe how to identify the last and therefore effective assignment for a literal of a characteristic type at a pin, i.e. the assignment that eventually defines whether a label is available. It is only necessary to map this assignment for the triple of pin, characteristic type and literal on the particular node. Afterwards, we describe how assignments map to a `characteristic` rule for output pins, i.e. how the label lookup on output pins is formalized.

Finding the effective assignment of an ordered set of assignments is necessary because formalizing a sequence of assignments that override effects of previous assignments is hard to do in Prolog, which we use as the underlying formalism. However, it is not necessary to represent assignments that do not have an effect on the final labels. We motivate this by the example originally given in Listing 5.6 on page 48. First of all, assignments can never refer to the result of a previous assignment within the same behavior because the left-hand side always has to refer to an output pin and the right-hand side can only refer to input pins. If we want to know if the *UserAirline* literal of the characteristic type *class* will be available at the output pin, it is sufficient to only consider the last assignment because it is the last assignment for this literal. This assignment will always assign a value, so previous assignments for the *UserAirline* literal of the characteristic type *class* will always be overridden. Therefore, it is pointless to consider them for this particular combination of characteristic type and literal. If we want to know if the *User* literal of the same characteristic type will be available, it is sufficient to only look at the second assignment.

Previous assignments will be overridden anyway and the assignment afterwards does not consider the *User* literal. To summarize, it is sufficient to only look at the last assignment that assigns a truth value to the triple of pin, characteristic type and literal on a particular node. Consequently, we only have to consider that assignment during the mapping, which we describe later.

The logic to find the last assignment that has an effect on a particular literal of a particular characteristic type is given in Algorithm 5.1. The algorithm yields the effective assignment for a given pin, characteristic type and literal based on a list of assignments. Essentially, the algorithm traverses the list of assignments in reversed order and returns the first assignment that matches, i.e. assigns a truth value to the triple of pin, characteristic type and literal. Whether an assignment matches or not depends on the left-hand side and right-hand side of the assignment. Both sides have to match the requested triple. The left-hand side matches if it considers the same pin, characteristic type and literal. The left-hand side also matches if wildcards are used, i.e. the literal is omitted or the literal and the characteristic type are omitted. The logic to match the right-hand side of an assignment is given in Algorithm 5.2. Essentially, the right-hand side is compatible if it is a constant, all terms contained by a logic term (and/or/not) are compatible and if the characteristic references to data/pins or nodes are compatible. The references are compatible if there cannot be an error when instantiating wildcards. This is guaranteed in three cases: i) If there are no wildcards involved, there cannot be a failure in binding the wildcards to particular values. ii) If both, characteristic type and literal are wildcards, binding these wildcards is always possible by using the characteristic type and the literal from the left-hand side. iii) If there is only a literal wildcard, the characteristic type of the right-hand side has to refer the same enumeration as the given characteristic type. Using the same enum means that the value range of the characteristic type, i.e. the available literals, are the same, so instantiation is always possible.

During the mapping from the extended DFD syntax to the first-order logic semantics, the previously described algorithms are used. In general, the mapping works as follows: There is exactly one `characteristic` rule for each item of the cross product of output pins, characteristic types and literals of that characteristic type at a particular node. This means that there is one rule to determine if a label, i.e. the tuple of characteristic type and literal, is available on an output pin. If the rule evaluates to true, the label is available. The clauses to be fulfilled as part of the rule are given by the last effective assignment for the triple of pin, characteristic type and literal. We determine this assignment by the previously described Algorithm 5.1.

To map an assignment, we first instantiate wildcards by inserting the characteristic type and literal of the given triple into all wildcards. We already described wildcard instantiations in the description of the syntax in Section 5.1.2. After that, we map the left-hand side of the assignment to the head of the `characteristic` rule and the right-hand side to the clauses in the rule body. The rule head is shown in line 2 of Listing 5.16. The lower case arguments `n`, `pin`, `ct` and `v` become constants based on the node and the given triple. `S` is the flow tree for the given pin and `VISITED` is the set of already visited data flows to break data flow cycles. Examples of the mapping of terms on the right-hand side of an assignment are

Algorithm 5.1 Identification of effective assignment for label on given pin.

```
function LASTMATCHING($assignments, pin, characteristicType, literal$)
    for all $assignment \in$ REVERSE(assignments) do
        if MATCHES($assignment, pin, characteristicType, literal$) then
            return $assignment$
        end if
    end for
end function
function MATCHES($assignment, pin, characteristicType, literal$)
    $lhs \leftarrow$ lhs of $assignment$
    $rhs \leftarrow$ rhs of $assignment$
    $p \leftarrow$ pin of $lhs$
    $ct \leftarrow$ characteristic type of $lhs$
    $l \leftarrow$ literal of $lhs$
    if $p \neq pin$ then
        return false
    else if $ct$ is defined and $ct \neq characteristicType$ then
        return false
    else if $l$ is defined and $l \neq literal$ then
        return false
    end if
    return ISCOMPATIBLE($rhs, characteristicType, literal$)
end function
```

Listing 5.16: Mapping examples of terms in output characteristic rule.

```
1 % lower case arguments become constants
2 characteristic(n, pin, ct, v, S, VISITED) :-
3   true,                                   % True (Constant)
4   false,                                  % False (Constant)
5   nodeCharacteristic(n, ct2, v2),         % ContainerCharacteristicReference
6   characteristic(n, pin3, ct3, v3, S0, VISITED). % DataCharacteristicReference
```

given in Listing 5.16 starting with line 3. In general, constants also become constants in the rule body. Logical terms become their Prolog counterparts. References to characteristics of nodes become a `nodeCharacteristic` clause, in which the node `n`, characteristic type `ct2` and literal `v2` refer to the elements specified in the reference. References to characteristics of data become a `characteristic` clause, in which the node `n`, pin `pin3`, characteristic type `ct3` and literal `v3` refer to the elements specified in the reference. All of these references to elements are given by constants, which are the identifiers of the corresponding elements. The set of visited flows is passed unchanged to the `characteristic` clause referring to the input pin.

The flow tree `S` of the output pin consists of a concatenation of flow trees for all input pins. The flow trees of the input pins are used in the `characteristic` clauses in the body of the

Algorithm 5.2 Compatibility check of right-hand side of assignment for assignment to given label at pin.

```
function isCompatible($term$, $characteristicType$, $literal$)
    $enumType$ ←enum of $characteristicType$
    switch typeOf($term$) do
        case $Constant$
            return true
        case $LogicTerm$
            for all $subterm \in term$ do
                if not isCompatible($subterm$, $characteristicType$, $literal$) then
                    return false
                end if
            end for
            return true
        case ($EnumCharacteristicReference$)
            $ct$ ←characteristic type of $term$
            $l$ ←literal of $term$
            $e$ ←enum of $ct$
            if $ct$ and $l$ are defined then
                return true
            else if $ct$ is defined and $l$ is undefined and $e = enumType$ then
                return true
            else if $ct$ and $l$ are undefined then
                return true
            else
                return false
            end if
end function
```

`characteristic` rule of the output pin as can be seen in line 6 of Listing 5.16. The variable `S0` is the flow tree of the input pin `pin3`. In order to construct the valid flow tree `S`, all input flow trees have to be available. An example of such a specification for the *book* process is shown in Listing 5.17. First of all, an incoming flow `F0`/`F1` has to be found for every input pin. We use the previously described predicate `inputFlow/5` to find these flows, which all belong to the same set of input flows `FLOWS`. A flow tree for an input pin is then given by using the incoming flow as a head of a list and leaving the tail unspecified. The benefit of not fully specifying the flow tree is to increase efficiency. The flow tree has only be resolved up to the point at which we know that the label will not change anymore. For instance, assigning a constant is such a point. The full flow tree of the output pin is given by concatenating the flow trees of the input pins. The given formalization is necessary to ensure that reported labels are really available for the given flow tree. Therefore, every `characteristic` rule of an output pin contains flow tree construction clauses as illustrated in Listing 5.17. If there are no input pins on the given node or the node is an actor, the flow tree `S` is the empty list.

Listing 5.17: Example of clauses determining parts of the flow tree needed for label lookup at output pins of book process of running example.

```
inputFlow('book', 'ccd', FLOWS, F0, VISITED),
inputFlow('book', 'flight', FLOWS, F1, VISITED),
S0 = [F0|_],                            % flow tree for ccd input pin
S1 = [F1|_],                            % flow tree for flight input pin
S = [S0,S1].                            % flow tree for output pin
```

Listing 5.18: Characteristic rule for output pin of *process booking* process.

```
 1 characteristic('processBooking','booking','class','UserAirline',S,V) :-
 2   inputFlow('processBooking','ccd',FLOWS,F0,V),
 3   inputFlow('processBooking','flight',FLOWS,F1,V),
 4   S0 = [F0|_],
 5   S1 = [F1|_],
 6   S = [S0,S1],
 7   (
 8     characteristic('processBooking','ccd','class','UserAirline',S0,V),
 9     (
10       characteristic('processBooking','flight','class','UserAirline',S1,V);
11       characteristic('processBooking','flight','class','UserAirlineTA',S1,V)
12     );
13     characteristic('processBooking','flight','class','UserAirline',S1,V),
14     (
15       characteristic('processBooking','ccd','class','UserAirline',S0,V);
16       characteristic('processBooking','ccd','class','UserAirlineTA',S0,V)
17     )
18   ).
```

To give a complete example, we illustrate the resulting rule for the *process booking* process of the *Airline* from the running example given in Figure 5.7 on page 47. The process uses the join behavior introduced in Listing 5.1 on page 42. Briefly explained, the behavior has two input pins and one output pin. The classification label on the output pin is the highest classification label received at any input pin. In Listing 5.18, we illustrate the `characteristic` rule for the output pin of the *process booking* process for the literal *UserAirline* of the classification characteristic type. This literal is the medium level because the *User* level is higher and the *UserArilineTA* level is lower. Lines 2 to 6 build the flow trees for the input pins by finding an incoming flow for every input pin. The following clauses are the result of mapping the assignment. There are two cases that can yield the medium classification level: The clauses in lines 8 to 12 say that the resulting label is *UserAirline* if the incoming credit card data has this level and the incoming flight data has the same or the lower level. The clauses in lines 13 to 17 are the same with the flight and credit card data swapped. The clauses illustrate the interplay between the flow trees and the `characteristic` clauses: The flow tree `S0` for the input pin receiving credit card data is passed to the `characteristic` clauses that refer to this input pin. The same holds for the flight data.

ID	Description	User	Semantics Definition
R2.1	every element covered	–	clauses for viewpoints
R2.2	derivation of properties	analysis	label lookup
R2.3	origin of properties	analysis	flow tree
R2.4	analyses based on goals	expert	–
R2.5	analyses based on goals	architect	–
R2.6	tracing of properties	architect	flow tree
R2.7	automated analyses	architect	–
R2.8	information flow	expert	–
R2.9	access control	expert	–

Table 5.2.: Overview on semantics definitions and met requirements by DFD semantics.

To simplify usage of the `characteristic/6` predicate, we define an additional `characteristic/5` predicate, which omits the last parameter. The last parameter contains the already visited data flows, which is always empty in the beginning. Therefore, the `characteristic/5` predicate is realized by a rule that evaluates to true, if the `characteristic/6` predicate with the same arguments and an empty list of already visited flows evaluates to true. The new predicate simplifies formulating queries for characteristics.

5.2.3. Requirements Coverage by Semantics

The semantics described before already cover half of the requirements for these semantics. Table 5.2 gives an overview on the requirements and how the semantics meet them.

The semantics assigned a meaning to all metamodel elements (R2.1). The core of the semantics is the derivation of properties (R2.2), i.e. the labels on data and nodes. The label assignment to nodes and the label lookup for labels on data meet this requirement. The flow tree provides the origin of data (R2.3) as well as trace information about all involved data flows (R2.6).

The remaining requirements state that the semantics shall support various kinds of analyses. Essentially, the semantics already meet these requirements: Security experts can already define analyses based on analysis goals (R2.4) because the Prolog code that emerges from mapping an extended DFD as explained in Section 5.2.2 already automatically derives labels and it is only necessary to formulate a Prolog query to define an analysis. In theory, software architects can use the same means as the security expert to define analyses (R2.5) but expecting software architects to have expertise in Prolog might not be realistic. Therefore, we address this particular requirement later in Section 6.5. Because the mapping into Prolog code as well as the execution of the query can be automated, the resulting analyses can also be automated (R2.7). As motivated in Section 4.2.3, labels can cover many relevant properties for deciding about violated requirements of information flow (R2.8) and access control (R2.9). However, we did not demonstrate meeting these requirements by the definition of particular analyses based on the semantics yet. We do this as part of the analysis definition described in Chapter 6. The description will cover the procedure of defining analyses as well as the particular analyses.

5.3. Assumptions and Limitations

This section discusses limitations of the extended DFD syntax and the corresponding semantics, as well as assumptions regarding its usage.

Confidentiality properties as discrete value sets. One of the most fundamental limitation of the extended DFD and its semantics is that confidentiality properties have to be represented as sets of discrete values. This works well for confidentiality properties such as classification levels or roles because the instances of these properties are indeed discrete values. However, this modeling approach does not work for confidentiality mechanisms that define policies on arbitrary properties. For instance, the age of data might be a relevant property in mechanisms such as ABAC. Representing every possible age as discrete value is possible but certainly not feasible. However, usually not every single value has a different meaning with respect to confidentiality. For instance, there might be several ranges of data ages that can be treated in the same way. Representing these ranges of data ages instead of the individual ages is feasible and might even improve the comprehensibility of the resulting model. In addition, it is at least questionable if detailed information about particular values is even available during the early design time.

Explicit data flows. Analysis or prediction results based on models are always limited to what can be discovered on the modeled information. Therefore, the semantics can only reason about data flows that have been modeled explicitly. With respect to the popular confidentiality mechanism of information flow control, this limits detectable violations to explicit flows instead of implicit flows caused by side-channels. Model-based analyses always sacrifice finding the absolute truth in favor of feasibility. A model capable of covering all types of side-channels would certainly have to be more detailed and consequently would require more information to be available. However, this information might not be available when creating the model, especially when talking about the early design time. Therefore, the analyses on the presented semantics cannot and are not meant to replace analyses on artifacts developed in later development phases. Instead, the analyses focus on detecting violations early that would also occur in follow-up artifacts. The earlier such violations are detected, the more efficient they can be fixed.

Independent data flow paths. The DFD semantics consider all possible combinations of alternative data flows, i.e. flow trees. However, this can lead to false positives because combinations that might never appear in realistic applications are checked as well. For instance, the decision on where to send data might depend on the characteristics of data. Consequently, the propagated labels might be different depending on these characteristics.

The syntax does not provide means for expressing such dependencies for two reasons: First, we aim for worst-case analyses. Because of implementation errors or faulty conditions on the characteristics, data might still be sent to another connected node. The analysis should reveal such potential issues, so the architect can decide whether the risk of a potential faulty implementation and the resulting consequence is acceptable or the architecture should be changed. Second, the specification becomes more complex when introducing such data routing concepts. As soon as specifications require too much effort or are hard to create, architects will less likely use the modeling and analysis approach.

Limited effect of loops. Data flow loops in a DFD potentially lead to endless recursions in label propagations. The presented semantics break such loops by stopping the evaluation of a path as soon as a loop is detected. This implies a limitation but is reasonable: If loops continuously change labels, it is hard to know when to stop the evaluation because a steady state, i.e. a state in which no labels change anymore, cannot be reached. The meaning of such oscillating labels or even when to use them as part of a label comparison is unclear. Therefore, we do not consider DFDs introducing such oscillating labels as valid. If the labels produced in a loop do not change anymore after the first iteration, the semantics consider these labels: Visiting all data flows in the loop is possible because no flow in the loop has already been visited. After one iteration of the loop, the semantics will not visit the loop again, i.e. will not select the same data flow again, but will choose another data flow to continue.

Availability of information. An assumption made by all model-based approaches is that the information required to create the models is available. The information required for creating an extended DFD are the system structure, the confidentiality policy as well as relevant properties and behaviors. The structure of the system is known to the architect in the required level of detail. The confidentiality policy might not be specified in full detail but at least an abstraction of the intended security policy will certainly be available. The more coarse-grained this policy is, the more coarse-grained the results will be but at least there will be results. An architect might not be capable of defining the relevant properties and behaviors to represent relevant parts for establishing confidentiality. However, security experts that understand the underlying principles of the confidentiality policy and the mechanisms that realize them can create properties and behaviors. It is also possible to provide and reuse definitions for common access control and information flow policies, so even that knowledge does not always have to be available from a security expert.

5.4. Summary

In this chapter, we presented the syntax and semantics of an DFD extended by confidentiality concepts. The syntax and semantics meet the requirements for both, which we identified in Section 4.1. The presented extended DFD is in line with the suggested solution from Section 4.2.

The extended syntax described in Section 5.1 addresses three viewpoints that we identified as relevant for the software architect and the security expert. The first viewpoint addresses the concern of describing the system by introducing the concepts of pins to enable reuse of nodes as well as the concept of actor processes to describe user behavior. The second viewpoint addresses the concern of defining confidentiality primitives by introducing the concepts of characteristics and behavior definitions. Characteristics represent strongly typed labels. Behavior definitions represent label propagation functions. The third viewpoint addresses the concern of meeting confidentiality requirements by introducing binding concepts of characteristics and behaviors to nodes and data.

The denotational semantics described in Section 5.2 formalize the semantics of the extended syntax in first-order logic. The semantics cover the existence of DFD nodes and edges, as well as their behavior. The behavior is given by a formalization of the label propagation function. To increase efficiency, the concept of *label lookup* is introduced, which is one way to achieve the same results as by applying label propagation. The lookup is potentially more efficient in presence of multiple alternative data flow paths through the system. To yield valuable results, all of these different paths, as well as combinations, are considered by the semantics. A *flow tree* describes one particular combination of paths through the system.

The underlying assumptions as well as assumptions regarding the usage of the syntax and semantics are discussed in Section 5.3. Limitations exist with respect to expressible confidentiality properties, which are limited to sets of discrete values. Regarding the results, there exist limitations implied by the considered data flows, the data flow paths as well as the data flows that are part of a loop. A fundamental assumption shared with other model-based analysis approaches is that the information required for modeling the system is available.

6. Confidentiality Analyses based on Label Propagation

The extended DFD syntax and corresponding semantics covered in Chapter 5 provide the foundation for modeling software architectures and analyzing them for confidentiality violations. This chapter describes how to define and conduct analyses of such DFDs.

The analysis procedure described in Section 6.1 specifies the interaction between security experts, software architects and automated tooling. Section 6.2 covers particular analyses for detecting violations of confidentiality requirements regarding established information flow and access control mechanisms. Besides those two common confidentiality mechanisms, we describe how to integrate encryption into these analyses in Section 6.3 as an additional option to protect information. In addition to the integration of encryption, a combination of multiple confidentiality mechanisms and their corresponding analyses can be useful to improve the protection of confidential information. Section 6.4 describes this combination of analyses.

To not only rely on security experts, software architects shall also be capable of defining analyses. The DSL introduced in Section 6.5 provides the architect with means to specify analyses without the need for expertise in the formal DFD semantics.

The analysis procedure, the particular analysis and the DSL address requirements for the DFD semantics. Section 6.6 gives an overview on how the requirements are met. Section 6.7 describes the assumptions and limitations of the analyses and the DSL. Eventually, Section 6.8 summarizes the chapter.

6.1. Procedure for Analyses

In order to conduct analyses, the security expert and the software architect have to collaborate. Figure 6.1 visualizes this collaboration in the Business Process Modeling Notation (BPMN). Roughly said, the security expert has to provide confidentiality primitives, i.e. characteristics, behavior types and a label comparison function. The software architect uses these primitives to define a system including the aspects, which are relevant for confidentiality. Automated tooling then analyzes the defined system for violations of confidentiality requirements and reports violations to the software architect. The software architect uses the reported violations to adjust the system in order to meet the confidentiality requirements. In the following, we describe these steps in more detail.

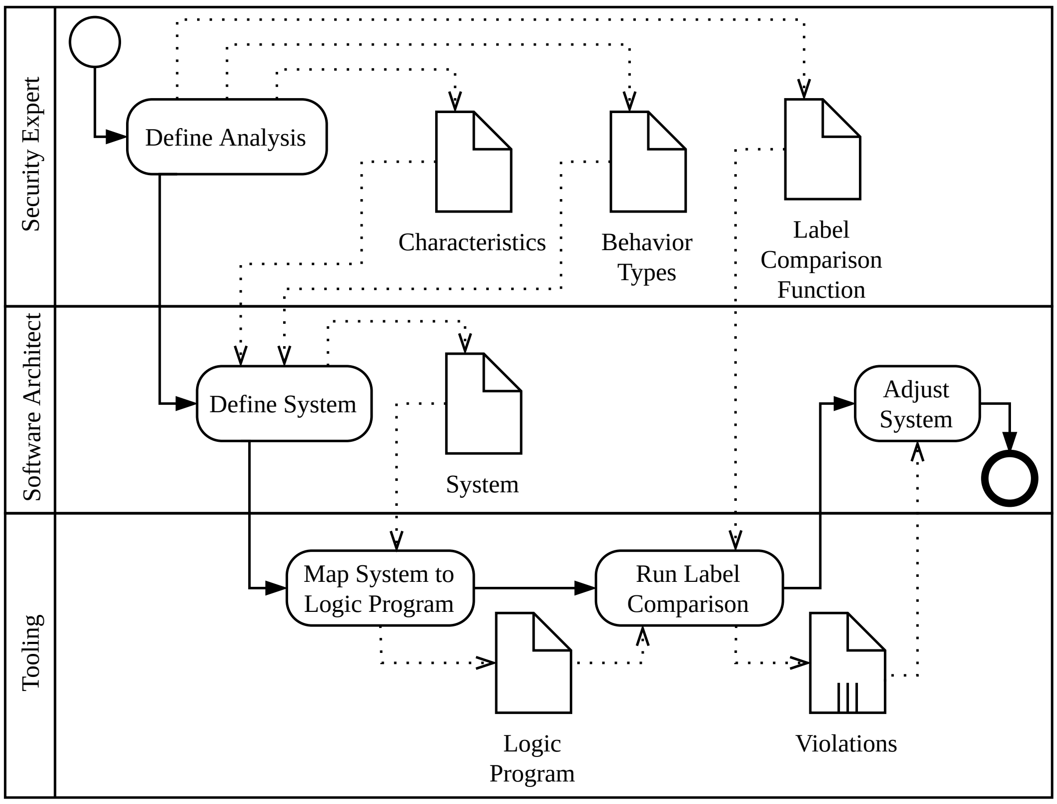


Figure 6.1.: Overview on analysis procedure given as BPMN diagram.

Analysis Definition The security expert defines the analysis by providing confidentiality primitives, which are specific to particular confidentiality mechanisms, to the software architect, who uses these primitives later to describe confidentiality aspects of the system. More precisely, the security expert provides the characteristics and the behavior types, which the software architect uses in the binding view type, as well as a label comparison function, which detects violations. When talking about an *analysis definition*, we always refer to a set of characteristics, characteristic types, behavior types and label comparison function. The characteristic types have to be part of an analysis definition because these types are necessary to define characteristics. The label comparison function includes the confidentiality requirements and a definition on how to detect a violation of these requirements. Depending on the particular confidentiality mechanism and the requirements, the analysis definition can be reused. This means that a security expert only has to be involved in the analysis of systems if such a reusable analysis definition is not available yet.

System Definition The software architect defines the system in a DFD and uses the confidentiality primitives defined by the security expert. The result is a DFD of the system that describes the structure, usage and deployment as well as the behavior in terms of processes, which change data characteristics.

System Mapping The tooling maps the system given as extended DFD to a logic program by applying the mapping rules for assigning semantics to syntax elements from Section 5.2.2. The tooling does not require assistance of the software architect or the security expert to execute the mapping rules because the mapping and all decisions are given as algorithms, which do not need human inputs except for the system definition. Consequently, the mapping can be fully automated. The result of the system mapping is a Prolog program.

Label Comparison The tooling performs the actual analysis by executing a label comparison within the logic program. The label comparison is essentially a query to the logic program that compares the labels of data and nodes with expected labels. The security expert can provide further clauses to define such expected labels in a query. The clauses provide additional information about confidentiality requirements, which would have to be encoded in the query otherwise. However, the query still formulates the analysis goal (R2.4). The logic program already contains all necessary clauses to determine all labels of all exchanged data via all possible data flow paths. The query triggers the label lookup and compares the results. All of these steps can be fully automated by executing the query, i.e. the label comparison, on the logic program within a Prolog interpreter. The result is a list of detected violations within the logic program, which represents the system.

System Adjustment The software architects use the reported violations to adjust the system in order to meet the confidentiality requirements. The violations contain information about the flow tree, which allows the architects to trace the origin of labels to locate the underlying design issue.

As mentioned before, the execution of the analysis can be fully automated as long as the software architect has defined a system and the security expert has defined the label comparison function. Therefore, the approach meets the requirement of providing the software architect with automated analyses (R2.7).

6.2. Label-based Confidentiality Analysis Definitions

Because there are various ways to protect the confidentiality of information in software systems, we cannot cover all possible mechanisms but focus on the most prominent mechanisms. According to Shostack [Sho14, p. 154], encryption is the predominant way of protecting information outside of a software system and access control is the predominant way of protecting information inside a software system. However, Sabelfeld and Myers [SM03] argue that these mechanisms are limited: Access control often only decides about access to information in one particular place in the system and does not protect the information after a user got access to it. Encryption cannot protect information after it has been decrypted. On the other side, information flow considers such possible leaks of information.

In the following, we cover the common information flow analysis for noninterference including declassification in Section 6.2.1. In Section 6.2.2, we define analyses for the four most common access control models. We see encryption as a supporting confidentiality mechanism, which protects data outside of systems or when transmitting data between systems. Therefore, we do not consider encryption by a dedicated analysis but in combination with information flow or access control analyses. We cover the integration of encryption into these analyses in Section 6.3. Compared to our previous publication of analysis definitions [Sei+22], we extended the descriptions of analyses by instructions on how to deal with particular variants of information flow and access control requirements. Namely, we extend the descriptions by a discussion of arbitrary lattices in information flow analyses, the delegation of rights in DAC, the Needs-To-Know model in MAC, Hierarchical and Constraint RBAC as well as hierarchies and constraints in ABAC.

Information flow and access control usually not only consider confidentiality but also integrity. We do not cover the integrity aspects of these mechanisms in the following because the goal of our approach is to detect confidentiality violations in software systems. Every time we claim support for a certain mechanism, we mean support for the confidentiality aspect of the mechanism.

6.2.1. Information Flow Analyses

Information flow has been studied for decades, which lead to various notions of a secure information flow [SM03]. In this section, we focus on the predominant notion *noninterference* including ways to weaken the implied strong restrictions by *declassification*. We recap the important fundamentals of this notion in the following. Afterwards, we show how to define analyses for simple and more complex information flow requirements in Section 6.2.1.1 and Section 6.2.1.2, respectively.

Noninterference is the predominant notion of secure information flows according to Hedin and Sabelfeld [HS12]. Every piece of information is classified by a certain label. Simply said, noninterference limits how information classified by different labels may influence each other. Influencing means that there is an information flow. An information flow can be explicit or implicit. An explicit information flow means that classified information directly flows into other classified information such as by variable assignments. An implicit information flow appears if classified information affects the control flow and thereby implies different observable behavior for different values of classified information. Allowed information flows are usually specified by lattices [Den76]. A lattice can be interpreted as a directed, acyclic graph of labels L. A label is assigned to every piece of information. The meaning of an edge $a \longrightarrow b$ with $a, b \in L$ is that information labeled with a is allowed to influence information labeled with b. This relation is transitive and reflexive. Often, label b is called *higher* than label a if information labeled with a is allowed to influence information labeled with b. A *declassification* allows to reclassify information to a lower classification to avoid violating the information flow requirements given by the lattice. A practical example is a user, who allows sharing medical information with an electronic health record service in order to use it. Such declassifications are always restricted to who can declassify information, what can be declassified, where information can be declassified or when information can be declassified [HS12]. Otherwise, the lattice would not imply restrictions anymore.

We solely focus on explicit information flows. First of all, implicit information flows as defined by Hedin and Sabelfeld [HS12] occur when classified information affects control flows. DFDs do not provide information about the control flow apart from a potential execution order implied by data dependencies. Therefore, these types of system models do not contain the necessary information to reason about implicit flows. However, this does not mean that the decision to use DFDs was wrong: Reasoning about implicit flows is a highly complex topic because all observable effects on the system behavior caused by classified information imply an implicit flow. The problem is known since about a half century [Lam73] but still one of the biggest challenges in information flow control up today [SM03]. Reasoning about such implicit flows is hard even in presence of source code and the real execution environment. Such detailed information is certainly not available during the early design phase, so software architects simply cannot specify enough information. We focus on explicit information flows because this is what software architects can know and model.

An information flow is relevant for analyses if it might be observable. We already defined that we can observe effects of explicit information flows, so this answers *what* we can observe. However, we also have to define *who* can observe an effect. In a DFD, any type of node can observe the effect of an explicit information flow by inspecting exchanged data. If a node can observe an effect, this means that there is an information flow from data to the node. Consequently, not only data but also nodes need a classification. To avoid ambiguities, we use the term *classification* for the classification of data and use the term *clearance* for the classification of a node. By evaluating the relations in the lattice, it is possible to determine if an information flow is allowed: If data d with classification c_d flows to a node n with clearance c_n, the flow is allowed if the lattice contains a (transitive) relation $c_d \longrightarrow c_n$.

6.2.1.1. Linear Ordered Lattice

A linear ordered lattice is a directed, acyclic graph of labels, in which there is one start label, which has no incoming and exactly one outgoing edge, a stop label, which has exactly one incoming and no outgoing edge, and an arbitrary number of labels, which have exactly one incoming and one outgoing edge. The labels in the running example form such a linear ordered lattice: *UserAirlineTA* is the lowest label that has an edge to the *UserAirline* label. The *User* label is the highest label that is the target of an edge from the *UserAirline* label. The lattice is *UserAirlineTA* $\longrightarrow$ *UserAirline* $\longrightarrow$ *User*. For instance, this means that a node having clearance for *UserAirline* is allowed to observe information with classification *UserAirlineTA* and also *UserAirline*. Table 6.1 visualizes the meaning of the lattice in terms of an access control matrix: A node with a clearance given by the column may have access to information with a classification given by the row if the resulting cell contains a checkmark.

Classification \ Clearance	User	UserAirline	UserAirlineTA
User	✓		
UserAirline	✓	✓	
UserAirlineTA	✓	✓	✓

Table 6.1.: Information flow requirements for the running example given as access control matrix.

The general idea of a confidentiality analysis looking for violations of such a linear ordered lattice in DFDs is to 1) assign a clearance label to every node, 2) provide initial classifications for data, 3) describe the propagation of data classifications within behaviors and 4) compare the clearance of a node with the classification of incoming data. To identify a violation, the comparison can simply test if the clearance of a node is less than the classification of data. In the following, we introduce the characteristic types, characteristics, behaviors and the comparison function required to realize the analysis. We discuss the analysis definition in a generic way but give concrete examples based on the running example.

Characteristic Types. As motivated before, the required characteristic types are the *clearance* and the *classification*. Both types share the same enumeration holding the clearance/classification levels in ascending order, i.e. the lowest level comes first. There have to be at least as much levels as there are clearance and classification levels. A too high number is not problematic because unused levels do not affect the analysis. Therefore, it is possible to define a set of generic levels and use these generic levels in behaviors. However, it is usually useful to give levels appropriate names to foster comprehensibility. In the running example, the levels are *UserAirlineTA*, *UserAirline* and *User* given in that order. If levels have system-specific names, the enumeration of levels is not reusable but the characteristic types still are.

Characteristics. A node can only have exactly one clearance level at a time. Data can only have exactly one classification level at a time. Assuming there are *n* levels, there are exactly *n* characteristics of the clearance characteristic type and *n* characteristics of the classification characteristic type. In the running example, there is one clearance characteristic and one classification characteristic for every level, i.e. *User*, *UserAirline* and *UserAirlineTA*. The characteristics are reusable in other systems if levels have generic names. Otherwise, the characteristics are system-specific and therefore only reusable for systems using similar clearance and classification levels.

Behaviors. There are three fundamental behaviors to cover system behavior with respect to a linear ordered lattice: The *Forward* behavior simply propagates an incoming data classification to its output. There is no change in the classification because there is no additional information and the data processing does not produce a declassification effect. In the running example, most of the processes do not affect the classification and therefore use a forwarding behavior. The *Declassify* behavior explicitly sets the data classification to a predefined classification. This behavior is reasonable for data processing that is meant to explicitly change the classification such as approvals by users. In the running example, the *declassify CCD* process uses a declassifying behavior to explicitly reduce the classification of credit card data based on the consent of the user. The *Join* behavior applies the highest of multiple incoming data classifications to its outputs. This means that the most restrictive classification level is used. This is reasonable when multiple data inputs are used and the result uses information from multiple inputs. For instance, the *create booking* process uses the joining behavior to combine a flight and credit card data into a booking in the running example. All behaviors but the declassifying behavior do not refer to particular classification levels, which means they are not system-specific and, therefore, reusable for specifying other systems. The declassifying behavior can only be reused for systems that use a similar set of classification levels.

Label Comparison. The label comparison function compares the classification label of incoming data with the clearance label of the receiving node. If the classification label is higher than the clearance label, there is a violation. Listing 6.1 presents this comparison in terms of the semantics formalized in Prolog. Consequently, the comparison is given as query to a logic program. To determine the clearance of a node `N`, the query uses the `nodeCharacteristic/3` predicate with the characteristic type representing the clearance in line 1. This line finds the clearance level `V_CLEAR` of the node. Line 2 determines the index of the clearance level, which we use to determine whether a level is higher or lower compared to another level. Because we are interested in finding violations when receiving data, line 3 identifies an input pin `PIN` of node `N`. For this input pin, line 4 determines the classification level `V_CLASS` by using the `characteristic/5` predicate. The index `N_CLASS` of this classification level is determined in line 5. Eventually, line 6 tests whether the classification index `N_CLASS` is bigger than the clearance index `N_CLEAR`, which implies that the data classification of received data is higher than the clearance of the receiving node. An answer to this query implies a violation of the information flow requirements given by the lattice. Because the order to classification and clearance levels already defines the lattice, we do not have to represent it explicitly. The query is capable of identifying all violations by reevaluating it within a Prolog interpreter because neither the node, the input pin nor the flow tree is bound to specific values before issuing the query. Therefore, the solution algorithm of Prolog finds all possible combinations that lead to a violation. The query does not refer to particular clearance or classification levels, so it is reusable for analyzing multiple systems using a linear ordered lattice.

6.2.1.2. Arbitrary Lattice

Arbitrary lattices do not adhere to the restrictions given by linear ordered lattices, which we discussed before. Instead, the lattice can be any directed, acyclic graph of labels. Compared to the previously discussed linear ordered lattice, the specification effort increases because the lattice has to be explicitly represented as part of the label comparison function by

Listing 6.1: Query for detecting violations of information flow requirements given by linear ordered lattice.

```
?- nodeCharacteristic(N,'clear',V_CLEAR),          % clearance of node
characteristicTypeValue('clear',V_CLEAR,N_CLEAR),  % index of clearance
inputPin(N,PIN),                                   % input pin of node
characteristic(N,PIN,'class',V_CLASS,S),           % classification of data
characteristicTypeValue('class',V_CLASS,N_CLASS),  % index of classification
N_CLASS > N_CLEAR.                                 % check of lattice
```

additional clauses and there is no generic *Join* behavior. We discuss both points as part of the following analysis definition.

Characteristic Types. The required characteristic types are the same as for the linear ordered lattice. Namely, a classification and a clearance characteristic type are necessary. An enumeration holds all available levels. In contrast to the linear ordered lattice, the order of the literals in the enumeration has no meaning anymore.

Characteristics. Data can only have one classification label and nodes can only have one clearance label. Therefore, the required characteristics are the same as for the linear ordered lattice. There is one characteristic of the classification characteristic type for each level and one characteristic of the clearance characteristic type for each level.

Behaviors. The *Forward* and the *Declassify* behavior are the same as for the linear ordered lattice. A *Join* behavior is still necessary but its realization cannot be as generic as discussed in the linear ordered lattice. It is not possible to create a generic realization because there is no generally applicable ordering relation that supports finding the highest of two labels. For instance, the lattice can be a disconnected graph, in which it is impossible to decide which of two labels selected from two disconnected parts is semantically higher. As a consequence, the joining behavior has to specify the processing effect for each possible tuple of incoming labels in the worst case. However, exploiting the order of labels is still possible for subsets of the literals, which can reduce the specification effort. Because the joining behavior is tailored to the set of levels, it is only reusable for specifying systems that use the same set of levels.

Label Comparison. The general idea of the label comparison is the same as for the linear ordered lattice: The query finds the clearance of a node and the classification of incoming data and reports a violation if an information flow violates the requirements given by the lattice. To detect such a violation in an arbitrary lattice, the comparison looks for a missing edge from the classification label to the clearance label in the transitive closure of the lattice graph. In order to do so, it is necessary to encode the lattice by additional clauses. Listing 6.2 exemplifies this by the linear ordered lattice. In lines 1 to 2, the lattice is given by describing the edges of the graph. The rules in lines 3 to 4 build the transitive closure by introducing the `connected/2` predicate. The actual query definition shown in

Listing 6.2: Clauses defining the arbitrary lattice and the transitive closure.

```
edge('UserAirlineTA', 'UserAirline').             % lattice definition
edge('UserAirline', 'User').                      % lattice definition
connected(X, X).                                  % relation is reflexive
connected(SRC, DST) :-                            % relation is transitive
   edge(SRC, X),
   connected(X, DST).
```

Listing 6.3: Query for detecting violations of information flow requirements given by arbitrary lattice.

```
1 ?- nodeCharacteristic(N,CT_CLEAR,V_CLEAR),        % clearance of node
2 characteristicTypeValue(CT_CLEAR,V_CLEAR,N_CLEAR), % index of clearance
3 inputPin(N,PIN),                                  % input pin of node
4 characteristic(N,PIN,CT_CLASS,V_CLASS,S),         % classification of data
5 characteristicTypeValue(CT_CLASS,V_CLASS,N_CLASS), % index of classification
6 \+ connected(N_CLASS, N_CLEAR).                   % check of lattice
```

Listing 6.3 uses this predicate. The query is the same as for the linear ordered lattice but replaces the index check in line 6 by a check of the transitive closure. A violation occurs if the classification label is not connected to the clearance label. In the given excerpt, we used the index values of the literals to query the lattice graph but using the literals themselves would also be possible. The query itself is reusable for analyzing other systems but the lattice depends on the particular levels. Therefore, the lattice can only be reused for system sharing the same lattice, i.e. the same confidentiality requirements.

6.2.2. Access Control Analyses

Access control is a mechanism to restrict access to information within a software system. The particular restrictions are given by a security model. To make clear that we talk about security models used in access control, we simply use the term *access control* model. Because access control has a long history, many different access control models have been defined. However, only three access control models have been successful in practice up to now according to Jin, Krishnan, and Sandhu [JKS12]: DAC, MAC and RBAC. Additionally, researchers as well as practitioners have shown high interest in ABAC because of its increased flexibility compared to previously mentioned access control models. There has also been work [Fur08, pp. 79] [JKS12] on how to represent the established access control models in ABAC. In the following, we briefly recap the essential aspects for these four commonly used access control models and describe corresponding analysis definitions.

6.2.2.1. Discretionary Access Control (DAC)

DAC [Fur08, pp. 61] explicitly assigns access permissions between dedicated subjects and dedicated objects. Permissions can be stored with the subjects, i.e. the subject knows which objects it is allowed to access, or with the objects, i.e. the object knows which subjects are allowed to access it. Besides the direct assignment of access rights between subjects and objects, there are two additional principles: the ownership of information, i.e. the creator of an object becomes its owner, and the delegation of rights, i.e. subjects holding certain rights can delegate these rights to other subjects.

The general idea of a confidentiality analysis looking for violations of requirements given in DAC without considering owners and delegation of rights is to 1) assign an identity label to all actors, 2) assign labels for representing read and write access to all stores, 3) trace back data from and to stores to actors and 4) compare the identity of actors with the read and write access of the store. Identity labels (1) are necessary to clearly identify a subject because there can be multiple actors, which represent the very same subject. It is reasonable to represent objects as stores (2) because these DFD elements represent persisted data, which matches the meaning of an object in a data-oriented system description. Directly assigning access rights to the objects, i.e. the stores, is the core concept of DAC. Tracing data (3) allows to find pairs of actors and stores, which exchange data. If data flows from an actor to a store, it is reasonable to see this data flow as a data flow writing information into the store. A data flow in the opposite direction means that the actor receives data from a store, which certainly requires reading data from the store. The analysis does not require the propagation of labels on data because an analysis of the structure is sufficient.

When considering the owner and the delegation of rights, the analysis becomes more complex. The general idea described before still applies but in addition it is necessary to 5) assign an owner label to a store, 6) assign labels for adding an owner, read rights or access rights to data and 7) compare the identity of actors accessing a store with the access rights added via data. The newly introduced labels for data (6) are control messages. If a data item arrives at a store that has a label to add an owner, read rights or access rights, the access rights already assigned to a store are extended by the new access rights. The owner label (5) is necessary to ensure that only owners change the access rights. The extended analysis (7) now requires the propagation of labels on data. The previously described idea does not consider the order, in which owners and permissions are added, or the removal of them because this would require knowledge about the order of such messages. However, DFDs do not provide means to describe control flows, so there are also no means to describe the order of messages. We discuss this point in more detail in Section 6.7. In the following, we explain the elements of the analysis definition in more detail.

Characteristic Types. All following characteristic types use an enumeration of subject identities as range of values. Because the enumeration is tailored to the particular system, the enumeration itself is not reusable for other systems except for systems that contain a subset of the identities. The characteristic types themselves are, however, reusable in other systems. The *Identity* characteristic type assigns an identity to an actor. This is useful because software architects can then represent the same subject by multiple actors, which can simplify the modeling and can improve the visualization of user behavior. The *ReadAccess* and *WriteAccess* characteristic types add access rights to a store, which represents an object. DAC demands such a direct assignment of access rights. The *Owner* characteristic type assigns owners to a store, which is necessary to decide if an actor is allowed to change access rights. The *AddOwner*, *AddReadAccess* and *AddWriteAccess* characteristic types represent control information in data. This control information is required to adjust the access rights.

Characteristics. Characteristics for all previously mentioned characteristic types except for the *Identity* characteristic type can hold multiple values but at least one. This is reasonable because it is possible to add multiple owners and access rights by one message. In contrast, it is not useful to assign more than one identity to an actor because this would violate the direct assignment of access rights between subjects and objects, which DAC demands. All characteristics refer to particular identities, so they are only reusable in systems that use a subset of these identities.

Behaviors. Data processing does not have an effect on the propagated labels on data. Therefore, only the *Forward* behavior, which propagates all received labels from the input to the output is necessary. Because the behavior does not depend on a particular characteristic type it can be reused in other systems.

Label Comparison. The label comparison has to test for violations of an actor by reading data from a store and by writing data into a store. Listing 6.4 presents the query to do so. To improve comprehensibility, we added one rule to identify a violating read (line 1) and one rule to identify a violating write (line 9). It is also possible to merge these two rules into the query in lines 17 to 18 that asks for violations of any of these types. The rule for identifying a read violation checks whether an actor `A` is allowed to receive data from a store `STORE`. An actor receives data from a store if at least one flow tree of one of his/her input pins (line 5) contains the store (line 6). A violation occurs if the identity `Y` of the actor (line 7) does not have read access to the store (line 8). We explain the `readAccess/2` predicate in the next paragraph. The rule for identifying a write violation checks whether an actor `A` is allowed to send data into a store `STORE`. An actor sends data into a store if at least one flow tree of one of the input pins of the store (line 13) contains the actor (line 14). A violation occurs if the identity `Y` of the actor (line 15) does not have write access to the store (line 16). We explain the `writeAccess/2` predicate in the next paragraph. The query does not depend on particular identities, so it is reusable for analyzing other systems.

Determining read or write access for a given identity is possible in two ways, which are illustrated in Listing 6.5: If owners and the delegation of rights is not considered, the bodies of the `readAccess/2` rule in line 4 and the `writeAccess/2` rule in line 7 would only consist of the first clause, which queries the access rights directly assigned to the store. The remaining clauses of Listing 6.5 would not be necessary. If owners and the delegation of rights are considered, it becomes important to determine whether an actor is an owner of a store. The rule in line 1 does this. An owner is either statically assigned to the store via a node characteristic or he/she is dynamically added via a control message. The `dynamic/3`

Listing 6.4: Query for detecting a violation of access control specified in DAC.

```
readViolation(A, STORE, S) :-          % identify illegal read of A at STORE
  store(STORE),                        % find store
  actor(A),                            % find actor
  inputPin(A, PIN),                    % find input on actor
  flowTree(A, PIN, S),                 % find flow tree to input
  traversedNode(S, STORE),             % find store in flow tree
  nodeCharacteristic(A, 'identity', Y), % find identity of actor
  \+ readAccess(Y, STORE).             % check read permission of actor
writeViolation(A, STORE, S) :-         % identify illegal write of A at STORE
  store(STORE),                        % find store
  actor(A),                            % find actor
  inputPin(STORE, PIN),                % find input on store
  flowTree(STORE, PIN, S),             % find flow tree to input
  traversedNode(S, A),                 % find actor in flow tree
  nodeCharacteristic(A, 'identity', Y), % find identity of actor
  \+ writeAccess(Y, STORE).            % check write permission of actor
?- readViolation(A, STORE, S);         % report read violation or
   writeViolation(A, STORE, S).        % report write violation
```

rule in line 11 checks whether a change of an owner or an access permission took place and whether this change was allowed. To identify the change, the labels of data arriving at the store `STORE` are checked for a literal `V` of a characteristic type `CT` in line 13. For instance, to find a label for adding an owner, the characteristic type would be *AddOwner* and the literal would be the identity, which shall be added as owner. To check whether the actor that initiated the change is an owner, the flow tree of the received message is found in line 15. If an actor is part of the flow tree (line 16), the identity `Y` of that actor is found in line 17. Eventually, line 18 checks whether the found identity is an owner. This lookup is recursive, so all added owners and the initial owners are considered. The rules that check the read and write access operate in the same way: before considering the effect of the permission change, the sending actor is verified as owner. The presented clauses do not refer to particular identities. This means the clauses and the query presented in the paragraph before are reusable for analyzing other systems.

6.2.2.2. Mandatory Access Control (MAC)

MAC [Fur08, pp. 64] defines a set of mandatory rules that aim to not only control the access to data but also the flow of information. There are two prominent access control models for MAC: the military security model and the Need-to-Know model. The military security model is essentially the same model as the information flow analysis using an ordered lattice. We already described the analysis definition in Section 6.2.1.1, which can also be used to model and analyze the military security model. Therefore, we only focus on the Need-to-Know model in the following.

The underlying idea of the Need-to-Know model is to only allow subjects to access objects if the subjects need the objects for their assigned work. A set of topics, also called

Listing 6.5: Rules for determining read and write access in DAC.

```
owner(V, STORE) :-                         % V is owner of store
  nodeCharacteristic(STORE, 'owner', V);   % check characteristic on node
  dynamic(STORE, 'addOwner', V).           % check characteristic via data
readAccess(V, STORE) :-                    % V can read from store
  nodeCharacteristic(STORE, 'read', V);    % check characteristic on node
  dynamic(STORE, 'addRead', V).            % check characteristic via data
writeAccess(V, STORE) :-                   % V can write to store
  nodeCharacteristic(STORE, 'write', V);   % check characteristic on node
  dynamic(STORE, 'addWrite', V).           % check characteristic via data

dynamic(STORE, CT, V) :-                   % permission added via data
  inputPin(STORE, PIN),                    % find input pin of store
  characteristic(STORE, PIN, CT, V, S),    % find permission on data
  actor(A),                                % find actor
  flowTree(STORE, PIN, S),                 % find flow tree to store
  traversedNode(S, A),                     % find actor in flow tree
  nodeCharacteristic(A, 'identity', Y),    % find identity of actor
  owner(Y, STORE).                         % check that actor is owner
```

compartments, defines the work areas. Subjects as well as objects have topics assigned. The topics of the subject s are the needs to know N_s. The topics of the object o are its compartments C_o. A subject s is allowed to read an object o if $C_o \subseteq N_s$.

The general idea of a confidentiality analysis looking for violations of requirements given in the Need-to-Know model is to 1) assign a set of topics to actors, 2) provide initial topics for data, 3) describe the propagation of data topics within behaviors and 4) compare the topics of data with the topics of actors. The comparison only needs to consider actors and their activities, i.e. the actor processes, because these represent subjects. Consequently, only actors and the actor processes (1) require assigned topics, i.e. the needs to know. Transmitted data is the information, which actors would like to access, so data requires compartments (2), which can change based on data processing (3). For every actor or actor process, the comparison (4) has to identify the set of topics on the data and report a violation if this set is not a subset of the topics on the actor or actor process. In the following, we introduce the characteristic types, characteristics, behaviors and the comparison function required to realize the analysis.

Characteristic Types. The required characteristic types are the *needs to know* and the *compartments*. The value range of both types is a set of topics. The order of the topics has no meaning. The topics are usually specific for the system under design. Therefore, reusing the enumeration that defines the value range is only possible if other systems support similar activities and are in the same application domain. The characteristic types themselves are generic, so they can be reused for modeling other systems.

Characteristics. The particular characteristics depend on the system under design because a reasonable combination of topics for actors depends on their tasks. Any subset of the available topics can build a reasonable characteristic for actors or data. Consequently, the particular characteristics are only reusable for systems supporting similar activities and operating in the same application domain.

Behaviors. Data processing can affect the topics of exchanged data. There are three relevant behavior types: The *Forward* behavior does not affect the topics of data, so it just propagates the same set of topics from its input to its output. For instance, a validation procedure for data could have such a behavior. The *Join* behavior combines incoming data into a new data item. Because we cannot guarantee that information from an incoming data item cannot be recovered, the safest assumption is that all incoming topics still apply to the result of joining data. Consequently, the behavior applies the union of all incoming topics to the output. A procedure, which creates a report of sick days for an employee, could be an example of such a behavior because it combines information from a personal topic, a health topic and a work planning topic into a report containing all three topics. The *Declassify* behavior removes certain topics from an incoming data item and only propagates the remaining topics. A procedure, which aggregates individual sick days into a sum of sick days, can be an example of such a behavior because it removes the work planning topic from the incoming data. All behaviors except for the declassifying behavior are reusable for defining other software systems. The declassifying behavior is not reusable because it refers to particular topics, which vary depending on the particular system.

Label Comparison. The label comparison function has to identify data received by actors, which do not need all the topics of the received data to do their work. Listing 6.6 presents the query to do so. The query considers all inputs (line 2) of all actors and actor processes (line 1) because only actors and their activities can violate the need-to-know rule. For all possible flow trees of the identified inputs (line 3), the compartments `L_COMP` of the incoming data, i.e. the data topics, are found (line 4). The topics `L_NTK`, which the actor needs to know, are also collected (line 5). To collect `L_COMP` and `L_NTK`, the second-order logic predicate `findall/3` is used. The predicate finds all individual solutions `X` (first argument) to a query template (second argument) and yields the list of all individual solutions (third argument). Because `findall/3` yields a list instead of a set of elements, the results have to be transformed to a set before using set operations. The built-in predicate `sort/2` takes a list as first argument and yields a sorted set as second argument as shown in line 6. The query reports a violation if the set of compartments `COMP` of data are no subset of the set of needs to know `NTK` of an actor. The `subset/2` predicate (line 6) is a built-in predicate that evaluates to true if the first argument is a subset of the second argument. The query does not refer to particular topics and can be reused for analyzing other systems. The query does not need additional information about the confidentiality requirements, i.e. additional clauses, because the Need-to-Know model only demands the subset relation between the compartments and the needs to know and the query fully covers this.

Listing 6.6: Query for detecting a violation of access control specified in Need-to-Know (MAC).

```
?- (actor(N);actorProcess(N, _)),                          % actor (process)
inputPin(N, PIN),                                          % actor input
flowTree(N, PIN, S),                                       % input flow tree
findall(X, characteristic(N,PIN,'compartment',X,S), L_COMP), % data topics
findall(X, nodeCharacteristic(N,'needs to know',X), L_NTK),  % actor topics
sort(L_COMP, COMP), sort(L_NTK, NTK), \+ subset(COMP, NTK).  % subset test
```

6.2.2.3. Role-based Access Control (RBAC)

RBAC [Fur08, pp. 70] is a commonly used access control model in organizations. The main benefit of RBAC compared to DAC is the decoupling of access rights and users through roles. The introduction of roles simplifies the administration of access control because the access rights assigned to roles change less frequently than the assignment of roles to users. The roles of users often simply stem from the position of a user in the organizational structure, so assigning roles to users is fairly simple. Compared to DAC, the management of access rights is centralized, which simplifies keeping track of assigned permissions as well as revoking permissions.

There are three types of RBAC. *Core RBAC* describes the introduction of roles, the assignment of access rights to roles and the assignment of roles to users. *Hierarchical RBAC* adds hierarchies for roles. Users, which have been assigned a senior role, automatically also have the corresponding junior roles assigned. Consequently, senior roles inherit access permissions from junior roles. *Constraint RBAC* introduces static constraints on the assignments of roles and dynamic constraints on the activation or usage of roles during runtime.

The general idea to analyze Core RBAC is to assign roles to nodes and to assign permitted roles to exchanged data. This closely represents the decoupling of users and permissions by roles. Data processing can change the permitted roles, e.g. because a data item derived from two other data items requires more protection. If the assigned and permitted roles do not share at least one role, accessing data is forbidden and if a node still accesses the data, a violation has been identified. The general idea to analyze Hierarchical RBAC is the same but before every comparison of role sets, the set of assigned roles is extended by all inherited roles according to the role hierarchy. The effect of extending the set is that all permissions of junior roles are also considered during the comparison, which matches the semantics of the role hierarchy. The general idea to analyze static constraints in RBAC is the same as for Core RBAC but in addition, the constraints are checked for every node in the system. To identify violations, the constraints can be negated, which means that all nodes not adhering to the constraints are reported. For instance, a constraint saying that an actor must not have two roles at the same time would be checked by looking for an actor, which has these two roles at the same time. Analyzing dynamic constraints requires detailed information about individual actors as well as dynamic activation and deactivation of roles for individual users. DFDs and many languages targeting the architectural design phase do not provide means to specify individual users but only classes of users. Therefore, there is no concept to analyze dynamic constraints. We will discuss this limitation and possible alternatives for analyzing individual users in Section 6.7. In the following, we describe the analysis definition for the three types of RBAC excluding dynamic constraints.

Characteristic Types. The required characteristic types are the *assigned roles* for nodes and the *permitted roles* for data. Both characteristic types share the same value range, which is the set of all available roles. The order of the roles does not imply any meaning. The particular roles are only reusable if the organization using the system has similar organization structures. The structure often implies the used roles, so the roles should be similar as well. The characteristic types using these roles are generic and can be reused for defining other systems.

Characteristics. The particular characteristics on data and nodes depend on the useful combinations of assigned roles as well as permitted roles. In theory, any subset of the available roles is a valid characteristic. However, only a small amount of these subsets is reasonable in realistic scenarios. Because reasonable combinations depend on the particular system and organization structure, the characteristics can only be reused for similar systems and organization structures.

Behaviors. Data processing can affect the permitted roles of exchanged data. There are three relevant behavior types: The *Forward* behavior does not affect the permitted roles because it propagates the received permitted roles to the output. The behavior is necessary because many processing steps of a system do not affect the permitted roles. For instance, a validation of data does not affect the permitted roles. The *Join* behavior combines incoming data into a new data item. A reasonable assumption for the resulting data item is that the roles, which have access to all inputs also have access to the combination of these inputs. Therefore, the behavior builds the intersection of permitted roles of all incoming data items and applies the resulting permitted roles to the output. The *Declassify* behavior explicitly adds roles to or removes roles from the outgoing data item. This is reasonable if the result of a data processing step yields a less confidential item, e.g. because data has been aggregated and individual inputs cannot be reconstructed anymore, or if the processing yields a more confidential item, e.g. because certain combinations allow drawing more conclusions than possible by looking at the individual data items. All behaviors except the declassification do not depend on particular roles, so they can be reused in describing other systems. The declassification assigns particular roles to or removes particular roles from data, so the behavior depends on the particular system. Reusing the declassification is only possible for systems using a similar set of roles with the same meaning.

Label Comparison (Core RBAC). The label comparison for Core RBAC compares the permitted roles of received data and the assigned roles of a node to report a violation if both sets of roles do not share at least one role. The query in Listing 6.7 shows the corresponding query. First, the query looks for an input pin of a node (line 1) and a possible

Listing 6.7: Query for detecting a violation of access control requirements specified in Core RBAC.

```
1 ?- inputPin(N, PIN),                                       % input of any node
2 flowTree(N, PIN, S),                                       % flow tree of input
3 findall(R, nodeCharacteristic(N,'assigned roles',R), L_AR),    % roles on node
4 findall(R, characteristic(N,PIN,'permitted roles',R,S), L_PR), % roles on data
5 sort(L_AR, AR), sort(L_PR, PR), intersection(AR, PR, []). % empty intersection
```

path, on which data arrives at the node (line 2). For the arriving data, all permitted roles `L_PR` are identified (line 4). We use the `findall/3` predicate to collect all solutions `L_PR` to the query template given as second argument. The permitted roles are compared with the assigned roles `L_AR` of the node (line 3) by looking for the intersection between the two sets of roles (line 5). Because the set operations require sets instead of lists of elements, we build the sorted sets `AR` and `PR` by using the built-in `sort/2` predicate. The `intersection/3` predicate is a built-in predicate that evaluates to true if the intersection of the sets given as first and second argument is equal to the set given as third argument. If the intersection of the two sets is empty, a violation has been found because this means that the node does not have at least one role that is permitted to access the data. The query is not tailored to the particular system, so it can be reused to analyze other systems. No additional clauses are necessary because comparing the intersection of permitted and assigned roles is sufficient to identify all violations of the access control requirements, which restrict access to data based on permitted roles.

Label Comparison (Hierarchical RBAC). The label comparison for Hierarchical RBAC is based on on the previously described comparison for Core RBAC. Instead of just comparing the roles assigned to a node with the permitted roles of data, the set of assigned roles is extended by the inherited roles. Besides the previously presented query, additional clauses are necessary to represent the role hierarchies and handle them properly. Listing 6.8 gives an example of the `senior/2` predicate, which we use to define the role hierarchies. The role given as first argument is a senior role for the role given as second argument. The meaning of the example is that the project lead role is senior to the engineering role. This means that a project lead also has the rights of an engineer. All roles, to which a particular role is transitively senior, have to be considered when comparing assigned roles with permitted roles. Therefore, we have to find all transitive junior roles for the roles assigned to a node. We do this by replacing the variable `AR` in line 5 of Listing 6.7 by the variable `ER`, which is bound by the clause `effectiveRoles(AR, ER)`. In the following, we explain the definition of the `effectiveRoles/2` predicate in Listing 6.9. The rule in line 2 takes a list of roles as first argument and yields a list of effective roles. To do so, the transitive closure `H_ROLES` of junior roles for a role `H` is found (line 3) and the remaining roles `T` are considered in a recursion (line 4). In the end, the union `HT_ROLES` of all transitive closures is built (line 5) and returned as sorted list of roles (line 6). The purpose of the fact in line 1 is to stop the previously mentioned recursion. To find the transitive closure of junior roles for a given senior role, the `includedRoles/2` predicate in line 8 starts a recursion via the `includedRoles/3` predicate. The predicate takes a senior role as first argument

Listing 6.8: Example of clause defining a role hierarchy in RBAC.

```
senior('Project Lead', 'Engineer').  % project lead is senior role of engineer
```

and a list of already considered junior roles as second argument. The third argument yields the transitive closure of junior roles. When starting the recursion, the already considered list of junior roles is empty. Tracking considered roles is important to create a set instead of a list of roles. The recursively considered rule in line 11 finds a junior role, ensures that the role has not been considered yet and continues the recursion by adding the found role to the list of considered roles. The recursion terminates if there are no more junior rules that have not been considered yet (line 15). The `junior/2` predicate describes a reflexive (line 21) and transitive relation (line 22) between two roles. If the predicate evaluates to true, the role given as first argument is a transitive junior role of the second argument. The clause is similar to the `connected/2` relation in Listing 6.2 on page 75. The only difference is the clause starting in line 25, which ensures that only one path to prove the relation is considered. The considered path always uses the smallest intermediate roles according to their natural order (`@<`). This avoids reporting duplicate results, which would not be harmful or lead to wrongly reported violations but would lead to multiply reported violations. The presented clauses do not depend on particular roles, so they can be used for analyzing other systems as well. The particular role hierarchies depend on particular roles and are, therefore, only reusable for systems operating in similar domains and organization structures.

Label Comparison (Constraint RBAC). Enforcing the static constraints regarding role assignments does not interfere with the previously presented queries to analyze Core RBAC and Hierarchical RBAC. Therefore, the query for detecting violations of the static constraints is a second query to execute before or after the previously presented queries. Essentially, a definition of the constraints and a corresponding check is necessary. A simple form of constraint is a set of roles, which must never be assigned to a node at the same time. Listing 6.10 defines such an illegal combination of roles by the `illegalCombination/1` predicate. In the example, an actor must never hold a *requestor* and an *approver* role together. The query to detect such a violation is shown in Listing 6.11. It identifies a node `N` of any type (line 1), determines the assigned roles `AR` (line 2) and identifies the effective roles `ER` as described before (line 3). A violation occurs, if any illegal set of roles `C` (line 4) is a subset of the effective roles (line 5). The query does not depend on particular levels, so it is reusable. The particular constraints depend on the particular system under design, so they are only usable for systems operating in similar domains and organization structures.

6.2.2.4. Attribute-based Access Control (ABAC)

ABAC [Fur08, pp. 74] decouples access rights from particular subjects or objects by so-called *identifiers*. An identifier contains a set of attribute selectors, which describe criteria

Listing 6.9: Clauses for handling role hierarchies in RBAC.

```
effectiveRoles([], []).                % no effective roles for no roles
effectiveRoles([H|T], ROLES) :-        % effective roles for role list
  includedRoles(H, H_ROLES),           % find for list head
  effectiveRoles(T, T_ROLES),          % recursive solving for list tail
  union(H_ROLES, T_ROLES, HT_ROLES),    % ensure found roles are set
  sort(HT_ROLES, ROLES).               % sort found roles

includedRoles(R, ROLES) :-             % transitive closure for R
  includedRoles(R, [], ROLES),         % start recursion, no visited roles
  sort(ROLES, ROLES).                  % sort found roles
includedRoles(R, ROLES, RESULT) :-     % find roles not contained in ROLES
  junior(X, R),                        % junior role X for R found
  intersection(ROLES, [X], []),        % X not part of ROLES
  includedRoles(R, [X | ROLES], RESULT). % extend ROLES by X and recurse
includedRoles(R, ROLES, ROLES) :-      % stop recursion
  \+ (                                 % no new roles available
    junior(X, R),                      % junior role X for R found
    intersection(ROLES, [X], [])       % role X already in ROLES
  ).

junior(X, X).                          % reflexive relation
junior(X, Y) :-                        % transitive relation
  senior(Y, Z),                        % Y is senior for intermediate Z
  junior(X, Z),                        % X is junior for intermediate Z
  \+ (                                 % no other intermediate Z2
    senior(Y, Z2),
    junior(X, Z2),
    Z2 @< Z                            % intermediate Z2 smaller than Z
  ).
```

Listing 6.10: Example of static constraint on role assignments in Constraint RBAC.

```
illegalCombination(['requestor', 'approver']).          % constraint
```

for matching subjects or objects. A *subject identifier* contains selection criteria based on attributes of a subject. An *object identifier* contains selection criteria based on attributes of an object. An identifier can match multiple subjects or objects. Subjects or objects can have multiple matching identifiers. Identifiers can be part of a hierarchy, in which specific identifiers inherit the selection criteria from generic identifiers. An *authorization* refers to one subject identifier and one object identifier and defines permissions of the matched

Listing 6.11: Query for detecting a violation of role assignment constraints in Constraint RBAC.

```
?- (process(N);actor(N);store(N)),                     % find any node N
findall(R, nodeCharacteristic(N,'assigned roles',R), AR), % assigned roles
effectiveRoles(AR, ER),                                 % consider hierarchy
illegalCombination(C),                                  % find constraint C
subset(C, ER).                                          % test for violation
```

subjects to the matched objects. In addition to the identifiers, an authorization can contain conditions that must hold in order to use the permissions. Various works [JKS12] [Fur08, pp. 79] see ABAC as a generalization of the previously described access control models DAC, MAC and RBAC and provide instructions on how to describe the corresponding requirements in ABAC.

The general idea to analyze ABAC is to 1) assign attributes to actors, 2) provide initial attributes for data, 3) describe the propagation of data attributes within behaviors, 4) define the subject and object identifiers by selection criteria, 5) define authorizations based on subjects and object identifiers and 6) find an authorization for every input data of an actor. It is reasonable to only consider actors (1) because subject descriptors often only consider actors. However, it is also possible to consider other nodes as long as these nodes have attributes assigned. Exchanged data closely matches the definition of an object, which shall be accessed, so it is reasonable to cover the attributes of data (2) as well as potential changes by data processing (3) in the system description. The subject and object identifiers (4) as well as the authorizations (5) describe the requirements in ABAC, so representing them is necessary to identify violations. A violation occurs if we cannot find an authorization for incoming data of an actor (6). In the following, we introduce the characteristic types, characteristics, behaviors and the comparison function required to realize the analysis.

Characteristic Types. The characteristic types have to describe the attributes of subjects and objects. These attributes highly depend on the particular system. Because of the flexibility of ABAC regarding the considered attributes, it is not useful to prescribe any characteristic types. However, the characteristic types have to distinguish attributes for subjects and objects.

Characteristics. Because there are no prescribed characteristic types, prescribing characteristics is not possible. When defining characteristics, it can be useful to provide characteristics based on the subject and object identifiers. There is a high chance that the characteristics representing identifiers are used multiple times.

Behaviors. The *Forward* behavior is a reasonable behavior to include. It propagates the received labels of input data to the same labels of output data. It is reasonable to include this behavior because many data processing steps in a system do not affect any labels but just document the data processing to be implemented in the development phase. Providing more behaviors is not possible because generic descriptions of the effect of joining or declassifying data are not available for yet unknown attribute types. Instead, security experts have to define system-specific behaviors to cover all relevant data processing.

Label Comparison. The label comparison consists of two parts: First, the subject identifiers, object identifiers and authorizations describe the ABAC requirements. We need additional clauses to represent this information. Second, the label comparison looks for missing authorizations. The comparison is realized as query, which uses the additional clauses.

Listing 6.12 presents examples of the ABAC requirements. Lines 2 to 5 describe subject identifiers. The `matchSubject/2` predicate provides a name for the subject identifier as first argument and takes a node identifier `N` as second argument. A node argument is necessary to test the given subject identifier for a particular node. The *Clerk* identifier matches nodes, which have a *Role* characteristic with the value *Clerk* assigned. The *Manager* identifier matches nodes, which have a *Role* characteristic with the value *Manager* assigned. Lines 8 to 10 describe object identifiers. The `matchObject/4` predicate provides an name for the object identifier as first argument and takes a node identifier `N`, a pin `PIN` and a flow tree `S` as second, third or fourth argument, respectively. The triple of the node, pin and flow tree arguments identifies data from or to a particular node via a particular flow tree. Because data items are the objects, to which we would like to limit access, it is reasonable to consider this identifying information in order to test the object identifier on particular objects. The *Regular* identifier matches data, which only has the *Regular* value of the *Status* characteristic type applied. The used `exactCharacteristicValues/5` predicate is a helper clause to collect all values of a given characteristic type and to compare the values with a given set of values. The predicate evaluates to true if the set of values of data and the given set of values are identical. More details on the predicate are available in Appendix A. The *All* identifier matches all data. Establishing an identifier hierarchy is possible by adding the generic identifier, i.e. the `matchSubject/2` or `matchObject/4` clauses of the generic identifier, in the body of the rule, which represents the specific identifier. In the example, we only consider authorizations to read, so we introduce the `read/3` predicate. The predicate takes all information to identify a node as well as data coming to or leaving the node as arguments. The predicate evaluates to true if read access for the data at the node is granted. The authorization in line 13 states that the subjects identified by the *Manager* identifier have access to objects identified by the *All* identifier. To do so, the rule requires the subject and the object identifiers to match. The authorization in line 16 refers to the subject identifier *Clerk* and the object identifier *Regular* and establishes a condition for the authorization. The condition also has to hold, so it is added in a conjunction to the identifiers of the subject and object. The condition states that the *Location* `L` of the node has to be the same as the *Origin* of the data.

The label comparison shown in Listing 6.13 identifies violations caused by missing authorizations. In line 1, an actor `A` and one of his/her input pins `PIN` is found. A flow tree `S` describes how data items arrive at the input pin (line 2). The actor `A`, pin `PIN` and flow tree `S` identify data. A violation occurs if there is any data that arrives at the actor, for which no authorization can be found. Line 3 tests this condition by proving that there is no solution for `read/3` with the specified data. Finding no solution means that there is no authorization.

Listing 6.12: Examples of subject and object identifiers as well as authorizations for ABAC.

```
% subject identifiers for node N with given name
matchSubject('Clerk', N) :-
  nodeCharacteristic(N, 'Role', 'Clerk').
matchSubject('Manager', N) :-
  nodeCharacteristic(N, 'Role', 'Manager').

% object identifiers for data on pin PIN of node N via flow tree S
matchObject('Regular', N, PIN, S) :-
  exactCharacteristicValues(N, PIN, 'Status', ['Regular'], S).
matchObject('All', _, _, _).

% authorizations for reading data on pin PIN of node N via flow tree S
read(N, PIN, S) :-
  matchSubject('Manager', N),                         % subject identifier
  matchObject('All', N, PIN, S).                      % object identifier
read(N, PIN, S) :-
  matchSubject('Clerk', N),                           % subject identifier
  matchObject('Regular', N, PIN, S),                  % object identifier
  nodeCharacteristic(N, 'Location', L),               % start condition
  exactCharacteristicValues(N, PIN, 'Origin', [L], S).  % end condition
```

Listing 6.13: Query for detecting a violation of ABAC requirements.

```
?-  actor(A), inputPin(A, PIN),      % find input pin PIN for actor A
flowTree(A, PIN, S),                 % find flow tree for input
\+ read(A, PIN, S).                  % test for missing read permission
```

6.3. Consideration of Encryption in Confidentiality Analyses

According to Bauer [Bau05a], encryption converts a plaintext, which everyone can understand, to a ciphertext, which is incomprehensible without further processing. Decryption converts the ciphertext back into plaintext. Usually, a key is another parameter for a cryptosystem, i.e. the encryption and decryption, to remove the obligation to keep the encryption and decryption algorithms secret. Instead, only the key has to be kept secret. There are symmetric and asymmetric cryptosystems. In a symmetric cryptosystem [Kal05b], the same key is used for encryption and decryption. In an asymmetric cryptosystem [Kal05a], different keys are used for encryption and decryption.

According to Shostack [Sho14, p. 154], encryption should be used to protect information outside of a system because a system cannot protect the information out of its scope. Consequently, encryption is also appropriate to protect information in systems of systems because a single system cannot control the information in other systems. In DFDs, we would represent systems of systems by a chain of processes, which belong to different systems. Characteristics can represent the relation of a process to a system, e.g. by annotating the system as a characteristic to the process.

With respect to the information flow and access control analyses already presented in Section 6.2, encryption can be seen as declassification because it makes information inaccessible to unauthorized actors and thereby lowers the needs to protect that information. We already described the concept of declassification before but only gave examples of data processing, which potentially has a declassifying effect on data characteristics. In the following, we describe how to integrate encryption into the previously described information flow and access control analyses. The description focuses on the required extensions with respect to characteristic types, behaviors and label comparison functions. In Section 6.3.1, we describe a simple form of encryption that does not consider the key handling. For instance, this simple form can represent systems using a symmetric cryptosystem with shared keys. Section 6.3.2 extends the key-less encryption by key pairs of public and private keys. This representation of encryption represents an asymmetric cryptosystem.

6.3.1. Encryption Without Keys

The core idea of using encryption and decryption is to temporarily hide information and restore the information later. This helps protecting the information from unauthorized access e.g. during a transmission. In the context of DFDs and label propagation, hiding information means that previously applied labels do not apply anymore. However, the labels shall be restored later upon decryption.

During an analysis, we do not really have to hide these labels but have to make clear that the labels shall be hidden. Therefore, it is sufficient to introduce new characteristic types, which share the value range with the affected characteristic types, change the characteristic type of a label to the newly introduced characteristic type upon encryption and reverse that change upon decryption. In addition, the encryption might add a label of the old characteristic type, which expresses that the contained information is not accessible anymore. For instance, consider a data item with a *high* label of the characteristic type *classification.* Upon encryption, we can add a *high* label of a newly introduced characteristic type *old classification* to the data item and replace the original classification label with a *low* label. This means that now nodes only having clearance for *low* data are allowed to access the data item because the information contained in the data item would not be accessible to the node in a real system. Upon decryption, we revert this effect, i.e. we add the *high* label of the characteristic type *classification* and remove the label of *old classification.* This is reasonable because the data item provides access to the contained information after decryption.

In the following, we describe how to extend an existing analysis definition to consider encryption in existing information flow and access control analyses.

Characteristic Types. For every characteristic type, which refers to information hidden by an encryption, a second characteristic type using the same value range is necessary. With respect to the previously presented confidentiality analyses, the classification (information flow analyses), the compartments (Needs-to-Know) and the permitted roles (RBAC) are characteristic types, which require a second characteristic type. All of these characteristic types describe the information contained in a data item. This information is not accessible anymore after encryption. The remaining characteristic types such as the ones to add additional permissions in DAC do not require a second characteristic type.

Behaviors. There are two additional behaviors. The *Encrypt* behavior takes an input and yields an output. The behavior forwards all labels but replaces the characteristic type of all labels by the newly created second characteristic types if one is available. In addition, the behavior adds new labels for the old characteristic types, which correctly characterize the encrypted content. With respect to the previously presented confidentiality analyses, there would be a new label with the lowest classification level (information flow analyses), an empty compartments label (Needs-to-Know) and a set of all roles for the permitted roles characteristic type (RBAC). The *Decrypt* behavior also takes an input and yields an output. The behavior reverts the effect of the encrypting behavior. The behavior forwards all labels but removes the new labels added during the encryption and replaces the characteristic type of all labels referring to the newly created second characteristic types with the original ones. The *Forward* and *Declassify* behaviors are not changed. The *Join* behavior can often not be applied to encrypted data in a reasonable way. However, in cases where data is just bundled in a tuple, the behavior should treat the labels for the newly introduced second characteristic types in the same way as if they were specified for the original characteristic types. For instance, joining encrypted data items, which had compartments in the Need-to-Know access control model, should yield encrypted data, in which the labels describing the original compartments are the union of all original compartments of incoming data. After the decryption, the joined data item has the same labels as if the input to the joining behavior was unencrypted data. Without this extended joining logic, the effect of creating a tuple from the data would be lost during the decryption.

Label Comparison. The label comparisons introduced in the analysis definitions for information flow and access control remain the same because the additional labels are only necessary between encryption and decryption and the effect of encryption and decryption is also visible on the existing labels. No additional checks regarding encryption are necessary. For instance, if the travel agency in our running example receives encrypted credit card data, the existing label comparison function would, correctly, not report a violation because the travel agency cannot access the contained information and the classification of the encrypted data would, consequently, be the lowest level. The travel agency has a clearance for the lowest level, so access to the encrypted credit card data is fine.

6.3.2. Encryption With Key Pairs

The encryption using key pairs has the same goals as the previously presented encryption without keys but now keys are mandatory inputs to the encryption and decryption processes. The idea to consider keys on top of the keyless encryption is twofold: First, the encryption and decryption should only have an effect if the correct keys are given. Second, the label comparison should detect incorrect key usage to reveal errors in applying the encryption. We assume an encryption by public keys and a decryption by private keys. Using a public and private key is a common approach for asymmetric encryption. The encryption can use any public key to encrypt the content for one or more people, which the given public key(s) represent. The decryption can only use any private key that matches a public key, for which the data item has been encrypted. An additional label comparison detects decryption attempts by wrong private keys. In the following, we describe the key usage and label comparison as extension to the encryption without keys.

Characteristic Types. To represent keys, it is necessary to declare its type, i.e. private or public, and the identity, which the key represents. We introduce the characteristic types *public key of* and *private key of* with a common value range of identities. The order of the identities has no meaning. A public key has a *public key of* label assigned. A private key has a *private key of* label assigned. After the encryption, we have to keep record of the public keys, for which the data item has been encrypted. To do so, a *decryptable by* characteristic type, which also uses the identities as values, represents the identities, which can decrypt the data item using their private key.

Behaviors. All but the following behaviors remain the same. The *Encrypt* behavior takes a public key as an additional input. In addition to the previously described effects, the encrypting behavior creates one *decryptable by* label for each *public key of* label on the received public key data. There can be multiple labels because the data item can be encrypted for multiple public keys. The *Decrypt* behavior takes a private key as an additional input. In addition to the previously described effects, the decrypting behavior only has an effect if the *private key of* label of the private key data has a value, which is also a value of any *decryptable by* label on the data to encrypt. Otherwise, the data item remains encrypted. The *Join* behavior also has to consider the *decryptable by* label. If two data items arrive, the intersection of the *decryptable by* labels of both inputs is applied to the output. This is reasonable because only people, who can decrypt both data items, can decrypt the joined data item.

Label Comparison. The label comparisons of the extended analysis definitions do not have to be adjusted. However, we suggest adding a second query, which can identify failed decryptions. A decryption fails if there is no private key of an identity, which is listed as being able to decrypt the data. Listing 6.14 presents this query. First of all, the query is only applicable to nodes, which have the *Decrypt* behavior. The clause in line 1 selects nodes, which have this behavior. In line 2, two different input pins of the same node `N` are found. The identities represented by the private keys given as incoming data of `PIN0` are found in line 3. We use the `findall/3` predicate to find all identities. In line 4, we find all identities, which are allowed to decrypt the incoming data. Because the private keys and

Listing 6.14: Query for detecting invalid usage of private keys in decryption.

```
1 ?- behavior(N, 'Decrypt'),                                   % decryptor
2 inputPin(N, PIN0), inputPin(N, PIN1), PIN0 \== PIN1,         % pins
3 findall(X, characteristic(N,PIN0,'private key of',X,S0), L_PROV), % prov. keys
4 findall(X, characteristic(N,PIN1,'decryptable by',X,S1), L_REQ),  % req. keys
5 sort(L_PROV, PROV), sort(L_REQ, REQ),                         % sets
6 intersection(PROV, REQ, []).                                  % violation
```

the data item to be decrypted arrive at a node via different pins, the flow trees `S0` and `S1` are different. Because `findall/3` yields lists instead of sets, we build the sets `PROV` and `REQ` using the built-in predicate `sort/2` in line 5. A violation occurs if the intersection between the identities of the private keys `PROV` and the identities allowed to decrypt `REQ` is empty. Line 6 tests this by the built-in predicate `intersection/3`.

6.4. Mixing Existing Confidentiality Analyses

A mixed confidentiality analysis means that a single analysis considers multiple confidentiality mechanisms. Mixing multiple confidentiality mechanisms can be useful or even required to cover realistic systems. For instance, a system of systems does not require all systems using the same confidentiality mechanism but every system can decide on its own depending on its particular confidentiality requirements. Systems can also decide to combine multiple confidentiality mechanisms to achieve certain requirements in an efficient way by combining the benefits and strengths of individual confidentiality mechanisms. For instance, there are existing works [XBS06; Wan+09] that demonstrate the benefits of combining information flow and access control.

In the following, we describe the required steps to combine multiple confidentiality analyses. As a running example, we will use the combination of RBAC with a taint-analysis. A taint-analysis taints, i.e. marks, data and taints all data, which gets in touch with tainted data. Tainted data must not reach critical system parts. The semantics of a taint is often that tainted data originates from untrusted sources and must, therefore, not be used in critical system parts without proper validation. A validation removes the taint and critical system parts can use the resulting data. Essentially, the taint-analysis is an information flow analysis using a linear ordered lattice: The lattice is *not-tainted* → *tainted*. Critical nodes are cleared for *not-tainted* data, which means that *tainted* data must not flow to such critical nodes. The RBAC analysis works as already described in Section 6.2.2.3.

To combine two analyses, a security expert has to merge the corresponding analysis definitions. In the following, we assume that both analyses do not have side-effects on each other. If analyses have such side-effects, the resulting mixed analysis is actually a new analysis and security experts should define it like they define a new analysis. To merge side-effect free analyses, the security experts have to merge the characteristic types and characteristics first. Thereto, the security experts build the union of characteristic types or characteristics respectively. In the example, the characteristic types would be the *permitted*

Listing 6.15: Query for detecting violations of RBAC requirements and requirements regarding the use of tainted data.

```
 1  % generic selection of input pin and incoming flow tree
 2  ?- inputPin(N, PIN), flowTree(N, PIN, S),
 3  (
 4    ( % RBAC analysis detecting missing assigned roles to access data
 5      findall(R, nodeCharacteristic(N, 'assigned roles', R), AR_L),
 6      findall(R, characteristic(N, PIN, 'permitted roles', R, S), PR_L),
 7      sort(AR_L, AR), sort(PR_L, PR), intersection(AR, PR, [])
 8    );
 9    ( % information flow analysis detecting illegal flow of tainted data
10      nodeCharacteristic(N, 'clear', V_CLEAR),
11      characteristicTypeValue('clear', V_CLEAR, N_CLEAR),
12      characteristic(N, PIN, 'class', V_CLASS, S),
13      characteristicTypeValue('class', V_CLASS, N_CLASS),
14      N_CLASS > N_CLEAR
15    )
16  ).
```

roles and *assigned roles* from RBAC as well as the *clearance* and *classification* from the information flow analysis. Because it is possible to assign multiple characteristics to nodes as well as data, building the union of existing characteristics is reasonable. In contrast, nodes can only have one behavior. Therefore, it is necessary to merge behaviors by merging the contained assignments. In the example, there would still be the *Forward*, *Join* and *Declassify* behaviors. The *Forward* behavior is identical in both analysis definitions, so no changes are necessary. The *Join* behavior has to assign the intersection of the permitted roles as well as the highest classification to the output. Because *Declassify* behaviors are specific to particular systems and situations, it is reasonable to keep these behaviors separated. Therefore, there will still be behaviors, which add or remove permitted roles and there will be a behavior lowering the classification of data. The label comparison is the disjunction of the clauses of both individual queries. However, duplications can be removed. A disjunction is necessary because the query shall report a violation of either the RBAC or the information flow analysis. Listing 6.15 presents the result for the example. First, the common parts of selecting an input and a corresponding flow tree for a node are extracted from both individual queries (line 2). Afterwards, the clauses of the RBAC query (lines 4 to 8) and the clauses of the information flow query (lines 9 to 15) are connected by a disjunction. This means, a violation occurs if any group of the analysis clauses evaluates to true. An alternative solution would be to execute both queries of the individual analyses after each other.

6.5. DSL for Defining Custom Analyses

The main motivation for creating a DSL for specifying label comparisons is to provide software architects with means to specify analyses (R2.5) without the need for deep knowledge about the formal semantics and logic programming. Custom analyses are a benefit compared to the state of the art, which either focuses on fixed, predefined analyses [TSB19; HSS14] or only supports simple well-formedness constraints given in query languages such as the Object Constraint Language (OCL) [AGI13]. In contrast, a label comparison function triggers the label lookup and is, therefore, more complex than simple well-formedness constraints. A DSL provides a tailored specification language focused on the architectural domain, which software architects know well. Based on the specification given in the DSL, an automated mapping generates a query for the logic program, which identifies violations.

The adjusted procedure of confidentiality analyses using the DSL is as shown in Figure 6.2. The security expert still provides the characteristic types and behaviors but the architect now not only specifies the system but also a constraint. The constraint is then transformed to a label comparison function, which identifies violations. To transform the constraint, the trace of the system mapping, i.e. a record describing which element from the system has been mapped to which element in the logic program, is necessary. Otherwise, the constraint could not refer to characteristic types or system elements. The procedure shows that all aspects related to the logic program are still hidden from the architect but he/she can now define a constraint, which eventually defines the label comparison function.

In the following, we define the scope of the DSL in Section 6.5.1. The abstract syntax and the concrete syntax, which represent the domain concept, are covered in Section 6.5.2 and Section 6.5.3. We briefly explain the mapping procedure from a specification given in the DSL to a query in the logic program in Section 6.5.4. All explanations are based on previously published work [Hah+21].

6.5.1. Scope

DSLs are languages “of limited expressiveness focused on a particular domain” [FP11, p. 27]. Without sacrificing expressiveness, a DSL would only be another concrete syntax for the concepts presented by the underlying domain, which would be Prolog in the context of this thesis. The challenging part is to select an appropriate subset of domain concepts to cover important use cases in a concise manner. We justify the selection of domain concepts (D*n*) by the following discussion of the scope of the DSL.

Confidentiality is violated as soon as unauthorized subjects access data. The extended DFD represents information to decide about the authorization by labels on nodes and data. The previously presented access control and information flow queries compare the labels of nodes and the labels of arriving data in queries. Consequently, the DSL has to provide means to D1) select a node and D2) select incoming data in order to select the corresponding labels. In addition, it can be necessary to also consider the origin of arriving data. Therefore, the DSL has to provide a way to D3) select the origin of data, i.e. a node in the flow tree.

In a query, the label comparison describes a pattern based on labels, which must not appear in the DFD. To correspond to this, the DSL also has to provide means to D4) specify a

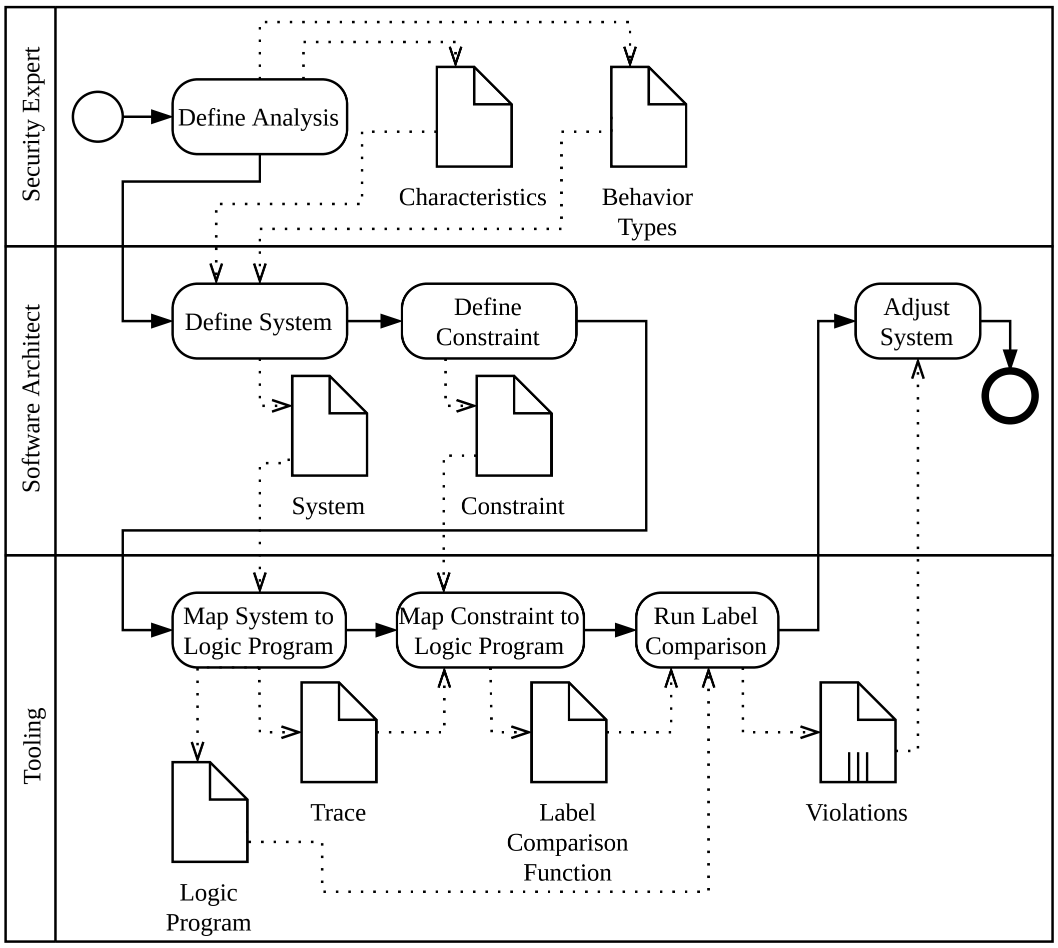


Figure 6.2.: Overview on analysis procedure when using the DSL given as BPMN diagram.

pattern, which indicates a violation of confidentiality requirements in a DFD, i.e. a pattern, which must not appear.

There are different kinds of patterns. A fixed pattern identifies data and nodes based on fixed labels. For instance, a pattern could look for a node cleared for low data that receives data classified high. The *low* label on the node and the *high* label on data are fixed. In contrast, a flexible pattern determines the labels on a node and a data item and compares these labels. For instance, a pattern could capture the clearance of a node and the classification of data and test whether the classification label is higher than the clearance label based on the index in the enumeration, which defines the labels. The pattern does not use any fixed label. Mixing static and flexible labels is also possible. The DSL has to support D5) fixed patterns, D6) flexible patterns and D7) mixed patterns using fixed and flexible parts. Because the DSL refers to labels, it has to have a mechanism to D8) reference existing characteristic types.

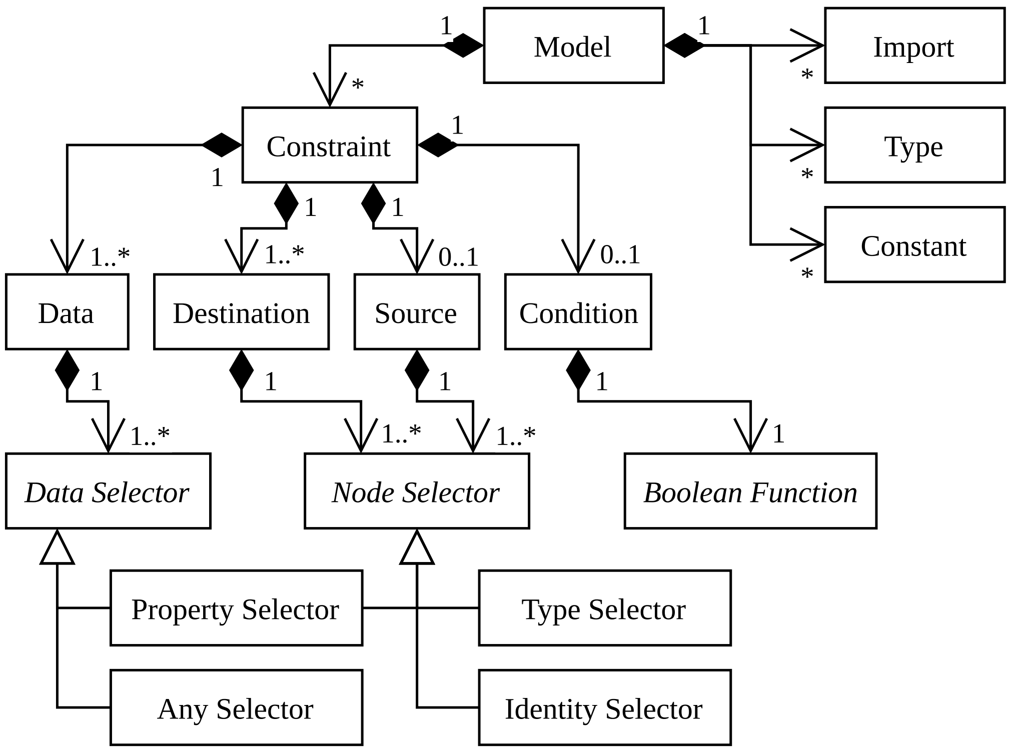


Figure 6.3.: Overview on metamodel of DSL given as UML class diagram.

6.5.2. Abstract Syntax

The abstract syntax describes the domain concepts, their relations and properties. We use a metamodel given as UML class diagram to describe the abstract syntax like we already did for describing the abstract syntax of extended DFDs. An overview on the most important parts of the metamodel is given in Figure 6.3. In the following, we explain the elements of the metamodel, their intuitive semantics and how the elements map to the domain concepts (D*n*) collected in Section 6.5.1.

The software architect specifies one or multiple constraints. A Constraint describes a situation in the DFD, which shall not appear, i.e. a violation (D4). This means a constraint represents one particular query for detecting violations in the logic program. Considering multiple constraints is reasonable because it can be easier to write multiple specific constraints than writing a large, combined constraint. We essentially created two queries for representing DAC in Section 6.2.2.1 (one query for detecting read violations and one query for detecting write violations) because the resulting queries were easier to understand than a merged query.

The constraint specifies a label pattern that shall never appear in the DFD. A pattern has to specify a set of labels on data and a set of labels on nodes. To specify relevant data items (D2), the DSL uses a set of Data Selectors as part of a Data description. To specify a node, which receives data (D1), the DSL uses a set of Node Selectors as part of a Destination description. If required, the DSL can also describe a node, which is (one possible) source

(D3) of the received data item, by a set of Node Selectors as part of a Source description. In order to select a particular data item or a node, all selectors have to apply, i.e. have to evaluate to true. Besides logical conjunction, it would also be possible to provide means to specify logical disjunction between selectors. In order to reduce the complexity of the DSL, we decided to only support conjunction. However, extending the DSL by disjunction or negation would be possible.

There are various types of selectors to represent commonly used selection criteria for data and nodes. The only information for selecting a data item are the labels applied to the data item. Therefore, a selection either has to consider any data item or provide selection criteria based on labels. The Any Selector selects all data items without any restriction. The Property Selector selects a data item based on the applied labels. One or multiple labels, which must refer to the same characteristic type, can be specified. If labels referring to different characteristic types shall be available, multiple Property Selector can be combined. The specified labels describe fixed patterns of labels (D5). However, it is also possible to only specify the characteristic type in a Property Selector and capture available labels, which refer to the characteristic type, in a variable. Later, the DSL allows to specify criteria on such variables. This provides means to specify flexible patterns of labels (D6). Because fixed labels and variables can be mixed, mixed patterns of labels (D7) are also supported. Nodes can also be selected by a Property Selector because they also have labels applied. In addition to labels, it is also possible to select nodes based on their type by using a Type Selector. This is reasonable because some violations can only appear at actors or stores. For instance, the ABAC query in Section 6.2.2.4 also limited its scope to actors or actor processes. In case of constraints, which only affect individual nodes, it can be useful to select nodes based on their identity. An Identity Selector provides this ability.

To formulate flexible and mixed patterns, the DSL has to provide means to compare the variables of the Property Selectors. The Condition allows to use a Boolean Function to do so. If the function evaluates to true, the condition holds, which means that a violation has been identified. A boolean function is a function, which yields a boolean value. If a function requires a parameter of a certain type and another function yields this type, the functions can be nested. The DSL provides the boolean functions shown in Table 6.2. The first three functions are logical connections, which allow to model conjunctions, disjunctions and negations. This set of boolean functions is functional complete [End01, p. 49], i.e. all truth tables, which can be constructed based on the given boolean parameters, can be expressed. This expressiveness is useful because conditions are often not as simple as performing one single test on variables. Besides the logical connections, the DSL provides functions for performing tests on sets of labels, individual labels and on integers.

The functions for performing tests on sets of labels shall also be functional complete, i.e. all tests covered by the algebra on sets shall be expressible. The functions set union $\cup$, set intersection $\cap$ and set complement X^C are functional complete to construct sets based on existing sets [Sto79, p. 18]. The DSL covers all of these functions as shown in the list of supported functions for constructing sets in Table 6.3. The provided complement function provides a relative complement, i.e. a complement of a given set X to another set U, for which $X \subseteq U$ holds. The elements in U depend on the particular context. Therefore, the

Function	First Input	Second Input
And	Boolean	Boolean
Or	Boolean	Boolean
Negation	Boolean	Boolean
EmptySet	CharacteristicSetReference	–
Subset	CharacteristicSetReference	CharacteristicSetReference
ElementOf	CharacteristicReference	CharacteristicSetReference
Equality	CharacteristicReference	CharacteristicReference
Inequality	CharacteristicReference	CharacteristicReference
GreaterThan	Integer	Integer
LessThan	Integer	Integer

Table 6.2.: DSL functions yielding a boolean result.

Function	First Input	Second Input
Intersection	CharacteristicSetReference	CharacteristicSetReference
Union	CharacteristicSetReference	CharacteristicSetReference
Subtract	CharacteristicSetReference	CharacteristicSetReference
Complement	Characteristic Types [1..*]	CharacteristicSetReference
CreateSet	CharacteristicReference	–

Table 6.3.: DSL functions yielding a reference to a characteristic set.

function in the DSL takes a set of characteristic types as a first argument to construct U as a union of all literals of the characteristic types. The second argument is the set X of labels, for which a complement shall be constructed. The subset test $\subseteq$ is feature complete to perform tests on existing sets [Sto79, p. 20]. The DSL supports the subset test as shown in Table 6.2.

To initially define sets, either variables or the CreateSet function (see Table 6.3) can be used. Variables can stem from data and node selectors or from Constants. Considering variables from the selectors is necessary to evaluate the available labels on nodes and data, which is essential for identifying violations. Constants provide sets of fixed labels, which can be used to construct new sets or compare existing sets with. For instance, a constant can provide the empty set or a set of all labels of a certain characteristic type. Both sets are valuable in comparisons. The CreateSet function converts a single label to a set, which is necessary to compare it with sets.

The remaining boolean functions EmptySet, ElementOf, Equality and Inequality from Table 6.2 are just shorthands for combinations of the previously described functions. Testing, whether a set X is empty, can be expressed by $X \subseteq \{\}$, where a constant provides the empty set. Testing whether an element x is part of a set X can be expressed by $\{x\} \subseteq X$, where the CreateSet function constructs a set from an element x. Testing for equality of two individual elements x and y can be expressed by $\{x\} \subseteq \{y\} \wedge \{y\} \subseteq \{x\}$. Testing for

Function	First Input	Second Input
Index	CharacteristicReference	—

Table 6.4.: DSL functions yielding an integer result.

inequality of two elements is given by negating the equality test. The remaining characteristic set function Subtract from Table 6.3 also just provides a shorthand. Constructing a new set by subtracting set B from set A, i.e. A/B, can be expressed by $A \cap B^C$, where the complement of B is defined with respect to the set of all elements of A and B, i.e. $A \cup B$.

Using the index of a single label in a comparison is a useful feature, which we have used in defining the label comparison for information flow analyses in Section 6.2.1.1. To support this, the DSL provides the Index function shown in Table 6.4, which yields the index of the label in the corresponding enumeration as integer. The boolean functions GreaterThan and LessThan test whether integer i is greater than j or whether integer i is less than integer j, respectively. Because the only function providing integers in the DSL is the Index function, those two boolean functions essentially always compare indexes of labels.

The only domain concept, which we have not explained so far, is the reuse of existing characteristic types (D8). Essentially, we already referred to characteristic types in the previous explanations and assumed that they are available. To actually make them available, we have to specify the location of the characteristic types as well as which of the characteristic types at the specified location shall be used. To specify the location, the DSL provides an Import statement. To specify the characteristic types to be used, the DSL provides a Type statement. The type statement allows to rename the characteristic type in the context of the DSL to make specifications more concise.

6.5.3. Concrete Syntax

The concrete syntax assigns one possible representation to the elements of the abstract syntax. The sole purpose of the concrete syntax in this thesis is to make examples given in the DSL readable and comprehensible. Therefore, we do not consider the concrete syntax a contribution. Nevertheless, we define the concrete syntax according to common best practices [Kar+09]. One of the provided guidelines suggest to use the same style over multiple languages, which might be used. We use a textual syntax because the style of writing expressions in textual notations is more common than writing them in graphical notations. We extensively use expressions in conditions, which contribute a considerable amount of expressiveness to the DSL. Therefore, it is reasonable to tailor the language to writing expressions.

In the following, we introduce the concrete syntax using two examples. The first example is a simple information flow analysis that detects when data, which has been classified as *high*, arrives at a node, which has been cleared for *low* data. The example demonstrates the usage of fixed label patterns to formulate constraints. The second example is the

Listing 6.16: Constraint requiring data classified high to never flow to nodes cleared for low data.

```
1 import "characteristicTypes.xmi"
2 type Classification : "Classification"
3 type Clearance : "Clearance"
4
5 constraint NoFlowHighToLow {
6    data.property.Classification.High
7       NEVER FLOWS
8    node.property.Clearance.Low
9 }
```

information flow analysis using a linear ordered lattice as demonstrated in Section 6.2.1.1. The example demonstrates the usage of flexible label patterns to formulate constraints. We explain the remaining DSL elements not covered by the two examples afterwards.

The constraint definition for the first example is given in Listing 6.16. The constraint refers to particular labels, which are a high data classification and a low node clearance. Before the architect can use these labels, he/she has to specify the location of the characteristic types via the `import` statement followed by the location (line 1). Afterwards, the architect can specify the characteristic types he/she wants to use by the `type` statement. The statement defines the name, which shall be used to refer to the characteristic type, in constraints. This is beneficial because the names can be shortened to make the constraint definition more concise. In lines 2 to 3, the characteristic types effectively do not get a new name because the new name written first is the same as the old name written second. A constraint can now refer to the characteristic type. To define a constraint, the keyword `constraint` is used followed by the name of the constraint. In line 5, the name is `NoFlowHighToLow`. The definition of the constraint is placed within a block delimited by curly brackets, which is a commonly used notation in various programming languages.

The constraint definition for fixed label patterns is usually shorter than for flexible label patterns because only few specification elements are necessary. Every constraint definition has to contain at least one data selector to specify the data to be tested and at least one destination selector to specify the node to be tested. A data selector always starts with the keyword `data` as shown in line 6. A data selector for a label continues with the keyword `property` followed by the characteristic type of the label. In our example, we would like to select data, which is classified high. Therefore, we append the `High` literal to the characteristic type. All parts of the data selector are connected by dots, which is also a commonly used notation for navigating through properties of elements in programming languages. A node selector always starts with the keyword `node` as shown in line 8. A node selector for a label continues with the keyword `property` followed by the characteristic type of the label. In our example, we select the `Low` literal from the `Clearance` characteristic type. The selectors are connected by the `NEVER FLOWS` keywords (line 7).

The constraint definition for flexible label patterns is based on the previously described language elements but uses conditions and variables in addition. The example shown in Listing 6.17 presents the constraint used to detect violations of information flow require-

Listing 6.17: Constraint requiring that data classifications must never be lower than node clearances.

```
 1 import "characteristicTypes.xmi"
 2 type Classification : "Classification"
 3 type Clearance : "Clearance"
 4
 5 constraint NoFlowAgainstLattice {
 6    data.attribute.Classification.$CLASS
 7       NEVER FLOWS
 8    node.property.Clearance.$CLEAR
 9       WHERE
10    index(CLASS) > index(CLEAR)
11 }
```

ments given by a linear ordered lattice. The language constructs up to including line 5 are the same as in the previous example. The first difference is the data selector in line 6. Instead of a fixed literal, the selector defines a variable `CLASS`. To indicate that the name is not the name of a literal but the name of a variable, a dollar sign `$` is prepended. Various programming languages such as PHP use dollar signs to mark variables. The node selector in line 8 also introduces a variable instead of using a fixed literal. As already explained as part of the abstract syntax, the meaning of a variable is that the particular classification or clearance label is stored in the variable. To compare the variables, which makes the label pattern flexible, architects have to formulate a condition on the variables. A condition always starts with the `WHERE` keyword (line 9), which is also used in query languages such as SQL to start conditions. The actual condition is given by a boolean expression. In our example, the condition is that the index of the data classification literal is greater than the index of the node clearance literal. The boolean expression is given by the greater function. The operands are integers yielded by the `index` functions. The greater function is an infix operation taking two arguments, which is the commonly used definition for this function in many programming languages.

Constants provide the means to define variables holding labels without referring to particular data or nodes. Constants are useful if predefined sets of labels shall be used in multiple comparisons. In ABAC, such predefined sets can represent the subject and object identifiers. Listing 6.18 illustrates various constant definitions. Constants are defined after types and before constraints. A constant always starts with the keyword `const` followed by the identifier of the constant. If a constant describes a set of labels, the identifier has curly brackets as a postfix. This is the same notation as for defining set variables in data and node selectors. To define the constant, an equal sign followed by the characteristic type and a literal selector has to be given. If a constant only holds one label like shown in line 1, the literal can be directly connected to the characteristic type via a dot. If multiple labels shall be part of a set like shown in line 2, the literals are written within square brackets and separated by commas. If all available literals of a characteristic type shall be selected, an asterisk can replace the literal selector as shown in line 3. The notation to select labels is the same as the selection of fixed labels in node and data selectors.

Listing 6.18: Excerpt of concrete syntax demonstrating the definition of constants.

```
const HIGH           = Classification.High
const HIGH_AND_LOW{}  = Classification.[Low,High]
const ALL{}           = Classification.*
```

Listing 6.19: Excerpt of concrete syntax demonstrating the use of selectors.

```
1 constraint Demonstration {
2    data.any
3       NEVER FLOWS
4    node.type.Actor & node.property.Clearance.LOW
5       FROM
6    node.property.Clearance.HIGH
7 }
```

Within the constraint, a software architect can use more data and node selectors than shown in the previous two examples. Listing 6.19 shows an excerpt of a constraint requiring that any type of data must never flow from a node with high clearance to an actor node, which has low clearance. To not restrict the considered data, the `any` selector can be used on data as shown in line 2. Besides the properties of a node, its type can also be considered as shown in line 4. Possible choices of node types are `Actor`, `ActorProcess`, `Store` and `Process`. To select a node, from which the selected data has been transitively received, the `FROM` keyword can be used after the node selectors of the destination as shown in line 5. After the keyword, all node selectors, which can also be used to select the destination node, can be used as shown in line 6.

Besides the already presented functions, all functions introduced in Section 6.5.2 have a corresponding concrete syntax and can be used. To solve ambiguities when using multiple functions, we define the function precedence as well as the associativity as shown in Table 6.5. Functions with precedence i are evaluated before functions with precedence j if $i > j$. We use the infix notation, i.e. we place the function symbol between the two operands, if common programming languages also use this notation. For instance, it is a common approach to place the function symbols for logical conjunction and disjunction between the two operands. The same holds for comparisons known from arithmetic such as testing for greater, less, equal and inequal operands. For all remaining functions, we use the prefix notation. This is reasonable for set functions because these functions use dedicated symbols when provided in infix notation. However, using special symbols not available in the ASCII character set makes using the DSL more complicated. To meet the standards of common programming languages, we made the logical conjunction and disjunction left-associative and the logical negation right-associative. The remaining functions are not associative because there is no ambiguity to solve: the boolean infix functions and the remaining prefix functions cannot appear directly after each other or clearly mark their arguments using parentheses.

Precedence	Associativity	Notation	Functions
1	left	infix	Or
2	left	infix	And
3	right	prefix	Negation
4	none	infix	Greater, Less, Equality, Inequality
4	none	prefix	all remaining functions

Table 6.5.: Function precedence, associativity and notation in DSL.

6.5.4. Mapping to Logic Program

We assign formal semantics to the elements of the DSL by describing how to map these elements to Prolog clauses. Using a mapping to describe the formal semantics of elements of an abstract syntax is the same approach as we have used in assigning formal semantics to the elements of the extended DFDs. In the following, we describe one fundamental assumption for the mapping and explain the mapping rules afterwards.

The following mapping rules assume that the mapping from the DFD, on which the query shall be executed, to the logic program already has been executed and that a transformation trace is available. The trace describes which DFD element has been mapped to which element in the logic program. This information is necessary to resolve references from the DSL constraint to elements of the extended DFD. In order to resolve, for instance, a reference to a characteristic type in a constraint, the mapping has to look up the clause in the logic program, which represents that particular characteristic type. The clauses resulting from mapping the constraint to a logic program can then refer to the correct characteristic type clause.

A constraint describes a pattern for detecting certain data, which arrives at a certain node. To allow to evaluate that pattern, a rule taking the node, the input pin and a flow tree is useful. This triple of information uniquely identifies an arriving data item. Because there can be multiple constraints, it is necessary to distinguish multiple rules for multiple constraints. An argument, which contains the name of the constraint, is a good approach because this preserves the particular constraint name and, therefore, makes it easy for a software architect to relate a result to a particular constraint later. Encoding the constraint name in the rule name would also be possible but provides no benefit compared to the argument containing the constraint name. The resulting rule looks like illustrated in the first mapping rule shown in Figure 6.4. The upper part shows an excerpt of a constraint given in the DSL. The lower part shows an excerpt of the resulting logic program. The body of the rule binds the variables given in the head of the rule by using the `inputPin/2` and `flowTree/3` clauses. The two clauses are necessary for considering all possible combinations of nodes `N`, input pins `PIN` and flow trees `S` by reevaluating the rule. The following mapping rules extend the rule body by the selection criteria. Because the selection criteria have to match the data and node, the criteria is added to the conjunction of clauses in the rule body, i.e. the clauses have to be true.

```
constraint Foo {
  // content omitted
}
```

$$\Downarrow$$

```
constraint('Foo', N, PIN, S) :-
  inputPin(N, PIN),
  flowTree(N, PIN, S).
```

Figure 6.4.: Example of mapping from empty DSL constraint to logic program.

```
const HIGH           = Classification.High
const HIGH_AND_LOW{} = Classification.[Low,High]
const ALL{}          = Classification.*
const EMPTY{}        = []
```

$$\Downarrow$$

```
constraint('Foo', N, PIN, S) :-
  % omitted standard clauses
  Var_HIGH             = 'High',
  VarSet_HIGH_AND_LOW = ['Low', 'High'].
  VarSet_ALL           = ['Low', 'High'],
  VarSet_EMPTY         = [],
```

Figure 6.5.: Example of mapping constants from DSL to logic program.

Constants provide variables to be used in comparisons or set functions inside conditions. A constant contains an arbitrary number of literals. It is reasonable to consider literals instead of labels, i.e. the tuple of characteristic type and literal, because this enables flexible comparisons: All analyses described in Section 6.2 compare literals from different characteristic types, which is possible because the value range, i.e. the underlying enumeration, is the same. Therefore, constants can be mapped to literals or sets of literals as shown in Figure 6.5. A constant, which refers to a literal, is mapped to a variable, which represents exactly this literal. In this and all following examples, the identifiers of the literals and characteristic types are just their names for the sake of simplicity. In the actual mapping, the identifiers would be unique and would have to be looked up in the transformation trace of the DFD. A set constant is mapped to a variable unified with a list. If particular literals are given, these literals are added to the list. A wildcard $*$ is mapped to all literals of that particular characteristic type. Empty set constants are also possible and mapped to an empty list. All variables resulting from the mapping are added to the rule body after the standard clauses as presented in Figure 6.4. This is reasonable because all following clauses can refer to these variables, which is the expected usage scope of global constants.

Data selectors specify selection criteria on data. There are two types of selectors. The Any Selector selects any data, which means that the constraint shall consider all possible incoming data items. Because the standard clauses already consider all possible incoming data items, no further clauses are necessary. The Foo constraint in Figure 6.6 uses the Any Selector, which does not lead to any additional clauses in the corresponding Prolog rule.

```
 1 constraint Foo {
 2   data.any NEVER FLOWS // remainder omitted
 3 }
 4 constraint Bar {
 5   data.property.Classification.High &
 6   data.property.Classification.$CLASS{} NEVER FLOWS // remainder omitted
 7 }
```

⇓

```
 8 constraint('Foo', N, PIN, S) :-
 9   % omitted standard clauses
10   % no added clauses based on any selector
11 constraint('Bar', N, PIN, S, VarSet_CLASS) :-
12   % omitted standard clauses
13   (
14     characteristic(N, PIN, 'Classification', 'High', S),
15     findall(V, characteristic(N, PIN, 'Classification', V, S), VarSet_CLASS)
16   )
```

Figure 6.6.: Example of mapping data selectors from DSL constraint to logic program.

The Property Selector selects data items based on their assigned labels. A selector referring to a particular label such as the selector in line 5 of Figure 6.6 requires that a particular label is available on the data item. The `characteristic/5` clause shown in line 14 has the same meaning: the clause evaluates to true if the particular label is available for the node, pin and flow tree under consideration. Consequently, the mapping generates such a `characteristic/5` clause for every property selector referring to a particular label. If the property selector introduces a characteristic set variable, the intended meaning is that the variable captures all literals for the given characteristic type on the incoming data item. It does not influence the selection of data directly because only conditions evaluating the introduced variable affect the selection. In line 6 of Figure 6.6, all classification literals shall be captured in the variable `CLASS`. In Prolog, the clause in line 15 has the same semantics. The `findall/3` clause finds all solutions for the goal template given as second argument. The variable given as first argument represents a single solution in the goal. The variable in the third argument is unified with the list of solutions, i.e. all solutions. The goal template is the same clause as for matching particular literals but the goal template uses the solution variable in place of the particular literal. The solutions are available via the third argument, which is the variable `VarSet_CLASS` in Figure 6.6. For every variable introduced in the DSL, an additional argument is added to the head of the `constraint/4` clause. This is necessary to report the contents of the variable back to the software architect. The meaning of multiple selectors in the DSL is that all individual selectors have to match. Using a conjunction to connect the individual clauses resulting from property selectors has the same meaning in Prolog.

Node selectors specify selection criteria for nodes. The selectors can specify the Destination of data as well as the Source of data. In the following, we explain the usage of the selectors to specify the Destination and will explain the usage for specifying the Source in the next

```
 1 constraint Foo {
 2   // data selector omitted
 3   NEVER FLOWS node.property.Clearance.Low
 4 }
 5 constraint Bar {
 6   // data selector omitted
 7   NEVER FLOWS node.type.Actor & node.property.Clearance.$CLEAR{}
 8 }
```

$$\Downarrow$$

```
 9 constraint('Foo', N, PIN, S) :-
10   % omitted standard and data selector clauses
11   nodeCharacteristic(N, 'Clearance', 'Low').
12 constraint('Bar', N, PIN, S, VarSet_CLEAR) :-
13   % omitted standard and data selector clauses
14   (
15     actor(N),
16     findall(V, nodeCharacteristic(N, 'Clearance', V), VarSet_CLEAR)
17   )
```

Figure 6.7.: Example of mapping node selectors from DSL constraint to logic program.

paragraph. There are three types of selectors. The Property Selector selects nodes based on assigned labels. A fixed label in the property selector as illustrated in line 3 of Figure 6.7 means that that fixed label has to be assigned to a node. The `nodeCharacteristic/3` clause in line 11 with the label, i.e. the characteristic type and the literal as second and third parameter, tests this criteria on a node `N`. Besides fixed labels, variables can capture literals of a certain characteristic type applied to a node. The second selector in line 7 introduces the `CLEAR` variable to capture all applied clearance labels of the node. In Prolog, a `findall/3` clause as shown in line 16 captures the literals in the variable `VarSet_CLEAR`. Node selectors, which introduce variables, do not imply restrictions for selecting nodes but just capture information to be evaluated later. The Type Selector selects nodes based on their type. The first selector in line 7 selects actor nodes. In Prolog, the corresponding counterpart is the clause, which introduced the identifier of a node of a certain type. In the particular example, the `actor/1` clause as shown in line 15 ensures that the node identifier `N` belongs to an actor. The mapping rules for other node types are analogous. The Identity Selector selects nodes based on their identity. To map such a selector to Prolog, the mapping first looks up the node in the trace to find its identifier in Prolog. Second, a unification of the node variable `N` with this identifier is added. The meaning of multiple selectors in the DSL is that all individual selectors have to match. Using a conjunction to connect the individual clauses resulting from the selectors has the same meaning in Prolog.

To specify the Source of data, the same types of node selectors can be used as for specifying the Destination of data. The major difference in mapping the selectors for sources is that an additional argument `N_FROM` is added to the head of the constraint as shown in line 5 of Figure 6.8. Adding the argument is necessary to report the identified source node back to software architects. The clauses resulting from the mapping of the node selectors are the

```
1 constraint Foo {
2   // data and node selector omitted
3   FROM node.type.Actor
4 }
```

$\Downarrow$

```
5 constraint('Foo', N, N_FROM, PIN, S) :-
6   % omitted standard and data selector clauses
7   actor(N_FROM).
```

Figure 6.8.: Example of mapping origin node selectors from DSL constraint to logic program.

same but all usages of the variable `N`, which represents the destination node, are replaced by the variable `N_FROM`, which represents the source node. An example of the resulting clause for the type selector used in line 3 is given in line 7.

The condition specifies selection criteria based on the variables used in property selectors. The clauses resulting from mapping the condition are added to the conjunction of previous clauses. The mapping of individual DSL functions to Prolog clauses is as shown in Table 6.6. Functions representing boolean logic and boolean functions are directly mapped to their Prolog equivalents. The functions and Prolog predicates in the same row of the table have the same meaning. The only difference is the mapping of the `EmptySet` function. The function is mapped to the Prolog clause `length/2` with `0` as a fixed second argument. Because a length of zero elements is the same as an empty set, the meaning remains the same. Functions not yielding booleans cannot be mapped as straight forward as boolean functions because Prolog does not support nesting functions like the DSL does. Others such as Cabot, Clarisó, and Riera [CCR14] also recognized this challenge when mapping functional to logic expressions. Logic clauses store the result of a function not yielding a boolean value in a dedicated variable. All functions yielding sets as well as the `Index` operation yielding an integer have a dedicated argument in their signature. Therefore, a function f having another function g as an argument in the DSL refers to the result variable of g in the logic program. In Prolog, it is not necessary that the result of function g is available before evaluating function f because Prolog can determine valid inputs, which make f succeed. If the predicted input is no output of g, the solving algorithm uses backtracking to determine other possible input values for f. However, evaluating the functions in applicative order [ASS96, p. 399], i.e. evaluating all arguments to a function before evaluating the function, can reduce the amount of used backtracking. In the following, we explain the mapping of nested functions using the applicative order.

The algorithm for mapping nested functions is essentially a depth-first search always starting at a boolean function taking at least one non-boolean argument. Only functions are considered in the algorithm. A function is mapped to the corresponding clause according to Table 6.6 and a temporary variable is introduced for representing the result if a result variable is necessary. In the example given in Figure 6.9, the non-boolean function of type `Union` is mapped to the `Union` clause and an anonymous result variable. The arguments of the functions are mapped based on the following rules: A characteristic set reference

Function	Predicate	Result Variable
And	,/2	—
Or	;/2	—
Negation	\+/1	—
ElementOf	memberchk/2	—
EmptySet	length/2	—
Equality	=/2	—
GreaterThan	>/2	—
Inequality	\=/2	—
LessThan	</2	—
Subset	subset/2	—
Complement	complement/3	✓
CreateSet	=/2 and []	✓
Index	characteristicTypeValue/3	✓
Intersection	intersection/3	✓
Subtract	subtract/3	✓
Union	union/3	✓

Table 6.6.: Mapping between functions in conditions of DSL and clauses in logic program.

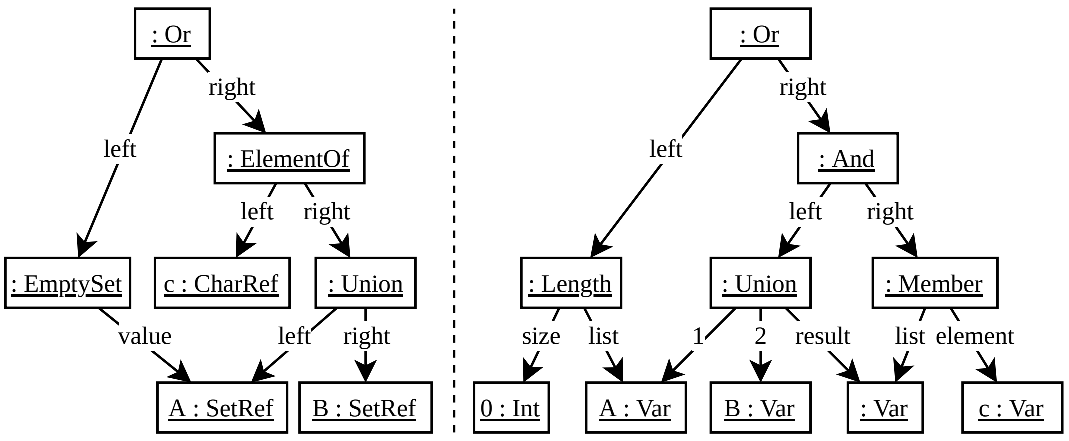


Figure 6.9.: Example of mapping between DSL condition (left) and Prolog (right) given as UML object diagram.

is mapped to a variable. In the example, the characteristic set reference *A* is mapped to a variable *A* in the logic program. A function (used as argument in another function) is mapped to the variable representing the result. In the example, the `Member` clause refers to the result variable of the `Union` clause. Because the depth-first search ensures that all arguments are evaluated before their use, the conjunction of all generated clauses in the order of their generation ensures an efficient evaluation without the need for extensive backtracking.

ID	Description	User	Covering Part
R2.1	every element covered	—	— (already met before)
R2.2	derivation of properties	analysis	— (already met before)
R2.3	origin of properties	analysis	— (already met before)
R2.4	analyses based on goals	expert	analysis procedure
R2.5	analyses based on goals	architect	DSL for custom analyses
R2.6	tracing of properties	architect	— (already met before)
R2.7	automated analyses	architect	analysis procedure
R2.8	information flow	expert	information flow analyses
R2.9	access control	expert	access control analyses

Table 6.7.: Overview on described parts and met requirements by analysis definitions.

6.6. Requirements Coverage

The analysis procedure, the DSL as well as the particular analyses cover the requirements regarding the semantics, which have not been covered so far. Table 5.2 gives an overview on the requirements and how the parts of this chapter address them. Requirements that are not addressed (indicated by dash in the last column) are already covered by the semantics as described in Section 5.2.3.

As part of the analysis procedure, we sketched how security experts can define analyses based on analysis goals (R2.4). We demonstrated the definition of analyses by the security expert for particular information flow analyses as well as access control analyses. Thereby, we showed that the underlying semantics support information flow analyses (R2.8) as well as access control analyses (R2.9). Software architects can define analyses (R2.5) based on their analysis goals using a DSL, which frees the architect from learning logic programming to formulate analyses. As the procedures for analyses defined by security experts as well as by software architects show, the defined analyses can be fully automated after their definition (R2.7). Software architects can execute the analyses without additional inputs besides the software architecture and the predefined analysis definition.

The DFD semantics meet all requirements regarding the semantics, which we demonstrated jointly in this section and in Section 5.2.3.

6.7. Assumptions and Limitations

This section discusses assumptions and limitations of the particular confidentiality analyses and the DSL for formulating analyses.

Consideration of incoming flows Constraints formulated in the DSL describe forbidden data flows to nodes. Consequently, the generated label comparison function tests for illegal incoming data flows, i.e. data flows arriving at an input pin. We assume that only considering incoming data is sufficient because all particular analyses presented in this chapter only consider incoming data flows. Outgoing data flows, i.e. data flows leaving an output pin, are not considered and software architects cannot write constraints for outgoing data. Testing outgoing data flows for confidentiality violations is counter-intuitive: If the node must not access the outgoing data item, it is likely that the node also was not allowed to access the incoming data items used to create the outgoing data item in the first place. If the node produces data, which it is not allowed to access, it is questionable how the node actually produced this data item. It is more likely that the intention of a constraint on outgoing data is to limit the data, which a node can inject into the system. However, such a limitation targets integrity rather than confidentiality.

Focus on confidentiality All presented analyses focus on confidentiality, which is also the focused security objective of this thesis. However, the confidentiality mechanisms usually also provide means for enforcing integrity. We demonstrated how an analysis definition can consider integrity aspects in the DAC analysis for write violations in Section 6.2.2.1 on page 75. Therefore, we are positive that the analysis definition can also capture integrity aspects of the confidentiality mechanisms. However, profound research on considering other security objectives such as integrity is not in the focus of this thesis but left to future work.

Confidentiality requirements in logic program The extended DFDs focus on representing relevant information about data and nodes via labels. The confidentiality requirements are encoded in the label comparison function. If comparing labels is not sufficient to represent the confidentiality requirements, security experts can provide additional information. However, security experts can only provide this information via additional clauses in the logic program. There are no dedicated, tailored models to represent the requirements. We do not see this as a crucial limitation because it is always possible to create a metamodel for representing the requirements. An automated mapping can then transform the requirements into the additional clauses in the logic program.

No state or time in analyses The DFD semantics do not provide a notion of state or time. Therefore, analyses cannot refer to a certain state and cannot build temporal relations. However, this would be necessary to represent specific aspects of confidentiality mechanisms such as the revocation of rights in DAC or changing assignments within RBAC sessions. Missing state and time limits the expressiveness of analyses but it favors system models with low complexity. Considering state or time would require more detailed system specifications, which are more challenging to create for software architects. The gaps in expressiveness only affect specific aspects of confidentiality mechanisms but not the fundamental concepts.

No instance information in analyses The DFDs describe systems, actors and data on a type level, which means they do not represent individual users or data. Consequently, analyses cannot refer to individual actors or data, which would be necessary to consider dynamic constraints in RBAC that affect role assignments to individual users. This limitation is the result of a trade-off between expressiveness and the required complexity for creating the DFDs. It is questionable whether software architects actually have such detailed information about individual actors and data while creating the software architecture.

6.8. Summary

In this chapter, we presented the means for defining analyses and we presented particular analyses. Together, both meet the remaining open requirements regarding the DFD semantics defined in Section 4.1 or at least demonstrate that the semantics meet the requirements. We elaborated on the requirements in Section 6.6.

The analysis procedure covered in Section 6.1 specifies the interaction between a security expert, a software architect and automated tooling. Security experts provide reusable characteristics, behaviors and a label comparison function. Software architects bind these elements to system elements in the software architecture to enrich the architecture by information relevant for confidentiality. The automated tooling maps the enriched software architecture and the label comparison function to a logic program based on the mapping described in Section 5.2.2. Executing the query, i.e. the label comparison function, yields violations.

Representing particular analyses in terms of labels, label propagation and label comparison functions requires security expertise. We illustrate how to define analyses for common information flow and access control mechanisms in Section 6.2. These mechanisms can be extended by encryption as an additional option to protect information, which we describe in Section 6.3. Combining multiple confidentiality mechanisms, and therefore also the corresponding analyses, can provide improved protection of confidentiality. We describe how to integrate multiple analyses as part of Section 6.4.

To support software architects in defining confidentiality analyses, we introduce a DSL, which does not require expertise in logic programming, in Section 6.5. The DSL sacrifices expressiveness compared to label comparison functions specified by logic programming in favor of comprehensibility and low initial learning effort. Constraints specified in the DSL are mapped to a query to the logic program, which can then be used to identify violations.

The major assumption, which we discuss in Section 6.7, is that only incoming data flows are relevant for detecting confidentiality violations. We justify this assumption by the analyses for the most common information flow and access control mechanisms. Major limitations also discussed in Section 6.7 are that analyses cannot refer to state, time or instance level information. This limits expressiveness regarding some specific aspects of confidentiality mechanisms. We do not consider this limitation crucial because the aspects, which we cannot express, only represent a small amount of aspects within the respective confidentiality mechanisms and because information for creating detailed models containing state, time and instances cannot be expected to be available while creating the software architecture.

7. Integrating DFD Analyses in Architectural Description Languages

The DFD syntax and semantics described in Chapter 5 and the DFD analyses described in Chapter 6 provide powerful means to analyze DFDs for violations of confidentiality requirements. However, the restriction to DFDs as ADL is too limiting for software architects because they often use other ADLs as a survey on the use of ADLs [Ozk18] shows. Therefore, we provide guidelines on how to integrate the DFD-based analyses into existing ADLs.

The integration guidelines in Section 7.1 introduce the role of a tool engineer and provide him/her with a process for extending an existing ADL. We refer to this process as *integration procedure*. To demonstrate that the procedure is applicable to ADLs, which use communication based on control flows, and to ADLs, which use communication based on data flows, we apply the procedure to the Palladio ADL in Section 7.2. The Palladio ADL [Reu+16] uses communication based on control flows but can also make use of data flows with a recent extension [WSK20]. By applying the procedure to a subset of Palladio, which uses control flows, and to another subset of Palladio, which uses data flows, we can show the applicability to both types of ADLs. In addition, we show how to make the DSL for formulating custom analyses usable with the ADLs. The integration guidelines as well as the applications to Palladio are based on existing publications [SHR19; Sei+21]. In contrast to our previous publications, we detail the descriptions and provide more fine-grained guidelines.

Eventually, we show that the integration guidelines meet the requirements for these guidelines in Section 7.3. We discuss assumptions and limitations in Section 7.4 and summarize the chapter in Section 7.5.

7.1. Integration Guidelines

The fundamental idea to realize the integration of the DFD analyses into ADLs is to map architectures given in the ADL to a DFD and reuse the analysis capabilities presented in Chapter 5 to identify confidentiality violations. The approach is beneficial because the existing analysis framework as well as the analysis definitions for DFDs can be reused. The prerequisite for defining the mapping between ADL and DFD is that the ADL provides all domain concepts, which are necessary to define the DFD. It can be necessary to extend the existing ADL to bridge potential gaps by adding missing concepts. The DSL for defining constraints can be reused as well but it has to be adjusted to comply with the terminology of the ADL. In the following, we explain these steps of the integration in more detail.

We introduce the *tool engineer* role, which is responsible for executing the integration procedure illustrated in Figure 7.1. The first activity is to compare the domain concepts, which are available in the ADL, with the essential domain concepts for conducting DFD-based analyses. For instance, descriptions of users are one essential domain concept. Next, the tool engineer addresses the identified missing concepts by an ADL extension. A missing concept is a concept, which is required for confidentiality analyses but is not available in the ADL, i.e. the ADL does not provide the required information. The comparison of existing with required concepts lowers the amount of necessary changes in the ADL and increases reuse (R3.3). The resulting extended ADL provides concepts to represent all information required for creating a DFD including confidentiality aspects. Based on the extended ADL, the tool engineer creates a mapping to a DFD. After this step, software architects can map architectures given in the ADL to DFDs and can then use the existing analysis capabilities for DFDs. Reusing the analysis framework for DFDs supports the tool engineer in creating an analysis framework for the ADL, which is as powerful as the analysis framework for DFDs (R3.4). This means software architects can analyze architectures given in the extended ADL for the same confidentiality violations as they already can for DFDs. To formulate custom analyses, software architects need an adjusted DSL for formulating analyses. The tool engineer creates a new so-called constraint DSL, which only uses concepts from the extended ADL. The DSL for the ADL is based on the DSL for DFDs. The DSL together with the mapping allows the architect to model and analyze architectures without knowledge about DFDs and logic programming (R3.5 and R2.5). In conclusion, the integration procedure yields an extended ADL, a mapping form the ADL to a DFD and the constraint DSL. In the following, we give more details on the analysis procedure, the essential concepts and the steps of the integration procedure.

Software architects can conduct analyses in a similar way as for DFDs after the tool engineer has applied the integration procedure to an existing ADL. The analysis procedure shown in Figure 7.2 is an extended version of the analysis procedure for DFDs, which we presented in Section 6.1. The definition of analyses is still the task of the security expert and the activities to be done by the software architect are also the same as for DFDs: he/she has to define the system, might define a custom constraint and has to adjust the system in case of identified violations. The only difference in the analysis procedure is that the tooling has to carry out two mappings: First, the tooling maps the architecture given in the ADL to a DFD. As a result, the step produces a DFD as well as a trace, which links elements from the architecture to newly created elements in the DFD. Second, the tooling maps the DFD to a logic program. As a result, the step produces a logic program as well as a transitive trace, which links elements from the architecture to elements in the newly created logic program. After these two mappings, the remaining analysis procedure is the same as for DFDs. If the software architect has defined a custom constraint, the tooling maps the constraint to a label comparison function. The mapping now uses the transitive transformation trace because the architect used elements from the architecture to define the constraint instead of DFD elements. Afterwards, the tooling runs the label comparison by propagating labels and comparing the labels using the comparison function.

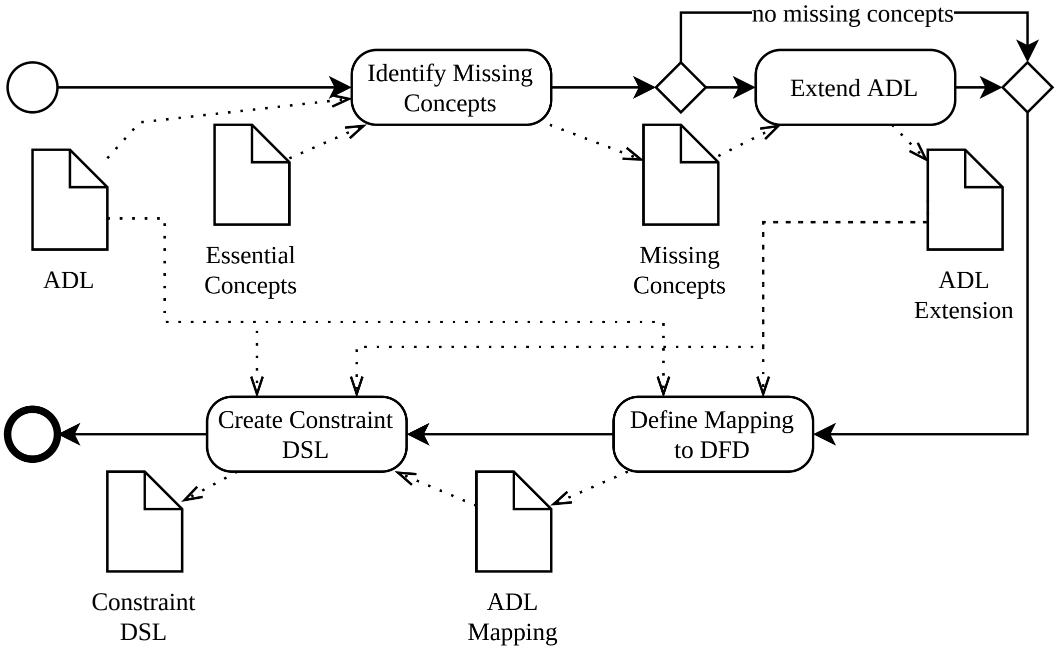


Figure 7.1.: Overview on integration procedure given as BPMN diagram.

The software architect uses the detected violations to adjust the system. The benefit of this procedure is that we can reuse most of the existing tooling as well as existing analyses. The software architect does not have to be aware of the DFD or even logic programming because the tooling does all steps involving DFDs or logic programming automatically without intervention of the software architect. Consequently, the analysis of a modeled architecture can be fully automated (R2.7).

Before we give details on the individual activities of the integration procedure, we collect the essential concepts for deriving a DFD from an architecture given in an ADL. First of all, the ADL has to provide means for describing processing steps (I1) and communication between these steps (I2). These concepts are necessary to derive the system structure consisting of processes and data flows. The processing steps can be given as coarse-grained components or fine-grained activities. The communication can be calls, exchanged events or exchanged data. We do not restrict the communication paradigm here as requested by the requirements to support ADLs using control flows (R3.1) as well as ADLs using data flows (R3.2). A notion of stores, e.g. by describing databases or filesystems, is helpful to recognize stores and recreate them in the DFD but this is not essential. To derive actors, a notion of a user (I3) and his/her interaction with the system (I4) is necessary. Users are crucial because they start and terminate data flows. Covering their interaction with the system is necessary to determine the data flows from and to the users. The concepts up to now represent the DFD structure. ADLs usually already provide these concepts because it is essential to describe the structure and behavior of the architecture. To summarize, the following concepts are essential for representing the DFD structure:

I1) processing steps

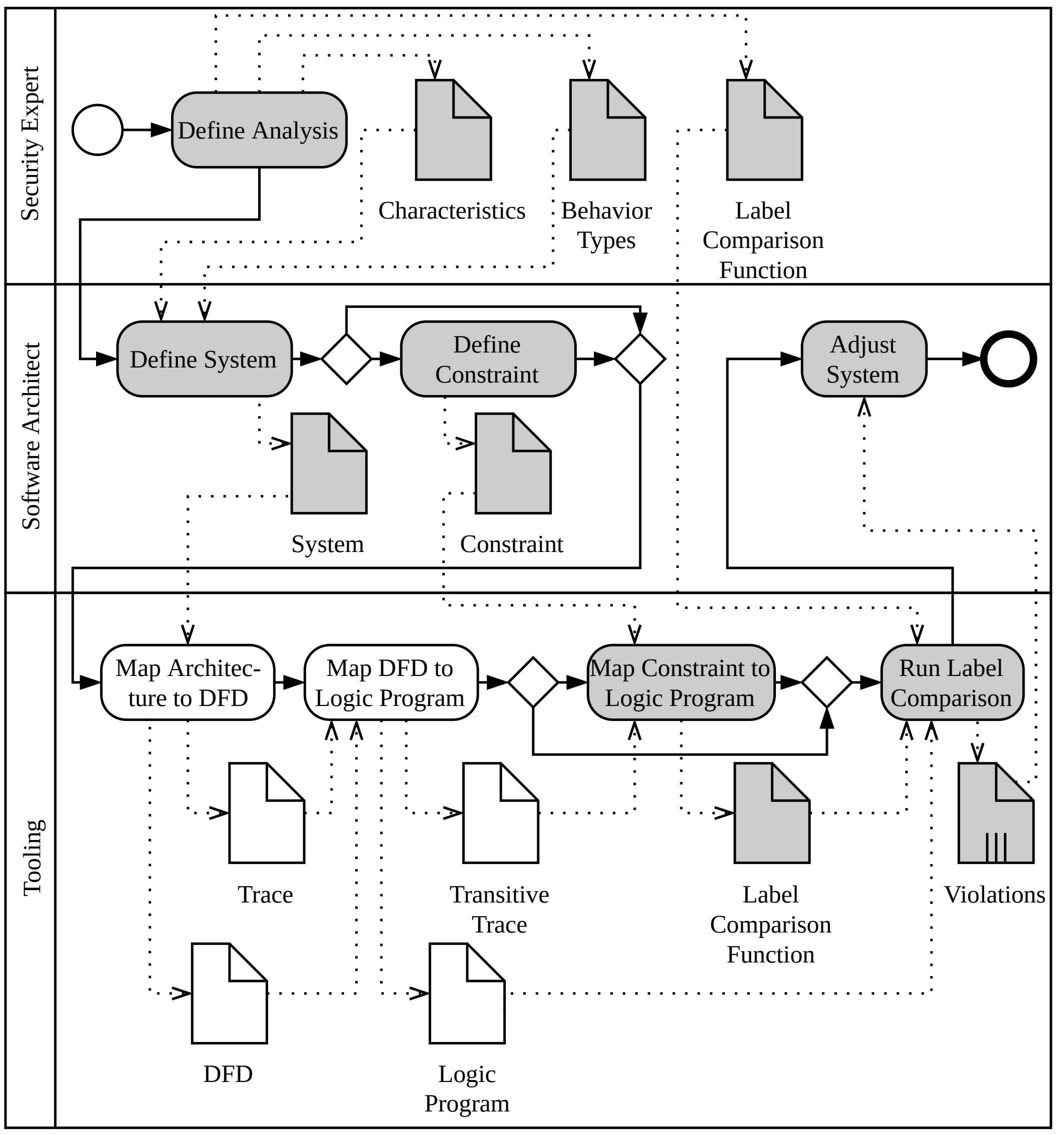


Figure 7.2.: Overview on analysis procedure for extended ADL given as BPMN diagram (gray elements are already part of the DFD analysis procedure).

I2) communication between processing steps

I3) users

I4) user activities

The ADL has to contain concepts to represent information relevant for expressing and analyzing confidentiality. The properties of nodes (I5), which are relevant for confidentiality, have to be part of the ADL. It is not important whether these properties are predefined, given by particular node types or encoded in textual annotations as long as the used modeling mechanism allows to represent discrete values. In the analysis definitions presented in Section 6.2, the properties of nodes were essential to detect violations in the label comparison. Besides properties of nodes, properties of data are the other input to label comparisons. The ADL has to provide concepts for representing initial properties of data (I6) as well as the effect of data processing on data properties (I7). Both concepts are essential for executing the label propagation because the propagation requires initial labels as well as propagation functions. Again, it does not matter whether the modeling mechanism uses predefined behaviors or flexible means for representing the behavior as long as the mechanism can describe the effect on data properties. Together, these three concepts provide the confidentiality-specific information for the DFD. To summarize, the following concepts are essential for representing the DFD structure:

I5) node properties

I6) initial data properties

I7) effect of data processing

In the following, we describe the individual activities of the tool engineer within the integration procedure in more detail. While executing the activities, he/she uses the previously described information about essential concepts to reason about necessary extensions.

Identify Missing Concepts In order to identify the missing concepts, the tool engineer looks for the previously mentioned essential concepts I*n* but he/she also captures other concepts, which can represent equivalent information or at least a part of the required information. In particular, the parts about communication, users, behavior and annotation mechanisms are of high interest. Often, concepts for representing the structure, behavior and user are available but the annotation of properties as well as the behavior description regarding such properties is missing. The tool engineer compares the identified concepts and the essential concepts I*n*. An ADL concept matches an essential concept if it provides at least the information required by the essential concept and the meanings of the concepts are not contradicting. For instance, if a concept describes a forbidden data flow, it provides information about data flows but does not have the same meaning as a data flow in a DFD. The concept can still be useful but further investigations on how to reuse the concept in an ADL extension are necessary. If an essential concept is missing, the tool engineer adds it to the list of missing concepts. If there are concepts in the ADL, which partially represent an essential concept, the essential concept is still missing but the identified partial match helps in building the ADL extension in the next step. The result of the activity is a list of missing essential concepts in the ADL.

Extend ADL The tool engineer has to extend the modeling language, i.e. the ADL, but also the corresponding architecture development process. To extend the modeling language, the tool engineer uses the list of missing concepts as well as the ADL concepts, which partially represent required information. The challenge in extending the ADL is to keep the introduced modeling mechanisms consistent to existing modeling mechanisms. A modeling mechanism is a way of representing information in an ADL. For instance, in order to model properties of nodes, the tool engineer can introduce strongly typed characteristics as used in the DFD or he/she can also add free text annotations to the elements. If the ADL already uses free text specifications in several places, it is reasonable to stick to this modeling mechanism to create a consistent modeling language. If the ADL does not provide means for specifying additional information at all, it is reasonable to choose the modeling mechanism, which matches the mechanisms used in the DFD. The main reason to reuse DFD mechanisms is to ease the mapping between the ADL and the DFD. Because the software architect is not aware of a particular mechanism for representing flexible annotations such as node properties yet, it does not matter to him/her, which modeling mechanism the tool engineer uses. The overall goal is to introduce as less new concepts and modeling mechanisms as possible but as much as necessary. To extend the modeling process, the tool engineer has to define the responsibilities of roles for creating model elements that represent the essential concepts and he/she has to specify when confidentiality analyses shall be conducted. If the modeling process for an ADL already specifies roles, the tool engineer tries to reuse the existing roles. Usually, there are roles for specifying the structure, behavior and interaction of users with the system. Often, it is necessary to introduce a security expert, who can specify the analysis definition as described in Section 6.1. If the ADL is already used for quality analyses, it is reasonable to conduct confidentiality analyses together with other quality analyses. If such an activity is not available, the tool engineer has to introduce it in the existing modeling process. The result of the activity is an ADL extension of the modeling language as well as of the corresponding modeling process.

Define Mapping to DFD The tool engineer uses the knowledge about the modeling concepts in the ADL and the DFD to define a mapping from the ADL to the DFD. The mapping of communication to data flows is not straight forward because not all ADLs natively support or use data flows. Instead, the tool engineer has to identify data flows from existing communication and has to map these implicitly specified data flows to data flows in the DFD. Because the analyses operate on DFDs, the tool engineer cannot always map all aspects of all ways of communication. We discuss these limitations in Section 7.4. The mapping of the remaining concepts is simpler because there is a high chance that the semantics of the concepts match well after the ADL has been extended. The result of the activity is a mapping description, which can be executed in a fully automated way.

Create Constraint DSL The tool engineer can reuse the constraint DSL as presented in Section 6.5 to a large extent. He/she has to replace the DFD concepts with ADL concepts. Especially, this affects the concrete syntax as well as the type and identity selector in the abstract syntax. The type selector refers to DFD node types but ADLs usually use other types of nodes. Therefore, the type selector has to refer to the new types. Because the information, which uniquely identifies nodes, is different for various node types, the identity selector also has to be changed. The concrete syntax uses DFD terminology such as the term `node`. The tool engineer has to replace these terms. In addition to the syntax, the mapping from the DSL to a logic program also has to be changed. Instead of resolving references to elements by looking up the trace from the DFD to the logic program, the mapping now has to consider the trace from the ADL to the DFD first. Thereto, the tool engineer creates a transitive trace that supports looking up identifiers in the logic program based on ADL elements. The result of the activity is an adjusted constraint DSL including an adjusted mapping procedure to a logic program.

7.2. Integrating DFD Analyses with Palladio

In this section, we apply the integration procedure resulting from the integration guidelines to the ADL Palladio [Reu+16]. We apply the integration procedure to illustrate the guidelines by an example in order to foster comprehensibility and also to demonstrate applicability. In addition, we demonstrate that the integration guidelines meet the requirements for the integration guidelines, which we defined in Section 4.1.3. In particular, we show that the resulting analysis framework for the ADL meets all requirements for the DFD-based analysis framework (R3.4), which we defined in Section 4.1.2.

We decided to use Palladio because it is an representative example for an ADL, it has been used before to predict quality properties of software architectures and it supports control flows as well as data flows. Palladio is representative because it shares fundamental concepts with many other ADLs such as components, defined interfaces and call-and-return communication. According to Ozkaya [Ozk18], Palladio considers all viewpoints found to be relevant for describing architectures, which means the ADL has good expressiveness. In addition, Palladio is used in practice according to an interview of practitioners [Mal+13]. Because Palladio is a representative ADL, the insights from applying the integration guidelines to it are also valid for other ADLs. Palladio focuses on essential concepts for predicting quality properties of software architectures. This is beneficial to reduce the overhead while applying the integration guidelines. In contrast, UML, which is used more often in practice, contains many different ways of expressing certain structures or behaviors. The tool engineer has to consider all different ways of expressing architectural aspects, which increases the effort for mapping the architecture to a DFD. From the perspective of a researcher, this additional effort does not pay off because it does not provide additional insights from applying the integration guidelines. It is realistic to assume that users of Palladio are interested in additional quality analyses because it already has been designed and extended to support various quality properties such as performance [BKR09], reliability [Bro+12] or maintainability [BSK15] in the past. Therefore, integrating confidentiality analyses in Palladio is a realistic application scenario. With the recent Palladio extension called *Indirections* [WSK20], Palladio supports communication via data flows in addition to control flows. An ADL supporting both communication paradigms is beneficial because this lowers the effort for demonstrating the integration guidelines for control flows (R3.1) and data flows (R3.2).

In the following, we describe the application of the integration guidelines to the subset of Palladio, which uses control flows in Section 7.2.1, and to the subset of Palladio, which uses data flows, in Section 7.2.2. For both applications, we use Palladio 5.1[1]. The last step of adjusting the DSL for formulating constraints does not depend on the communication paradigm, so we describe this step together for both previously mentioned integrations in Section 7.2.3.

7.2.1. Call and Return Communication

We demonstrate application of the integration guidelines for the subset of the Palladio ADL, which models systems based on control flows. We structure the description by the steps of the integration guidelines, which we described in Section 7.1: In Section 7.2.1.1, we identify the concepts, which are missing in the ADL. We bridge the identified gaps by an ADL extension, which we describe in Section 7.2.1.2. The mapping of architectures given in the extended ADL to an architecture given in an extended DFD is covered in Section 7.2.1.3.

7.2.1.1. Identify Missing Concepts

We identify concepts (I1–I7), which provide required information, based on publications on Palladio [Reu+11; Reu+16] as well as based on the Palladio metamodel [Pal21b]. We structure the discussion by the concepts.

Processing Steps (I1) describe the data processing excluding its effect on data. The effect is another required piece of information to be discussed later. In Palladio, data is exchanged via parameters in the control flow [Reu+16, pp. 263]. Parameters are sent when calling a service and received when a called service returns. Therefore, to identify processing steps, it is necessary to consider the ADL elements affecting the control flow as well as elements affecting the parameters. Various elements of the ADL work together to specify the control flow: Interfaces specify the services, which can be called [Reu+16, p. 45]. Components can provide interfaces, i.e. offer the services described in the interfaces, and can require interfaces, i.e. request services described in the interfaces [Reu+16, pp. 47]. Instances of components, so-called *Assembly Contexts* or short *Assemblies*, can be wired based on the provided and required interfaces [Reu+16, pp. 49]. The resulting network of assemblies builds the overall system, which provides interfaces in order to offer services to users [Reu+16, p. 50]. This means that starting from a user, the wired assemblies together with the parameters and return values of the called services define the control flow given by the structure of the software system. In addition, there is an abstract description of the service behavior for every provided service of a component, which is called *Service-Effect Specification (SEFF)* [Reu+16, pp. 53]. Within such a SEFF, a sequence of actions describes the behavior. The most important action affecting the control flow is a call action, which calls a service from a required interface [Reu+16, p. 102]. Apart from that, there are other

[1] `https://web.archive.org/web/20220129154253/https://sdqweb.ipd.kit.edu/wiki/PCM_5.1`

actions such as branching actions, which also affect the control flow [Reu+16, p. 100]. This means that the actions in a SEFF define the control flow within the component as well as the control flow between the corresponding component and potentially called components. To summarize, the ADL already provides model elements to describe data processing activities, which are components, assemblies, SEFFs and actions within SEFFs. Therefore, the information to represent I1 is available.

Communication between Processing Steps (I2) describes what data individual processing steps exchange. As already explained as part of the discussion of I1, the wiring of assemblies as well as the call actions in the SEFFs define the paths, over which data is exchanged. Call actions send data to the called service via parameters and receive data from a called service via the return value. The data to be exchanged is defined by the parameters as well as the return values. Because the communication is done via call-and-return, it is also clear, that all parameters have to be available when starting the communication, i.e. doing the call, and the return value has to be available when ending the communication, i.e. the call is returning. This covers all information required to describe the communication between processing steps. Therefore, the information to represent I2 is available.

Users (I3) describe the external actors, i.e. actors outside the system. In Palladio, there are dedicated usage models to describe external actors and their behavior. Within these models, there are usage scenarios, which describe the behavior of a group of actors. Because the model does not contain information about individual actors but only about the group of actors, this description represents a type of actor [Reu+16, pp. 56]. It is possible that there are multiple usage scenarios, which actually describe different behaviors of the same type of actor. However, the usage model does not provide means to group these usage scenarios. Therefore, every usage scenario can be seen as an individual type of actor. This covers all information to identify users and external actors. Therefore, the information to represent I3 is available.

User Activities (I4) describe the data processing done by external actors. As explained while discussing I3, the usage scenarios in the usage models describe the behavior of external actors. The usage scenarios consist of a sequence of actions [Reu+16, p. 56]. There are call actions as well as branch actions, which affect the control flow [Reu+16, pp. 103]. Call actions can call services provided by systems and the system delegates the call to the assembly, which provides the service. External actors use parameters to pass information to the system and use return values to receive information from the system. Branch actions introduce conditional executions and can also affect the control flow. The information provided by the usage scenario regarding the data processing activities of external actors is equivalent to the information provided by the SEFFs regarding the data processing activities of components. Therefore, the provided information is sufficient, i.e. all information to represent I4 is available.

Node Properties (I5) focus on node properties, which affect confidentiality, i.e. which are used to derive confidentiality properties of data or to identify violations of confidentiality requirements. Palladio provides two ways of specifying properties of system parts: deployment information and component parameters. Deployment information is specified by allocating an assembly on a node in the resource environment. Every assembly has to be transitively deployed on a node [Reu+16, p. 58]. Transitively means that an assembly can be nested and is, therefore, deployed on the same node as the nesting assembly. Component parameters introduce variables to components. A variable can hold multiple variable characterizations, which describes the properties of the variable and, therefore, also the properties of the component [Reu+16, p. 107]. The characterizations are limited to five types, which describe the value, byte size, number of elements, structure and type of the variable [Reu+11, p. 102]. The variable characterizations cannot represent all node properties, which affect confidentiality. For instance, describing the clearance of a node is not possible without changing the semantics of an existing characterization type, e.g. by encoding the clearance level in the characterization describing the structure of a variable. Therefore, concepts for describing properties of nodes, which focus on confidentiality, are still missing to fully represent I5.

Initial Data Properties (I6) define the properties of data when it is created. Creating data when only considering control flows in Palladio means that a parameter or return value is defined and it does not refer to other data in order to derive properties. In Palladio, parameters are created when defining a call action and return values are created when defining a so-called *SetVariableAction* [Reu+16, pp. 263]. Palladio specifies properties of parameters and return values in terms of variable characterizations, which we already discussed for I5. The characterizations are limited to five types of characteristics, which cannot cover all information, which is relevant to reason about confidentiality. Besides constants and logical connectors, the expressions to specify the values of the characterizations can also refer to the characterizations of other variables, i.e. return values of previous calls or parameters of the call to the provided service. Encoding the initial properties of data into the five predefined characterization types could be possible but this violates the intended semantics of the characterization types and does not guarantee type-safety anymore. Therefore, a type-safe way of defining arbitrary variable characterizations, which are not limited to a predefined set of five characterization types, is missing to fully represent I6.

Effects of Data Processing (I7) describe how data processing steps affect the properties of data. In Palladio, data, i.e. parameters and return values, have variable characterizations, which describe the properties of data. As already discussed for I6, these characterizations can be defined in call actions and actions for setting variables. To define the value of one of these characterizations, so-called *stochastic expressions* are used [Reu+16, p. 103]. Stochastic expressions define the values of characterizations and can refer to other characterizations to describe the propagation of data as well as the effect of data processing on the data properties. However, the restriction to the five predefined characterization types still applies. Therefore, a type-safe way of defining arbitrary variable characterizations, which are not limited to a predefined set of five characterization types, is still missing to fully represent I7.

7.2.1.2. Extend ADL

We have to extend the development process for creating and analyzing the software architecture and we have to extend the ADL to support the missing concepts.

The extension of the process for modeling and analyzing the software architecture is rather small: Palladio already defines the role of a *quality analyst* [Reu+16, p. 205], who supports the other roles involved in the development of the architecture, provides quality-specific information and conducts quality analyses. The responsibilities of the previously defined *security expert* for DFDs match this role. In our context, the quality analyst creates the analysis definitions and runs the analyses. The existing roles defined for Palladio (software architect, component developer, system deployer and domain expert) [Reu+16, pp. 12] bind the confidentiality primitives, i.e. characteristics and behaviors, to the Palladio elements. Please note that the existence of a dedicated quality analyst role does not imply that creating a software architecture and analyzing the quality properties have to be done by two dedicated persons. The person having the role of a software architect can also (partially) have the role of a quality analyst, which explicitly includes running analyses.

We identified three missing concepts, which we have to introduce in the Palladio ADL. The missing concepts are node properties (I5), initial data properties (I6) and effects of data processing (I7). In the following, we explain the extension of the Palladio ADL by these concepts. Because the syntax of Palladio is specified as a metamodel, we describe the extensions as metamodel extensions.

In DFDs, we used characteristics to describe properties of nodes (I5). First of all, it is necessary to define what a *node* means in the context of the Palladio ADL. From a structural point of view, the assemblies, i.e. the component instances, are the smallest unit of composition in the architecture. These assemblies are deployed on hardware resources, the so-called *ResourceContainers*. The wired network of assemblies builds the system. All of the mentioned model elements can be seen as nodes, which can have different properties. However, defining properties of the whole system actually means defining properties, which apply to all parts of the system. Such properties have the character of global properties, which can also be represented in the confidentiality requirements. Therefore, only considering assemblies and resources as owners of properties is sufficient to cover all non-global properties of nodes within the system. The only elements, which can be classified as nodes outside of the system, are users. Users can also have properties, so it is reasonable to also consider them in the extension for node properties.

Because Palladio does not provide flexible annotations of node properties or flexible definitions of properties and property types yet, we have to introduce a completely new concept. Therefore, we can reuse the DFD parts, which describe characteristic types and characteristics, without breaking with existing modeling conventions. To assign

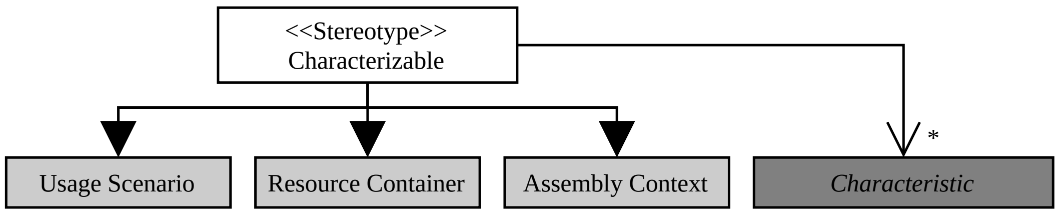


Figure 7.3.: Extension of Palladio metamodel to capture properties of nodes given as UML class diagram. Light gray elements are already part of Palladio, dark gray elements are reused elements of the DFD syntax and non-filled elements are newly introduced elements.

these characteristics, we define the stereotype as illustrated in Figure 7.3. A stereotype is an extension mechanism known from UML [Obj20, pp. 252], which allows to extend a UML model element by additional attributes or references. EMF Profiles [Lan+12] provide stereotypes for the Eclipse Modeling Framework (EMF) [Ste09], which Palladio uses as meta-language to define its metamodel. The stereotypes of EMF Profiles are one of the suggested extension mechanisms for Palladio [HSR21]. The meaning of the stereotype visualized in Figure 7.3 is as follows: The stereotype has a reference to multiple characteristics. The stereotype can be applied to the metamodel elements Usage Scenario, Resource Container and Assembly Context. When applying the stereotype, the metamodel elements are virtually extended by an additional reference to the characteristics. Therefore, instances of the metamodel elements can refer to characteristics, which describes their properties. The three metamodel elements represent the relevant nodes as discussed in the paragraph before. Therefore, the extension provides the missing concept of node properties (I5).

Palladio already provides means to describe initial data properties (I6) and effects of data processing (I7) but the types of properties are fixed and can, therefore, no cover the properties, which are required to express and analyze confidentiality. Nevertheless, we aim to reuse the modeling concepts and extend them. The existing modeling concepts are the definition of characteristics of sent parameters and received return values. The available characteristics in Palladio are not flexible or extensible. In our extension, we aim to provide modeling concepts to use the characteristic types and characteristics as defined for DFDs. This is reasonable because we already use these characteristics and characteristic types for describing node properties. Additionally, there does not exist any flexible way of defining characteristics in Palladio yet. Therefore, we do not break with existing modeling concepts for characteristics.

The Palladio extension to cover data properties and the effect of data properties is illustrated in Figure 7.4. The existing model element to capture characteristics of parameters and return values is the Variable Usage. It contains an Abstract Named Reference, which represents the name of the parameter or return value to be characterized. An actual characteristic is specified by a Variable Characterization, which is contained in the Variable Usage. In order to extend the usable characteristics, we introduce the new model element Confidentiality Variable Characterization, which is a subclass of a Variable Characterization. The benefit of this inheritance relation is that the new element can be used together

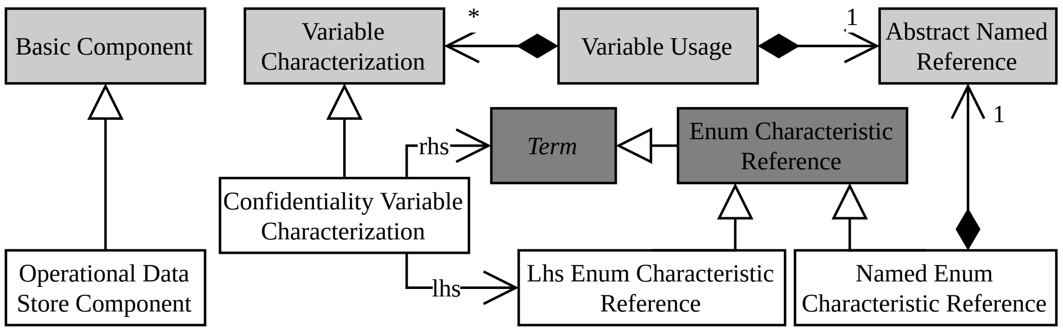

Basic Component
Variable Characterization
*
Variable Usage
1
Abstract Named Reference
1
rhs
Term
Enum Characteristic Reference
Confidentiality Variable Characterization
lhs
Operational Data Store Component
Lhs Enum Characteristic Reference
Named Enum Characteristic Reference

Figure 7.4.: Extension of Palladio metamodel to capture stores and properties of data given as UML class diagram. Light gray elements are already part of Palladio, dark gray elements are reused elements of the DFD syntax and non-filled elements are newly introduced elements.

with the old model elements. Therefore, existing software architectures do not become incompatible but only have to be extended by new model elements. In contrast to the old characterization, the new Confidentiality Variable Characterization defines characteristics based on characteristic references and terms as already known from the DFD extension described in Section 5.1.2. More precisely, such a characterization is equivalent to an Assignment in a DFD: a term on a right-hand side assigns a boolean value to a boolean variable represented by a characteristic reference on the left-hand side. If the boolean value is true, the label, i.e. the tuple of characteristic type and literal, is available on the variable specified in the corresponding variable usage. Most of the terms can be reused from the DFD syntax. Only the DataCharacteristicReference (see Figure 5.5 on page 42) has to be replaced because there are no pins in Palladio. Instead, we use a Named Enum Characteristic Reference, which uses a name instead of a pin to refer to characteristics of other parameters or return values. The Lhs Enum Characteristic Reference does not require a name because it refers to a characteristic of the variable, i.e. parameter or return value, which is specified by the Variable Usage. It must only be used on the left-hand side of a Confidentiality Variable Characterization. The extension fully provides the missing concepts for describing initial data properties (I6) and effects of data processing (I7). By reusing parts of the DFD syntax (definition of characteristic types, characteristics and terms), the extension is small and streamlined with the extension for node properties.

All missing essential concepts are covered by the previously described ADL extensions. However, the integration guidelines in Section 7.1 mention the non-essential but helpful concept of a data store. The concept is non-essential because it can be replaced by a node, which uses the behavior of a store. However, having a dedicated element to represent a store is potentially more comprehensible than replicating the behavior of a store. Therefore, we add an Operational Data Store Component, which inherits from Basic Component. Such a store has limited features compared to a regular component: The store must only provide one interface and must not require an interface. The interface must have exactly two operations. One operation takes a parameter of a certain data type but does not return anything. This operation represents an operation for adding data to the store. One operation takes no parameters but returns data of the same data type as used in the other operation. This operation represents an operation for reading data from the store. There does not have to be a behavior specification for these operations, i.e. SEFFs, because the behavior is always the behavior of a store. We elaborate on this when defining the mapping to a DFD in Section 7.2.1.3.

7.2.1.3. Define Mapping to DFD

The goal of mapping the architecture given in Palladio to a DFD is to make use of existing DFD-based analyses. Consequently, the mapping does not have to represent every aspect of the architecture but only the aspects required for analyzing confidentiality. The mapping has to yield a DFD, i.e. the structure given by nodes and data flows, the properties of nodes as well as the behavior given as label propagation functions. In the following, we structure the explanation of the mapping by these three parts of the DFD.

Prerequisite: Characteristic Types. In the following mappings, we always assume that the characteristic types are available. This implies no limitations or excluded manual work because we reuse the metamodel elements for describing characteristic types and characteristics from the DFD syntax. Therefore, a mapping of the characteristic types is not necessary.

Prerequisite: Unique Identifiers. In the descriptions of the mappings, we use intuitive, short identifiers for elements for a sake of comprehensibility. When realizing the mapping, such identifiers have to be unique to avoid ambiguities. Because most model elements in a software architecture given in Palladio already have unique identifiers, constructing unique identifiers to be used in DFDs is possible. However, we do not describe this aspect because this is rather a technical than a conceptual issue. Instead, we assume that identifiers are unique in the following descriptions.

Additional Characteristic Types. Besides the characteristic types specified by the security expert, we add two additional characteristic types, which we use to represent information from the software architecture given in Palladio in the DFD. Such information can be useful for formulating custom analyses that refer to architectural information modeled in Palladio. The first information is the containing architectural element. We represent this information in a characteristic type, which we call *Containing*. For instance, an action in a SEFF is contained in a component and an action in a usage model is contained in a scenario behavior. We distinguish the containing elements *Scenario Behavior* and *Component*. This information is useful for analyses that only look for violations in the behavior of the user, for instance. Such analyses can select nodes to consider by an applied label *Scenario Behavior* of the characteristic type *Containing*. We explain how we assign such labels to nodes when describing the structural mapping of elements in the following. The second information to represent by an additional characteristic type is call-related information. We refer to this characteristic type by the name *CallRole*. A call always consists of a sending and a receiving part. The called element receives the call and returns the call. The

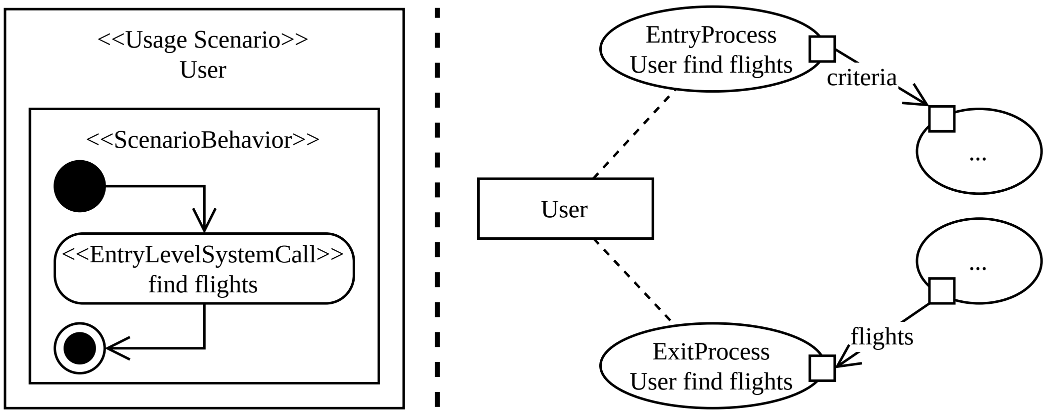


Figure 7.5.: Example of mapping usage scenarios and user actions from Palladio (left) to a DFD (right).

corresponding characteristic type provides means to describe the sending and receiving parts for the caller and the callee. An analysis, which only detects violations caused by communication between components, can make use of the resulting labels on nodes. We explain how we assign such labels to nodes when describing the structural mapping of elements in the following.

Structure: External Actors. In Palladio, software architects describe external actors in usage models. Usage models consist of multiple usage scenarios. A usage scenario describes a group of users, who interact with a system in the same way. We interpret such a group of users as a type of user. Therefore, we map each usage scenario to an external actor in the DFD. Figure 7.5 illustrates this mapping for an excerpt of the user behavior in our running example. The group of users described by the usage scenario performs all actions described in the corresponding scenario behavior. Therefore, it is reasonable to interpret all of these actions as actor processes, i.e. processes done by an actor. Figure 7.5 illustrates actor processes by a dashed line between the process and the actor, to whom the process belongs. We will explain the remainder of the illustration when explaining the mapping of call actions. Because the actor processes originate from the scenario behavior, we add a label of the *Containing* characteristic type with the value *Scenario Behavior* to all actor processes.

Behavior: External Actors. The external actors in the DFD do not own a behavior definition because the mapping only yields external actors without incoming or outgoing data flows. Instead, the behavior of actors is given by the actor processes, which we derive from the actions of the actors. This is not problematic because the actor processes are clearly related to the corresponding actors and analyses can look for violations on actor processes to identify violations of actors.

Structure: SEFF A SEFF describes the behavior of a service in a component. In a system, there can be multiple assemblies, i.e. component instances, which use the same SEFF.

A call always targets a SEFF of an assembly: A call from an assembly can only target a service of another assembly, i.e. a SEFF of an assembly. A call from a user targets a system service but the system directly delegates the call to an assembly, which provides this service. A SEFF consists of actions. Every action can make use of the parameters received via a call to the SEFF. After all actions have been executed, the SEFF returns the return value. It is important to represent this receiving of parameters and returning of values because these actions can violate confidentiality requirements. To represent the SEFF in a DFD, we map every SEFF to two processes: One process receives the parameters and provides the parameters to processes originating from actions within the SEFF. We refer to the process, which receives parameters, as *entry process*. To recognized this process in the DFD, we add a label of the *CallRole* characteristic type with the value *SEFFEntry* to the process. One process receives the result value and provides the value to the calling processes. We refer to the process, which provides the return value to callers, as *exit process*. To recognized this process in the DFD, we add a label of the *CallRole* characteristic type with the value *SEFFExit* to the process. Because the SEFF is always part of a component, we add a label of the *Containing* characteristic type with value *Component* to both processes. If the service described by the SEFF does not receive parameters, we omit the entry process. If the service described by the SEFF does not return a value, we omit the exit process. Figure 7.6 illustrates this mapping. The SEFF *findFlights* of the airline is mapped to an entry and an exit process. Because the service described by the SEFF receives a query as parameter, the entry process receives query data via an input pin. For every received data, i.e. input pin, there is one output pin, which provides the received data to other processes, which originate from actions within the SEFF. The service described by the SEFF of the airline returns a value. Consequently, the exit process provides result data via an output pin. The mapping rules for deriving the input pins of the exit process are discussed later. Because there can be multiple component instances, i.e. assemblies, of the same component, there can also be multiple instances of the same service. In addition, assemblies can be nested. The example illustrated in Figure 7.7 demonstrates the nesting of assemblies for the *CreditCardCenter* component. The component consists of an instance of the *CreditCardCenterLogic* component and an instance of the *CreditCardCenterDB* component. When an instance of the *CreditCardCenter* component is called, the call is delegated to the *CreditCardCenterLogic* assembly. To uniquely identify an instance of a component, a SEFF or any action within a SEFF, the complete hierarchy of assemblies has to be given. For instance, to uniquely identify the SEFF of the *declassify* service, we have to know the assembly of the *CreditCardCenter* component as well as the assembly of the *CreditCardCenterLogic* component. Therefore, we apply the mapping rules described above to every tuple of a SEFF and an assembly hierarchy. We map every tuple of assembly hierarchy and action within a SEFF according to the descriptions in the following paragraphs.

Behavior: SEFF. The behavior of the entry and exit processes of SEFFs is the forwarding behavior. An entry process of a SEFF owns one input pin and one output pin for each parameter of the described service. The labels of an input pin are directly copied to the corresponding output pin. The forwarding behavior is appropriate here because data is

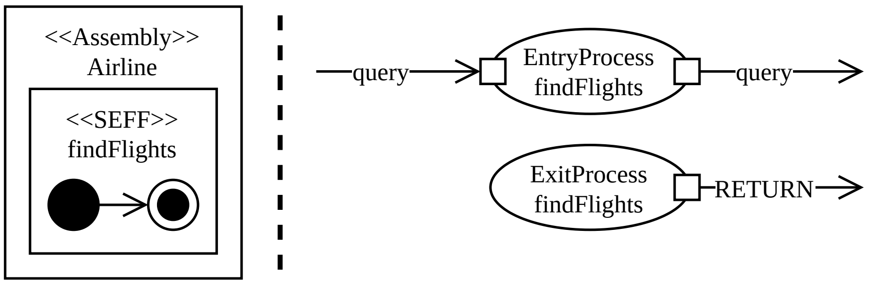


Figure 7.6.: Example of mapping a SEFF from Palladio (left) to entry and exit processes in a DFD (right).

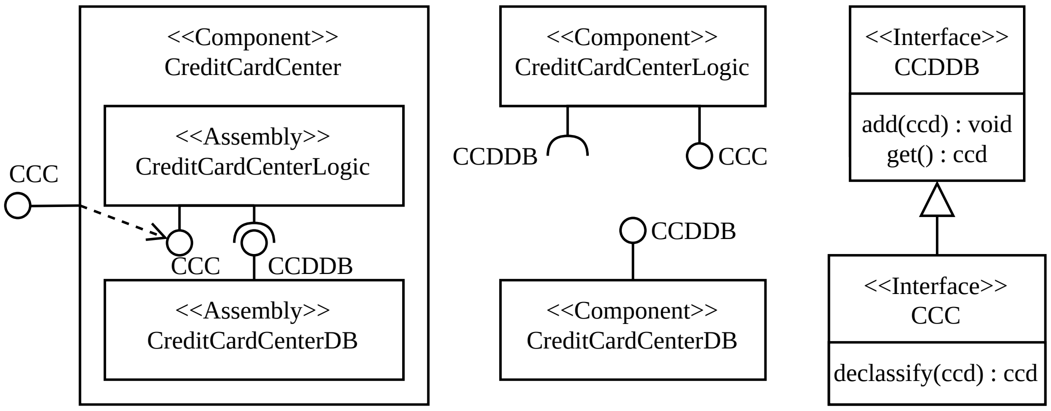


Figure 7.7.: Example illustrating nested assemblies in the CreditCardCenter component.

not changed by just sending it from a caller to a callee. An exit process of a SEFF owns one input pin and one output pin. The labels of the input pin are directly copied to the corresponding output pin. The forwarding behavior is appropriate here because data is not changed by just sending it to a caller.

Structure: Data Stores. We introduced OperationalDataStores to the ADL to mark a component as a data store. The store provides one interface containing a service for adding data and one service for receiving data. We map the corresponding SEFFs according to the mapping rules before, which yields one entry process for the service to add data and one exit process for the service to get data. The SEFFs of a data store are always empty because stores have a fixed behavior that must not be changed. Instead of allowing actions within the SEFFs, we create one Store in the DFD and add one data flow from the created entry process to the store and one data flow from the store to the created exit process. Figure 7.8 illustrates the mapping of the *FlightDB* data store from the running example according to the mapping rules given above. Because the data store component can be instantiated and nested multiple times, we execute the described mapping for every tuple of data store assembly and assembly hierarchy.

Behavior: Data Stores. The behavior of data stores is already given by the data store in the DFD. The behavior of the entry and exit processes, which result from mapping an

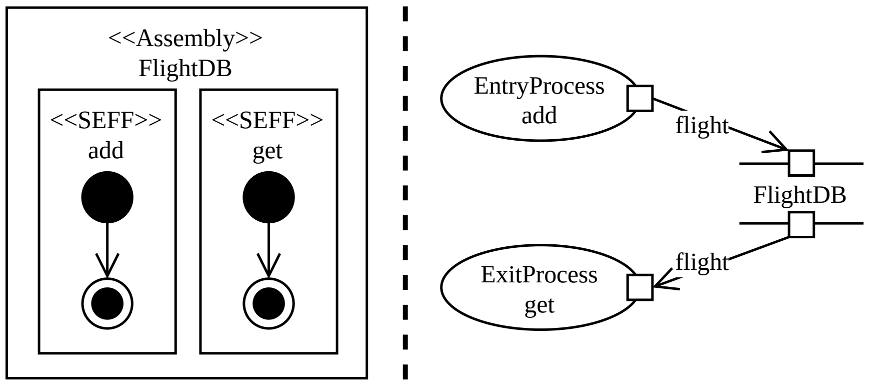


Figure 7.8.: Example illustrating the mapping of a data store in Palladio (left) to a store in a DFD (right).

Operational Data Store to a DFD, is the forwarding behavior as previously described for SEFFs.

Structure: Set Variable Actions. The SetVariableAction is an action used in a SEFF to specify the result value. The result is returned when the sequence of actions in a SEFF ends. Because the action consumes data and yields data without further communication with other actions in-between, the behavior of the action closely matches the semantics of a DFD process. Therefore, we can map the action to a single process. We add a label of the *Containing* characteristic type with value *Component* to indicate that the process is contained in a component. Because a SetVariableAction can only occur in a SEFF, we have to consider the nesting of assemblies. Therefore, we create one process for every tuple of action and assembly hierarchy.

Behavior: Set Variable Actions. The behavior of the process resulting from mapping the Set Variable Action depends on the Confidentiality Variable Characterizations for the *RETURN* variable. Such characterizations are the counterpart of assignments in DFDs and specify the labels of the output based on labels of the inputs. The behavior of the process derived from the Set Variable Action has one output pin as well as one input pin for every variable, i.e. data from parameters or other actions, used within the variable characterizations. The assignments of the behavior are created based on the Confidentiality Variable Characterizations. Because we use the same terms as for specifying the behavior in DFDs, the mapping is straight forward: A term of a certain type in Palladio is mapped to a term of the same type in the DFD. The only difference is the mapping of Named Enum Characteristic References. The Named Enum Characteristic Reference is mapped to a Characteristic Reference using the same characteristic type and literals. Instead of the variable name, the corresponding pin is used. In the example shown in Figure 7.9, the pins have the same name as the variables in Palladio. Therefore, the concrete syntax of the variable characterization looks the same as the concrete syntax of the assignment in the behavior. However, the names in the assignments of the DFD refer to pins instead of

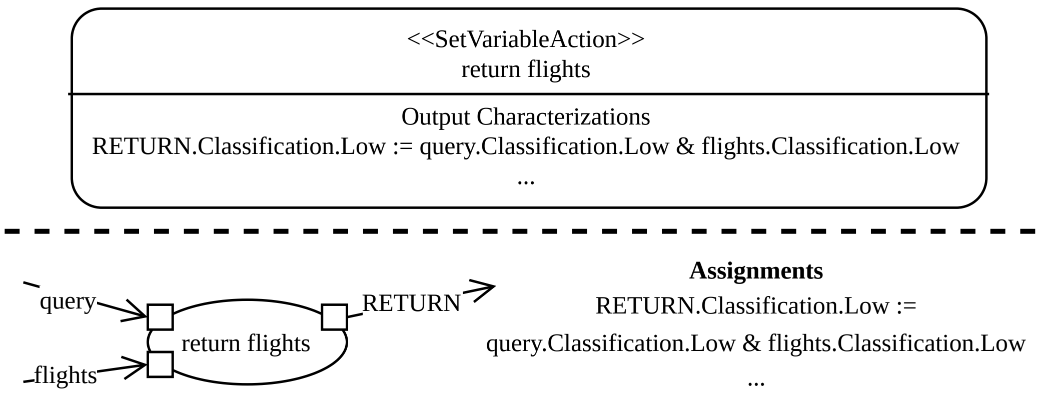

<<SetVariableAction>>
return flights
Output Characterizations
RETURN.Classification.Low := query.Classification.Low & flights.Classification.Low
...
query
flights
return flights
RETURN
Assignments
RETURN.Classification.Low :=
query.Classification.Low & flights.Classification.Low
...

Figure 7.9.: Example of mapping the behavior of a SetVariableAction (top) to the behavior of a process (bottom).

variables. The action in Palladio uses the *query* and *flights* variables. Therefore, the DFD process has two input pins.

Structure: Call Actions. Calls are a concept of the control flow paradigm. In Palladio, calls can occur in a scenario behavior or in a SEFF. The calls in the scenario behavior originate from users and target services of the system, which delegates calls to the responsible assemblies. The calls in SEFFs originate from assemblies and target services of other assemblies. There is no counterpart of calls in DFDs. However, parameters and return values exchanged via calls can be seen as data to be exchanged. Therefore, a call can be seen as a pair of processes: one process sends data (the parameters) to another process and one process receives data (the return value) from another process. Consequently, we map every call action to two processes as illustrated in Figure 7.5. We refer to the process sending data as *entry process* and to the process receiving data as *exit process*. We add a label of the *CallRole* characteristic type with value *CallSending* to the entry process and label with value *CallReceiving* to the exit process in order to indicate the role of the processes within the call-based communication. We add a label of the *Containing* characteristic type with value *Component* to indicate that the processes are contained in a component. If the call does not send parameters, we omit the entry process. If the call does not receive a return value, we omit the exit process. If a call action is placed within a SEFF, we have to consider the nesting of assemblies, so we have to create the entry and exit processes for every tuple of action and assembly hierarchy. We discuss data flows derived from calls later.

Behavior: Call Actions. We define behaviors for the entry as well as the exit processes. For entry processes, the variable characterizations of input parameters are important to consider. There is one variable usage for every input parameter of a call and each of these variable usages can contain multiple variable characterizations. The entry process has one input pin for every variable name, which is used in the variable characterizations of the inputs of the call. There is one output pin for every parameter of the called service.

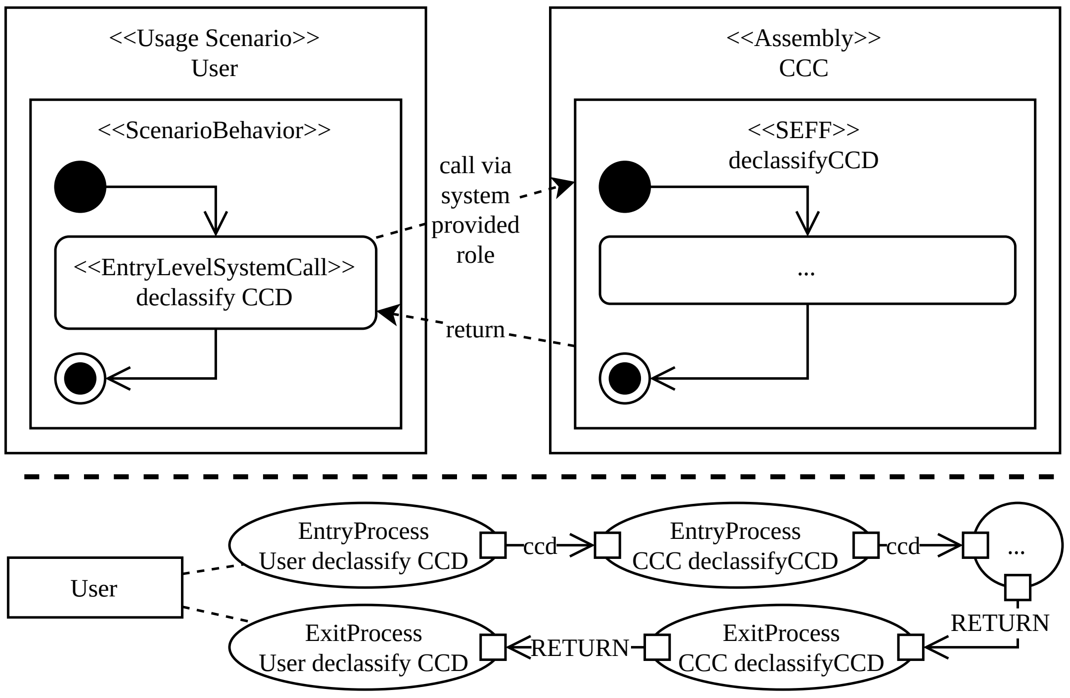


Figure 7.10.: Example of mapping a call action (top) to DFD processes (bottom).

The assignments of the behavior are created based on the variable characterizations as described for the SetVariableAction. For exit processes, the variable characterizations of so-called output variables are important to consider. A call action can define arbitrary output variables by multiple variable usages. The variable characterizations within the variable usages can refer to the return value of the called service but also to other variables or parameters within the SEFF. The exit process has one input pin for the result of the called service, potentially multiple input pins for other variables used within the variable characterizations and one output pin for every defined variable, i.e. VariableUsage. The assignments of the behavior are created based on the variable characterizations as described for the SetVariableAction.

Structure: Data Flows between Services. We map calls to data flows if a parameter or return value is exchanged via a call. When calling a SEFF of an assembly, there is one data flow from the entry process of the call to the entry process of the SEFF for every parameter of the service described by the SEFF. Figure 7.10 illustrates the mapping of calls to data flows for the declassification of credit card data in our running example. If the service provides a return value, there is one data flow from the exit process of the SEFF to the exit process of the call action. To identify the destination of a call, we can follow the connections between required and provided interfaces of the assemblies. The destination of a call is uniquely identified by the called SEFF and the assembly hierarchy.

Structure: Data Flows between Actions. Data flows between actions in usage scenarios or SEFFs occur by reading parameters, which have been received via a call to the SEFF, or by reading so-called *Variables*, which other actions define in Variable Usages. Therefore, we map every reading of these variables to data flows between actions. The example shown in Figure 7.11 illustrates the mapping for the service of finding flights at the airline in our running example. Reading parameters or variables is only possible within a Variable Characterization, i.e. the Terms, which we introduced as an extension of Palladio. To identify, which variables a process reads, we look for Named Enum Characteristic References and extract the variable name. The resulting set of variable names defines the required data. We add one data flow from the process, which defines the variable, to the process, which uses the variable in a characterization. The source of parameters, which have been received from a call to the SEFF, is the entry process of the SEFF. The source of result values of calls is the exit process of a call action. In Figure 7.11, the *return flights* action refers to the query parameter as well as the flights variable. The use of the parameter is mapped to a data flow from the SEFF entry process to the process representing the *return flights* action. The use of the variable is mapped to a data flow from the exit process of the call action. In this and all following examples, we use the variable name *RETURN* to refer to the result of a call. This is also the suggested convention in Palladio[2]. The *return flights* action defines the variable *RETURN*, which represents the result of the call to the SEFF *findFlights*. In order to make the result, which the *return flights* action defined, available to callers, we add a data flow from the *return flights* action to the exit process of the SEFF. The exit process makes the result available to the calling process. The call action *call DB* also uses a *RETURN* variable but this variable refers to the result of the call to be done by the action and not to the result of the currently executed SEFF. In consequence, the exit process of the call action receives a data flow from the exit process of the called SEFF. When mapping actions in SEFFs, we have to consider nested assemblies. Therefore, we execute these mappings for all tuples of actions and assembly hierarchies.

Node Properties. We extended Palladio by means to assign characteristics to elements in the software architecture. The assigned characteristics represent the properties of nodes, which result from the structural mapping explained before. Mapping the characteristics themselves is straight forward because we use the same metamodel elements as the DFD syntax to represent them. Therefore, mapping characteristics as well as characteristic types is a simple one-to-one mapping. However, there are two aspects to be defined: determining the effective characteristics and determining the covered nodes. Determining effective characteristics is necessary because we can assign characteristics to assemblies as well as to resources, which are organized in a hierarchy. The hierarchy of assemblies is given by their nesting. Resources are always the outermost elements because assemblies are allocated on resources. If there are multiple characteristics using the same characteristic type, a precedence rule is necessary to decide for an effective characteristic. For instance, if a resource is cleared for low data but an assembly, which is allocated on the resource,

[2] `https://web.archive.org/web/20220119133503/https://www.palladio-simulator.com/fileadmin/user_upload/palladio-simulator/videos-screencasts/pcm_8_returnvalue.mp4`

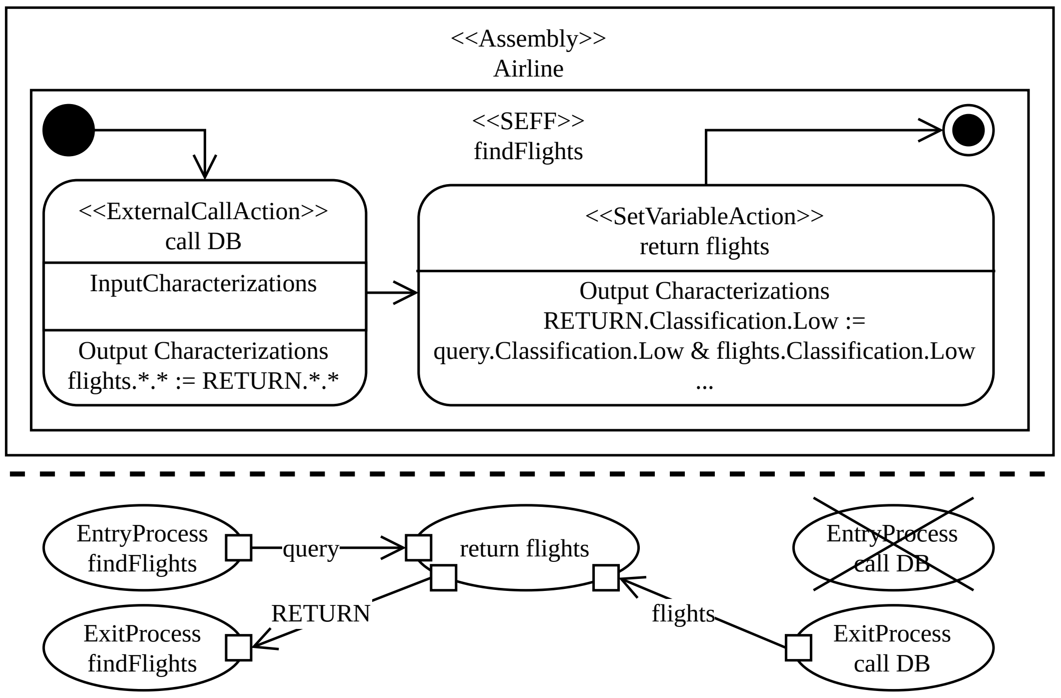


Figure 7.11.: Example of mapping actions (top) to data flows (bottom).

is cleared for high data, the clearance levels clash. To solve this conflict, we define the precedence of characteristics for the same characteristic type in a way that always uses the characteristic of the model element, which is nested the most. In the given example, the clearance of the assembly would override the clearance of the resource because the assembly is more nested than the resource. The precedence rules allow to define general applicable characteristics but also allow to define exceptions. Determining the DFD elements, which are covered by a characteristic, is simple: every DFD element, which has been mapped from a Palladio element, which is contained in the element having a characteristic, uses the characteristics of the containing Palladio element. For the scenario behavior, this means that the actor derived from the scenario behavior as well as all actor processes derived from the call actions within the scenario behavior use the characteristics of the scenario behavior. For SEFFs and actions within a SEFF, this means that processes and stores derived from them use the effective characteristics, which are determined based on the corresponding assembly hierarchy and the resource container.

Additional Prolog Clauses. The previously described mapping rules map a software architecture given in the Palladio ADL to a DFD. Later, the DFD is mapped to a logic program. To ease using the node properties of the *CallRole* and *Containing* characteristic types, we add the additional Prolog clauses shown in Listing 7.1 to the result of the mapping to a logic program. The clauses are shorthands for writing a `nodeCharacteristic/3` clause and not mandatory for analyses: Omitting these additional clauses would not lower the expressiveness because it is always possible to replace the additional clauses by the corresponding `nodeCharacteristic/3` clauses.

Listing 7.1: Additional Prolog clauses to simplify accessing additional node properties.

```
isACallSending(N) :- nodeCharacteristic(N, 'CallRole', 'CallSending').
isACallReceiving(N) :- nodeCharacteristic(N, 'CallRole', 'CallReceiving').
isASEFFEntry(N) :- nodeCharacteristic(N, 'CallRole', 'SEFFEntry').
isASEFFExit(N) :- nodeCharacteristic(N, 'CallRole', 'SEFFExit').
containedInScenarioBehaviour(N) :- nodeCharacteristic(N, 'Containing', 'Scenario
    Behavior').
containedInComponent(N) :- nodeCharacteristic(N, 'Containing', 'Component').
```

7.2.2. Data-oriented Communication

We demonstrate the integration procedure for the Palladio ADL, which has been extended by data flows as part of the *Indirections* project [WSK20]. We structure the description by the steps of the integration procedure, which we described in Section 7.1: In Section 7.2.2.1, we identify the concepts, which are missing in the ADL. We bridge the identified gaps by an ADL extension, which we describe in Section 7.2.2.2. The mapping of architectures given in the extended ADL to an architecture given in an extended DFD is covered in Section 7.2.2.3.

7.2.2.1. Identify Missing Concepts

We identify concepts (I1–I7), which provide required information, based on publications on Palladio [Reu+11; Reu+16], a publication on *Indirections* [WSK20] as well as based on the Palladio metamodel [Pal21b] and the metamodel of *Indirections* [Pal21a]. We structure the discussion by the concepts.

Users (I3) describe the external actors, i.e. actors outside the system. The *Indirections* extension does not extend the usage model, which means it does not introduce data flows for external users and their behavior. Therefore, the usage scenarios, which we already discussed in Section 7.2.1.1, are still the elements, which represent external users of systems in Palladio. Therefore, the information to represent I3 is available.

User Activities (I4) describe the data processing of external actors. The *Indirections* extension does not extend the usage model, which means it does not introduce data flows for external users and their behavior. Therefore, the call actions in the scenario behaviors, which we already discussed in Section 7.2.1.1, are still the elements, which represent the behavior of users, i.e. their data processing activities. Therefore, the information to represent I4 is available.

Processing Steps (I1) describe the data processing excluding its effect on data. The effect is another required piece of information to be discussed later. The *Indirections* extension introduces data channels to describe data processing within systems. A data channel consumes data, processes data and yields data. If a data channel can consume a certain data type, it provides a sink for this data type. If a data channel can yield a certain data type, it provides a source for this data type. Data channels are special types of components and have to be instantiated to be used within the system. An instantiated data channel is also called *assembly*. Connectors wire assemblies from sources to sinks. The resulting network of data channel assemblies describes the structure of data processing steps, i.e. their existence and their relations. Because the *Indirections* extension does not introduce data flows for users, users still use system services via calls. Therefore, there still have to be components, which provide services. SEFFs still describe these services but a component can also communicate with a data channel via sinks and sources for data. Components can interact with data channels through these sources and sinks after they have been instantiated and a connector between them has been created. SEFFs contain additional actions to explicitly create data from parameters or variables (CreateDateAction), to send data through sources to data channels (EmitDataAction) and to consume data through sinks from data channels (ConsumeDataAction). Because these actions are contained in a component and a component can be instantiated, there are also multiple instances of these actions. Each of these action instances describes a data processing by either creating or transmitting data. To summarize, the instances of data channels, SEFFs and actions within SEFFs provide the processing steps. Therefore, the information to represent (I1) is available.

Communication between Processing Steps (I2) describes what data individual processing steps exchange. As explained before, the instances of data channels and components are connected, i.e. sources are connected to sinks and required services are connected to provided services. These connections between the assemblies specify the data, which can be exchanged between the assemblies. For sources and sinks, the exchanged data type is made explicit. For required and provided services, the exchanged data is given by the sent parameters as well as by the return value. Within assemblies, i.e. between actions of a SEFF, the exchanged data is given by the parameters and return values, which are used to create a data item or which are sent or received. Because a SEFF still describes a control flow, it is clear that used data has to be available before executing an action. To summarize, the connections between assemblies as well as the used data of actions provide all information to describe the communication between processing steps. Therefore, the information to represent I2 is available.

Node Properties (I5) focus on node properties, which affect confidentiality, i.e. which are used to derive confidentiality properties of data or to identify violations of confidentiality requirements. The *Indirections* extension does not extend the annotation mechanisms for components, SEFFs, resources or usage scenarios. Therefore, the limitations of representing node properties already discussed in Section 7.2.1.1 still apply. This means that Palladio still cannot express properties except for deployment information. The newly introduced data channels provide means to add configurations to data channels. A configuration entry can be any string. However, using these configuration entries for representing node properties would differ from the intended semantics of the entries, which is passing parameters to the behavior of the data channel. Therefore, we do not consider the configurations as means to express node properties. To conclude, Palladio and the *Indirections* extension do not provide sufficient means to describe node properties. Therefore, the information to fully represent I5 is not available yet.

Initial Data Properties (I6) define the properties of data when it is created. Data in Palladio are either variables within SEFFs, parameters or return values. The *Indirections* extension additionally introduces data, which is exchanged between sources and sinks. The properties of data in Palladio are specified using characterizations, which are specified within the action, which creates or transmits data. Data exchanged via sources and sinks either originates from an action or from a data channel. The data properties of data going through actions are specified by the Palladio characterizations. The data properties of data going through data channels are specified by the behavior of the data channel but the behavior also uses the Palladio characterizations. These characterizations are limited to five predefined types of characterizations. As already discussed in Section 7.2.1.1, this limitation is too strict and prohibits expressing all data characteristics, which are necessary to analyze confidentiality. Therefore, the information to fully represent I6 is not available yet.

Effects of Data Processing (I7) describe how data processing steps affect the properties of data. The processing effects implied by user behaviors, SEFFs and actions are already covered in Section 7.2.1.1. The expressible effects are limited by the five predefined types of characteristics, which are not sufficient for analyzing confidentiality. The *Indirections* extension does not prescribe how the behavior of a data channel is described but provides an abstract metamodel element, which tool engineers have to implement. The extension provides one implemented metamodel element for describing the behavior, which uses Java code as description language. Obviously, Java code is expressive enough to describe all processing effects on data properties but the Java API of *Indirections* is focused on performance simulations and extending the code would require introducing new concepts. Therefore, a concept for describing the processing effect of data channels in terms of data properties is still missing. To summarize, the information to fully represent I7 is not available yet.

7.2.2.2. Extend ADL

We have to extend the development process for creating and analyzing the software architecture and we have to extend the ADL to support the missing concepts. The *Indirections* extension does not extend the development process of Palladio but uses the process as it is. This means that the extension of the development process for Palladio using control flows, which we already described in Section 7.2.1.2, is also applicable to Palladio using *Indirections*. Therefore, we only describe the required extensions of the ADL in this section.

We identified three missing concepts, which we have to introduce in the Palladio ADL including *Indirections*. The missing concepts are node properties (I5), initial data properties (I6) and effects of data processing (I7). In the following, we explain the extension of the Palladio ADL by these concepts. Because the syntax of Palladio is specified as a metamodel, we describe the extensions as metamodel extensions.

To describe node properties (I5), we can reuse the extension, which we have defined for the Palladio subset using control flows. The extension illustrated in Figure 7.3 on page 124 introduces a stereotype, which is applicable to Usage Scenarios, Resource Containers and Assembly Contexts. Because instantiated data channels are also represented by Assembly Contexts, the extension already considers the newly introduced data channels. Apart from the data channels, there are no new metamodel elements, which need dedicated assignments of node properties. Therefore, the already defined extension is sufficient to represent node properties (I5).

Palladio uses characterizations to describe properties of data (I6). To circumvent the limitation to five predefined characterization types, we introduced Confidentiality Variable Characterizations for the subset of Palladio, which uses control flows. Figure 7.4 on page 125 illustrates the extension, which allows to use the characteristic types and the expressions already known from the DFD syntax. Palladio including *Indirections* also uses the characterizations to specify properties of data in the newly introduced actions. Therefore, the extension is also applicable here. The extension also already covers the effects of data processing (I7) for all existing Palladio elements as well as all elements of *Indirections* except for data channels. We will address data channels in the next paragraph. The extension is sufficient to represent the properties of data (I6) and the effects of data processing (I7), for all Palladio elements except for data channels.

To represent the effects of data processing (I7) as well as data properties assigned by data channels (I6), we have to specify the behavior of data channels. As already explained in Section 7.2.2.1, the *Indirections* only provide means to specify the effects on performance for data channels by Java code or require a tool engineer to create alternative means. Because we do not want to introduce alternative means, which are incompatible to the Java-based solution, we decided to introduce the stereotype illustrated in Figure 7.12. The stereotype can be applied to any data channel implementation and is, therefore, compatible to the Java-based solution as well as to solutions, which might be introduced in the future. The stereotype Confidentiality Behavior links a Data Channel Behavior to a Data Channel. The Data Channel already contains sources and sinks, which have names. The Data Channel Behavior describes the effect of data processing by defining one Variable Usage for every data source provided by the data channel. A variable usage refers to the name of the data source and specifies Confidentiality Variable Characterizations, which we introduced in a previous extension illustrated in Figure 7.4 on page 125. These characterizations define the properties of data (I6). The expressions in the characterizations can refer to

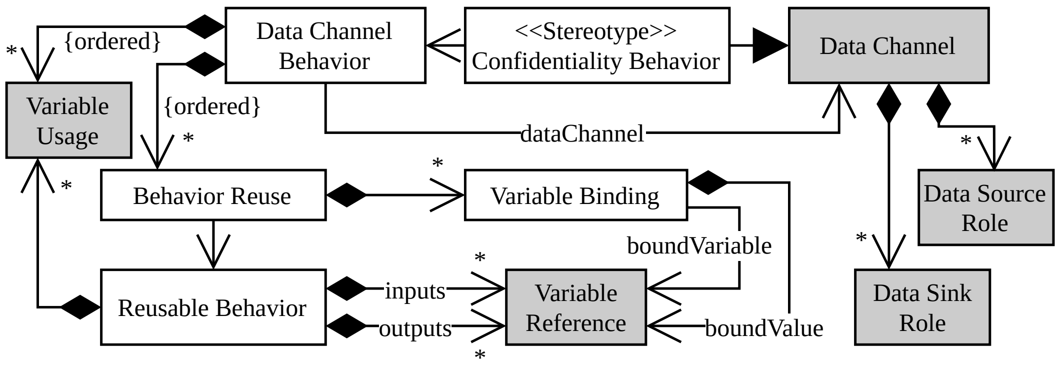


Figure 7.12.: Extension of Palladio metamodel to capture the data processing effect of data channels given as UML class diagram. Gray elements are already part of Palladio or the *Indirections* extension and non-filled elements are newly introduced elements.

the characteristics of incoming data by the name of the corresponding data sink of the data channel. This provides the means to specify the propagation of characteristics from the inputs to the outputs, which is also the effect of data processing (I7). Making use of the already introduced extension lowers the amount of required changes in the ADL and provides streamlined modeling of data properties and processing effects. Streamlined means that the same model elements can be used for the same purpose, i.e. describing the effect of data processing.

The previously explained subset of the extension shown in Figure 7.12 is already sufficient to represent all required information. However, modeling the same processing effects multiple times is cumbersome, prone to errors and increases maintenance efforts in case of required changes. A mechanism to specify types of processing effects (Reusable Behavior) and reusing them (Behavior Reuse) to describe behaviors of data channels addresses these problems. A Reusable Behavior specifies the effect of data processing by Variable Usages just like a Data Channel Behavior does. However, the Reusable Behavior shall not refer to particular sinks in the expressions of the characterizations of outputs because it shall be independent of a particular usage context. Instead, a Reusable Behavior introduces variables for inputs and outputs, which the expressions can use. A Behavior Reuse defines Variable Bindings, which bind a value, i.e. a name of a particular sink, to a variable defined in the Reusable Behavior. By binding the variables with values from the context, the behavior can be used in a particular context. It is possible to reuse multiple behaviors, i.e. define multiple Behavior Reuse elements for the same Data Channel Behavior. In this case, the processing effects are applied in the order of definition of the reuse elements. Processing effects executed later can override processing effects executed earlier, which are the same semantics as for assignments in DFDs or characterizations in Variable Usages. Variable Usages directly assigned to a Data Channel Behavior are always executed last, i.e. they can override the effects of all reused behaviors. This follows the commonly used practice that specific elements, i.e. the particular Data Channel Behavior, can override properties or definitions of generic elements, i.e. the reused behaviors.

The presented reuse approach for behaviors is more flexible than reusing whole data channels by instantiating them: The reuse of behaviors supports defining behaviors on the same level as we did in Section 6.2, e.g. we can define a forwarding behavior, a joining behavior and so on. The behaviors are applicable to any data channel, where the number of output variables matches the data sources and the number of input variables matches the data sinks. Reusing the whole data channel behavior, i.e. the Data Channel Behavior instead of Reusable Behavior, is only possible if the names of data sources and data sinks match the names used in the variable characterizations. Therefore, such a reuse approach would restrict the naming of data sources and data sinks, which can impact the comprehensibility negatively. Reusing a whole data channel is even more inflexible: It is possible to define data channels, which are essentially forwarding data channels or joining data channels. By instantiating them multiple times and connecting them to other data channels, it is possible to reuse the data channels. There are two problems with this approach: First, the names of data sources and data sinks will be, most probably, generic, which can affect comprehensibility negatively. Second, the data sources and data sinks refer to particular data types, which restricts the reuse of a data channel to contexts, in which the exact same data types are received and yielded. To circumvent this problem, either multiple versions of the same behavior have to be created for multiple combinations of data types or only generic data types such as *Object* have to be used. The first approach requires considerable effort and the second approach reduces comprehensibility. In contrast, the explicit definition of reusable behaviors and reusing them with the mechanism illustrated in Figure 7.12 does not imply such disadvantages.

7.2.2.3. Define Mapping to DFD

The goal of mapping the architecture given in Palladio to a DFD is to make use of existing DFD-based analyses. Consequently, the mapping does not have to represent every aspect of the architecture but only the aspects required for analyzing confidentiality. The mapping has to yield a DFD, i.e. the structure given by nodes and data flows, the properties of nodes as well as the behavior given as label propagation functions. Because Palladio using *Indirections* still requires control flow aspects such as the usage scenarios or SEFFs, we build the mapping to the DFD on top of the already described mapping for control flows in Section 7.2.1.3. This means that all mapping rules described for control flows also apply for a Palladio architecture using the data channels of *Indirections*. In the following, we only describe the additions required for properly handling data sinks, data sources and data channels.

Additional Characteristic Types. The mapping rules for communication via control flows introduced additional characteristic types. We can reuse all of these types but add the value *Data Channel* to the *Containing* characteristic type. This allows to identify process, which represent data channels.

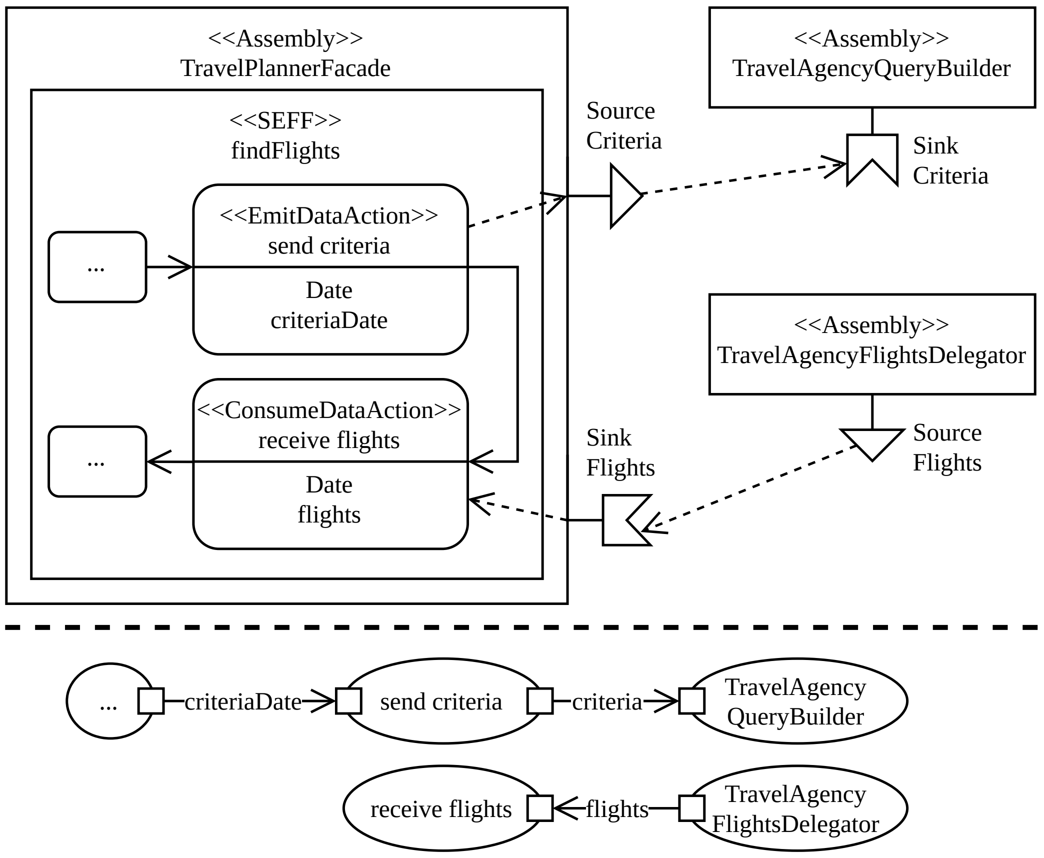


Figure 7.13.: Example of mapping data flows through sources and sinks (top) to data flows in a DFD (bottom).

Structure: Data Channels Data channels have dedicated data inputs (data sinks) and dedicated data outputs (data sources). The data processing within data channels uses data inputs and yields data outputs. This behavior closely matches the behavior of processes in DFDs, so we can map data channels to processes. We apply a label of the *Containing* characteristic type with value *Data Channel* to the processes to indicate that the processes originate from data channels. In addition, we map every data source of a data channel to an output pin and every data sink to an input pin. This mapping is useful because the semantics of a data source or data sink matches the semantics of a pin: all elements specify that a certain type of data shall be exchanged. Because data channels can be instantiated multiple times, we map every assembly of a data channel to a process. In the example shown in Figure 7.13, we map each data channel assembly (*TravelAgencyQueryBuilder* and *TravelAgencyFlightsDelegator*) to one dedicated process having the corresponding names.

Structure: Actions The newly introduced actions CreateDateAction, EmitDataAction and ConsumeDataAction consume data and yield data without communication with other actions in-between. Therefore, these actions match the semantics of DFD processes. We add a label of the characteristic type *Containing* with the value *Component* to the processes to indicate that the processes originate from model elements contained in a component.

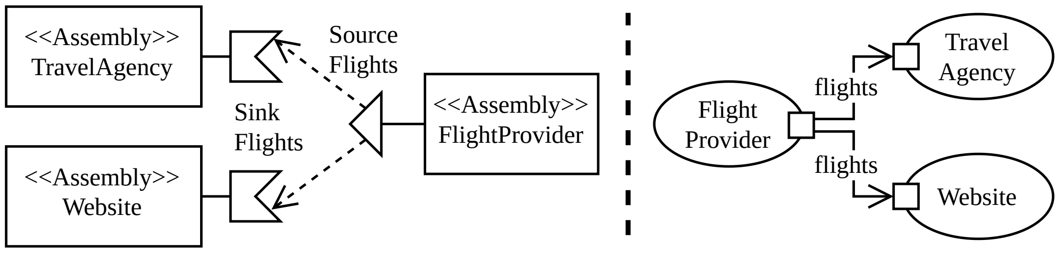


Figure 7.14.: Example of mapping data flows between assemblies of data channels (left) to data flows in a DFD (right).

Because these actions are part of SEFFs, we have to consider the assembly hierarchy in the mapping. Therefore, we map each tuple of action and assembly hierarchy to one process. The mapping of data flows between these and other actions within a SEFF already described in Section 7.2.1.3 still applies. This means, we create one input pin for each variable, which the CreateDateAction uses to specify the characterization of the created date. We create one input pin for the ConsumeDataAction and EmitDataAction action because both receive exactly one data item. We create one output pin for the all three actions because all yield exactly one data item.

Structure: Data Flows between Sources/Sinks of Data Channels Data flows as defined by *Indirections* always use data sources and data sinks. A data source or sink is always defined for a single data type. A connector between the data source of an assembly and the data sink of another assembly enables data flows between the assemblies. If both involved assemblies are data channels, a connector represents one data flow because the data channel sending data through a data source is the only provider of data for this data source and the data channel receiving data through a data sink is the only consumer of data for this data sink. We illustrate this situation in Figure 7.14. All assemblies in the illustration are data channels. We map each connector, which is illustrated as dashed edge, to a data flow between the processes of the two involved data channels. This mapping closely matches the semantics of such a connector by representing the direct data flow between the data channels. The *Indirections* do not define selection semantics for the case that multiple connectors originate from the same data source. Therefore, we do not restrict the data flows here but stick to the mapping, which creates one data flow for each connector. The data flows originating from the *FlightProvider* both use the same output pin of the process resulting from mapping the data channel.

Structure: Data Flows between Sources/Sinks of Components Data flows as defined by *Indirections* always use data sources and data sinks. A data source or sink is always defined for a single data type. A connector between the data source of an assembly and the data sink of another assembly enables data flows between the assemblies. If at least one of the involved assemblies is a component, a connector can represent multiple data flows because there can be multiple actions using the same data source or data sink. Figure 7.15 gives an example of such a situation. The assembly *TravelPlannerFacade* is a component.

Listing 7.2: Additional Prolog clause to simplify accessing additional node properties for data channels.

```
containedInDataChannel(N) :-
  nodeCharacteristic(N, 'Containing', 'Data Channel').
```

The remaining assemblies are data channels. In the SEFF *findFlights*, the actions send the criteria for flights two times and receive the list of flights two times. The actions for emitting data use the same data source and the actions for consuming data use the same data sink. The *Indirections* do not define semantics on how to distribute data received on the data sink to the actions, i.e. it is unclear if the first or second consume action shall receive data. Therefore, two data flows are possible. This means, we have to map the connector from the *TravelAgencyFlightsDelegator* to the *TravelPlannerFacade* to two data flows: Both data flows start at the output pin of the process *TravelAgencyFlightsDelegator*, which represents the data channel in the DFD. One of the data flows targets the input pin of the *receive flights 1* process, which represents the first consume action from the SEFF. The other data flow targets the input pin of the *receive flights 2* process. The connector from the *TravelPlannerFacade* to the *TravelAgencyQueryBuilder* also represents two potential data flows: One data flow originates from the first emit action and one data flow originates from the second emit action. Both data flows target the *TravelAgencyQueryBuilder*. Consequently, we have to map the connector to two data flows in the DFD. One data flow starts at the output pin of the process, which represents the first emit action. The other data flow starts at the output pin of the process, which represents the other emit action. Both data flows target the input pin of the *TravelAgencyQueryBuilder* process.

Additional Prolog Clauses. The previously described mapping rules map a software architecture given in the Palladio ADL to a DFD. Later, the DFD is mapped to a logic program. To ease using the node properties of the *CallRole* and *Containing* characteristic types, we already added additional Prolog clauses as part of the mapping of control flow communication of Palladio. We extend these clauses by the clause shown in Listing 7.2 to handle the value *Data Channel*, which we added to the *Containing* characteristic type.

7.2.3. DSL for Defining Custom Analyses

Software architects cannot directly use the DSL for defining custom analyses as introduced for DFDs in Section 6.5 to formulate custom analyses for software architectures given in another ADL. First of all, the mapping from the DSL to a query in the logic program has to be changed to use the transitive transformation trace to resolve referenced elements of the software architecture. For instance, a DSL constraint, which uses the clearance characteristic type from the running example, could not be mapped to a query in the logic program because the identifier of the characteristic type in the logic program would be unknown. Second, the abstract and concrete syntax do not reflect the types of elements, which are available in the ADL. For instance, the DSL refers to *processes* and *nodes*, which

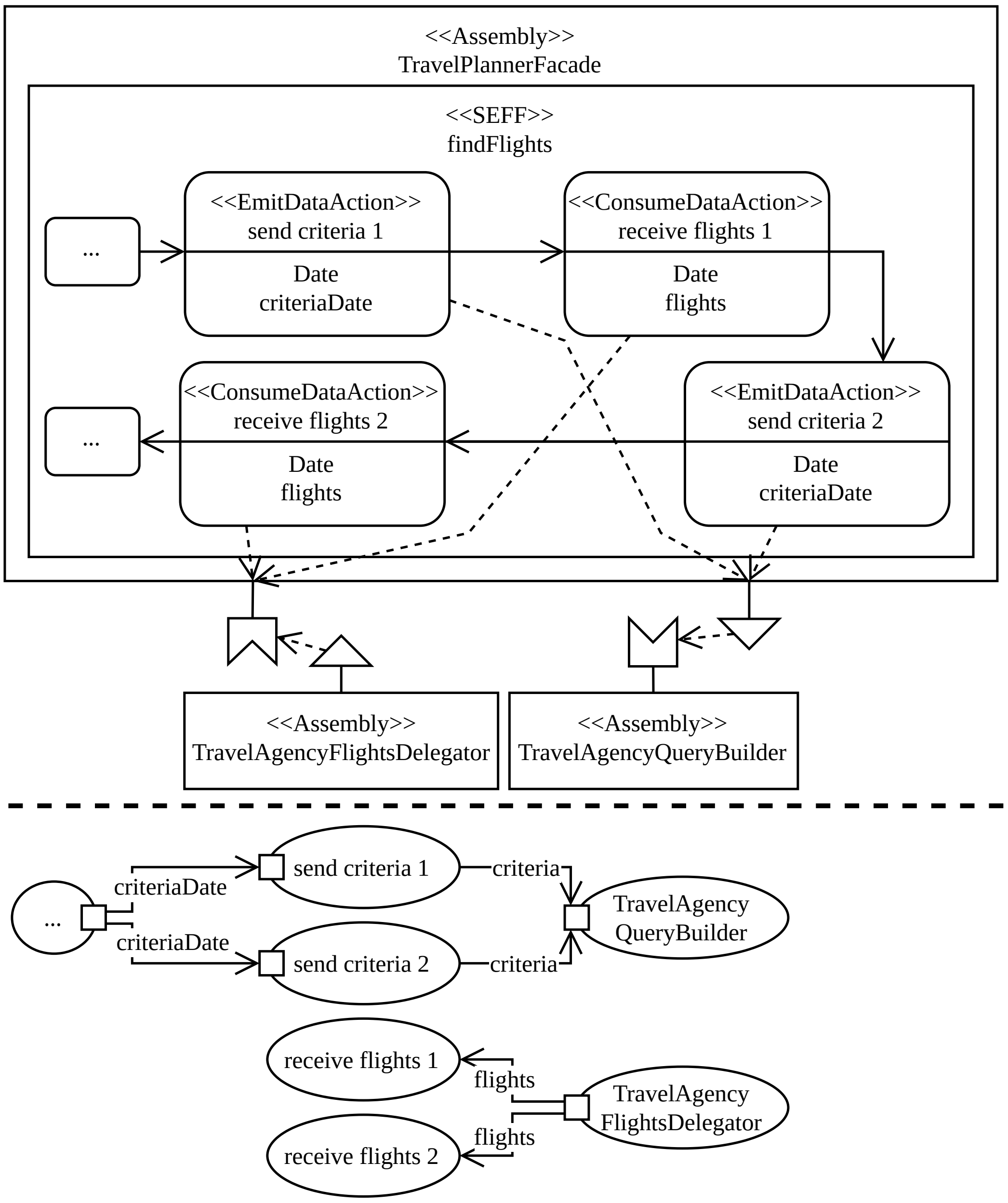


Figure 7.15.: Example of mapping between data flows from and to actions (top) to data flows between processes in a DFD (bottom).

are no domain concepts of the Palladio ADL. In addition, the DSL does not provide means to refer to actions in usage models or to particular SEFFs.

The previously mentioned shortcomings render the existing DSL useless but the tool engineer can define a new DSL, which is tailored to the ADL. However, the tool engineer does not have to recreate the DSL from scratch. He/she can reuse most parts of the abstract and concrete syntax as well as of the mapping and can address the shortcomings by changing the syntax or mapping.

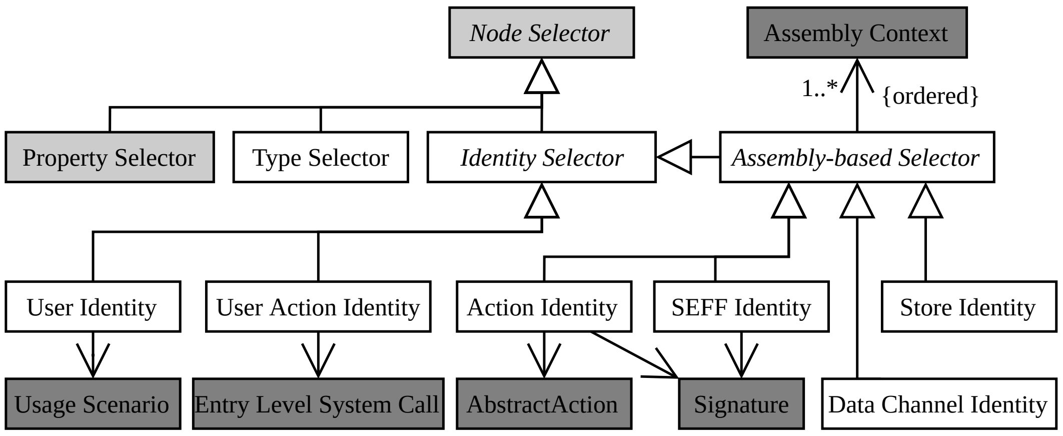


Figure 7.16.: Overview on adjusted elements of metamodel of DSL given as UML class diagram (light gray elements are part of the original DSL metamodel, dark gray elements are elements of the Palladio ADL and non-filled elements are newly defined or changed elements).

In the following, we describe the necessary changes in the abstract syntax, the concrete syntax and the mapping. We do not describe all parts of the newly created DSL because the descriptions in Section 6.5 already cover most aspects of the DSL. Instead, we only explain the adjusted parts of the DSL.

7.2.3.1. Abstract Syntax

The abstract syntax of the DSL is given as a metamodel. Most information described by the original metamodel, which we have introduced in Section 6.5.2, is still necessary and usable for describing analyses of software architectures given in an ADL: The selection of data as well as the conditions based on characteristic variables remain the same. However, the selection of nodes is different because the model elements, which represent nodes, are different. In the mappings described in Section 7.2.1.3 and Section 7.2.2.3, we consider six model elements of the Palladio ADL as nodes. We consider each of these model elements in the node selectors of types (Type Selector) and node selectors of identities (Identity Selector). Figure 7.16 visualizes an excerpt of the extended metamodel of the DSL to be used together with Palladio. In the following, we describe the six considered node types and how the extended metamodel represents them.

A Usage Scenario is mapped to an external actor. Consequently, the User Identity selector refers to a Usage Scenario to uniquely identify a user. An Entry Level System Call is a call action executed by a user. It is the only node of the user behavior, which is considered in the mapping. Consequently, the User Action Identity selector refers to an Entry Level System Call to uniquely identify that action.

The remaining nodes are represented by elements within the system. Because the system consists of assemblies, i.e. instances of components or data channels, the assembly hierarchy always has to be considered when uniquely identifying an element. The Assembly-based Selector is a supertype for all Identity Selectors, which refer to system elements. The selector requires an ordered list of Assembly Contexts, which represents the assembly hierarchy. The first element is the outermost assembly, i.e. the assembly directly contained in the system. The last element is the most nested assembly.

A store is represented by a special type of component. The assembly hierarchy of the Assembly-based Selector is already sufficient to uniquely identify a store. The store is always the last element of the ordered list of Assembly Contexts. Consequently, the Store Identity selector does not require additional information except for the assembly hierarchy provided by the Assembly-based Selector.

A data channel is also a special type of component. Therefore, the same considerations as for stores apply to data channels as well. Consequently, the Data Channel Identity selector only uses the assembly hierarchy provided by the Assembly-based Selector.

A SEFF describes the behavior of a component for a provided service. A provided service is identified by a Signature, through which a service can be called. This means that an assembly hierarchy uniquely identifies a component instance and a signature uniquely identifies a SEFF at such a component instance. Consequently, the SEFF Identity selector uses the assembly hierarchy provided by the Assembly-based Selector and a Signature to uniquely identify a SEFF.

An action within a SEFF can be identified by the action itself after the SEFF has been identified. Because the mapping not only considers call actions but also other types of actions such as actions for emitting data or defining return values, the Action Identity selector refers to an Abstract Action, which is the supertype of all actions within a SEFF. In addition, the Action Identity uses the assembly hierarchy and the Signature to uniquely identify the action.

The Type Selector also considers the six node types. The selector allows to choose any of the six previously mentioned node types (user, user action, store, data channel, SEFF and action). The semantics of this selection is that all nodes of that type are selected.

7.2.3.2. Concrete Syntax

Most parts of the concrete syntax, which we introduced in Section 6.5.3, remain the same. The parts of the concrete syntax, which we have to change or adjust, are the keyword for referring to nodes, the identity selectors and the type selector. We explain these changes in the following.

The keyword for referring to nodes was `node` in the original definition of the concrete syntax of the DSL. Because the term *node* is not used in the Palladio ADL, we do not use it anymore to avoid confusion. Instead, we use the keyword `element`, which does not use terminology of DFDs and can represent all six types of nodes, which we introduced in the discussion of the abstract syntax. In the following descriptions, we will also refer to *element* instead of *node*.

The abstract syntax contains various types of identity selectors for elements. To make them usable, we define a concrete syntax for all of these selectors. Examples of the concrete syntax are given in Listing 7.3. The imports in lines 1 to 3 are necessary because we refer to elements contained within these imported models. The concrete syntax of all identity selectors starts with the sequence `element.identity`. As already motivated for the original DSL, using dots to connect elements in an expression to navigate to a certain object or element is common practice, so we also connect the individual parts of the identity selectors by dots. Afterwards, the type of element has to be specified. In line 5, a user shall be selected by his/her identifier, so the keyword `SystemUser` is used. As can be seen in Listing 7.3, there are keywords for all six element types. After the keyword, the element has to be specified. For users (see line 5), the identifier only consists of the name of the user, i.e. the name of the user represents the reference to the Usage Scenario from the abstract syntax. For user actions (see line 8), the identifier consists of the identifier of the user followed by the name of the action of that user, i.e. the name of the action represents the reference to the Entry Level System Call from the abstract syntax. For stores (see line 11), the identifier consists of an ordered list of assembly names connected by dots, i.e. the sequence of names represents the ordered list of Assembly Contexts from the abstract syntax. In the given example, `FlightDB` is the name of the instance of the store. `Airline` is the name of the component instance, which contains the store. For data channels (see line 14), the identifier is essentially the same as for stores but the last assembly name has to refer to an instance of a data channel instead of a store. For SEFFs (see line 17), the identifier also uses the sequence of assembly names but adds the name of the signature, which the SEFF describes, to the end, i.e. the name of the signature represents the reference to the Signature from the abstract syntax. For actions (see line 20), the identifier also uses the sequence of assembly names and the name of the signature but adds name of the action to the end, i.e. the name of the action represents the reference to the Abstract Action from the abstract syntax.

The concrete syntax of a type selector is as shown in Listing 7.4. The selector starts with the sequence `element.type` followed by the selected type connected by a dot. In the example, the elements representing a user are selected by the keyword `SystemUser`. The keywords are the same keywords as used for identity selectors.

7.2.3.3. Mapping to Logic Program

The description of a mapping from the model elements of the abstract syntax of the DSL to clauses in the logic program serves two purposes: First, it assigns a meaning to an element of the abstract syntax. Second, it enables automated evaluations of the specified analyses within tooling. We already described the mapping of the DSL without the adjustments for Palladio in Section 6.5.4. The major part of this description still holds. Especially, the mapping of data selectors, conditions and node selectors for properties is still valid. However, we did not define the meaning for and the mapping of the newly introduced selectors for the identity and type of elements yet. We describe their meaning and mapping in the following but omit general explanations of default clauses such as

Listing 7.3: Examples of identity selectors in constraint DSL.

```
import "travelPlanner.usagemodel"
import "travelPlanner.system"
import "travelPlanner.repository"
constraint NoFlowsToUser {
 data.any NEVER FLOWS element.identity.SystemUser.User
}
constraint NoFlowsToUserAction {
 data.any NEVER FLOWS element.identity.UserAction.User.findFlights
}
constraint NoFlowsToStore {
 data.any NEVER FLOWS element.identity.Store.Airline.FlightDB
}
constraint NoFlowsToDataChannel {
 data.any NEVER FLOWS element.identity.DataChannel.Airline.FlightSelector
}
constraint NoFlowsToSEFF {
 data.any NEVER FLOWS element.identity.SEFF.Airline.AirlineLogic.addFlight
}
constraint NoFlowsToAction {
 data.any NEVER FLOWS element.identity.Action.Airline.AirlineLogic.addFlight.call
}
```

Listing 7.4: Examples of type selector in constraint DSL.

```
constraint NoFlowsToAnyUser {
 data.any NEVER FLOWS element.type.SystemUser
}
```

`inputPin/2` or `flowTree/3` because we already explained them in depth in the mapping of the DSL without the adjustments for Palladio.

User Identity Selector. An identity selector selects elements based on their identifier. In the DSL, the selector always refers to one particular element in the software architecture given in Palladio. Because we map model elements of Palladio to one or multiple elements, it is possible that an identity selector does not only refer to a single element in the DFD and also in the logic program but to multiple elements. The identity selector in Figure 7.17 selects the user named `User`. In our mapping of the extended Palladio ADL to a DFD, we mapped a usage scenario to an actor in the DFD. The calls of the user became actor processes in the DFD. The actor itself does not emit or consume data but uses the actor processes to process data. Consequently, when referring to a user in an analysis definition, it is reasonable and intuitive to consider all activities of this user. Figure 7.18 visualizes this for the user selector. This means, we map the identity selector for a user to all actor processes, which belong to the selected user. We use the transformation trace to look up the identifiers of the actor processes in the logic program, which match the selection. In the example in Figure 7.17, the nodes are the entry and exit processes of the calls of the user. The individual clauses that unify the node identifier `N` in the logic program with the

```
constraint NoFlowsToUser {
   data.any NEVER FLOWS element.identity.SystemUser.User
}
```

⇓

```
constraint('NoFlowsToUser', N, PIN, S) :-
  inputPin(N, PIN),
  flowTree(N, PIN, S),
  (
   N = 'User.findFlights entry';
   N = 'User.findFlights exit';
   N = 'User.getCCD entry';
   N = 'User.getCCD exit';
   N = 'User.bookFlight entry';
   N = 'User.bookFlight exit'
  ).
```

Figure 7.17.: Example of mapping an identity selector for a user from DSL constraint to logic program.

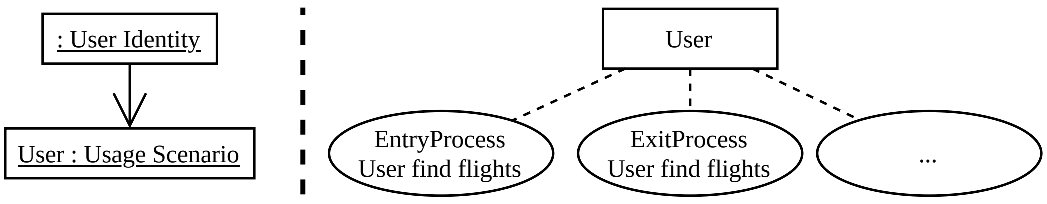


Figure 7.18.: DFD elements (right) considered by an identity selector for a user (left).

selected identifier, are collected in a disjunction. This means that any of the selected nodes can be used to identify a violation.

Remaining Identity Selectors. All identity selectors follow the structure illustrated in Figure 7.17, i.e. there is a disjunction of identified elements, which are derived from the selected Palladio element by looking up related elements in the transformation trace. However, the considered DFD elements are different for all types of identity selectors. For selectors of user actions, the entry and exit processes, which correspond to the selected call action of the user according to the transformation trace, are considered as shown in Figure 7.19. There are no other related elements to be considered. For selectors of stores, the DFD store mapped from the store in Palladio is considered as shown in Figure 7.20. It would also be possible to consider the entry and exit processes of the store but because there is only one data flow from the entry process and one data flow to the exit process and both data flows are connected to the store, it is sufficient to only consider the store. For selectors of data channels, the process mapped from the data channel instance in Palladio is considered as shown in Figure 7.21. There are no other elements related to the data channel, which could be considered. For selectors of SEFFs, the entry as well as the exit process mapped from the selected SEFF are considered as shown in Figure 7.22. We do not consider the actions within the selected SEFF because these elements can also be

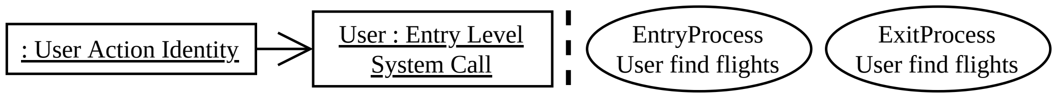

: User Action Identity
User : Entry Level System Call
EntryProcess User find flights
ExitProcess User find flights

Figure 7.19.: DFD elements (right) considered by an identity selector for a user action (left).

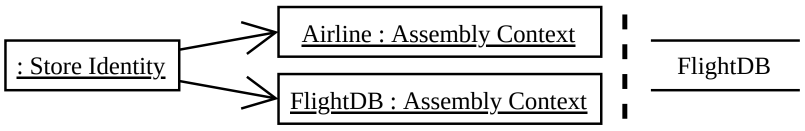


Figure 7.20.: DFD elements (right) considered by an identity selector for a store (left).

selected by the identity selectors of the actions and it can be useful to only consider the data exchanged by a SEFF via a call. The entry and exit processes are appropriate for this purpose. For selectors of actions within a SEFF, we consider the processes mapped from the selected action as shown in Figure 7.23. There are no other elements related to the action, which we could consider.

Type Selectors for Users/User Actions. The meaning of a type selector is that all model elements, which are of a certain type, shall be considered in the analysis. The types are given as types from the Palladio ADL. The available types of elements are the six types, which we already discussed for the identity selectors. The type selectors for users and user actions essentially map to the same clauses as shown in Figure 7.24. The mapping uses the clause `containedInScenarioBehaviour/1`, which we introduced in the mapping from the Palladio ADL to a DFD as an additional clause. The clause matches all identifiers of nodes N, which originate from call actions of a user. It is reasonable to not distinguish between the user and its actions because the user himself/herself does not have a dedicated behavior but his/her behavior is defined by the actor processes, which belong to him/her.

Type Selectors for Remaining Types. The mapping of the type selector for stores shown in Figure 7.25 is rather simple because we can simply test if a given node N is a store in the DFD by using the `store/1` clause. This selector matches all Palladio elements, which are mapped to stores. The mapping of the type selector for data channels shown in Figure 7.26 shall match all DFD elements, which have been mapped from a data channel. We use the `containedInDataChannel/1` clause, which we introduced in the mapping from the Palladio ADL to a DFD as an additional clause. The clause considers exactly all processes, which have been mapped from a data channel. The mapping of the type selector for SEFFs

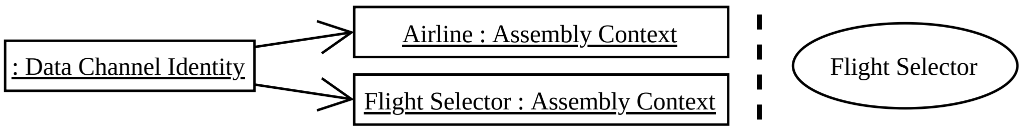

: Data Channel Identity
Airline : Assembly Context
Flight Selector : Assembly Context
Flight Selector

Figure 7.21.: DFD elements (right) considered by an identity selector for a data channel (left).

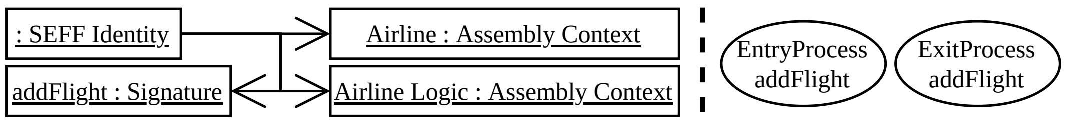


Figure 7.22.: DFD elements (right) considered by an identity selector for a SEFF (left).

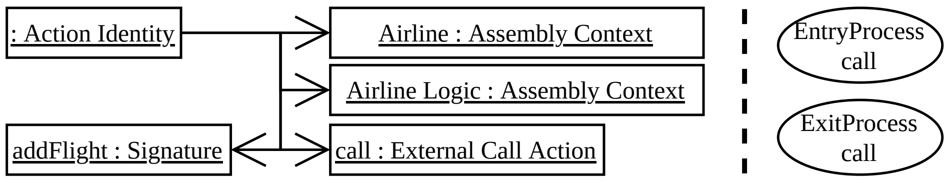

: Action Identity
addFlight : Signature
Airline : Assembly Context
Airline Logic : Assembly Context
call : External Call Action
EntryProcess call
ExitProcess call

Figure 7.23.: DFD elements (right) considered by an identity selector for an action in a SEFF (left).

shown in Figure 7.27 matches the entry and exit processes of SEFFs. It is reasonable to only match these two types of processes because they are additional processes only added to represent the call receiving and returning of a service. By only matching these two processes, it is possible to ignore the data processing within a SEFF but focus on the data transmissions when calling a service as well as when returning from a service. We use the clauses `isASEFFEntry/1` and `isASEFFExit/1`, which we introduced in the mapping from the Palladio ADL to a DFD as an additional clause, in a disjunction. This means all entry and exit processes are considered. The mapping of the type selector for actions within SEFFs also shown in Figure 7.27 matches the remaining processes, which have been mapped from actions within SEFFs. We use the `containedInComponent/1` clause, which we introduced in the mapping from the Palladio ADL to a DFD as an additional clause, to identify processes, which have been created by a mapping from an element within a component, i.e. actions and SEFF entry and exit processes. Because the SEFF entry and

```
constraint NoFlowsToAnyUser {
  data.any NEVER FLOWS element.type.SystemUser
}
constraint NoFlowsToAnyUserAction {
  data.any NEVER FLOWS element.type.UserAction
}
```

⇓

```
constraint('NoFlowsToAnyUser', N, PIN, S) :-
 inputPin(N, PIN), flowTree(N, PIN, S),
 containedInScenarioBehaviour(N).
constraint('NoFlowsToAnyUserAction', N, PIN, S) :-
 inputPin(N, PIN), flowTree(N, PIN, S),
 containedInScenarioBehaviour(N).
```

Figure 7.24.: Example of mapping a type selector for users or user actions from DSL constraints to logic programs.

```
constraint NoFlowsToAnyStore {
   data.any NEVER FLOWS element.type.Store
}
```

$$\Downarrow$$

```
constraint('NoFlowsToAnyStore', N, PIN, S) :-
  inputPin(N, PIN), flowTree(N, PIN, S),
  store(N).
```

Figure 7.25.: Example of mapping a type selector for stores from DSL constraint to the logic program.

```
constraint NoFlowsToAnyDataChannel {
   data.any NEVER FLOWS element.type.DataChannel
}
```

$$\Downarrow$$

```
constraint('NoFlowsToAnyDataChannel', N, PIN, S) :-
  inputPin(N, PIN), flowTree(N, PIN, S),
  containedInDataChannel(N).
```

Figure 7.26.: Example of mapping a type selector for data channels from DSL constraint to the logic program.

exit processes do not originate from actions, we exclude them by requesting that a node N originated from a component but is not a SEFF entry or exit process.

```
constraint NoFlowsToAnySEFF {
   data.any NEVER FLOWS element.type.SEFF
}
constraint NoFlowsToAnyAction {
   data.any NEVER FLOWS element.type.Action
}
```

$$\Downarrow$$

```
constraint('NoFlowsToAnySEFF', N, PIN, S) :-
  inputPin(N, PIN), flowTree(N, PIN, S),
  (isASEFFEntry(N); isASEFFExit(N)).
constraint('NoFlowsToAnyAction', N, PIN, S) :-
  inputPin(N, PIN), flowTree(N, PIN, S),
  (containedInComponent(N), \+ isASEFFEntry(N), \+ isASEFFExit(N)).
```

Figure 7.27.: Example of mapping a type selector for SEFFs or actions within SEFFs from DSL constraints to logic programs.

ID	Description	Covering Part
R3.1	support control flow ADLs	guidelines and application to Palladio
R3.2	support data flow ADLs	guidelines and application to Palladio
R3.3	high reuse of ADL elements	identification/extension steps
R3.4	analysis framework	see Table 7.2
R3.5	stay on architecture level	mapping and analysis DSL

Table 7.1.: Overview on requirements on the integration guidelines and how they are met.

7.3. Requirements Coverage

The integration guidelines presented in this chapter meet all requirements, which we defined in Section 4.1.3. Table 7.1 gives an overview on the requirements as well as how the integration guidelines meet these requirements.

The requirements to support ADLs using control flows (R3.1) and ADLs using data flows (R3.2) are met by the integration guidelines, which we introduce in Section 7.1. We show that the guidelines are applicable to ADLs using control flows by applying them to the subset of Palladio, which uses control flows, in Section 7.2.1. We show that the guidelines are applicable to ADLs using data flows by applying them to the subset of Palladio, which uses data flows, in Section 7.2.2. To reuse as much ADL elements as possible (R3.3), we introduced a step to capture missing but mandatory concepts of the existing ADL in the integration guidelines in Section 7.1. The step to extend the ADL only introduces new concepts if there is no matching concept available. Therefore, the tool engineer has to make use of as much existing concepts as possible, which also implies a high reuse of ADL elements. The architect does not have to be aware of DFDs and Prolog, i.e. only has to be aware of domain concepts of the architectural design level (R3.5), because everything, which uses concepts not present in the ADL, is hidden from the architect: The guidelines in Section 7.1 yield a mapping for the software architecture given in an ADL to a DFD as well as a DSL for formulating custom analyses. Because the mapping can be automated completely and the DSL avoids writing Prolog code, the architect can model and analyze architectures without the need to be aware of any underlying concept. The requirement on the analysis framework (R3.4) is that the analysis framework yielded by the integration guidelines shall meet all requirements on the analysis framework for DFDs. Table 7.2 provides an overview on how the analysis framework resulting from applying the integration guidelines meet these requirements. We discuss the table in the following.

The analysis framework yielded by the integration guidelines describes the semantics of the ADL elements by a mapping to a DFD. We demonstrated this for Palladio in Section 7.2.1.3 and Section 7.2.2.3. The description of the mapping only covers ADL elements, which actually have an effect on data flows. However, the mapping still describes the semantics for all ADL elements (R2.1) because every ADL element, which is not explicitly mentioned in the mapping, is a neutral element with respect to the data flows. This means it does neither affect the structure of the resulting DFD nor the behavior of the label propagation.

ID	Description	User	Covering Part
R2.1	every element covered	–	mapping to DFD
R2.2	derivation of properties	analysis	label lookup in DFDs
R2.3	origin of properties	analysis	flow tree from DFDs
R2.4	analyses based on goals	expert	analysis procedure for DFDs
R2.5	analyses based on goals	architect	DSL for custom analyses
R2.6	tracing of properties	architect	flow tree from DFDs
R2.7	automated analyses	architect	analysis procedure
R2.8	information flow	expert	analysis definitions for DFDs
R2.9	access control	expert	analysis definitions for DFDs

Table 7.2.: Overview on requirements on the analysis framework and how the analysis framework of the integration procedure meets them.

The mapping from an architecture given in an ADL to a DFD allows to reuse the analysis framework for DFDs. Therefore, we automatically meet the requirements regarding the derivation of properties (R2.2), the tracing of properties (R2.3 and R2.6), the definition of analyses based on goals by the security expert (R2.4) as well as the support for information flow analyses (R2.8) and access control analyses (R2.9).

The DSL for formulating custom analyses meets the requirement regarding the definition of analyses based on goals by the software architect (R2.5). As we describe in Section 7.2.3, the DSL does not require knowledge about DFDs or logic programming but only knowledge about software architectures.

The analysis procedure described in Section 7.1 supports automated analyses (R2.7). This is possible because all mappings as well as the label propagation and comparison can be fully automated.

7.4. Assumptions and Limitations

This section discusses assumptions and limitations of the integration guidelines as well as of the particular integrations into Palladio.

Implications of reuse The integration guidelines aim for reusing as much as possible of the analysis framework and the analysis definitions for DFDs. Therefore, the resulting integration shares the assumptions of limitations of the analysis framework, which we discuss in Section 5.3, as well as of the analysis definitions, which we discuss in Section 6.7.

No implicit flows In mappings from ADLs, which use control flows, to DFDs, it is necessary to derive data flows from the existing descriptions. The approach presented in the previous sections is to treat exchanged parameters and return values as data and the exchange as a data flow. We make these data flows explicit by mapping them to data flows in the DFD. However, there are also implicit information flows such as changed timing-behavior, which an attacker can observe. As already discussed in Section 6.2.1, we exclude such implicit flows because software architectures do not provide information with enough details to reason about implicit flows.

ADL elements without effect In the mappings from architectures given in an ADL to DFDs, usually not all ADL elements are mapped to a counterpart in a DFD. This is not surprising because ADLs often not only represent information to analyze confidentiality properties but also information for other quality properties. The assumption in the mappings is that the ADL elements, which are not mapped, do not affect confidentiality. This is not entirely true in our mappings because we do not represent ADL elements, which affect control flows, in the DFDs. However, elements such as branches can imply implicit flows. Instead, we only consider the explicitly exchanged data between actions within a control flow. However, the assumption that these elements do not affect confidentiality is true with respect to the confidentiality analyses, which we want to conduct. As described before, we exclude implicit flows anyways, so there is no effect on the confidentiality analyses.

Independent treatment of parameters As part of the integration of confidentiality analyses in Palladio, we map each parameter transmitted via a calling action to one data flow. If there are multiple calls to the same SEFF, there will be multiple data flows to the same pin. All of these data flows are treated independently. This means that the analysis also combines data flows from two different calls when propagating labels. The reason to do this is that a DFD process could potentially cache received parameters and combine cached parameters with newly received parameters. Thereby, the analysis overestimates potential confidentiality problems. It is possible to avoid this overestimation by providing special `flowTree/3` clauses, which only yield flow trees that use flows from the same source. However, we did not implement this option.

Overestimation of data flows via sinks and sources As already discussed in Section 7.2.2.3, we consider all possible combinations of data flows between data sources and data sinks if an action in a SEFF is involved. We do this because the Palladio extension *Indirections* does not provide semantics for the case that there are multiple actions using the same data sinks or data sources. Doing an overestimation is the most conservative handling of this situation. As soon as there are semantics for this situation, the mapping can consider them.

7.5. Summary

In this chapter, we presented how to integrate DFD-based confidentiality analyses into existing ADLs. A set of integration guidelines specifies the integration procedure and the application of the guidelines to the Palladio ADL demonstrated their applicability.

The integration guidelines in Section 7.1 provide a process to integrate DFD-based analyses into existing ADLs. The process consists of four steps: First, the tool engineer identifies concepts, which are essential but which are not represented in the ADL. An extension of the ADL introduces these concepts. A definition of a mapping from the ADL to a DFD enables reusing the existing analysis framework for DFD-based analyses. By adjusting the DSL for formulating custom analyses, the tool engineer provides the software architect with the means to define new analyses.

The application of the integration guidelines to the Palladio ADL demonstrates the applicability of the guidelines in Section 7.2. Because Palladio uses call-and-return communication as well as communication based on data flows, we can demonstrate the integration guidelines for the subset of the Palladio ADL, which uses control flows, in Section 7.2.1 as well as for the subset of the Palladio ADL, which uses data flows, in Section 7.2.2. In addition, we show how a tool engineer can adjust the DSL for formulating custom analyses in Section 7.2.3.

The integration guidelines meet the corresponding requirements, which we defined in Section 4.1.3. We explain how the integration guidelines meet the requirements in Section 7.3.

Because the integration guidelines aim to reuse the analysis framework for DFDs, they share the same assumptions and limitations but there are also additional assumptions and limitations, which we discuss in Section 7.4. The most prominent limitation is the exclusion of implicit information flows. The most prominent assumptions are that certain ADL elements do not affect confidentiality and that overestimations of data flows are reasonable.

8. Validation

The overall goal of the validation is to show that the contributions sufficiently answer the corresponding research questions. To ensure that the validation achieves this goal, we refine this high-level goal into further validation goals and discuss the data, which we have to collect to decide whether a goal is achieved. We present the validation goals and the necessary data in the overview on our validation in Section 8.1.

The major part of the validation is based on case studies. A case study always involves particular systems, to which the contributions are applied. To ensure that the case study systems support the validation, we derive requirements on the selection of the systems and the systems themselves. Afterwards we present the selected systems. We describe the requirements and the selected systems in Section 8.2.

We structure the presentation of the actual validations by the validation goals, which also align with the contributions. For every validation goal, we present the validation design and the results. Afterwards, we discuss these results as well as threats to validity. The validation of the extended DFD syntax (C1) is covered in Section 8.3. Section 8.4 describes the validation of the DFD analyses (C3). The validation of the DFD semantics (C2) is subject to Section 8.5. Eventually, we describe the validation of the ADL integration guidelines (C4) in Section 8.6. We summarize the results of the validations and the implications on the validation goals in Section 8.7.

8.1. Overview

The goal of the validation is to show that we sufficiently answered the research questions presented in Section 1.4. Therefore, we structure the validation by the contributions, which represent the answers to the research questions. We applied the Goal-Question-Metric (GQM) approach [BW84; BCR94], which we explain in the next paragraph, to break down the high-level validation goals into validation questions and corresponding metrics to answer these questions. These validation questions and metrics support a focused validation design. In this section, we focus on the resulting so-called GQM plan.

The GQM approach [BW84; BCR94] provides guidelines on how to effectively achieve a certain goal. In the context of a validation, the goal is usually to validate that a certain contribution meets certain quality standards or appropriately answers research questions. Usually, a measurement in an experiment or case study is not sufficient to completely achieve such a validation goal. Therefore, the approach suggests to define validation

Validation Goal	Contribution	Research Question
VG1	C1) DFD Syntax	RQ1) modeling access control
		RQ2) modeling information flow
		RQ3) modeling primitives
VG2	C3) DFD Analyses	RQ5) access control analyses
		RQ6) information flow analyses
VG3	C2) DFD Semantics	RQ4) analysis semantics for DFDs
VG4	C4) ADL Integration	RQ7) integration control flow ADLs
		RQ8) integration data flow ADLs

Table 8.1.: Relation between validation goals (VG), contributions (C) and research questions (RQ).

questions. An answer to such a question gives insights in whether a certain aspect of the goal has been achieved. By summing up the answers to all questions, we can decide whether a goal has been achieved. The answers to the questions are given in terms of metrics and a guideline on how to interpret these metrics. For instance, a metric collecting the duration of an execution is reasonable to rate the performance of an approach but it is not sufficient to answer a question about appropriate performance without a guideline on how to interpret such a duration. Such a guideline can be a maximum acceptable duration or a reference duration from another approach. A metric does not necessarily have to be based on a quantitative measurement but can also be a qualitative result as long as a clear guideline on how to interpret the results is available.

The goals in our GQM plan are to validate that the contributions C1–C4 sufficiently address the corresponding research questions RQ1–RQ8. Table 8.1 illustrates the relation between the validation goals (VG), the contributions and the research questions. We derive the validation questions to achieve the validation goals from the research questions and corresponding motivations as well as explanations in Section 1.4. The resulting GQM plan is illustrated in Figure 8.1 for the contributions C1, C2 and C3 as well as in Figure 8.2 for contribution C4. In the following paragraphs, we describe all parts of the plan and show that the questions and metrics are reasonable and sufficient to achieve the validation goals.

8.1.1. Validation Goal 1: Validate DFD Syntax

The validation goal about the DFD syntax is to show that the syntax sufficiently answers the research questions RQ1, RQ2 and RQ3. RQ1 asks what information is necessary to reason about access control. RQ2 asks the same for information flow control. In the DFD syntax, we decided that properties of nodes and behaviors formulated as label propagation functions are sufficient to reason about access control as well as information flow control. Therefore, we have to validate that these elements of the modeling language sufficiently describe systems in a way that reasoning about access control and information flow control

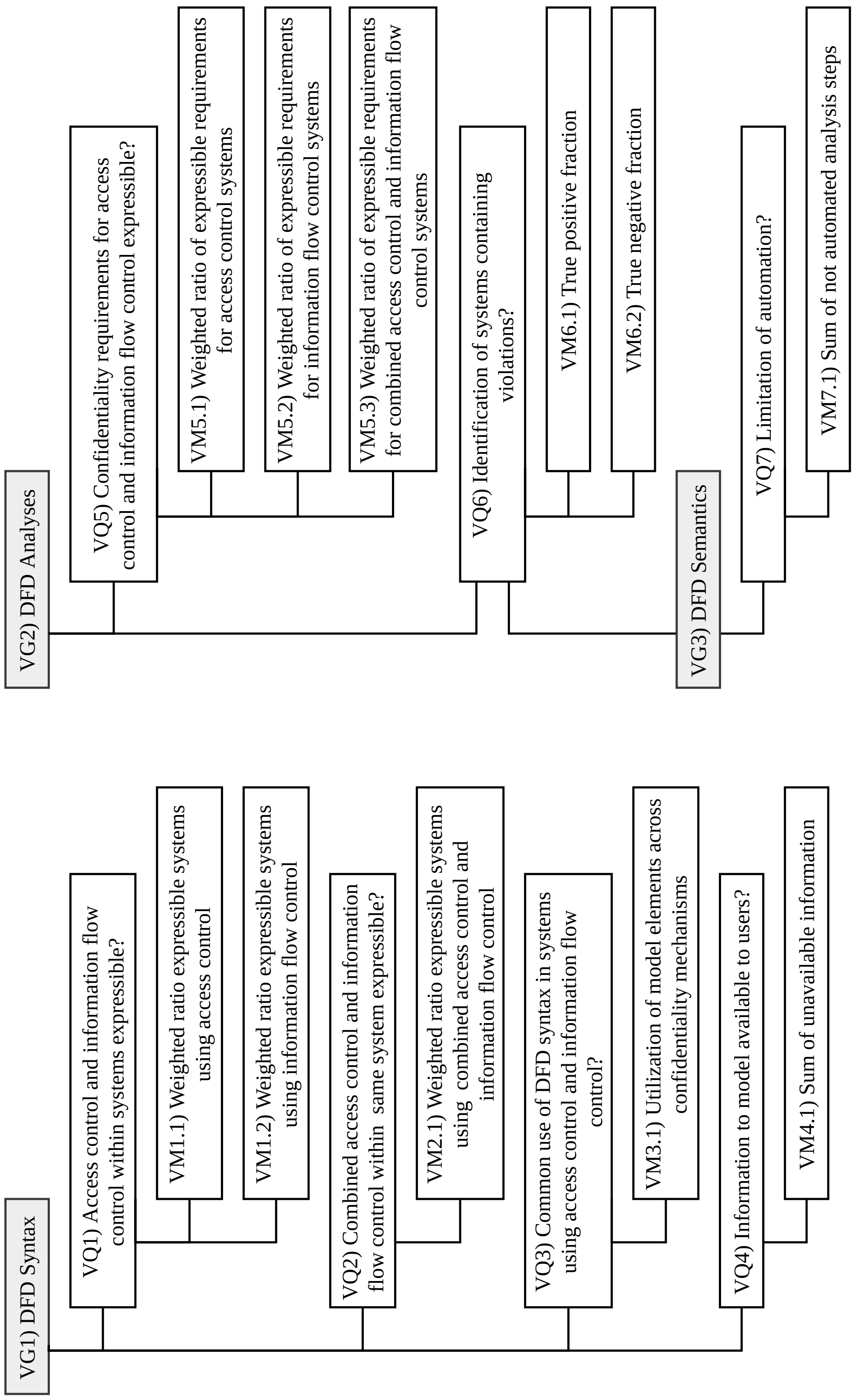

VG1) DFD Syntax
VQ1) Access control and information flow control within systems expressible?
VM1.1) Weighted ratio expressible systems using access control
VM1.2) Weighted ratio expressible systems using information flow control
VQ2) Combined access control and information flow control within same system expressible?
VM2.1) Weighted ratio expressible systems using combined access control and information flow control
VQ3) Common use of DFD syntax in systems using access control and information flow control?
VM3.1) Utilization of model elements across confidentiality mechanisms
VQ4) Information to model available to users?
VM4.1) Sum of unavailable information
VG2) DFD Analyses
VQ5) Confidentiality requirements for access control and information flow control expressible?
VM5.1) Weighted ratio of expressible requirements for access control systems
VM5.2) Weighted ratio of expressible requirements for information flow control systems
VM5.3) Weighted ratio of expressible requirements for combined access control and information flow control systems
VQ6) Identification of systems containing violations?
VM6.1) True positive fraction
VM6.2) True negative fraction
VG3) DFD Semantics
VQ7) Limitation of automation?
VM7.1) Sum of not automated analysis steps

Figure 8.1.: Overview on GQM plan to validate the contributions C1, C2 and C3.

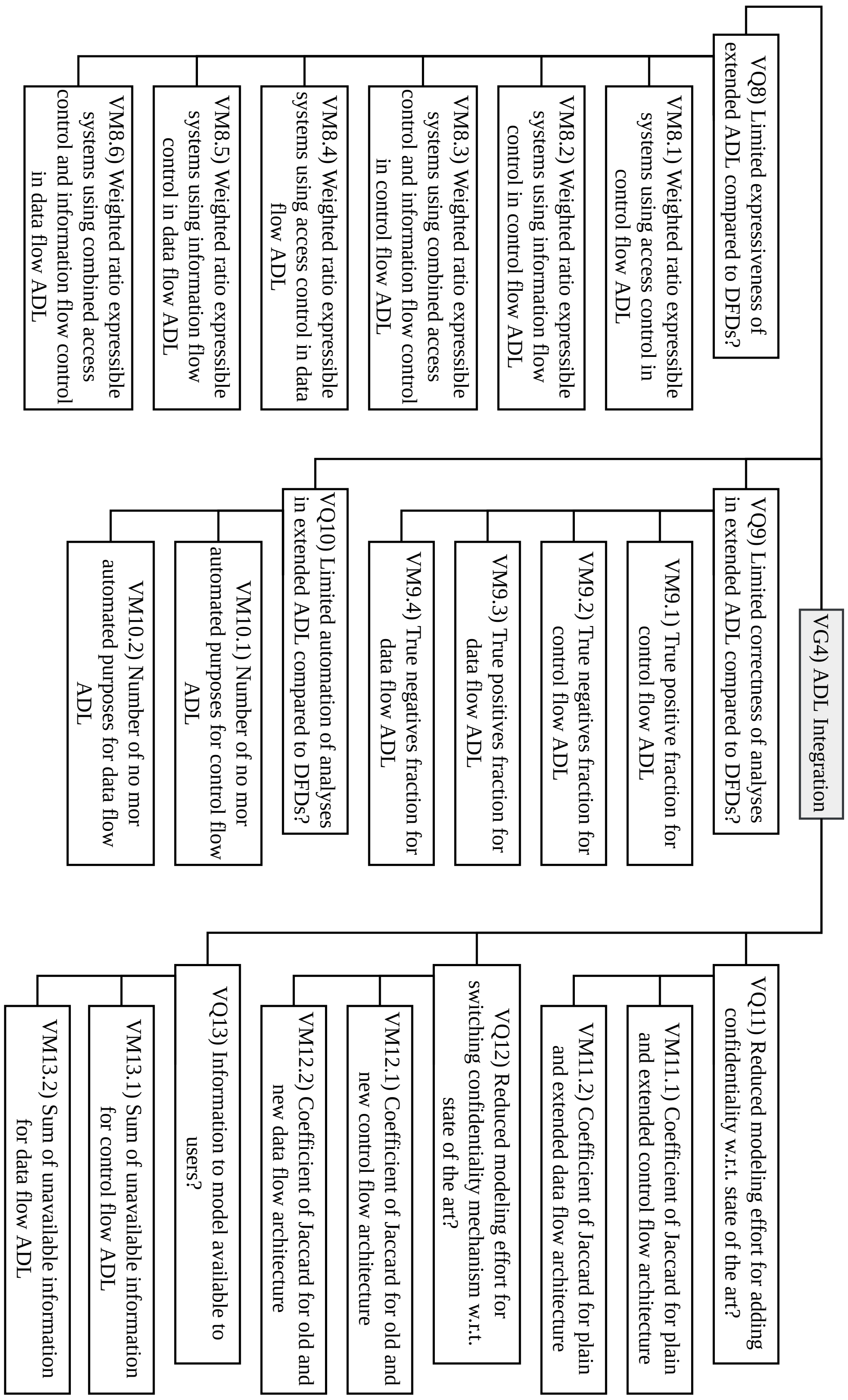

VG4) ADL Integration
VQ8) Limited expressiveness of extended ADL compared to DFDs?
VM8.1) Weighted ratio expressible systems using access control in control flow ADL
VM8.2) Weighted ratio expressible systems using information flow control in control flow ADL
VM8.3) Weighted ratio expressible systems using combined access control and information flow control in control flow ADL
VM8.4) Weighted ratio expressible systems using access control in data flow ADL
VM8.5) Weighted ratio expressible systems using information flow control in data flow ADL
VM8.6) Weighted ratio expressible systems using combined access control and information flow control in data flow ADL
VQ9) Limited correctness of analyses in extended ADL compared to DFDs?
VM9.1) True positive fraction for control flow ADL
VM9.2) True negatives fraction for control flow ADL
VM9.3) True positives fraction for data flow ADL
VM9.4) True negatives fraction for data flow ADL
VQ10) Limited automation of analyses in extended ADL compared to DFDs?
VM10.1) Number of no mor automated purposes for control flow ADL
VM10.2) Number of no mor automated purposes for data flow ADL
VQ11) Reduced modeling effort for adding confidentiality w.r.t. state of the art?
VM11.1) Coefficient of Jaccard for plain and extended control flow architecture
VM11.2) Coefficient of Jaccard for plain and extended data flow architecture
VQ12) Reduced modeling effort for switching confidentiality mechanism w.r.t. state of the art?
VM12.1) Coefficient of Jaccard for old and new control flow architecture
VM12.2) Coefficient of Jaccard for old and new data flow architecture
VQ13) Information to model available to users?
VM13.1) Sum of unavailable information for control flow ADL
VM13.2) Sum of unavailable information for data flow ADL

Figure 8.2.: Overview on GQM plan to validate the contribution C4.

is possible. To validate the capability of describing systems, we formulate the validation questions VQ1 and VQ2, which we describe in the next paragraphs. The validation of the analyses and semantics in VQ6 of VG3 will show that the modeled systems can be used to reason about access control or information flow control. RQ3 asks what modeling primitives, i.e. elements of the modeling language, are required. We have to validate that every element of the modeling language is necessary to model systems containing access control and information flow control. We formulate the validation question VQ3, which we describe in the next paragraphs, to validate this. Having a set of required elements of a modeling language is only helpful if an architect or security expert has the required information to create models by using these elements. Therefore, we have to validate that the required information to use the model elements is available while creating a software architecture. To validate this, we formulate VQ4, which we describe in the next paragraphs.

VQ1) Can the DFD syntax express access control and information flow control mechanisms within systems?

The focus on expressing mechanisms in the context of a particular system is important because architects are usually interested in analyzing a particular system under design. We already demonstrated in Section 6.2 that the syntax can express access control and information flow control mechanisms in general. However, this did not demonstrate that the chosen way of expressing mechanisms fits the needs of particular systems. Assume we have a set of systems S, where each system $s \in S$ uses a certain mechanism. A reasonable metric to answer the validation question could be the ratio r of the expressible systems $S_e \subseteq S$ to the total amount of systems S, i.e. $r = \frac{|S_e|}{|S|}$. A metric value of 1.0 means that the DFD syntax can express all mechanisms within systems. However, the metric is problematic if there are multiple systems using the same mechanisms: Imagine that the DFD syntax can express DAC very well but certain other mechanisms not. If the set of systems contains many systems using DAC but only a few systems containing other mechanisms, the metric value would hide the bad expressiveness regarding other mechanisms. To avoid this effect of an unbalanced set of systems with respect to the used mechanism, we use a normalizing metric, i.e. a weighted ratio metric. First, we define a set of mechanisms M and extend the ratio metric r to only consider systems containing a certain mechanism $m \in M$, i.e. $r(m)$. Based on that, the weighted ratio metric $\tilde{r}$ is $\tilde{r} = \sum_{m \in M} \frac{r(m)}{|M|}$. A metric value of 1.0 means that the DFD syntax can express all mechanisms within systems. Assuming that all systems stem from the state of the art and have been published together with a modeling approach, the expected value of the weighted ratio metric $\tilde{r}$ is also 1.0 because all of these systems can be modeled by state-of-the-art approaches. To decide whether we sufficiently addressed RQ1 and RQ2 individually, we define the following two dedicated metrics:

VM1.1) Weighted ratio $\tilde{r}$ of expressible systems using access control and the set of systems using access control.

VM1.2) Weighted ratio $\tilde{r}$ of expressible systems using information flow control and the set of systems using information flow control.

The previous validation question aims at validating that the DFD syntax can express systems using either access control or information flow control. However, combining multiple confidentiality mechanisms can be beneficial. Therefore, the DFD syntax should also support this as we already motivated while explaining the research questions in Section 1.4.1. Therefore, we formulate the following validation question:

VQ2) Can the DFD syntax express combinations of access control and information flow control mechanisms within the same system?

The validation question is about validating that the DFD syntax supports systems using combinations of mechanisms. In theory, all possible combinations of mechanisms are possible. The new set $M_P = P(M) \setminus (M \cup \emptyset)$ with the power set $P(M)$ of the set of mechanisms M describes these combinations. It is unrealistic to assume that there is at least one system for each $m \in M_P$ because M_P contains too many elements. Therefore, we only expect a subset $M_c \subseteq M_P$ to be available. We also cannot assume that there will be the same number of systems for each $m \in M_c$. Therefore, the weighted ratio metric $\tilde{r}$ as already used in VQ1 is appropriate. We redefine the ratio metric $r(m)$, which is used in $\tilde{r}$, to only consider systems using a combination of mechanisms $m \in M_c$. A metric value of 1.0 means that the DFD syntax can represent all investigated combinations of mechanisms. Because modeling approaches in the state of the art often do not support combinations of mechanisms, a metric value above 0.0 can already be considered a good result. This brings us to the following metric.

VM2.1) Weighted ratio $\tilde{r}$ of expressible systems using any combination of access control and information flow control mechanisms.

RQ3 asks for the necessary model elements in the DFD syntax in order to express systems using access control and information flow control. The previous validation questions VQ1 and VQ2 already validate that the model elements are sufficient to express such systems. However, the questions did not validate whether all model elements are actually necessary or whether there are model elements, which are specific for certain confidentiality mechanisms. One of the goals on the DFD syntax was to avoid such specific model elements as explained in Section 1.4.1. Therefore, we formulate the following validation question:

VQ3) Are all elements of the DFD syntax commonly used when modeling systems containing access control and information flow control?

A utilization metric for every model element, which can be used, i.e. instantiated, can provide an answer to the validation question. We are, especially interested identifying model elements, which are only used by one confidentiality mechanism and not necessarily only by one system. It is likely that such elements are specific to a particular confidentiality mechanism. To represent our interested in the utilization metric, we define the metric as the usage of model elements per confidentiality mechanism. Assume a set F of model elements of the DFD syntax, which can be used, i.e. instantiated, and a set $S_f \subseteq S$, which contains all systems that use a particular model element $f \in F$. Assume further a set of systems $S_m \subseteq S$, which use a certain confidentiality mechanism $m \in M$. The function $u(f, m)$ returns 1 iff $S_f \cap S_m \neq \emptyset$ holds, 0 otherwise. This means $u(f, m)$ returns 1 if there is at least one system, which uses model element f and the confidentiality mechanism

m. The utilization metric u_f is then defined as $u_f = \sum_{m \in M} u(f, m)$. The validation passes if u_f is never less than 2 for all $f \in F$. This means, there are at least two systems using different confidentiality mechanisms but the same model element f. This brings us to the following metric.

VM3.1) Utilization metric u_f of model element f across confidentiality mechanisms.

The previous validation questions focused on validating that the model elements are sufficient and necessary. Another important aspect to consider when validating that the provided model elements can express systems is whether software architects and security experts can instantiate these model elements while modeling the software architecture of the system. In particular, the required information to instantiate the model elements has to be available and accessible to the software architects and security experts. Otherwise, the DFD syntax fails its pragmatics of describing a software architecture in order to analyze it while creating the software architecture. Therefore, we formulate the following validation question:

VQ4) Is the information to be expressed by the DFD syntax available and accessible to its users?

Assuming we have a set of information I, which is necessary to instantiate the model elements of the DFD syntax, we can define a set $I_k \subseteq I$ of information, which can be known. A reasonable metric is to sum up all information, which is not known. The resulting sum $s_{\bar{k}} = |I \setminus I_k|$ has to be 0. A value greater 0 means that the DFD syntax requires unavailable information. This would render the syntax unusable. This brings us to the following metric.

VM4.1) Sum $s_{\bar{k}}$ of information, which cannot be known by software architects and security experts.

8.1.2. Validation Goal 2: Validate Analysis Definitions

The validation goal about the analysis definitions for access control and information flow control is to show that the analysis definitions sufficiently answer the research questions RQ5 and RQ6. The research questions ask how to define analyses using the DFD syntax and semantics. We presented analysis definitions for common access control and information flow control mechanisms in Section 6.2 and also demonstrated how to integrate encryption in Section 6.3 as well as how to combine analyses in Section 6.4 to make these analysis definitions more applicable. However, we did not demonstrate that these analysis definitions are applicable to particular systems including their confidentiality requirements. Here, *applicable* means that the analysis definitions provide the means to express confidentiality requirements and that the resulting analyses provide correct results. To do this validation, we formulate the validation question VQ5, which we describe in the following, and reuse the validation question VQ6, which we will describe as part of VG3.

VQ5) Do the analysis definitions provide the means to express confidentiality requirements based on access control and information flow control mechanisms within systems?

The focus on the application within systems is important because software architects are usually interested in identifying violations in particular systems under design. We already motivated that the analysis definitions can identify violations in Section 6.2 in general but we did not show yet that the analysis definitions fit the needs of particular systems, i.e. that we can express the confidentiality requirements of the system by using the analysis definitions. In contrast to VQ1, VQ5 focuses on expressing confidentiality requirements by analysis definitions and not on expressing confidentiality mechanisms within systems. Nevertheless, we can reuse the ideas for finding an appropriate metric from VQ1 to answer VQ5: To answer the validation question, we need to know how many confidentiality requirements of systems can be expressed using the analysis definitions. The ratio metric $\tilde{r}$ already defined for VQ1 is a good metric to summarize this data in order to answer the validation question. For calculating the metric, we define a system to be part of the set of expressible systems S_e if we could express the confidentiality requirements of the system. The remainder of the metric definition remains the same. The metric normalize the effect of a set of systems, which is unbalanced with respect to the used confidentiality mechanism. Without this normalization, a big amount of systems using the same mechanism and therefore similar confidentiality requirements could make the metric look more positive if only a single system is using a particular mechanism and therefore specific confidentiality requirements. To decide whether we sufficiently addressed RQ5 and RQ6 individually as well as combinations of both mechanisms, we define the following three dedicated metrics:

VM5.1) Weighted ratio $\tilde{r}$ of systems, for which the analysis definitions can express access control requirements, and the set of systems using access control.

VM5.2) Weighted ratio $\tilde{r}$ of systems, for which the analysis definitions can express information flow control requirements, and the set of systems using information flow control.

VM5.3) Weighted ratio $\tilde{r}$ of systems, for which the analysis definitions can express combined access control and information flow control requirements, and the set of systems using combined mechanisms.

An analysis definition is only useful if it can be used to identify violations. Therefore, we have to validate that analyses, which are expressed in terms of the analysis definition, can identify systems, which violate the confidentiality requirements. VQ6, which we define as part of VG3, essentially validates this, i.e. that analyses can identify systems containing violations. Therefore, we use the results of that validation question to decide whether the analysis definitions can identify violations. With respect to the defined metrics *true positive fraction* (VM6.1) and *true negative fraction* (VM6.2), we aim for a value of 1.0, which means no false negatives or false positives. This is possible because the analyses can provide exact results and should do so. An explanation of why exact results are possible, is part of the description of the validation question for VG3.

8.1.3. Validation Goal 3: Validate DFD Semantics

The validation goal about the DFD semantics is to show that the semantics sufficiently answer the research question RQ4. RQ4 asks what DFD semantics allow detecting violations of confidentiality requirements. In Section 5.2, we presented our semantics and motivated why they support analyses. However, we did not show yet that the semantics support automated analyses of confidentiality mechanisms used in systems. Therefore, we have to validate that the semantics support analyses of systems, which yield correct results, and that the semantics do not limit the automation of analyses. To do this validation, we formulate the validation questions VQ6 and VQ7, which we describe in the next paragraphs.

VQ6) Can analyses based on the DFD semantics correctly identify systems containing violations?

As motivated before, we focus on detecting violations in the context of particular systems because software architects use the semantics for this purpose. The only purpose of the DFD semantics is to enable analyses for identifying such violations. Therefore, it is reasonable to focus on validating the support for particular analyses. We consider an analysis to be supported if it provides correct results. A result is correct if it correctly classifies a system as containing or not containing violations and it classifies it for the right reason. An analysis classifies a system for the right reason if i) no violations are reported for a system, which does not contain an issue, and if ii) all violations reported for a system, which does contain an issue, are caused by the issue and at least one issue is reported. Metz [Met78] suggests various metrics to rate the quality of analyses with only two possible outcomes, i.e. binary classifiers. The suggested metrics can handle unbalanced data sets such as a data set with many systems without an issue and only few systems with issue. A metric, which solely focuses on the right output of the analysis, would rate an analysis, which never reports a violation, good in such a setting. Although, the analysis is not useful at all. One suggested combination of metrics, which addresses this issue, is the true positive fraction (also called sensitivity) and the true negative fraction (also called specificity). We define both metrics in the following. Assume there exists a set S_i of systems, which contain an issue, and a set $S_{\bar{i}}$ of systems, which do not contain an issue. The set $S'_i \subseteq S_i$ describes the set of systems, which the analysis correctly classified as containing violations according to the definition given above. The set $S'_{\bar{i}} \subseteq S_{\bar{i}}$ describes the set of systems, which the analysis correctly classified as not containing violations according to the definition given above. The true positive fraction TPF is defined as $TPF = |S'_i|/|S_i|$. The true negative fraction TNF is defined as $TNF = |S'_{\bar{i}}|/|S_{\bar{i}}|$. The DFD analyses are meant to yield exact solutions, which is possible because DFDs are simple descriptions of software systems that reduce the complexity of the analysis by focusing on the most important aspects. The reduced complexity with respect to implemented software systems is no disadvantage but it is necessary to keep the complexity of modeling and the required information for modeling low enough to be applicable in the early phases of creating software architectures. Because of that reduced complexity, it is possible to clearly classify a result, i.e. add the system either to S'_i or $S'_{\bar{i}}$. As a consequence, we can expect and demand

$TPF = TNF = 1.0$ because there are no heuristics involved. A single false positive or false negative is already not acceptable. Both metrics provide the required insight to answer VQ6, so we use these:

VM6.1) The ratio of systems, which have been correctly classified as violating confidentiality requirements, compared to the total amount of systems, which actually violate confidentiality requirements (TPF).

VM6.2) The ratio of systems, which have been correctly classified as not violating confidentiality requirements, compared to the total amount of systems, which actually do not violate confidentiality requirements (TNF).

One key benefit of specified semantics is giving a precise meaning to model elements of the DFD syntax. A precise meaning is an enabler for automating reasoning steps, which would require heuristics or human interpretation otherwise. The precise meaning has to cover all model elements and has to be applicable in all reasoning steps in order to create a fully automated analysis. Therefore, it is important to validate that the semantics do not limit the automation of analyses by missing semantic specifications, weak typing of information or by not covering important usage scenarios. We formulate the following validation question:

VQ7) Do the DFD semantics limit the automation of analyses?

To answer the validation question, we have to identify the steps required to automate an analysis of a system. Assuming we have a set A of required analysis steps, we can define a set $A_a \subseteq A$ of analysis steps, which can be automated. The semantics do not limit the automation of analyses if there are no analysis steps, which are not automated, i.e. $A = A_a$. A metric to represent this relation is the number of not automated steps $\bar{a} = |A \setminus A_a|$. If the value is 0, the semantics do not limit the automation of analyses because there are no steps, which are not automated. Therefore, we answer VQ7 by the following metric:

VM7.1) The number of analysis steps, which cannot be automated ($\bar{a}$).

8.1.4. Validation Goal 4: Validate ADL Integration Guidelines

The validation goal about the integration guidelines of DFD analyses in existing ADLs is to show that the guidelines sufficiently answer the research questions RQ7 and RQ8. RQ7 asks how the DFD analyses can be integrated into existing ADLs using the control flow paradigm. RQ8 asks the same for ADLs using the data flow paradigm. The integration guidelines described in Chapter 7 answer these questions and the integration into the Palladio ADL demonstrated the applicability of the integration guidelines. A dedicated validation of the integration guidelines is not possible in an objective way because there are too many human factors involved in executing the integration procedure resulting from the integration guidelines. Instead, we validate the result of the integration procedure, i.e. the extended ADLs, because we can analyze these artifacts in an objective way and can infer whether the integration guidelines cover all important aspects. We did not show yet that the resulting extended ADLs meet the functional and non-functional requirements, which we aim to achieve by executing the integration procedure as described in Section 1.4.3. The implicit functional requirements, which are essential to provide a useful ADL, are (FR1) expressiveness with respect to confidentiality mechanisms and confidentiality requirements, (FR2) correctness of analysis results and (FR3) automation of analyses. The mentioned non-functional requirements are (NFR1) reduced effort for integrating confidentiality mechanisms into existing models, (NFR2) reduced effort for switching confidentiality mechanisms and (NFR3) usage of architecture level information only. To achieve the validation goal, we define a validation question for every functional or non-functional requirement, which we describe in the following paragraphs. To decide whether we sufficiently validated the contributions for RQ7 and RQ8 individually, we always define dedicated metrics for the ADL integration for ADLs using control flows and ADLs using data flows.

Validating that the extended ADLs provide good expressiveness with respect to confidentiality mechanisms and requirements (FR1) is important because low expressiveness means that architects cannot integrate the confidentiality mechanism, which fits best the needs of the system, but have to choose the mechanism, which they can express. Limited expressiveness with respect to confidentiality requirements implies the same disadvantage. Because the analyses for violations operate on DFDs, the expressiveness of DFDs implies an upper bound for the expressiveness of the extended ADLs. Therefore, it is reasonable to validate that the expressiveness of the extended ADL is not worse than the expressiveness of the DFDs. We formulate the following validation question:

VQ8) Is an extended ADL less expressive than an extended DFD with respect to confidentiality mechanisms and requirements?

To answer VQ8, we have to compare the expressiveness of extended ADLs with the expressiveness of DFDs. This covers the expressiveness of confidentiality mechanisms within systems and the expressiveness regarding confidentiality requirements. We can use the weighted ratio metric $\tilde{r}$, which we already used to answer the related validation questions VQ1 and VQ2 regarding the expressiveness of confidentiality mechanisms within DFDs. The metrics VM8.1–VM8.6 must have the same values as the corresponding metrics VM1.1, VM1.2 and VM2.1 because this means that the extended ADLs imply no limited expressiveness compared to the DFDs. We do not have to explicitly validate the expressiveness regarding confidentiality requirements because the requirements are still formulated in terms of the analysis definitions, which means that the expressiveness does not change compared to the already validated expressiveness in VQ5. This brings us to the following metrics:

VM8.1) Weighted ratio $\tilde{r}$ of expressible systems using access control and the set of systems using access control if a control flow ADL is used.

VM8.2) Weighted ratio $\tilde{r}$ of expressible systems using information flow control and the set of systems using information flow control if a control flow ADL is used.

VM8.3) Weighted ratio $\tilde{r}$ of expressible systems using any combination of access control and information flow control mechanisms and the set of systems using such a combination if a control flow ADL is used.

VM8.4) Weighted ratio $\tilde{r}$ of expressible systems using access control and the set of systems using access control if a data flow ADL is used.

VM8.5) Weighted ratio $\tilde{r}$ of expressible systems using information flow control and the set of systems using information flow control if a data flow ADL is used.

VM8.6) Weighted ratio $\tilde{r}$ of expressible systems using any combination of access control and information flow control mechanisms and the set of systems using such a combination if a data flow ADL is used.

The correctness of analysis results in the extended ADLs (FR2) is crucial because software architects cannot rely on the results, otherwise. Unreliable results imply additional effort, which degrades the benefits of using an automated analysis. Because analyses operate on DFDs, the upper bound regarding the correctness is the correctness of corresponding analysis results on a DFD. Therefore, it is reasonable to validate that the correctness of analysis results for the extended ADL is not worse than the correctness of analysis results for DFDs. We formulate the following validation question:

VQ9) Is the correctness of analysis results for an extended ADL worse than for DFD-based analyses?

To answer VQ9, we have to compare the correctness of analysis results based on the extended ADLs with analysis results based on DFDs. To rate the correctness of DFD-based analysis results, we use the true positive fraction TPF and true negative fraction TNF as described for the metrics VM6.1 and VM6.2. We can also use these metrics for rating the correctness of the ADL-based analysis results, which simplifies comparing the results with the DFD results. This is reasonable because the DFD-based analyses set the upper bound for the correctness of analysis results and the goal of VQ9 is to ensure that the correctness is not worse compared to the DFD-based analyses. This brings us to the following metrics:

VM9.1) The ratio (TPF) of systems, which have been correctly classified as violating confidentiality requirements, compared to the total amount of systems, which actually violate confidentiality requirements, if a control flow ADL is used.

VM9.2) The ratio (TNF) of systems, which have been correctly classified as not violating confidentiality requirements, compared to the total amount of systems, which actually do not violate confidentiality requirements, if a control flow ADL is used.

VM9.3) The ratio (TPF) of systems, which have been correctly classified as violating confidentiality requirements, compared to the total amount of systems, which actually violate confidentiality requirements, if a data flow ADL is used.

VM9.4) The ratio (TNF) of systems, which have been correctly classified as not violating confidentiality requirements, compared to the total amount of systems, which actually do not violate confidentiality requirements, if a data flow ADL is used.

Automating analyses can lead to lower manual effort, less human errors and reproducible results. Therefore, we aim for supporting automated analyses in the extended ADLs (FR3). The actual analysis operates on DFDs but the analysis of a software architecture given in an extended ADL requires additional steps. Therefore, the degree of automation can be limited compared to DFDs. We have to validate that the automation is not worse than for DFDs. We formulate the following validation question:

VQ10) Is the degree of automation for ADL-based analyses lower compared to DFD-based analyses?

We can answer VQ10 if we know the steps, which are not automated in analyses of software architectures specified in the extended ADL, and if we know how these steps relate to the analysis steps of DFD-based analyses. Assume we have a set A of analysis steps required to analyze DFDs and a set $A_a \subseteq A$ of automated analysis steps for DFDs. Analogously, we define a set A' of analysis steps required to analyze architectures given in an extended ADL and a set $A'_a \subseteq A'$ of automated analysis steps for ADLs. Then, we define a set P of purposes, which an analysis step serves. Based on that, we define a mapping $m : A \cup A' \mapsto P$, which maps every analysis step to a purpose. The automation of ADL-based analyses is worse than the automation of DFD-based analyses if the ADL-based analyses do not automate a purpose, which the DFD-based analyses automate. Counting automated steps or automated purposes and comparing the numbers between DFD and ADL analyses cannot answer VQ10 because this procedure neglects that new steps or purposes cannot be simply compared. In contrast, identifying previously automated purposes that are now no longer automated provides a clear measure for detecting degraded automation. Therefore, the number of no longer automated purposes $p_{\bar{a}}$ with $p_{\bar{a}} = | \cup_{a' \in A' \setminus A'_a} m(a') \cap \cup_{a \in A_a} m(a)|$ is a good metric for detecting degraded automation. The value has to be 0 to show that the extended ADL does not impose limited automation compared to DFDs. This brings us to the following metrics:

VM10.1) Number of no longer automated purposes $p_{\bar{a}}$ in control flow ADLs.

VM10.2) Number of no longer automated purposes $p_{\bar{a}}$ in data flow ADLs.

A non-functional requirement for the extended ADL is that introducing a confidentiality mechanism requires less effort compared to the state of the art (NFR1). The scenario to consider is that a software architect created a software architecture in an ADL. After extending the ADL, an architect can add a confidentiality mechanism to the already modeled architecture. It is important to validate this aspect because it provides a major benefit compared to the situation that an existing ADL, which does not support confidentiality yet, has been used to model a software architecture and there is no extended ADL. In such cases, the software architecture has to be modeled from scratch or a transformation into the analysis model is required. Creating such a transformation is part of our integration guidelines, so the validation does not demonstrate a benefit to this approach. Nevertheless, the validation can show a benefit compared to remodeling a software architecture from scratch, which can be reasonable if a software architect has not enough expertise to create such a transformation or a tool developer having these competences is not available. We formulate the following validation question:

VQ11) Does an extended ADL require reduced modeling effort for adding confidentiality mechanisms compared to the state of the art?

As described while motivating VQ11, we are interested in comparing the modeling effort to the situation, in which a model has to be created from scratch. The experienced effort highly depends on many human factors such as the amount of expertise. Therefore, measuring effort in an objective way is hard. However, it is reasonable to assume that the amount of required changes in a model correlates to the required effort for creating the model. Heinrich et al. [Hei+18] demonstrate that it is reasonable to estimate effort by first collecting changes in a model, asking experts to estimate the effort for so-called atomic change operations in the model and eventually derive the total effort. It is important to note that the effort implied by two arbitrary sets of model changes M and N is not comparable without the assignment of effort to atomic change operations: Even if set M only contains one model change and set N several hundred model changes, it is still possible that the change from M requires more effort than all other changes from N. However, it is possible to compare the effort of a set of model changes M if either $N = \emptyset$ or $M \subseteq N$ holds under the assumption that a model change always requires either no or more than no effort. The baseline from the state of the art is that the software architect has to recreate the whole model from scratch. For this baseline, $M \subseteq N$ holds because creating a model from scratch implies a set of model changes N, which contains all necessary model changes. Consequently, applying the model changes M requires at most as much effort as applying the model changes N but it is more likely that it requires less effort. We need a metric that detects the similarity of two models and that can indicate if there are model parts, which have been reused. Reused parts imply no model changes. Changed parts imply model changes. The coefficient of Jaccard [LW71] has already been used in other publications [Hei20; Mon+21] to rate the similarity of models. To calculate the coefficient, we interpret a model as a set of model elements. We assume that two models U and V exist and that equal model elements are identical with respect to set algebra, i.e. $u \equiv v \implies u = v \;\; \forall u \in U \land \forall v \in V$. In that case, the coefficient $j = \frac{|U \cap V|}{|U \cup V|}$ calculates the ratio between equal model elements and the combination of model elements from both models U and V. A value greater than 0 means that there are model elements, which are equal. In our case, this means that there were model elements, which could be reused while adding confidentiality mechanisms to the software architectures. The validation successfully showed an improvement with respect to the state of the art, if all calculated coefficients of Jaccard are greater than 0. This brings us to the following metrics:

VM11.1) The coefficient of Jaccard j for the software architecture without confidentiality mechanisms and the software architecture with confidentiality mechanisms, both specified in a control flow ADL.

VM11.2) The coefficient of Jaccard j for the software architecture without confidentiality mechanisms and the software architecture with confidentiality mechanisms, both specified in a data flow ADL.

Low effort for switching a confidentiality mechanism (NFR2) is important in case of evolutionary changes of the software architecture. If the requirements on the confidentiality mechanism change, the software architect should not hesitate to switch to a new mechanism in order to identify how well the new mechanism meets the new requirements. If the effort for switching the confidentiality mechanism is high, an architect might decide to work around the issue, which might lead to bad design decisions. In the state of the art, modeling languages often do not support multiple confidentiality mechanisms, which implies that software architects either have to remodel the whole software architecture in another modeling language or have to create mappings between the two modeling languages. In our approach, creating the mapping is not necessary because an extended ADL already supports multiple confidentiality mechanisms. Therefore, our integration approach implies a benefit compared to the mapping approach by design. Nevertheless, we have to validate that the extended ADL reduces the effort compared to recreating a model of a software architecture. Therefore, we formulate the following validation question:

VQ12) Does an extended ADL require less modeling effort for switching confidentiality mechanisms compared to the state of the art?

As already discussed for VQ11, measuring effort in an objective way is hard because of many human factors affecting the experienced effort. Instead, we measure the amount of reused elements when switching a confidentiality mechanism in an existing model of a software architecture. If at least one element is reused, the measurement indicates an improvement compared to the state of the art, in which models usually have to be recreated from scratch. We use the coefficient of Jaccard as introduced for VM11.1 and VM11.2 because we have to measure the same type of information and the coefficient provides the required insights into the degree of reuse. Again, a metric value greater than 0 implies a successful validation. This brings us to the following metrics:

VM12.1) The coefficient of Jaccard j for two software architectures representing the same system but containing different confidentiality mechanisms, both specified in a control flow ADL.

VM12.2) The coefficient of Jaccard j for two software architectures representing the same system but containing different confidentiality mechanisms, both specified in a data flow ADL.

As already discussed for VQ4, it is important to ensure that software architects and security experts have access to the information required to model the software architecture (NFR3). In case of the extended ADL, this means that all required information has to be available while creating the software architecture. Without this information, modeling a software architecture by using the extended ADL is not possible and the modeling language fails its pragmatics of describing software architectures. Therefore, we formulate the following validation question:

VQ13) Is the information to be expressed by an architecture using an extended ADL available and accessible to its users?

The metrics to answer VQ13 are the same as for answering VQ4 because we need the same type of information in order to answer the question. We use the sum of unknown information $s_{\bar{k}}$ as defined for VM4.1. The validation succeeds if the value is 0, which means that all required information is available to the users of the extended ADL. This brings us to the following metrics:

VM13.1) Sum $s_{\bar{k}}$ of information required to use an extended control flow ADL, which cannot be known by software architects and security experts.

VM13.2) Sum $s_{\bar{k}}$ of information required to use an extended data flow ADL, which cannot be known by software architects and security experts.

8.2. Case Study Systems

Many validation metrics and validation questions presented in Section 8.1 aim for validating contributions in the context of particular systems. Case studies are an appropriate way of collecting such information because they aim for getting insights into particular cases, which usually implies particular systems. According to various surveys [Ngu+15; Ber+17], case studies are commonly used to validate model-based security approaches, which explicitly includes approaches for establishing confidentiality. We briefly explain the rationale of using case studies for answering the validation questions when describing the validation design within Sections 8.3 to 8.6.

The foundation of case studies are appropriate systems to apply an approach to. We discuss the requirements, which make a system an appropriate system for our validation, in Section 8.2.1. Afterwards, we introduce the systems in Section 8.2.2 and report on how they match the previously defined requirements.

8.2.1. Requirements on Case Study Systems

There are three types of requirements regarding the case study systems. The Overall Requirements (ORs) define requirements on the set of case study systems such as how many systems of a certain type are required in order to calculate the validation metrics and answer the validation questions, which we defined in Section 8.1. We describe the Overall Requirements in Section 8.2.1.1.

The descriptions of the systems also have to meet Description Requirements (DRs) such as that a system has to use a confidentiality mechanism. We cannot use system descriptions, which do not meet these requirements, in a validation because they do not provide all information to calculate the validation metrics. We describe the Description Requirements in Section 8.2.1.2.

The last group are Source Requirements (SRs) and focus on the source of the system and the system description. These are no must-requirements but meeting them reduces the threats to validity. We describe the Source Requirements in Section 8.2.1.3.

8.2.1.1. Overall Requirements (ORs)

The overall requirements are requirements on the set of systems and not on the systems themselves. We derive these requirements from the validation metrics.

The set of systems has to contain systems using different, commonly used confidentiality mechanisms. This is important to rate the expressiveness regarding different access control mechanisms (VM1.1) and information flow control mechanisms (VM1.2) within systems as well as for rating the expressiveness regarding confidentiality requirements in the context of different access control mechanisms (VM5.1) and information flow control mechanisms (VM5.2). The corresponding requirement is as follows:

OR1) The set of case study systems shall contain at least one system for each commonly used confidentiality mechanism.

Besides having systems for every commonly used confidentiality mechanism, the set of systems also has to contain at least one system, which uses a mix of at least one access control and one information flow control mechanism. This is necessary to validate the expressiveness of the syntax (VM2.1) and analyses (VM5.3) regarding such combinations. The corresponding requirement is as follows:

OR2) The set of case study systems shall contain at least one system using a combination of an access control and an information flow control mechanism.

To validate the correctness of analysis results, we have to be able to classify a system as having or not having an issue, which leads to a violation of confidentiality requirements. The metrics for capturing the true positive fraction (VM6.1) and true negative fraction (VM6.2) need this classification as a foundation to classify the analysis results. To ensure that the analyses do not yield correct results by accident, e.g. because there are only systems without issues and the analysis always reports no violation without even analyzing a system, it is reasonable to require a variant with issue and a variant without issue for every system. The corresponding requirement is as follows:

OR3) The set of case study systems shall contain a variant with an issue and a variant without an issue for every system.

To validate the modeling effort for switching the confidentiality mechanism (VM12.1 and VM12.2), it is necessary to have at least two variants of the same system, where the variants use different confidentiality mechanisms. Only variants of the same system allow to calculate the amount of required changes that a software architect would have to do. The corresponding requirement is as follows:

OR4) The set of case study systems shall contain two variants of the same system, which use different confidentiality mechanisms.

8.2.1.2. Description Requirements (DRs)

The description requirements define requirements for the description of systems. The descriptions exceed the pure description of usual architectural information about systems but includes requirements on the type of system or includes information to build variants of systems.

First of all, we aim for systems, which describe a solution to a problem in a certain application domain. This explicitly excludes unit test models or toy examples. Without a certain degree of complexity, the validations regarding expressiveness (VQ1, VQ2, VQ5 and VQ8) is unlikely to reveal an issue. If the system describes a potentially working system in an application domain, the complexity is at least as high as required in that domain. Therefore, such systems are more representative than small, tailored systems. This brings us to the first requirement:

DR1) The system description shall describe a solution for a problem in a certain problem domain.

The system description itself has to cover various aspects. First of all, the commonly used architectural information including the structure, behavior, deployment and usage of the system has to be given. Otherwise, modeling the system is not possible with our approach or with any other modeling approach for software architectures. In addition to the software architecture, information about confidentiality is necessary. Especially, the usage of the included confidentiality mechanism has to be available. Otherwise, we cannot integrate the confidentiality mechanism, which is necessary to validate the expressiveness (VQ1, VQ2, VQ5 and VQ8). The confidentiality requirements also have to be given in terms of the used confidentiality mechanism. Otherwise, it is not possible to decide if the system violates any confidentiality requirements, which makes validations using the analysis results (VQ6 and VQ9) impossible. This brings us to the following requirements:

DR2) The system description shall describe the common architectural information including structure, behavior, deployment and usage.

DR3) The system description shall describe how the system integrates the confidentiality mechanism.

DR4) The system description shall describe the confidentiality requirements in terms of the confidentiality mechanism.

The more features of a confidentiality mechanism a system uses in its confidentiality requirements, the more likely it is to identify limitations regarding the expressiveness of these requirements (VQ5). Therefore, the requirements should use as many features as possible. This brings us to the following requirement:

DR5) The system description should use as much features of the confidentiality mechanism as possible to formulate the confidentiality requirements.

In order to build the variants of the systems with and without an issue, the system description has to include information about a potential issue and the resulting violations. If this information is missing, only one variant of the system is available. The variant either contains an issue or does not contain an issue. Without both variants, we cannot eliminate the chance that the correctness of the analysis results is rated to positively: The results of an analysis that never finds a violation, e.g. because it reports this result independently of the system to be analyzed, would be classified correct if the analysis is only validated with systems without issues. This would be a threat to the validity, especially for the validations regarding the correctness of analysis results (VQ6 and VQ9). This brings us to the following requirement:

DR6) The system description shall describe an issue, which can be added to the system, as well as resulting violations.

8.2.1.3. Source Requirements (SRs)

The source requirements specify quality criteria for the origin of a case study system. These requirements are not mandatory but meeting them increases the validity of the results.

A system, which originates from a third party, is less likely to be tailored to the approach, which shall be validated, because the system has not been designed with the limitations or capabilities of the approach in mind. Therefore, it is beneficial to use systems published by other authors. The requirement is not mandatory because it is always possible to create a system by ourselves. In that case, we have to discuss potential threats to validity arising from creating the system by ourselves. If possible, it is beneficial to use systems, which also have been implemented, because the corresponding software architectures and confidentiality requirements are realistic. This means that such systems represent systems, which software architects actually create. This brings us to the following requirements:

SR1) The system should have been defined by a third party, i.e. another author.

SR2) The system should also have been implemented.

Because system descriptions, which include confidentiality requirements, are often discussed in publications about security research, it is likely that these descriptions also include a discussion of an existing or potential issue. This is beneficial because creating a variant with and without an issue is much simpler if a realistic issue is already part of the description. This requirement is not mandatory because it is usually possible to invent and introduce an issue based on the confidentiality requirements. To decide if violations, which our analysis reports, are correct, it is beneficial to have information about the violations, which the issue implies. Obviously, this information is only available if the issue is also already available. This requirement is not mandatory because the effect of an introduced issue can usually be estimated. This brings us to the following requirements:

SR3) The system description should contain an issue, which leads to violations of confidentiality requirements.

SR4) The system description should contain violations of confidentiality requirements, which an issue in the system implies.

8.2.2. Selected Case Study Systems

The previously introduced requirements on the case study systems and the requirements on the set of selected systems provide guidelines on how to select case study systems that support our validations. Source requirements are the most challenging to meet because they solely depend on the availability of information from other publications. Therefore, we define the selection procedure around the source requirements: First, we identify closely related approaches and extract used case study systems from corresponding publications. Focusing on closely related approaches is beneficial because there is a high chance that the resulting systems are given as software architectures or high-level software designs. We can use such systems without changing the level of abstraction, which has the potential of introducing errors or simplifying the system too much. After we included all systems of closely related approaches, we look for missed overall requirements on the set of selected systems. For all missed requirements, we define case study systems on our own and add them to the selected case study systems. Defining systems on our own is reasonable because it is more important to meet the overall requirements than the source requirements. Missed overall requirements mean that we cannot do a part of our validation. Missed source requirements only introduce potential threats to validity, which we have to consider.

The result of the selection procedure is a set of seventeen case study systems as illustrated in Table 8.2. The approaches iFlow [Kat17] and SecDFD [TSB19] provide case study systems (CS1–CS9), which use information flow control to protect the confidentiality of data. The formulated confidentiality requirements are given in terms of non-interference with and without encryption. Most of the corresponding lattices are linear but there is also one case study system with an arbitrary lattice and encryption with key pairs as well as one, which defines non-interference between tenants of a system. Because this set of case study systems does not meet OR1, which requires at least one system per commonly used confidentiality mechanism, we have to create and add at least five systems for the commonly used access control mechanisms DAC, MAC (military and need-to-know model), ABAC and RBAC. The systems CS10–CS16 fill this gap. We defined multiple systems using RBAC (CS10–CS12) to meet OR4, which requires at least two variants of the same system using different confidentiality mechanisms. The resulting case study systems use the same system as the case study systems CS1–CS3. This means, we got three pairs of case study systems, which use the same system. To meet the last requirement about a case study system using a combination of at least two confidentiality mechanisms (OR2), we added CS17, which combines RBAC and a taint analysis, which is a simple form of non-interference. We meet the last remaining requirement about having a variant with and a variant without an issue for every case study system (OR3) either i) by using the variant containing an issue from the original description if the source meets SR3, ii) by defining a variant based on information from the original description if the source only

ID	System	Conf. Mech./Req.	SR1	SR2	SR3	SR4
CS1	TravelPlanner	Non-Int. Linear	●	●	◐	◐
CS2	DistanceTracker	Non-Int. Linear	●	○	◐	◐
CS3	ContactSMS	Non-Int. Linear	●	●	◐	◐
CS4	PrivateTaxi	Non-Int. Arb. Enc.	●	●	◐	◐
CS5	BankingApp	Non-Int. Tenant	●	●	●	●
CS6	FriendMap	Non-Int. Linear Enc.	●	●	●	●
CS7	Hospital	Non-Int. Linear Enc.	●	●	●	●
CS8	JPMail	Non-Int. Linear Enc.	●	●	◐	◐
CS9	WebRTC	Non-Int. Linear Enc.	●	●	◐	◐
CS10	TravelPlanner	RBAC	◐	◐	◐	◐
CS11	DistanceTracker	RBAC	◐	◐	◐	◐
CS12	ContactSMS	RBAC	◐	◐	◐	◐
CS13	ImageSharing	DAC	○	○	○	○
CS14	FlightControl	MAC Military Model	○	○	○	○
CS15	HealthRecord	MAC Need-to-Know	◐	○	○	○
CS16	BankBranches	ABAC	○	○	○	○
CS17	TravelPlanner	RBAC + Tainting	◐	◐	◐	◐

Table 8.2.: Overview on selected case study systems including the confidentiality mechanism used to formulate requirements as well as report on not meeting (○), partially meeting (◐) and fully meeting (●) the Source Requirements (SRs).

partially meets SR3 or iii) by defining such a variant on our own if the case study system does not originate from a third party, i.e. if it does not meet SR1.

The descriptions of the case study systems meet all Description Requirements (DRs), which we explain in this paragraph as well as in the detailed descriptions of the case study systems, which follow this paragraph. The systems are no toy examples (DR1) and provide all required information to describe their software architecture (DR2). The systems stemming at least partially from third parties (SR1) meet these requirements because they already have been used to validate closely related approaches and have been implemented often (SR2). For the remaining systems CS13–CS16, we demonstrate that these requirements are met by the following descriptions of the systems. The requirements about describing the integration of the confidentiality mechanism (DR3) and the description of confidentiality requirements (DR4) are also met for case study systems originating from third parties because these publications motivate the systems based on interesting confidentiality requirements and the detection of circumvented confidentiality mechanisms. For the systems created by ourselves (CS13–CS16) as well as for the system using combined confidentiality mechanisms (CS17), we discuss the usage of confidentiality mechanisms and the particular confidentiality requirements in the created descriptions. The requirement about describing a potential issue and the corresponding violations (DR6) is often not trivially met by the third party systems, as the considerable amount of only partially met source requirements SR3 and SR4 shows. Nevertheless, we can derive potential issues and violations from the existing descriptions of the third party systems because all descriptions cover how they protect confidential information. Consequently, a potential issue is circumventing or disabling these protections. For the systems defined by ourselves, we clearly describe issues and potential violations later. We discuss the usage of features of the confidentiality mechanisms within the confidentiality requirements (DR5) in the following descriptions.

8.2.2.1. CS1 TravelPlanner (Non-Interference Linear)

System Source. The system has been published as part of the iFlow project [Kat+13] and a PhD thesis [Kat17]. According to the PhD thesis, the system has been implemented[1].

System Description. The travel planner is a system consisting of three actors: A user wants to book a flight using his/her travel planner app as well as the credit card center app. A travel agency receives queries for flights from the travel planner app and returns matching flights. An airline provides information about flights to the travel agency and processes bookings for a given flight and credit card data.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference. Transmitted information has a classification level. Processing actors have a clearance level. The levels are *UserAirlineTravelAgency*, *UserAirline* and *User*.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirements: The credit card data is classified by the *User* level but has to be transmitted to the Airline, which is only cleared for *UserAirline*. To allow transmission, a declassification operation asks the user for permission and reclassifies the credit card data. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the airline accesses credit card data, to which it should not have access to.

[1] `https://web.archive.org/web/20220103091007/https://kiv.isse.de/projects/iflow/TravelPlannerSite/travelplanner.zip`

8.2.2.2. CS2 DistanceTracker (Non-Interference Linear)

System Source. The system has been published as part of the iFlow project [Ste+16] and a PhD thesis [Kat17]. According to the PhD thesis, the system has not been implemented.

System Description. The distance tracker is a system consisting of three actors: A user wants to track the distance, which he/she has run, and shares his/her location to enable the calculation of the distance. A distance tracker records these periodically transmitted locations and calculates a run distance based on these records. A tracking service records the run distance.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference. Transmitted information has a classification level. Processing actors have a clearance level. The levels are *OnlyDistance*, *User,DistanceTracker* and *User*.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirements: The location data is classified by the *User* level but has to be transmitted to the distance tracker, which is only cleared for *User,DistanceTracker*. To allow transmission, a declassification operation asks the user for permission and reclassifies the location data. The resulting location data is classified by *User,DistanceTracker*. The tracking service, which shall record the run distance, is cleared for *OnlyDistance*. To avoid a violation of the non-interference requirements, the distance tracker declassifies the locations by calculating the distance and reclassifying it by *OnlyDistance*. A potential issue is that one of these declassifications is not done, e.g. because a software architect forgot that this is necessary. The expected violation is either that the distance tracker accesses the location data, to which it should not have access to in case of circumventing the first declassification, or that the tracking service accesses the distance data, to which it should not have access to in case of circumventing the second declassification.

8.2.2.3. CS3 ContactSMS (Non-Interference Linear)

System Source. The system has been published as part of a PhD thesis [Kat17]. The PhD thesis indicates that the system has at least partially been implemented.

System Description. The contact SMS system consists of two actors: A user wants to manage contacts and send a SMS to the contacts. A SMS manager receives the number of a contact as well as the message and sends the SMS to the receiver.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference. Transmitted information has a classification level. Processing actors have a clearance level. The levels are *User,Receiver* and *User*.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirements: The contact data is classified by the *User* level but the SMS manager needs the phone number from the contact data, which is only cleared for *User,Receiver*. To allow transmission, a declassification operation extracts the number from the contact data and reclassifies the number by the *User,Receiver* level. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the SMS manager accesses contact data, to which it should not have access to.

8.2.2.4. CS4 PrivateTaxi (Non-Interference Arbitrary with Encryption)

System Source. The system has been published as part of a PhD thesis [Kat17]. The PhD thesis indicates that the system has at least partially been implemented.

System Description. The purpose of the private taxi system is to provide ride sharing. The private taxi system consists of four actors: Drivers publish their route and accept riders for ride sharing. Riders publish their route and select a driver for ride sharing. The private taxi system receives the routes, receives the distance between two routes from the distance calculation service, matches riders to drivers and mediates the communication between drivers and riders. The distance calculation service receives two routes and calculates the distance between them.

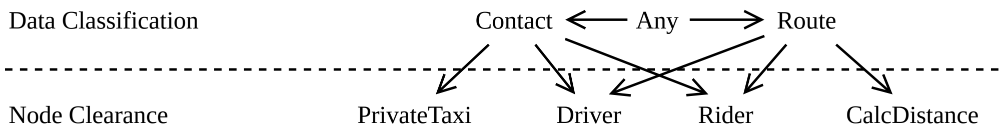


Figure 8.3.: Lattice used in the PrivateTaxi case study system.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference as well as encryption with key pairs. Transmitted information has a classification level. The classification levels are *Any*, *Contact* and *Route*. Processing actors have a clearance level. The clearance levels are *Driver*, *Rider*, *PrivateTaxi* and *CalcDistance*. Data might be encrypted for a set of public keys, which reduces the classification of data to *Any*. After decryption, the classification of data is the same as the classification level before the encryption.

Confidentiality Requirements. The confidentiality requirement is that a node with a clearance level a must only receive data with a classification level b if $a \geq b$. A level a is greater or equal to a level b if b is transitively connected to a in the lattice shown in Figure 8.3. Informally speaking, the lattice does not allow the distance calculation service to access contact data and does not allow the private taxi service to access routes. The requirement uses an arbitrary lattice and all features that information flow control enforcing non-interference with an arbitrary lattice provides.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirements: The route data is classified by the *Route* level and the private taxi service receives routes for forwarding it to the distance calculation service. However, the private taxi service is not cleared for data classified by the *Route* level. To allow transmission, drivers and riders encrypt their routes with the public key of the distance calculation service, which reduces the classification of the route data to *Any*. This is essentially a declassification operation. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the private taxi service accesses route data, to which it should not have access to.

8.2.2.5. CS5 BankingApp (Non-Interference Tenant)

System Source. The system has been published as part of a PhD thesis [Kat17]. The PhD thesis indicates that the system has at least partially been implemented.

System Description. The system consists of two actors: A user wants to interaction with his/her bank account, i.e. depositing and withdrawing money as well as querying the account balance, by using a banking app. A bank provides the services to interact with the bank account.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference. Transmitted information is always associated with a certain user. If data of groups of users shall be isolated, these groups are often called tenants [Fac+13]. The confidentiality mechanism ensures non-interference between these tenants.

Confidentiality Requirements. The confidentiality requirement is that tenants must not access data of other tenants. Essentially, the system shall behave as if there was only one tenant using the system. This is a common formulation of non-interference requirements between tenants, so we consider the requirement to use the relevant features of such types of requirements.

Potential Issue. The original description of the system describes an issue: Because of a wrong software design, the authenticity of the transmitted user identifier is not validated when accessing the account balance, which allows all users to access the balance of other users by sending different user identifiers. This violates the confidentiality requirement that only tenants are allowed to access their data. The violation is that other users, who are not the owner of the data, access the balance.

8.2.2.6. CS6 FriendMap (Non-Interference Linear with Encryption)

System Source. The system has been used to validate the SecDFD approach [TSB19]. There is an implementation [Fab17] of the system.

System Description. The friend map system provides users of social networks with a map of the geographical locations of their friends. The social network, the friend map and the map provider work together to generate and visualize the map: The social network provides the locations of friends and displays the generated map of friends. The map provider takes the locations and renders the map. The friend map orchestrates and mediates the previously described two systems.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference as well as encryption without considering keys. Transmitted information has a classification level. The levels are *Low* and *High*. Processing actors have a clearance level. In the original publication, the clearance levels are *AttackZone* and *TrustZone* but to improve comprehensibility, we use the clearance level *Low* instead of *AttackZone* and *High* instead of *TrustZone* . This does not reduce the complexity of the system or mechanism. Data might be encrypted, which reduces the classification of data to *Low*. After decryption, the classification of data is the same as the classification level before the encryption.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The map provider is considered to be in the attack zone, i.e. is cleared for *Low*. The location of friends is considered confidential, i.e. is classified by *High*. To allow passing the locations to the map provider, the locations are encrypted, which classifies them by *Low*. In consequence, the locations can be transmitted to the map provider. The original description of the system describes an issue: Because of a wrong software design, the locations of the friends are no longer encrypted, which means that a node cleared for *Low* receives data classified by *High*, which violates the confidentiality requirement. The violation is that the map provider accesses location data.

8.2.2.7. CS7 Hospital (Non-Interference Linear with Encryption)

System Source. The system has been used to validate the SecDFD approach [TSB19]. There is an implementation [Fab16] of the system.

System Description. The hospital system provides employees with patient lists, which are stored in a database. An employee can read and modify the patient list using his/her hospital app. The changes a stored in a database. The system assumes that an attacker has access to a system part, which reads the patient list from the database and passes the list back to the hospital app.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference as well as encryption without considering keys. Transmitted information has a classification level. The levels are *Low* and *High*. Processing actors have a clearance level. In the original publication, the clearance levels are *AttackZone* and *TrustZone* but to improve comprehensibility, we use the clearance level *Low* instead of *AttackZone* and *High* instead of *TrustZone* . This does not reduce the complexity of the system or mechanism. Data might be encrypted, which reduces the classification of data to *Low*. After decryption, the classification of data is the same as the classification level before the encryption.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The attacker is considered to be in the attack zone, i.e. is cleared for *Low*. The patient list is considered confidential, i.e. is classified by *High*. To protect patient data, the database encrypts the patient list before sending it to the reading part of the system, to which an attacker has access. This reduces the classification level of the patient list to *Low*. Therefore, the attacker only has access to data, which is classified *Low*. The original description of the system describes an issue: Because of a wrong software design, the patient list is no longer encrypted, which means that a node cleared for *Low* receives data classified by *High*, which violates the confidentiality requirement. The violation is that the attacker accesses the patient list.

8.2.2.8. CS8 JPMail (Non-Interference Linear with Encryption)

System Source. The system has been used to validate the SecDFD approach [TSB19]. There exists an implementation[2] [HAM06] of the system.

System Description. JPMail is a mail system consisting of mail clients, a SMTP server and a POP3 servers. The user Alice encrypts her mail and passes it to the SMTP server. The SMTP server sends the mail to the POP3 server. The POP3 server delivers the mail to Bob, who decrypts the mail.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference as well as encryption without considering keys. Transmitted information has a classification level. The levels are *Low* and *High*. Processing actors have a clearance level. In the original publication, the clearance levels are *AttackZone* and *TrustZone* but to improve comprehensibility, we use the clearance level *Low* instead of *AttackZone* and *High* instead of *TrustZone* . This does not reduce the complexity of the system or mechanism. Data might be encrypted, which reduces the classification of data to *Low*. After decryption, the classification of data is the same as the classification level before the encryption.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The

[2] `https://web.archive.org/web/20130731052551/http://siis.cse.psu.edu/jpmail/downloads/jpmail-full-latest.tgz`

requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirement: The SMTP server and the POP3 server are cleared for *Low*. The content of the mail is classified *High*. Because the content of the mail is encrypted before sending the mail to the servers, the confidentiality requirement is not violated. This is essentially a declassification operation. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the SMTP server and the POP3 server access the mail content, to which they should not have access to.

8.2.2.9. CS9 WebRTC (Non-Interference Linear with Encryption)

System Source. The system has been used to validate the SecDFD approach [TSB19]. There exists an implementation [Moz20] of the system.

System Description. WebRTC is a protocol for real-time communication including media exchange between browsers. The WebRTC system described in the following is a simplified version. The system consists of the users Alice and Bob, their browsers, their identity providers, STUN/TURN servers and a signaling server. Alice initiates a session through three components (her browser, the signaling server and the browser of Bob) by sending identity information from the identity provider. The ports to communicate are negotiated via the STUN/TURN servers. The actual communication takes place via direct connections between the browsers.

Confidentiality Mechanism. The system uses information flow control to enforce non-interference as well as encryption without considering keys. Transmitted information has a classification level. The levels are *Low* and *High*. Processing actors have a clearance level. In the original publication, the clearance levels are *AttackZone* and *TrustZone* but to improve comprehensibility, we use the clearance level *Low* instead of *AttackZone* and *High* instead of *TrustZone* . This does not reduce the complexity of the system or mechanism. Data might be encrypted, which reduces the classification of data to *Low*. After decryption, the classification of data is the same as the classification level before the encryption.

Confidentiality Requirements. The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all features that information flow control enforcing non-interference with a linear lattice provides.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirement: The STUN/TURN servers and the signaling servers are external, potentially public systems. Therefore, these systems are cleared for *Low*. The identities, session data and communication information is classified *High*. Because the session data is encrypted before sending it to the signaling server, the confidentiality requirement is not violated. This is essentially a declassification operation. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the signaling server accesses the session data, to which it should not have access to.

8.2.2.10. CS10 TravelPlanner (RBAC)

System Source. The system has been published as part of the iFlow project [Kat+13] and a PhD thesis [Kat17]. According to the PhD thesis, the system has been implemented[3]. We adjusted the system to use RBAC instead of information flow control to ensure confidentiality and published this adjusted system [SHR19]. We explain the adjustments when describing the confidentiality mechanism and requirements.

System Description. The travel planner is a system consisting of three actors: A user wants to book a flight using his/her travel planner app as well as the credit card center app. A travel agency receives queries for flights from the travel planner app and returns matching flights. An airline provides information about flights to the travel agency and processes bookings for a given flight and credit card data.

Confidentiality Mechanism. The system uses RBAC. We introduce one role for each actor in the system, which yields the roles *TravelAgency*, *Airline* and *User*. The actors get assigned their corresponding roles. The exchanged data has a set of roles attached, which describes the roles allowed to access the data. We map the classification levels to sets of roles: The *User* classification becomes the set consisting of the *User* role. The *UserAirline* classification becomes the set consisting of the *User* and *Airline* role. The *UserAirlineTravelAgency* classification becomes the set of all roles. The mapping closely matches the intention of the classification levels with respect to the actors that can access the information.

Confidentiality Requirements. The confidentiality requirement is that the role of the actor has to be in the set of roles, which have access to data, when the actor accesses data. This requirement only covers RBAC Core but not Hierarchical RBAC or Constraint RBAC. It would have been possible to introduce role hierarchies or constraints in an artificial way but the original system described in Section 8.2.2.1 does not contain any information to

[3] `https://web.archive.org/web/20220103091007/https://kiv.isse.de/projects/iflow/TravelPlannerSite/travelplanner.zip`

derive such hierarchies or constraints. To avoid creating an artificial system, we stick to RBAC Core.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirement: The credit card data is only accessible to the *User* role but has to be transmitted to the Airline, which has the *Airline* role. To allow transmission, a declassification operation asks the user for permission and explicitly adds the *Airline* role to the set of roles, which have access to the credit card data. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the airline accesses credit card data, to which it should not have access to.

8.2.2.11. CS11 DistanceTracker (RBAC)

System Source. The system has been published as part of the iFlow project [Ste+16] and a PhD thesis [Kat17]. According to the PhD thesis, the system has not been implemented. We adjusted the system to use RBAC instead of information flow control to ensure confidentiality and published this adjusted system [SHR19]. We explain the adjustments when describing the confidentiality mechanism and requirements.

System Description. The distance tracker is a system consisting of three actors: A user wants to track the distance, which he/she has run, and shares his/her location to enable the calculation of the distance. A distance tracker records these periodically transmitted locations and calculates a run distance based on these records. A tracking service records the run distance.

Confidentiality Mechanism. The system uses RBAC. We introduce one role for each actor in the system, which yields the roles *TrackingService*, *DistanceTracker* and *User*. The actors get assigned their corresponding roles. The exchanged data has a set of roles attached, which describes the roles allowed to access the data. We map the classification levels to sets of roles: The *User* classification becomes the set consisting of the *User* role. The *User,DistanceTracker* classification becomes the set consisting of the *User* and the *DistanceTracker* roles. The *OnlyDistance* classification becomes the set of all roles. The mapping closely matches the intention of the classification levels with respect to the actors that can access the information.

Confidentiality Requirements. The confidentiality requirement is that the role of the actor has to be in the set of roles, which have access to data, when the actor accesses data. This requirement only covers RBAC Core but not Hierarchical RBAC or Constraint RBAC. It would have been possible to introduce role hierarchies or constraints in an artificial way but the original system described in Section 8.2.2.1 does not contain any information to derive such hierarchies or constraints. To avoid creating an artificial system, we stick to RBAC Core.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirement: The location data is only accessible to the *User* role but has to be transmitted to the distance tracker, which has the role *DistanceTracker*. To allow transmission, a declassification operation asks the user for permission and explicitly adds the *DistanceTracker* role to the location data. The tracking service, which shall record the run distance has the role *TrackingService*. To avoid a violation of the RBAC requirement, the distance tracker declassifies the locations by calculating the distance and adding the role *TrackingService*. A potential issue is that one of these declassifications is not done, e.g. because a software architect forgot that this is necessary. The expected violation is either that the distance tracker accesses the location data, to which it should not have access to in case of circumventing the first declassification, or that the tracking service accesses the distance data, to which it should not have access to in case of circumventing the second declassification.

8.2.2.12. CS12 ContactSMS (RBAC)

System Source. The system has been published as part of a PhD thesis [Kat17]. The PhD thesis indicates that the system has at least partially been implemented. We adjusted the system to use RBAC instead of information flow control to ensure confidentiality and published this adjusted system [SHR19]. We explain the adjustments when describing the confidentiality mechanism and requirements.

System Description. The contact SMS system consists of two actors: A user wants to manage contacts and send a SMS to the contacts. A SMS manager receives the number of a contact as well as the message and sends the SMS to the receiver.

Confidentiality Mechanism. The system uses RBAC. We introduce one role for each actor in the system, which yields the roles *Receiver* and *User*. The actors get assigned their corresponding roles. The exchanged data has a set of roles attached, which describes the roles allowed to access the data. We map the classification levels to sets of roles: The *User* classification becomes the set consisting of the *User* role. The *User,Receiver* classification becomes the set of all roles. The mapping closely matches the intention of the classification levels with respect to the actors that can access the information.

Confidentiality Requirements. The confidentiality requirement is that the role of the actor has to be in the set of roles, which have access to data, when the actor accesses data. This requirement only covers RBAC Core but not Hierarchical RBAC or Constraint RBAC. It would have been possible to introduce role hierarchies or constraints in an artificial way but the original system described in Section 8.2.2.1 does not contain any information to derive such hierarchies or constraints. To avoid creating an artificial system, we stick to RBAC Core.

Potential Issue. The original description of the system does not describe an issue but describes the critical part for not violating the confidentiality requirement: The contact data is only accessible to the *User* role but the SMS manager, which needs the phone number from the contact data, only has the *Receiver* role. To allow transmission, a declassification operation extracts the number from the contact data and explicitly adds the *Receiver* role to the set of accessible roles of the number data. A potential issue is that this declassification is not done, e.g. because a software architect forgot that this is necessary. The expected violation is that the SMS manager accesses contact data, to which it should not have access to.

8.2.2.13. CS13 ImageSharing (DAC)

System Source. We created the system on our own because the related approaches do not provide a system using DAC. We derive the domain and features of the system from a common use case for DAC, which is access control in filesystems and filesharing applications [Fur08, pp. 61]. We did not implement the system but published it [Sei+22].

System Description. The system supports sharing images between users. The users of the system can read images from a store and write images into a store. The involved users are a mother, a dad, an aunt and an indexing bot of a search engine. A visualization of the system is available in Figure B.1 on page 263.

Confidentiality Mechanism. The system uses DAC with delegation of rights. Each user has an identity, which is either *Mother*, *Dad*, *Aunt* or *Indexing Bot*. The data store holds a list of identities, which are allowed to read images, a list of identities, which are allowed to write images, and a list of identities, which are owners of the data store. Owners can add identities to any of the three lists.

Confidentiality Requirements. The confidentiality requirements are that i) the identity of an actor has to be in the list of allowed readers if the actor wants to read images and that ii) the identity of an actor has to be in the list of allowed writers if the actor wants to add images. Initially, only the mother is owner of the data store. The mother and the dad are in the list of allowed readers. The mother is in the list of allowed writers. The requirements use all features of DAC with delegation of rights except for the revocation of access rights. We already recognized this limitation while describing the DAC analyses in Section 6.2.2.1 and discussed this limitation in Section 6.7. Integrating this feature in the requirements would only demonstrate an already known limitation, so it would not provide further insights as part of the validation. Nevertheless, we will mention this limitation when discussing the validation results.

Potential Issue. There are multiple potential issues. Because we focus on confidentiality, we only focus on violations caused by illegal reading of images. We see two types of issues: The first type of issue is caused by a missing static assignment of an identity to a list at the data store. The second type of issue is caused by missing dynamic assignment of an identity to a list at the data store by an owner. Because the second type of issue is more complex, we choose this type of issue. The introduced issue is that the mother does not add reading rights for the aunt anymore. The expected violation is that the aunt accesses images, although she should not have access to the images.

8.2.2.14. CS14 FlightControl (MAC Military Model)

System Source. We created the system on our own because the related approaches do not provide a system using MAC with the military security model. We derive the domain and features of the system from a common use case for the military security model, which is a military information system [FGL93]. We decided to model a system for flight control, because such systems process information with varying levels of confidentiality and for various purposes. We did not implement the system but published it [Sei+22].

System Description. The system supports the flight monitoring and control of civil as well as military planes. The system has three users: A clerk collects information about the weather, creates weather reports and stores them in a database. A civil flight controller registers planes, looks them up and determines routes for the planes. He/she considers the weather reports to reduce the risk of directing a plane into dangerous airspaces. The civil planes are stored in a database. A military flight controller has the same tasks as the civil flight controller but he/she determines routes for military planes. He/she considers the weather reports as well as the positions of the civil planes in a new route. A visualization of the system is available in Figure B.2 on page 264.

Confidentiality Mechanism. The system uses MAC with the military security model. There are three levels: *Unclassified*, *Classified* and *Secret*. Weather data is classified *Unclassified* and the clerk is cleared for data classified as *Unclassified*. Data about civil planes is classified *Classified* and the civil flight controller is cleared for data classified as *Classified*. Data about military planes is classified *Secret* and the military flight controller is cleared for data classified as *Secret*.

Confidentiality Requirements. The military security model in MAC is comparable to non-interference using a linear lattice, so the confidentiality requirement is the same: The confidentiality requirement is that no node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The order of the lattice is the same as the order of the levels mentioned in the description of the confidentiality mechanism. The requirement uses all confidentiality features of MAC using the military security model.

Potential Issue. To violate the confidentiality requirement, a user must get access to information, which is classified higher than his/her own clearance. A potential issue can be that the civil flight controller also uses the positions of military planes to determine routes for civil planes. The violation is that the civil flight controller gets to know information about military planes, which are classified higher than his/her own clearance. This issue is realistic because it is reasonable to assume that the calculation of routes requires information about all planes in the airspace.

8.2.2.15. CS15 HealthRecord (MAC Need-to-Know)

System Source. We derived the system from an existing publication [AF08], which uses a mix of access control mechanisms. We extract the part, which is about MAC using the Need-to-Know model. We could not find an implementation of the system.

System Description. The system is a simple electronic health record application. The system has three types of users: A physician creates a diagnosis based on the medical record of the patient and prescribes treatments. He/she also creates a list of treatments for a clerk. The clerk uses the list of treatments, treatment prices and the contact information of the patient to create an invoice. The patient receives the invoice after he/she has provided the patient history and the contact information.

Confidentiality Mechanism. The system uses MAC with the Need-to-Know model. Medical records are assigned the *Medical* compartment. Personal information about the patient are assigned the *Personal* compartment. Financial information is assigned the *Financial* compartment. A physician has a need to know for *Medical* information. A clerk has a need to know for *Personal* and *Financial* information. A patient has a need to know for all information.

Confidentiality Requirements. The confidentiality requirement is that a user of the system must only get access to data, which has a set of compartments, which is a subset of the needs to know of the user. Because this is the only confidentiality requirement specified for the Need-to-Know model, the requirement uses all features of that particular access control model.

Potential Issue. To violate the confidentiality requirement, a user must get access to information, which has at least one compartment that is not part of the needs to know of a user. A simple way to violate this requirement in the present system is that the physician, which prepares the list of treatments for the clerk, does no longer remove medical details from these treatments. This way, the declassification effect of the list creation gets lost, which means that the treatment list still has the *Medical* compartment. Because the clerk does not have a need to know for *Medical* information, this violates the confidentiality requirement. The expected violation is that the clerk accesses treatments including medical details, to which he/she should not have access to. This issue is realistic because it is reasonable to assume that the need to declassify this information is overseen by the architect.

8.2.2.16. CS16 BankBranches (ABAC)

System Source. We created the system on our own because the related approaches do not provide a system using ABAC. It would have been possible to derive a system from the previously described systems using access control because ABAC can represent DAC, MAC and RBAC [JKS12]. However, we would essentially just copy the previous systems, which impedes getting new insights during the validation. Therefore, we define a new system on our own. We did not implement the system but published it [Sei+22].

System Description. The system is a banking system for an international bank. Clerks can register customers and determine a credit line for them. Managers can use the same features but can use them for regular customers as well as celebrities. There are two branches of the bank: One branch serves the United States of America (USA) and one branch serves Asia. Managers can move customers between the branches. A clerk is always assigned to a certain branch office. A visualization of the system is available in Figure B.3 on page 265.

Confidentiality Mechanism. The system uses ABAC. The relevant attributes of subjects are the *Role*, which can be either a clerk or a manager, and the *Location*, which can be either USA or Asia. The relevant attributes of objects are the *Origin*, which can be either USA or Asia, and the *CustomerStatus*, which can be either a regular customer or a celebrity customer. Moving customer data to a new location changes the *Origin* attribute. When combining two or more data items, the resulting data item receives the union of all attributes of the incoming data items.

Confidentiality Requirements. There are two subject descriptors and two object descriptors: The *Clerk* descriptor matches subjects, which have the *Role* attribute set to a clerk. The *Manager* descriptor matches subjects, which have the *Manager* attribute set to a manager. The *Regular* descriptor matches objects, which have the *CustomerStatus* set to a regular customer. The *All* descriptor matches all objects regardless of particular attribute values. The confidentiality requirement is that a subject must only access objects if there exists an authorization for this access. The authorizations are that i) a *Manager* is allowed to access *All* objects and that ii) a *Clerk* is allowed to access *Regular* objects if the *Location* of the subject is the same as the *Origin* of the object. This requirement and the authorizations use all features but the hierarchical descriptors offered by ABAC. We will discuss this limitation as part of the discussion of the threat to validity of the validation.

Potential Issue. There are many options to introduce an issue. The issue we introduce is that a manager does not use the system feature for registering celebrity customers but uses the feature for registering regular customers. This overrides the separation between regular and celebrity customers. The expected violation is that clerks access information about celebrities, which they are not allowed to. The issue is realistic for two reasons: First, it is possible that managers use the wrong system feature by mistake. A software architect has to be aware of such a potential problem and has to create specifications for avoiding such mistakes (e.g. by using different designs or credentials for different features). Second, it is possible that the software architect would like to save implementation effort by not using two dedicated system parts for celebrities and regular customers without considering the impact on confidentiality.

8.2.2.17. CS17 TravelPlanner (RBAC + Tainting)

System Source. We could not find a system combining access control and information flow control in related approaches. Therefore, we defined a case base on CS10 (travel planner using RBAC) and integrated a taint analysis, which can be seen as a non-interference using a simple linear lattice. The combination of access control and taint analysis has already been done and also has been shown to be beneficial in the literature [Wan+09; ZG02]. We did not implement the system.

System Description. The travel planner is a system consisting of three actors: A user wants to book a flight using his/her travel planner app as well as the credit card center app. A travel agency receives queries for flights from the travel planner app and returns matching flights. An airline provides information about flights to the travel agency and processes bookings for a given flight and credit card data.

Confidentiality Mechanism. The system uses a combination of RBAC and taint analysis. The handling of roles has already been described in CS10. To extend the described mechanism by taint analysis, we do the following: We introduce a validation property on data. Initially data created by a user is *NotValidated*. Data created by the system is always considered *Validated*. After a node in a system has validated data, it is *Validated*. Speaking in terms of a taint analysis, the status *NotValidated* is equivalent to be tainted and the status *Validated* is equivalent to not being tainted. Nodes in the system have a criticality level, which can be *Low*, *DMZ* and *High*.

Confidentiality Requirements. The confidentiality requirements specified for CS10 still hold but in order to allow access of a node to data, an additional confidentiality requirement always has to hold: No node with a clearance level a receives data with a classification level b such that $a < b$. The order relation $<$ is given by the linear ordered lattice, where a level with a lower index is considered lower than a level with a higher index. The ordered lattice in ascending order is *Validated*, *High*, *NotValidated*, *Low* and *DMZ*. Simply said, data, which has not been validated, must be validated before it might be processed by nodes with high criticality. This requirement is comparable to the requirements formulated in the literature [Wan+09; ZG02]. The requirements use all features of the taint analysis and RBAC Core. As discussed for CS10, we only consider RBAC Core to stick closer to the original travel planner system described in CS1.

Potential Issue. We could use the same issue as used in CS10 but this would not provide more insights in the validation compared to only using CS10. Instead, we define an issue, which can only be shown in the current system. A realistic issue, which could appear because a software architect forgot to specify a validation for incoming data, is that the criteria for looking for flights has not been validated before its processing within the travel planner app. The expected violation is that the travel planner app processes the not validated criteria data, which it must not process.

8.3. Validation of Extended Data Flow Diagrams

The validation of the extended DFD syntax, which we described in Section 5.1, is the first validation goal (VG1). We describe the validation design for answering the validation questions VQ1–VQ4 in Section 8.3.1. The results of executing the designed validation are presented in Section 8.3.2 and discussed in Section 8.3.3. We discuss threats to the validity of our results and conclusions in Section 8.3.4.

8.3.1. Validation Design

The validation design describes the procedure to provide answers to all four validation questions, which support VG1. The first three validation questions VQ1, VQ2 and VQ3 ask about the expressiveness of the DFD syntax and require particular systems for collecting the data to answer the questions. A case study is appropriate to provide this information because it aims for gaining insights into the application of the approach in particular cases, which implies particular systems. We conduct the case study in six steps:

1) For each case study system mentioned in Section 8.2, we try to model the system and its usage of the confidentiality mechanism in the DFD syntax.

2) For each case study system, we classify the modeling result either as successfully modeled or as not successfully modeled. The successfully modeled case study systems are part of the set of expressible systems S_e. A system is classified as successfully modeled if the structure and deployment could be represented and if every behavior and usage, which affects the confidentiality, could be represented.
3) We group the tuples of case study system and its classification by the confidentiality mechanism mentioned in Table 8.2.
4) We calculate the weighted ratio metrics VM1.1, VM1.2 and VM2.1.
5) For each case study system, we collect the set of used model elements of the DFD syntax. We consider a model element as used if it has been instantiated or if any of its subclasses has been instantiated.
6) We calculate the utilization metric VM3.1.

We interpret the metric values as described in Section 8.1.1 and answer the validation questions VQ1, VQ2 and VQ3.

The last validation question VQ4 asks whether all information to create a system model is available to the users of the DFD syntax. To answer the question, we discuss the required information to instantiate every element of the DFD syntax and build groups of information. For instance, one group could be the structural information about the architecture. This brings us to the set I of necessary information. Afterwards, we discuss whether this information is available in the required granularity when creating the software architecture. We base our decision about availability by looking at the information, which other, established ADLs require, and by collecting commonly required information to use typical architectural viewpoints. This brings us to the set I_k of necessary information. After this discussion, we can classify each information as either known or unknown. The sum of unknown information is the required metric VM4.1. We adhere to the interpretation guidelines mentioned in Section 8.1.1 for answering the validation question based on the metric. A discussion is a reasonable method for collecting the data to calculate the metric because there is no formal definition of *information*, which is usable for answering the validation question. A discussion can cover many different influencing factors and provides valuable results as long as the line of argumentation is clear.

8.3.2. Validation Results

We could successfully express all but one case study system as illustrated in Table 8.3. The table presents the ratio metric $r(m)$ for a mechanism m. The top part covers the systems using access control, so it provides the data for calculating VM1.1. Because there are five access control mechanisms, $\tilde{r} = \frac{5}{5} = 1.0$. The middle part covers the systems using information flow control, so it provides the data for calculating VM1.2. Because there are four information flow control mechanisms, $\tilde{r} = \frac{3}{4} = 0.75$. The bottom part covers the systems using a combination of access control and information flow control, so it provides

Confidentiality Mechanism m	$\|S\|$	$\|S_e\|$	$r(m)$
DAC	1	1	1
MAC Military Model	1	1	1
MAC Need-to-Know	1	1	1
RBAC	3	3	1
ABAC	1	1	1
Non-Interference Linear	3	3	1
Non-Interference Linear with Encryption	4	4	1
Non-Interference Arbitrary with Encryption	1	1	1
Non-Interference Tenant	1	0	0
RBAC + Tainting	1	1	1

Table 8.3.: Overview on ratio $r(m)$ of expressible systems $|S_e|$ to total amount of systems $|S|$ per used confidentiality mechanism m.

the data for calculating VM2.1. Because there is only one combination of mechanisms, $\tilde{r} = \frac{1}{1} = 1.0$.

We calculated the utilization metrics u_f for all model elements $f \in F$. For the most model elements, u_f was 9, where 9 is the maximum possible value because there are nine different expressible confidentiality mechanisms. The model element True has a slightly lower utilization of 8 and Or a value of 7. The three model elements ActorProcess, And and False have a utilization of 6. The ContainerCharacteristicReference has a utilization of 3. The only model element below the threshold of 2 is the model element Not with a utilization of 0.

In order to calculate VM4.1, we collected the required information to use the model elements. Table 8.4 lists all model elements, the action to do with the model elements, the corresponding user and the category of information, which the user needs to know to perform the action. The triples of category, action and user build the elements of the set of information I. We can show for every triple $i \in I$ that the required information is available: Creating the structure of an architecture and defining its usage is covered by common architectural viewpoints. According to Rozanski and Woods [RW05, p. 36], the structure is covered by the functional viewpoint and the usage has to be covered by the information viewpoint in order to describe the manipulation of data. Because data flows represent the structure in terms of the wiring of components as well as the usage in terms of used components by users, they fall in both categories. Creating model elements, which describe the behavior, is the responsibility of the security expert. According to an explanation of the threat modeling process [Tor05], security experts have to collect various information during threat modeling, which maps to the information required to specify the BehaviorDefinition: The information about entry points as well as inputs and outputs matches the information required to create a Pin. The information on the behavior with respect to inputs and outputs matches the information required to create an Assignment including all remaining model elements used to create expressions for the assignments. The information to bind the BehaviorDefinition to the architecture is also known to the architect because he/she has to know the data processing of components as defined by

Category	Model Element	Action	User
Structure	Process	Create	Architect
	Store	Create	Architect
Usage	ExternalActor	Create	Architect
	ActorProcess	Create	Architect
Structure/Usage	DataFlow	Create	Architect
Behavior	BehaviorDefinition	Create	Sec. Exp.
		Bind	Architect
	Pin	Create	Sec. Exp.
	Assignment	Create	Sec. Exp.
	And	Create	Sec. Exp.
	Or	Create	Sec. Exp.
	Not	Create	Sec. Exp.
	True	Create	Sec. Exp.
	False	Create	Sec. Exp.
	ContainerCharacteristicReference	Create	Sec. Exp.
	DataCharacteristicReference	Create	Sec. Exp.
Properties	Enumeration	Create	Sec. Exp.
	Literal	Create	Sec. Exp.
	CharacteristicType	Create	Sec. Exp.
	EnumCharacteristic	Create	Sec. Exp.
		Bind	Architect

Table 8.4.: Overview on categories of information required to use model elements.

the information viewpoint [RW05, p. 36]. The security expert knows the information required to create properties because he/she also needs this information during threat modeling. According to [Tor05], a security expert has to be aware of the assets to be protected, deployment information as well as about trust levels of entry points. If the security expert knows the assets to protect, he/she also knows the properties of the assets, which allows to define CharacteristicTypes, Enumerations and Literals. If the security experts knows the trust levels of entry points as well as deployment information, he/she knows at last some properties of the components, which allows to define EnumCharacteristics. It is reasonable to assume that the security expert also knows remaining important properties if they are important for security analyses. In order to bind these properties to components, the software architect has to know if the component actually has these properties. The software architect certainly can bind the deployment information and all properties related to the deployment because this information is part of the deployment viewpoint [RW05, p. 36]. For the remaining properties, it is either reasonable to assume that the security expert provides guidelines on how to interpret the properties or that the security expert is still available to provide guidance on applying the properties. In both cases, the software architect has the information to bind the properties.

We can now calculate VM4.1. The set of information I consists of every triple of category, action and user from Table 8.4. As explained before, all information to create the corresponding model elements is available to the software architect and the security expert. Therefore, $I = I_k$ holds. Consequently, $s_{\bar{k}} = |I \setminus I_k| = |\emptyset| = 0$ holds.

8.3.3. Result Discussion

We structure the discussion of the results by the corresponding validation questions.

Discussion of VQ1. The validation question aims to validate that the expressiveness of the DFD syntax is not worse than the expressiveness of the state-of-the-art approaches. As shown by the value 1.0 of VM1.1, the DFD syntax supports all commonly used access control mechanisms. This means, we could not show worse expressiveness of access control mechanisms with respect to the state of the art. However, VM1.2 only has a value of 0.75, which means that the DFD syntax could not express all information flow control mechanisms. The syntax could successfully represent all case study systems using non-interference with linear or arbitrary lattices. Before discussing the case study system, which the DFD syntax could not express, we give more details on why the syntax could express the other systems.

For all case study systems, the characteristic types, characteristics and behaviors provided in Section 6.2 and Section 6.3 were sufficient to model the systems. There are only two differences to the provided behaviors: variants of the behaviors and an additional synchronizing behavior. The variants affect the joining and forwarding behavior and support more inputs and outputs. Creating these variants is simple because the behaviors work essentially the same as the ones described by us: The forwarding behaviors still forward all labels from an input to an output but there are now multiple pairs of input and output. The joining behaviors still combine labels in the way described by us (such as building an intersection or finding the highest label) but the expressions consider more inputs. The synchronizing behavior is a forwarding behavior, which takes an additional input. However, the additional input is not considered when determining the labels of the output. Therefore, the synchronizing behavior just documents an additional input, which a process receives. The synchronizing behavior could also be replaced by the forwarding behavior from an analysis point of view.

As shown by Table 8.3, the not-supported confidentiality mechanism is non-interference between tenants. Tenants are groups of legitimate users of the same system functions [Fac+13]. This means that these individual tenants are represented by the same type of user. The DFD syntax describes systems on such a type level, which means that it cannot express individual users, i.e. tenants. Consequently, integrating a confidentiality mechanism, which can distinguish between individual users, is not possible. The case study system originates from the iFlow approach [Kat17, pp. 187], which can make use of source code stubs for more detailed analyses. An approach to be used in the architectural design phase cannot make use of such detailed implementation information because it is not available at that time. The part of iFlow [Kat17, pp. 81], which operates on models, shares such a limitation and defines one tenant as the whole group of legitimate users and another tenant as an external, illegitimate user. In consequence, the violations, which can be detected, are limited compared to the analysis on source code. The DFD syntax could also express such an attacking user. However, this representation of the system would be incomplete with respect to the original definition of the confidentiality requirements – just like the model part of the iFlow approach. Therefore, we cannot conclude that the expressiveness of the DFD syntax is worse than the expressiveness of state-of-the-art approaches as long as only the architecture- and design-time models are considered.

Discussion of VQ2. The validation question aims to validate that the DFD syntax can represent combinations of access control and information flow control mechanisms within the same system. As the value 1.0 of VM2.1 shows, the syntax could successfully represent all case study systems using combinations of both types of confidentiality mechanisms. The characteristic types, characteristics and behaviors provided by the descriptions of the mechanisms in Section 6.2 were sufficient to express the systems. The only exception are joining and forwarding behaviors with more inputs and outputs than we originally described. However, this does not indicate a problem of the descriptions because extending the behaviors by more inputs and outputs is simple as discussed for the previous validation question. A combination of access control and information flow control within the same system is an improvement compared to the state of the art.

Discussion of VQ3. The validation question aims to validate that the DFD syntax does not contain highly specific elements for certain confidentiality mechanisms. The validation revealed that only the `Not` element is used in less than two systems. In fact, the `Not` element has never been used in any case study system. This means that the element could be removed without affecting the expressiveness of the DFD syntax with respect to the case study systems. However, the element has been added to gain functional completeness [End01, p. 49], i.e. all truth tables, which can be constructed based on boolean parameters, can be expressed. The DFD syntax intentionally does not only provide a minimal set of boolean connectives (negation and either logical disjunction or logical conjunction would be sufficient) but all commonly used three logical connectives to simplify specifications. To improve VM3.1, we could remove either the logical conjunction or the logical disjunction and transform every expression into the logical equivalent. Because `Not` has not been added to support a specific confidentiality mechanism and it can be replaced by other existing model elements, we can answer VQ3 by stating that there are no model elements, which are tailored to specific confidentiality mechanisms.

Discussion of VQ4. The validation question aims to validate that users of the DFD syntax have access to the information required to use it. As discussed in Section 8.3.2, the value of VM4.1 is 0, which means that all required information is available to the software architect and the security expert while creating the architecture.

8.3.4. Threats to Validity

Because the majority of the validation took place as a case study, we structure the discussion of threats to validity according to the guidelines of Runeson and Höst [RH09, pp. 153] for discussing the validity of a case study.

Internal Validity is concerned with how well a taken measure supports a cause-effect relationship and especially whether there are alternative explanations for the effect. In the context of VG1, we expect the DFD syntax to be the cause of an effect. The effects are expressiveness, good metamodel utilization and availability of required information. We discuss potential alternative explanations of these effects, i.e. other possible influencing factors, in the following.

The *expressiveness* (VM1.1, VM1.2 and VM2.1) is certainly affected by the DFD syntax because it provides the means to express systems. We measure the amount of systems, which we could express in the DFD syntax, to determine expressiveness. Besides the DFD syntax, there are other potential influencing factors: The selection of case study systems influences the expressiveness measure. For instance, removing CS5 from the set of case study systems would influence the expressiveness metric positively. We cannot rule out this factor completely but we mitigated overly positive results by sticking to the system selection procedure described in Section 8.2.2. The set of case study systems covers all commonly used confidentiality mechanisms and avoids cherry-picking by always selecting all case study systems from closely related approaches. The aspects of confidentiality mechanisms, which the case study systems do not cover, focus either on the confidentiality requirements (hierarchies in RBAC and ABAC) or require stateful modeling (revocation of rights in DAC). The confidentiality requirements only affect VG2 and VG3. The stateful modeling is an intentional limitation of our approach, which we already discussed in Section 6.7. The skill of the software architect and security expert, which create the DFDs, is another influencing factor. Because we are interested in the upper bound of expressiveness, this factor is only important if the skill is too low and thereby lowers the measured upper bound. We can exclude this factor because the person, who created the DFDs, is the author of the DFD syntax, which makes low skill in using the syntax unlikely. Another influencing factor is the chosen abstraction of the system. By choosing a very high level of abstraction, a complex system can become expressible even if it misses important details. We mitigated this factor in two ways: First, we pick case study systems from closely related approaches, which use a comparable level of abstraction. Second, we reuse issues from existing case study systems or create issues based on critical aspects of the system. If an aspect of a system is important for reasoning about confidentiality, it is unlikely that we can successfully create a system variant containing an issue when omitting this aspect. For the systems, which we created on our own, we stuck to common application scenarios and issues.

The *metamodel utilization* (VM3.1) is certainly affected by the DFD syntax because it provides the elements of the metamodel to be used. For every element, we count the confidentiality mechanisms, for which at least one system exists, which uses the element.

We report elements, which are used at most once. Besides the DFD syntax, there are other potential influencing factors: The selection of case study systems directly influences the metamodel usage. For instance, a set of systems, which do not require a storage, makes the utilization of the Storage element 0. As explained for the expressiveness, we carefully selected the case study systems according to the selection procedure described in Section 8.2.2 to avoid biased selections or small toy examples, which would lead to an unrepresentative set of systems. Another influencing factor are the choices done while modeling the system. With respect to the structure of the system, we do not expect many choices to be available because of the small amount of model elements in the DFD syntax and missing alternatives. With respect to the behavior definitions, there are often multiple ways of formulating expressions. For instance, $\neg a \vee b$ is the same as $\neg(a \wedge \neg b)$. In the first case, negation and logical disjunction is used. In the second case, negation and logical conjunction is used. Therefore, the metamodel usage would be different. This means, the metric is susceptible to modeling choices. Therefore, it is not sufficient to answer the validation question solely by the metric value but to discuss, whether the model element is only not used because of design choices. We did this discussion for the only model element, which has been used never.

The *information availability* (VM4.1) depends on the DFD syntax because the syntax prescribes, which information shall be expressed. For every model element, we determine the information, which is required to create and use the model element and discuss the information availability. Besides the DFD syntax, there is another potential influencing factor: The classification of the availability of information directly influences the metric. If the classification is wrong, the metric is also wrong. To avoid a wrong classification, we discussed the availability of every information and justified the classification by literature.

External Validity is concerned with the generalization of case study results to other contexts. Because a case study does not use a representative sample but a limited set of cases, results cannot be generalized to arbitrary other contexts. However, the results can be generalized to other cases with comparable characteristics. We discuss these characteristics in the following.

The result of the *expressiveness* validation (VM1.1, VM1.2 and VM2.1) is that the DFD syntax can express systems using the mentioned confidentiality mechanisms. This finding can be generalized to systems using data flows and using one of the mentioned confidentiality mechanisms. The restriction on data flows is important because a DFD cannot represent pure control flows, which means that confidentiality mechanisms represented by the DFD syntax always work on exchanged data. A system, in which activities cannot be represented appropriately by data flows and data processing, cannot be represented in a DFD. This is a general limitation of DFDs. However, this limitation is not too restrictive because many different systems have been initially designed using DFDs.

The result of the *metamodel utilization* validation (VM3.1) is that there are no elements specifically tailored to a particular confidentiality mechanism in the DFD syntax. Because the validation already covered all model elements of the DFD syntax and we could show that every element is at least used by three types of confidentiality mechanisms or only has a low usage because of arbitrary design decisions while modeling, we could disprove the hypothesis of the existence of specifically tailored model elements. When executing the same validation with a different set of systems, the results could be different. Therefore, the results, i.e. the usage values, can only be generalized if the set of systems is comparable with respect to the amount of systems, system sizes and used confidentiality mechanisms. However, the finding, i.e. that there are no specifically tailored model elements, is generalizable without restrictions because giving examples is sufficient to disprove the mentioned hypothesis.

The result of the *information availability* validation (VM4.1) is that all information required to express a system in the DFD syntax is available to the software architect and the security expert. Because the corresponding validation is not based on a case study but a general discussion, the results are generalizable to any other system that is expressible.

Construct Validity is concerned with the appropriateness of taken measures to make statements about the research objective.

The weighted ratio metrics of the *expressiveness* validation (VM1.1, VM1.2 and VM2.1) measure the ratio between expressible systems and the total amount of systems normalized by the amount of systems using the same confidentiality mechanism. The statement to be made based on the metric value is that the DFD syntax can express the same amount of confidentiality mechanisms within systems as state-of-the-art approaches. The metric is appropriate to provide the information for the statement because a value smaller than 1 means that not all confidentiality mechanisms could be expressed within all systems, which is worse than state of the art. Apart from this threshold-based interpretation of the metric, the metric also gives an idea of how bad the expressiveness is, which is valuable for a discussion. It is reasonable to not use a simple ratio metric of expressible systems compared to the total amount of systems because the amount of systems using a certain confidentiality mechanism varies and we are especially interested in how well the DFD syntax can express confidentiality mechanisms within systems. Therefore, not supporting two systems with two different confidentiality mechanism is worse than not supporting two systems with the same confidentiality mechanism. The metric respects this and handles unbalanced sets of systems with respect to the confidentiality mechanism.

The utilization metric of the *metamodel utilization* validation (VM3.1) measures for every element the confidentiality mechanisms, for which at least one system exists, which uses the element. The statement to be made is that there are no elements only used by systems using one particular mechanism. The metric is appropriate to provide the information for this statement: The metric correctly handles unbalanced sets of systems with respect to the used confidentiality mechanism. For instance, if only two systems using the same confidentiality mechanism make use of a model element, its utilization metric value is still only 1. We cannot use a simple sum of systems using an element because a large amount of systems using the same confidentiality mechanism would yield a high usage value, which does no longer provide the information required to support our statement.

The unknown information metric of the *information availability* validation (VM4.1) measures the sum of required information, which cannot be known by software architects and security experts. The statement to be made is that all required information can be known by software architects and security experts. The metric provides the required information because it gives the sum of unknown information, which can be compared to the expected value 0.

Reliability is concerned with the dependency between the collected data and the conducting researcher. Best reliability is achieved if the collected data as well as the conclusions are completely independent of the conducting researcher.

The data collected in the *expressiveness* validation (VM1.1, VM1.2 and VM2.1) certainly depends on the skills of the modeler because two different modelers will most probably not create the very same model. However, this is not necessary because the validation is about finding an upper bound of expressiveness. Therefore, it is only necessary that other researchers can understand the decision of whether a system has been successfully expressed or not. We provide all created models as part of a data set [Sei22], so other researchers can decide whether the classification of a system as expressible was correct. The corresponding metrics can be calculated objectively, which we also did as part of the validation application in our data set [Sei22].

The data collected in the *metamodel utilization* validation (VM3.1) does not depend on a particular researcher because checking for the usage of a model element can be done objectively. We have fully automated this step as part of the validation application in our data set [Sei22].

The data collected in the *information availability* validation (VM4.1) depends on the discussion of the availability of information. As part of this discussion, we explain, why information is available and provide references to literature, which supports our statements. Other researchers can reproduce the results by consulting the references.

8.4. Validation of DFD Analyses

The validation of the DFD analyses, which we described in Section 6.2, is the second validation goal (VG2). We describe the validation design for answering the validation question VQ5 in Section 8.4.1. The results of executing the designed validation are presented in Section 8.4.2 and discussed in Section 8.4.3. We discuss threats to the validity of our results and conclusions in Section 8.4.4. Because VQ6 belongs to VG2 as well as to VG3 and uses the DFD analyses as well as the DFD semantics, we describe the aforementioned aspects for this validation question only in one place, which is in Section 8.5.

8.4.1. Validation Design

The validation design describes the procedure to provide answers to VQ5. The validation question asks about the expressiveness of the DFD analyses and requires defining the confidentiality requirements of particular systems. A case study is appropriate to provide this information because it aims for gaining insights into the application of the approach in particular cases, which implies particular systems.

In the case study, we reuse the systems, which have already been modeled to answer VQ1 and VQ2. We use these systems instead of all systems mentioned in Table 8.2 because to formulate confidentiality requirements for a system, the system has to use a confidentiality mechanism, which can actually meet these requirements. For instance, requiring non-interference for a system, which uses DAC or no confidentiality mechanism at all, is not reasonable because the system can only meet this requirement by accident and the system description does not provide the necessary information to reason about non-interference. In contrast, the systems modeled to answer VQ1 and VQ2 use the appropriate confidentiality mechanisms and provide the necessary information to reason about non-interference.

We conduct the case study as follows: For every system resulting from VQ1 and VQ2, we formulate the confidentiality requirements, which have been described in the case study system descriptions in Section 8.2.2, as a Prolog query. We use the queries from Section 6.2 and Section 6.3 and replace names of characteristic types and literals if the names do not already match. If expected by the analysis definitions, we create the additional Prolog clauses for covering all confidentiality requirements of the system. For CS17, which combines access control and information flow control, we apply the combination procedure described in Section 6.4. Afterwards, we classify the created confidentiality requirements in either successfully expressed or not expressed. We classify the confidentiality requirements as successfully expressed i) if the query is the same as the one presented in the corresponding analysis definition when ignoring differences in names and ii) if the additional clauses follow the structure described in the analysis definition. Afterwards, we calculate the metrics VM5.1, VM5.2 and VM5.3.

8.4.2. Validation Results

We could successfully represent all confidentiality requirements of the case study systems CS10–CS16, which covers requirements for systems using the confidentiality mechanisms DAC, MAC Military Model, MAC Need-to-Know, RBAC and ABAC. Therefore, the weighted ratio for VM5.1 is $\tilde{r} = \frac{5}{5} = 1.0$.

We could successfully represent all confidentiality requirements of the case study systems CS1–CS4 as well as CS6-CS9, which covers requirements for systems using the confidentiality mechanisms non-interference with linear lattice, arbitrary lattice including encryption, and linear lattice including encryption. As explained in the validation design, CS5 is not part of the considered case study systems for this validation. The weighted ratio for VM5.2 is $\tilde{r} = \frac{3}{3} = 1.0$.

We could successfully represent the confidentiality requirement of the case study system CS17, which covers the requirements for a system using a combination of RBAC and a taint analysis. Therefore, the weighted ratio for VM5.3 is $\tilde{r} = \frac{1}{1} = 1.0$.

8.4.3. Result Discussion

Validation question VQ5 aims to validate that the analysis definitions for DFDs are expressive enough to cover confidentiality requirements of particular systems. As the metric values for VM5.1, VM5.2 and VM5.3 show, the DFD analyses provide the necessary means to express the confidentiality requirements of the case study systems. The only necessary adjustment to be made to the Prolog queries was changing the names of characteristic types. However, this does not indicate a weakness of the analysis definitions. Security experts can always decide to use different names for characteristic types if they think, these names fit the particular system better. The analysis definitions provide the primitives to model systems and execute analyses. The names used in the analysis definitions do not influence the results of analyses and are, therefore, interchangeable.

8.4.4. Threats to Validity

Because the validation took place as a case study, we structure the discussion of threats to validity according to the guidelines of Runeson and Höst [RH09, pp. 153] for discussing the validity of a case study.

Internal Validity is concerned with how well a taken measure supports a cause-effect relationship and especially whether there are alternative explanations for the effect. In the context of VG2, we expect the analysis definitions for DFDs to be the cause of an effect. The expected effect of these analysis definitions is expressiveness with respect to confidentiality requirements of systems. We discuss potential alternative explanations of this effect, i.e. other possible influencing factors, in the following.

The *expressiveness* (VM5.1, VM5.2 and VM5.3) is certainly affected by the analysis definitions because they provide the primitives to formulate the requirements. We measure the amount of systems, for which we could express the confidentiality requirements using the analysis definitions, to determine expressiveness. Besides the analysis definitions, there are other potential influencing factors: The selection of case study systems also has an influence on the measured expressiveness because every system having confidentiality requirements, which cannot be expressed, lowers the measured expressiveness. As already discussed in Section 8.3.4, we cannot completely eliminate this influencing factor but minimize its effect by a careful selection procedure for case study systems, which covers systems using the most common information flow control and access control mechanisms as well as corresponding confidentiality requirements. In addition, the systems have a considerable size and describe solutions for common problems in various application domains. Therefore, we consider the results to be representative. However, we did not cover role hierarchies in RBAC and selector hierarchies in ABAC. Therefore, we cannot draw conclusions for these particular features. Excluding CS5 is a logical consequence of a lack of expressiveness regarding this system, which we already showed and discussed in Section 8.3: If we cannot express a system, we cannot express confidentiality requirements for that system. Another potential influencing factor is the expertise of interpreting confidentiality requirements and mapping them to common confidentiality mechanisms. For instance, the case study systems originating from the SecDFD approach [TSB19] distinguish the classification and the zone of nodes by having different sets of values. However, when also using the classification values for describing the clearance of nodes, the analysis can detect the same violations and matches the definition of a non-interference analysis using a linear lattice. Recognizing this, requires some expertise. A lack of expertise can have a negative impact on the measured expressiveness. Because we are interested in the upper bound of expressiveness and we could not find confidentiality requirements, which we could not express, this factor had no effect. Another potential influencing factor is that the formulated confidentiality requirements do not represent the confidentiality requirements of the case study system. We mitigate this factor by VQ6, which validates that an analysis based on the requirements detects violations for the issue contained in the case study system. This does not fully mitigate the threat but weakens it by showing that the formulated requirements at least cover the critical aspect of the requirements. Because the issues contained in the case study system usually demonstrate the most important aspects of the confidentiality mechanism and requirements used by the systems, we think that the treat of formulating too simple requirements is negligible.

External Validity is concerned with the generalization of case study results to other contexts. Because a case study does not use a representative sample but a limited set of cases, results cannot be generalized to arbitrary other contexts. However, the results can be generalized to other cases with comparable characteristics. We discuss these characteristics in the following.

The result of the *expressiveness* validation (VM5.1, VM5.2 and VM5.3) is that the analysis definitions can express the requirements of systems. This finding can be generalized to systems using data flows, using one of the mentioned confidentiality mechanisms and specifying confidentiality requirements in terms of data. The restriction on data flows is important because a DFD cannot represent pure control flows, which means that confidentiality mechanisms represented by the DFD syntax always work on exchanged data. A system, in which activities cannot be represented appropriately by data flows and data processing, cannot be represented in a DFD. Consequently, confidentiality requirements also have to be specified in terms of data. This is a general limitation of analyses based on DFDs. However, this limitation is not too restrictive because many different systems have been initially designed using DFDs and threat modeling, which is one of the most prominent approaches to identify security issues, also uses requirements based on data.

We can explicitly not generalize the conclusions to confidentiality mechanisms, which use role hierarchies in RBAC or use selector hierarchies in ABAC because no case study system covered these aspects.

Construct Validity is concerned with the appropriateness of taken measures to make statements about the research objective.

The weighted ratio metrics of the *expressiveness* validation (VM5.1, VM5.2 and VM5.3) measure the ratio between systems, of which we could express confidentiality requirements, and the total amount of systems. The statement to be made based on the metric value is that the analysis definitions for DFDs can express commonly used confidentiality requirements in the context of particular systems. The metric is appropriate to provide the information for the statement because non-expressible requirements lower the metric value and indicate a problem. The threshold for considering the validation to succeed is 1, which means that all confidentiality requirements have to be expressible. Apart from this threshold-based interpretation of the metric, the metric also gives an idea of how bad the expressiveness is, which is valuable for a discussion. It is reasonable to not use a simple ratio metric of system, for which we could express all confidentiality requirements, compared to the total amount of systems because the amount of systems using a certain confidentiality mechanism and thereby having the same type of confidentiality requirements varies and we are especially interested in how well the analysis definitions can express confidentiality requirements for different confidentiality mechanisms within systems. Therefore, not supporting the confidentiality requirements of two systems with two different confidentiality mechanism is worse than not supporting the requirements of two systems with the same confidentiality mechanism. The metric respects this and handles unbalanced sets of systems with respect to the confidentiality mechanism.

Reliability is concerned with the dependency between the collected data and the conducting researcher. Best reliability is achieved if the collected data as well as the conclusions are completely independent of the conducting researcher.

The data collected in the *expressiveness* validation (VM5.1, VM5.2 and VM5.3) certainly depends on the expertise in interpreting and modeling confidentiality requirements. A person with less expertise might not be able to express the same amount of requirements as we did. We already mentioned this problem while discussing the internal validity. However, it is not necessary that another person produces the exactly same results because the validation is about finding an upper bound of expressiveness. Therefore, it is only necessary that other researchers can understand the decision of whether confidentiality requirements have been successfully expressed or not. We provide all created requirements as part of a data set [Sei22], so other researchers can decide whether the classification of the requirements as expressible was correct. The corresponding metrics can be calculated objectively, which we also did as part of the validation application in our data set [Sei22].

8.5. Validation of DFD Semantics

The validation of the DFD semantics, which we described in Section 5.2, is the third validation goal (VG3). We describe the validation design for answering the validation questions VQ6 and VQ7 in Section 8.5.1. The results of executing the designed validation are presented in Section 8.5.2 and discussed in Section 8.5.3. We discuss threats to the validity of our results and conclusions in Section 8.5.4.

8.5.1. Validation Design

The validation design describes the procedure to provide answers to both validation questions, which support VG3.

The first validation question VQ6 asks whether analyses, which are built upon the DFD semantics, can correctly identify systems, which violate confidentiality requirements. A case study is appropriate to provide this information because it aims for getting insights into the application of the approach in particular cases, which implies particular systems. We reuse the systems, which have already been modeled as part of the syntax validation (VQ1 and VQ2), as well as the confidentiality requirements created as part of the analysis validation (VQ5). In the validation, we run the analysis for every variant of every case study system, for which we have a modeled DFD and the corresponding confidentiality requirements. All case study systems CS1–CS17 except for CS5 meet these requirements. It is reasonable to exclude CS5 because it is not possible to run an analysis on a model, which does not exist. For every of these case study systems, we have two variants: one variant containing no issue and one variant containing an issue. Running an analysis for a variant means running the automated analysis steps described in Section 6.1. This covers the mapping of the system, i.e. the DFD, to a logic program and the execution of the label comparison, i.e. running the query in the logic program. The automated analysis steps yield a list of violations.

To calculate the true positive fraction (VM6.1) and the true negative fraction (VM6.2), we classify the reported violations. All violations reported for the variants without issue are wrong because there cannot be violations without an underlying issue in our case study systems. The violations reported for the variant with issue have to be rated individually. For every such reported violation, we check if the violation matches the expected violations, which we described for every case study system in Section 8.2. If a violation does not match the expected violations, we consider it wrong.

We consider every system variant, for which at least one wrong violation has been reported, as having wrong results. To calculate the true positive fraction, we collect all variants containing an issue in a set S_i and all of these variants, which do not have wrong results, in a set $S_i' \subseteq S_i$. The true positive fraction is then $TPF = {}^{|S_i'|}/_{|S_i|}$. To calculate the true negative fraction, we collect all variants not containing an issue in a set $S_{\bar{i}}$ and all of these variants, which do not have wrong results in a set $S_{\bar{i}}' \subseteq S_{\bar{i}}$. The true negative fraction is then $TNF = {}^{|S_{\bar{i}}'|}/_{|S_{\bar{i}}|}$.

The second validation question VQ7 asks whether the DFD semantics limit the automation of analyses. To answer this question, we have to know the set of analysis steps A and the set of automated analysis steps $A_a \subseteq A$. We define the analysis steps A as part of a discussion. Afterwards, we derive the set of automated analysis steps A_a from our prototypical implementation by comparing the set of analysis steps with features of our implementation. After having both sets, we can identify the amount of not automated steps $\bar{a} = |A \setminus A_a|$, which is also the required metric VM7.1.

8.5.2. Validation Results

The analyses based on DFD semantics could successfully identify correct violations for systems containing an issue, which means that the TPF (VM6.1) is 1.0. The analyses based on DFD semantics could also successfully identify that there are no violations in systems not containing an issue, which means that the TNF (VM6.2) is 1.0.

In order to calculate VM7.1, we collected the steps to be done during and analysis and checked whether our prototypical implementation automates these steps. The input of the analysis is the system to analyze, i.e. the DFD including applied confidentiality mechanisms, and the confidentiality requirements, i.e. the query for unwanted combinations of labels. The output of the analysis is a set of violations. Everything between receiving the inputs and producing the outputs is part of the analysis. The required steps are:

A1 Transforming the DFD into a logic program

A2 Propagating the data including its labels through the system

A3 Comparing the labels of data with labels of nodes

A4 Reporting identified violations

All of the previously mentioned steps are automated within our prototype. We described the mapping of a DFD to a logic program (A1) in Section 5.2.2. The mapping does not require human interpretation or heuristics. Therefore, we could implement the mapping as a model transformation in our prototype. The propagation of labels (A2) is the core of the semantics. The semantics require clear specifications of the propagation functions in the analysis definitions. Because the propagation functions are limited to logic operations, constants and references to other labels, the propagation functions can be executed in a fully automated way. The logic to identify paths through the DFD, on which labels can be propagated, is also specified in form of a clear algorithm. Both parts do not require human interaction. The propagation of labels is automated within the logic program. The comparison of labels (A3) uses the propagated labels, which can be determined automatically, and a comparison function. Because the comparison function is given as a query to a logic program, it can be executed fully automatically. The violations are also automatically reported (A4) by the Prolog execution environment, which finds solutions to the label comparison function.

Based on that discussion, we can calculate the sum of not automated analysis steps (VM7.1). Because the set of analysis steps A and the set of automated analysis steps A_a are the same, i.e. $A = A_a$, the amount of not automated steps $\bar{a} = |A \setminus A_a|$ is 0.

8.5.3. Result Discussion

We structure the discussion of the results by the corresponding validation questions.

Discussion of VQ6. The validation question aims to validate that analyses based on the DFD semantics can correctly identify systems containing violations. As the true positive fraction $TPF = 1.0$ shows, the analyses could successfully identify all variants of case study systems, which contain an issue, as containing violations. Because we request that all reported violations for each individual system are correct in order to consider the results for a system variant as true positive, we can also state that the analyses did not report a wrong violation. As the true negative fraction $TNF = 1.0$ shows, the analyses could successfully identify that all variants not containing an issue do not contain violations. Both metric values are also the required values for answering the validation question positively. The results support both, the validation goal about the DFD analyses (VG2) and the DFD semantics (VG3): The results show that the DFD semantics are a good foundation for realizing confidentiality analyses, which yield correct results. It would be unlikely that weak semantics together with the high degree of automation of the analyses always yield correct results. Therefore, the results support VG3. The results also show that the DFD analyses can correctly detect violations within systems. Too simple analysis definitions or wrongly specified analysis definitions would, most probably, not always yield correct results. Therefore, the results support VG2.

Discussion of VQ7. The validation question aims to validate that the semantics do not limit the automation of analyses. As the number of not automated steps $\bar{a} = 0$ shows, we could automate all steps of the analysis assuming that the required inputs (DFDs, analysis definitions and confidentiality requirements) are available. In the discussion about the automation, we clearly stated that the analysis does not rely on human intervention or heuristics. Because all steps could be automated, the semantics cannot have limited the automation of analyses.

8.5.4. Threats to Validity

Because one part of the validation took place as a case study, we structure the discussion of threats to validity according to the guidelines of Runeson and Höst [RH09, pp. 153] for discussing the validity of a case study. The categories to be discussed are also suitable for other types of validation designs, such as the discussion of automation.

Internal Validity is concerned with how well a taken measure supports a cause-effect relationship and especially whether there are alternative explanations for the effect.

In the context of the *correctness* of analyses (VQ6), we expect the DFD semantics and the analysis definitions for DFDs to be the causes of an effect. The expected effect of both are correct analysis results. We discuss potential alternative explanations of this effect, i.e. other possible influencing factors, in the following. The DFDs to be analyzed can have an effect on the correctness of the analysis results. We distinguish two cases: If the modeled DFD shall contain an issue but actually does not, we would classify the result as wrong, which would lower the TPF metric. The same holds if the DFD contains an issue but the issue is not the expected issue, i.e. the resulting violations are not expected. The described case did not occur in our case study, which can be seen from the TPF, which is 1.0. The second case occurs if the modeled DFD shall not contain an issue but actually does contain an issue. In this case, we would classify the result as wrong, which would lower the TNF metric. The described case did not occur in our case study, which can be seen from the TNF, which is 1.0. Besides the classification guidelines for systems, the classification guidelines for violations also influence the result. If the guidelines accept violations as correct even if the violations are actually wrong, the TPF looks more positive than it actually is. We mitigated this problem by specifying the expected violations as part of the case study system descriptions in Section 8.2. These descriptions provide clear guidelines on how to classify violations.

In the context of the *automation* of analyses (VQ7), we expect the DFD semantics to be the causes of an effect. The expected effect of the DFD semantics is the capability to automate all analysis steps. We discuss potential alternative explanations of this effect, i.e. other possible influencing factors, in the following. The scope of the analysis is one potential influencing factor. We defined the scope of the analysis as reporting violations based on a given DFD and confidentiality requirements. Another definition could be to provide assistance while creating the DFD by reporting potential violations. The latter definition would most probably not allow full automation because incomplete DFDs and confidentiality requirements require additional human input or intervention. However, even in the real-time analysis, there is one analysis part, which analyzes the DFD and confidentiality requirements as they are, to identify the need for human input. Therefore, the discussion of automation is still applicable to these scenarios. The confidentiality requirements are another influencing factor. If the requirements are not given in a formal specification, automation becomes impossible. However, analyses always require inputs to be given in a particular formalization. We already validated that expressing the confidentiality requirements of the case study systems is possible in VQ5 and created the requirements in the expected format. Therefore, a lack of formalization of the requirements is no influencing factor here.

External Validity is concerned with the generalization of case study results to other contexts. Because a case study does not use a representative sample but a limited set of cases, results cannot be generalized to arbitrary other contexts. However, the results can be generalized to other cases with comparable characteristics. We discuss these characteristics in the following.

The result of the *correctness* validation (VM6.1 and VM6.2) is that the DFD analyses based on the DFD semantics can correctly identify systems containing violations. This finding can be generalized to systems, which are given in the DFD syntax and which use the confidentiality mechanisms and features of the confidentiality mechanisms, which were considered in the validation. The confidentiality requirements have to be given in terms of a query to the logic program and optional additional facts. We cannot draw conclusions for other confidentiality mechanisms.

The result of the *automation* validation (VM7.1) is that the DFD semantics do not limit the automation of analyses. Because this insight is based on a general applicable discussion, it is not limited to particular cases but is generally applicable.

Construct Validity is concerned with the appropriateness of taken measures to make statements about the research objective.

In the *correctness* validation, we use the true positive fraction TPF (VM6.1) and the true negative fraction TNF (VM6.2). The statement to be made is that the analyses using the DFD semantics can correctly identify systems containing violations. The true positive fraction provides the necessary information to make this statement: Every correctly identified system containing violations is counted and the ratio between the correctly identified systems and the systems containing violations is calculated. A value of 1.0 means that all systems, which should have been identified, have been identified. However, the true positive fraction does not cover falsely reported systems. This means, an analysis, which always reports a system to contain a violation without even analyzing the system, still has a true positive fraction of 1.0. Therefore, it is necessary to also calculate the true negative fraction, which builds the ratio between systems, which have been correctly not identified as containing violations, and the total amount of systems, which actually do not contain violations. If the true negative fraction is also 1.0, it means that also systems containing no violations are correctly classified by the analysis. This means that the analysis actually analyzes the system. As already discussed by Metz [Met78], the combination of both metrics is good to rate the quality of binary classifiers, which is essentially what the validation question is about.

In the validation of the degree of *automation*, we use the sum of analysis steps, which are not automated. The statement to be made is that the DFD semantics do not limit the automation of analyses. The metric is simple and provides all information to support the statement. As long as the sum of non-automated analysis steps is 0, the statement is supported.

Reliability is concerned with the dependency between the collected data and the conducting researcher. Best reliability is achieved if the collected data as well as the conclusions are completely independent of the conducting researcher.

The data collected in the *correctness* validation (VM6.1 and VM6.2) depends on the conducting researcher when it comes to classifying the reported violations. Classifying the violations is necessary because a wrongly reported violation lowers the metric values and therefore influences the conclusions to be drawn from the data. To classify the violations, a researcher has to understand the issue introduced in the case study systems as well as the implications of this issue. To improve the reliability, we described each issue as well as its implications, i.e. the expected violations, as part of the descriptions of the case study systems in Section 8.2. In addition, we added the classification of violations to the application for reproducing the validation results in our data set [Sei22]. For every case study system, we clearly encode the classification guidelines in Java, so other researchers can check our classifications. When using the same classification guidelines, the resulting data is the same because the remaining parts of the validation design are fully automated by the application for reproducing the validation. Based on the data, the same conclusions can be drawn.

The data collected in the validation of the *automation* (VM7.1) depends on the identified analysis steps as well as on the discussion about automating them. The analysis steps can be identified based on the provided discussion and based on the explanation of the analysis procedure in Section 6.1, so other researchers can identify the same analysis steps. In the discussion about the automation of analysis steps, we often refer to the implementation of our prototype, which shows that an analysis step can be automated. Other researchers can check these statements based on the source code and by executing the application to reproduce the validation results, which we both provide in our data set [Sei22].

8.6. Validation of ADL Integration Guidelines

The validation of the integration guidelines for existing ADLs, which we described in Chapter 7, is the fourth validation goal (VG4). The validation of the guidelines takes place by a validation of integrations, i.e. extended ADLs, which result from applying the integration guidelines to existing ADLs. We describe the validation design for answering the validation questions VQ8–VQ13 in Section 8.6.1. The results of executing the designed validation are presented in Section 8.6.2 and discussed in Section 8.6.3. We discuss threats to the validity of our results and conclusions in Section 8.6.4.

8.6.1. Validation Design

The validation design describes the procedure to provide answers to all six validation questions, which support VG4. The integration guidelines support ADLs using control flows and ADLs using data flows. Because these types of ADLs have fundamental differences, we have to ensure that the validation results are valid for both types of ADLs. Thereto, all metrics to answer the validation questions belonging to VG4 distinguish two scenarios: In one scenario, metrics for an ADL focusing on control flows shall be calculated. In the other scenario, metrics for an ADL focusing on data flows shall be calculated. The extended

Palladio ADL presented in Section 7.2 supports both scenarios. In the first scenario, we limit the use of the Palladio ADL to the subset, which focuses on control flows. We treat this subset like an individual ADL in the following descriptions. In the second scenario, we use the communication via data flows as often as possible. We cannot solely use data flows because Palladio does not support communication based on data flows between the users and the system as described in Section 7.2.2. We also treat this usage of the ADL as an individual ADL in the following descriptions. Based on these definitions, we describe the design for answering the questions in the following.

Case Study for VQ8. The first validation question asks whether the expressiveness of the extended ADL syntax and semantics (VQ8) is lower than the expressiveness of DFDs. The expressiveness of the DFD syntax and the semantics has been validated by a case study using particular systems. In order to allow a comparison of expressiveness, it is reasonable to also conduct a case study using the same case study systems and the same metrics. We conduct the case study in three steps:

1) For each case study system described in the overview on case study systems in Section 8.2, we try to model the system and its usage of the confidentiality mechanism in the extended Palladio ADL twice: one time using the Palladio ADL focusing on control flows and one time using the Palladio ADL focusing on data flows. Because modeling a DFD is not directly possible in Palladio using control flows or data flows, we adapt the structure of the system as well as its usage if necessary. The adapted systems still have to represent the same functionality as specified in the description of the case study system. In addition, the critical aspects for analyzing confidentiality still have to be represented. For instance, data still has to be joined or declassified.
2) For each tuple of case study system and used ADL subset (control flow or data flow), we classify the modeling result either as successfully modeled or as not successfully modeled. A system is classified as successfully modeled if the structure, deployment and confidentiality requirements could be represented and if every behavior and usage, which affects the confidentiality, could be represented.
3) We calculate the metrics to answer VQ8.
    3.1) We only consider systems modeled in the control flow ADL when calculating the weighted ratio metrics VM8.1–VM8.3 because the metrics focus on the expressiveness of the control flow ADL. For calculating VM8.1, we only consider systems using access control mechanisms. For calculating VM8.2, we only consider systems using information flow control mechanisms. For calculating VM8.3, we only consider systems using a mix of access control and information flow control mechanisms. The weighted ratio metric normalizes the weight of a modeling result based on the used confidentiality mechanism. Therefore, we group the systems, which are considered for calculating the metrics, by the particular confidentiality mechanism (e.g. RBAC). The information to do so is available in the overview on the case study systems in Section 8.2. Each group has the same weight in the final metrics.

3.2) We only consider systems modeled in the data flow ADL when calculating the weighted ratio metrics VM8.4–VM8.6 because the metrics focus on the expressiveness of the data flow ADL. For calculating VM8.4, we only consider systems using access control mechanisms. For calculating VM8.5, we only consider systems using information flow control mechanisms. For calculating VM8.6, we only consider systems using a mix of access control and information flow control mechanisms. The weighted ratio metric normalizes the weight of a modeling result based on the used confidentiality mechanism. Therefore, we group the systems, which are considered for calculating the metrics, by the particular confidentiality mechanism (e.g. RBAC). The information to do so is available in the overview on the case study systems in Section 8.2. Each group has the same weight in the final metrics.

Case Study for VQ9. The second validation question asks whether the correctness of the analysis results (VQ9) is worse compared to the results of DFD-based analyses. We rated the correctness of DFD-based analyses by running them on particular systems in a case study. Therefore, we have to do the same in order to get comparable results. The DFD-based case study used the same case study systems as the systems considered by the validation of the expressiveness (VQ8). Therefore, we can run the analyses on the systems, which already have been modeled for answering VQ8. If we cannot express a system in VQ8, we exclude it from the case study for VQ9. This is reasonable because an analysis always requires an input and if that input is not available, the analysis cannot produce results. Considering a system, which cannot be expressed, does not provide any insights beyond that our analyses require a system as an input. For every expressible system, there are two variants. One variant contains an issue, which leads to violations of confidentiality requirements. The other variant contains no issue and does not violate confidentiality requirements. The introduced issues are available in the descriptions of the case study systems in Section 8.2.2. We use both variants in the case study, which we conduct by executing the following three steps:

1) We run the analysis for violations of confidentiality requirements on all four models of a case study system: Every system has been modeled using the control flow ADL and the data flow ADL. For both system models, there are two variants (one with issue and one without issue), which results in a total of four system models. Running an analysis means that the automated tooling takes the modeled system as well as the label comparison function and reports detected violations.

2) We classify all reported violations as correct or wrong. A reported violation for a system variant without issue is always wrong because there are no issues that could lead to a violation. In order to be correct, a reported violation for a system variant with issue has to be within the expected violations, which we explain in Section 8.2.2 for every system. A violation is within the expected violations if it occurs in the expected locations, which are described in in Section 8.2.2. Otherwise, the reported violation is wrong.

3) We calculate the metrics to answer the validation question. We always classify whole system variants based on the classifications of the reported violations.

   3.1) The true positive fraction TPF (VM9.1) considers system variants with issues modeled in a control flow ADL. A system variant is classified as true positive if at least one violation is reported and all reported violations are correct.

   3.2) The true negative fraction TNF (VM9.2) considers system variants without issues modeled in a control flow ADL. A system variant is classified as true negative if no violations have been reported.

   3.3) The true positive fraction TPF (VM9.3) considers system variants with issues modeled in a data flow ADL. A system variant is classified as true positive if at least one violation is reported and all reported violations are correct.

   3.4) The true negative fraction TNF (VM9.4) considers system variants without issues modeled in a data flow ADL. A system variant is classified as true negative if no violations have been reported.

Discussion of VQ10 VQ10 asks whether the degree of automation of ADL-based analyses is lower compared to DFD-based analyses. We already rated the degree of automation for DFD-based analyses by a discussion of analysis steps and their automation for VQ7. In order to compare the degree of automation, we use the same validation approach: We collect all analysis steps for an ADL and classify an analysis step as automated if the prototype for the Palladio ADL automates this step. Because the analysis steps might differ between DFD-based analyses and ADL-based analyses, we assign every analysis step a purpose. After that, we can collect purposes, which are not automated for DFD-based analyses, and we can collect purposes, which are not automated for ADL-based analyses. A purpose is classified as not automated if there is at least one not automated analysis step that serves this purpose. By identifying the purposes, which are no longer automated for ADLs, we can calculate the metrics. We do the previously described steps for the analysis procedure of control flow ADLs to calculate VM10.1 and for the analysis procedure of data flow ADLs to calculate VM10.2.

Case Study for VQ11 VQ11 asks whether the extended ADL lowers the modeling effort for adding confidentiality mechanisms to software architectures compared to the state of the art. As discussed in Section 8.1.4, we can show reduced modeling effort by showing that software architects do not have to remodel the software architecture from scratch for adding a confidentiality mechanism. Therefore, it is sufficient to validate that at least one model element can be reused. Because the integration into Palladio reuses many ADL elements to describe the structure, behavior, deployment and usage of the system, it is reasonable to assume that software architects can reuse a considerable amount of already modeled software architectures, which lowers the overall modeling effort. This means software architects should always be able to reuse parts of existing software architectures because of the way we constructed the ADL integration. However, we would also like to show the validity of this statement for particular systems and give some numbers on how much can actually be reused. A case study is appropriate to provide the answer to the validation question because it focuses on gaining insights into particular applications of an approach, which is necessary because the amount of reuse can vary depending on the particular systems. We conduct the case study by executing the following three steps:

1) We create variants of case study systems, which do not contain confidentiality mechanisms. These variants represent software architectures, which the software architect already modeled without integrating confidentiality. We create the variants by modifying the software architectures, which we already modeled for answering VQ8: We remove or replace all model elements, which belong to the ADL extension. We replace operational data stores with regular components. We remove all applied stereotypes, i.e. characteristics of nodes and behaviors of data channels, and all variable characterizations describing confidentiality. Because the removed elements are no mandatory elements of the Palladio ADL and the elements do not replace other mandatory elements of the Palladio ADL, the resulting software architecture is still a valid software architecture. We create these variants for the case study systems CS1, CS2 and CS3 based on the modeled software architectures using control flows as well as data flows. Using these three systems is beneficial because each of these systems shares the represented system with at least one other case study system. For instance, CS1 represents the *TravelPlanner* system just like CS10 and CS17. This allows us to not only compare the variant without a confidentiality mechanism with the case study system, from which we derived the variant, but also with other case study systems. This is useful because the validation would only show that we did not replace or remove all model elements while creating the variant, otherwise.

2) We calculate the Jaccard Coefficients. To identify equal elements, we use the comparison approach of EMF Compare [BP08] consisting of two phases: First, we match elements of two models, which shall be equal. We match elements based on their unique identifiers. This works in our case because we created models of case study systems, which represent the same system, by copying an existing model and adjusting it to use another confidentiality mechanism. Second, we compare the matched elements to classify them as equal or not equal. Two elements are equal if all attributes and references of the elements are equal. Attributes are equal if the values are equal. References are equal if the target of the reference in the first model and the target of the reference in the second model have been matched to each other.

    2.1) We compare the variant of CS1 with the modeled case study systems for CS1, CS10 and CS17.

    2.2) We compare the variant of CS2 with the modeled case study systems for CS2 and CS11.

    2.3) We compare the variant of CS3 with the modeled case study systems for CS3 and CS12.

    2.4) We do the comparisons separately for the models using control flows and the models using data flows.

3) We compare the Jaccard Coefficient with the threshold in order to answer the validation question regarding the state of the art. For this comparison, we assume that many approaches from the state of the art require the software architect to remodel the whole system, which brings us to a threshold of 0, i.e. we consider a Jaccard Coefficient greater than 0 as good.

Case Study for VQ12 VQ12 asks whether the extended ADL lowers the modeling effort for switching between confidentiality mechanisms in software architectures compared to the state of the art. We use the amount of models elements, which have to be changed, to reason about the modeling effort. If not all model elements have to be changed, the modeling effort can be considered lower than for state-of-the-art approaches, which require remodeling the whole system. Because the amount of required changes depends on the particular system and the involved confidentiality mechanisms, we conduct a case study. A case study is a good approach because it aims for gaining insights into the application of an approach in particular contexts. With respect to this validation, different contexts mean different software architectures, i.e. case study systems. We conduct the case study by executing the following three steps:

1) We identify all possible pairs of case study systems, which represent the same system. For instance, CS2 and CS11 both use the *TravelPlanner* system. We use the system models, which we created for answering VQ8. We do not mix models using control flows and data flows in pairs but create dedicated pairs.
2) We calculate the Jaccard Coefficients (VM12.1 and VM12.2) for each pair of models. We use the comparison procedure already described for VQ11 to identify equal model elements.
3) We compare the Jaccard Coefficient with the threshold in order to answer the validation question regarding the state of the art. For this comparison, we assume that many approaches from the state of the art require the software architect to remodel the whole system when switching confidentiality mechanisms. This brings us to a threshold of 0, i.e. we consider a Jaccard Coefficient greater than 0 as good.

Discussion of VQ13 VQ13 asks whether all information required to model a software architecture using the extended ADL is available to the software architect and security expert. To answer the question, we discuss the required information to instantiate every newly introduced element of the extended ADL syntax and build groups of information. It is not necessary to consider elements of the non-extended ADL in the validation because we can assume that an existing ADL is usable by software architects and the required information to create software architectures using the ADL is available. For instance, one group could be the information about the behavior of the architecture. We use the groups, which we identified in the validation of VQ4 in Table 8.4 on page 197 whenever possible because VQ4 essentially asks the same question as VQ13 but with respect to DFDs. This brings us to the set I of necessary information. Afterwards, we discuss whether this information is available in the required granularity when creating the software architecture. We base our decision about availability by looking at the information, which other, established ADLs require, and by collecting commonly required information to use typical architectural viewpoints. This brings us to the set I_k of necessary information. After this discussion, we can classify each information as either known or unknown. We build the sum of unknown information for model elements used in the control flow ADL as metric VM13.1. The sum of unknown information for model elements used in the data flow ADL is metric VM13.2. A metric value greater than 0 means that information is necessary but unknown by the users, which means a failed validation. A discussion is a reasonable method for collecting the data to calculate the metric because there is no formal definition of *information*, which is usable for answering the validation question. A discussion can cover many different influencing factors and provides valuable results as long as the line of argumentation is clear.

8.6.2. Validation Results

We structure the presentation of the validation results by the validation questions.

Expressiveness of ADL (VQ8). We tried to model all case study systems mentioned in Section 8.2 in Palladio using control flows as well as in Palladio using data flows. We could successfully model all systems using access control mechanisms in Palladio using control flows, which means that the weighted ratio metric (VM8.1) is $\tilde{r} = 1.0$. We could model all systems using information flow control mechanisms in Palladio using control flows except for the BankingApp system. This brings us to a weighted ratio metric (VM8.2) of $\tilde{r} = 0.75$. We could model all systems using mixed access control and information flow control mechanisms in Palladio using control flows, which means that the weighted ratio metric (VM8.3) is $\tilde{r} = 1.0$. The results for Palladio using data flows are the same: We could successfully model all systems using access control as well as all systems using combined access control and information flow control mechanisms. This means the weighted ratio metrics VM8.4 and VM8.6 are both $\tilde{r} = 1.0$. We could model all systems using information flow control mechanisms except for the BankingApp system. This brings us to a weighted ratio metric (VM8.5) of $\tilde{r} = 0.75$.

Correctness of Analysis Results (VQ9). We executed the automated analyses on all systems resulting from VQ8. We classified the reported analysis results, which brings us to the following metric values. The true positive fraction of the analysis results stemming from models using control flows is $TPF = 1.0$ (VM9.1). The true negative fraction of the analysis results stemming from models using control flows is $TNF = 1.0$ (VM9.2). The true positive fraction of the analysis results stemming from models using data flows is $TPF = 1.0$ (VM9.3). The true negative fraction of the analysis results stemming from models using data flows is $TNF = 1.0$ (VM9.4).

Automation of Analyses (VQ10). To reason about the automation of analyses, we first collect the activities to be done in ADL-based analyses. The activities are visualized in Figure 7.2 on page 116. We do not consider creating inputs for the analysis as an activity of the analysis itself but as necessary preparations. Addressing the violations by identifying the underlying issue and changing the system design is also not part of the analysis itself but a follow-up activity. Because the analysis procedure for ADL-based analyses is built upon the analysis procedure for DFD-based analyses, the analysis activities overlap. In Section 8.5.2, we list the following four activities:

A1 Transforming the DFD into a logic program

A2 Propagating the data including its labels through the system

A3 Comparing the labels of data with labels of nodes

A4 Reporting identified violations

All of these activities are still valid and part of the analysis procedure for ADLs. In order to reuse these activities, the analysis procedure for ADLs introduces one additional activity, which has to be done before the other activities:

A0 Transforming the ADL model into a DFD

By transforming the ADL to a DFD, we can reuse the analysis procedure for DFDs. This means, there are five activities for ADL-based analyses (A0–A4) and four activities for DFD-based analyses (A1–A4). The purposes of these activities are as follows:

P1 Prepare analysis execution

P2 Conduct the analysis for violations

P3 Present the analysis result

The mapping of the analysis activities to the purposes is as follows. The mappings of the ADL model to a logic program (A0 and A1) as well as the mapping of the DFD to a logic program (A1) serve purpose P1 because these activities do not yet look for violations but only map the models into an artifact, which the tooling can analyze. The activities of propagating labels and comparing labels for ADLs (A2 and A3) as well as for DFDs (A2 and A3) serve purpose P2 because they actually look for violations by considering the system behavior. The activities of reporting the identified violations for ADLs (A4) as well as for DFDs (A4) serve purpose P3 because they make the analysis results accessible to the software architect.

We already discussed that the analysis activities for DFD-based analyses are completely automated in Section 8.5.2. Because the ADL-based analyses simply reuse these activities, these activities are also automated for ADL-based analyses. The mapping of the ADL model into a DFD (A0) is also automated in the prototypical implementation of the Palladio integration. This is possible because the mapping does not require heuristics or human interaction and it provides rules for all relevant parts of the architecture. This means that there are no activities, which have not been automated in the ADL-based analyses. Consequently, there is no non-automated purpose, i.e. $p_{\bar{a}} = 0$.

Metric	CS1	CS2	CS3	CS10	CS11	CS12	CS17
VM11.1	0.49	0.59	0.52	0.52	0.60	0.53	0.33
VM11.2	0.47	0.44	0.54	0.50	0.44	0.56	0.40

Table 8.5.: Overview on the Jaccard Coefficients for adding confidentiality mechanisms to case study systems (CS) using control flows (VM11.1) and data flows (VM11.2).

Metric	CS1/CS10	CS1/CS17	CS10/CS17	CS2/CS11	CS3/CS12
VM12.1	0.81	0.50	0.54	0.86	0.75
VM12.2	0.88	0.66	0.71	0.86	0.88

Table 8.6.: Overview on the Jaccard Coefficients for switching confidentiality mechanisms between case study systems (CS) using control flows (VM12.1) and data flows (VM12.2).

Effort for Introducing Confidentiality Mechanisms (VQ11) We compared the case study systems with their corresponding variant without integrated confidentiality mechanisms and calculated the Jaccard Coefficient. The results are shown in Table 8.5. We discuss the results in Section 8.6.3.

Effort for Switching Confidentiality Mechanisms (VQ12) We compared all case study systems with each other that represent the same system and calculated the Jaccard Coefficient. The results are shown in Table 8.6. We discuss the results in Section 8.6.3.

Availability of Information (VQ13) In order to calculate the metric values of VM13.1 and VM13.2, we collected the required information to use the newly introduced model elements. Table 8.7 lists all model elements, the action to do with the model elements, the corresponding user and the category of information, which the user needs to know to perform the action. The triples of category, action and user build the elements of the set of information I. We can show for every triple $i \in I$ that the required information is available: The knowledge required to create data stores is available to software architects because storing files or data is a common activity of software systems, which has to be considered when creating the software architecture. Palladio extensions for analyzing storage performance [Bus+15] or performance of database transactions [MS14] also assume knowledge about such a storage. The information required to create the model elements describing the behavior is available to the security expert as we already discussed in Section 8.6.3. We only discuss the relation from the ADL elements to DFD elements here and refer to the discussion of available information for the DFD elements. All DFD elements mentioned in the following can be created by the security expert as discussed in Section 8.6.3. The Confidentiality Variable Characterization is the counterpart of an Assignment in DFDs. The model elements representing terms are the same as for DFDs. Because the extended ADL does not use pins but names to refer to data, the Named Enum Characteristic Reference and Lhs Enum Characteristic Reference is the counterpart of the Data Characteristic Reference in DFDs. The Data Channel Behavior and the Reusable Behavior are the counterpart

Category	Model Element	Action	User
Structure	Operational Data Store Component	Create	Architect
Behavior	Confidentiality Variable Charact.	Create	Sec. Exp.
	And	Create	Sec. Exp.
	Or	Create	Sec. Exp.
	Not	Create	Sec. Exp.
	True	Create	Sec. Exp.
	False	Create	Sec. Exp.
	Container Characteristic Reference	Create	Sec. Exp.
	Named Enum Characteristic Reference	Create	Sec. Exp.
	Lhs Enum Characteristic Reference	Create	Sec. Exp.
	Data Channel Behavior	Create	Sec. Exp.
	Data Channel Behavior	Bind	Architect
	Confidentiality Behavior	Apply	Architect
	Reusable Behavior	Create	Sec. Exp.
	Behavior Reuse	Create	Architect
	Variable Binding	Create	Architect
Properties	Characterizable	Apply	Architect
	Characteristic	Create	Sec. Exp.
	Characteristic	Bind	Architect

Table 8.7.: Overview on categories of information required to use model elements of the extended ADL.

of Behavior Definitions in DFDs. None of the mentioned ADL elements, which describe the behavior, require more information to be created than their counterparts used in DFDs. Because a security expert has the necessary knowledge to create the DFD elements, he/she also has the knowledge to create the corresponding ADL elements. The information to bind the Data Channel Behavior to the architecture by applying the Confidentiality Behavior and referring to the Data Channel Behavior is also known to the architect because he/she has to know the data processing of components as defined by the information viewpoint [RW05, p. 36], which is a common viewpoint required to create software architectures. Because the knowledge of the structure of the software architecture as well as the knowledge for selecting a behavior is available to the architect, he/she can also reuse behaviors by creating Behavior Reuse elements and Variable Binding elements to bind variable names to the reused behaviors. Because creating and binding characteristics works the same way as in DFDs and the procedure requires the same knowledge in ADLs as well as in DFDs, we can also assume that the security expert can create the characteristics and that the software architect can bind them to the structural elements of the software architecture. We refer to the discussion for DFDs in Section 8.3.3 for a detailed explanation on why the knowledge is available. Because we could not identify knowledge, which is not available but required to create the model elements, we can conclude that the set I of required information is equal to the set I_k of known information, i.e. $I = I_k$ holds. This means that the sum of unknown information for control flow ADLs (VM13.1) is 0 and that the sum of unknown information for data flow ADLs (VM13.2) is 0.

	Access Control		Information Flow		Mixed	
	Syntax	Analyses	Syntax	Analyses	Syntax	Analyses
DFD	1.0	1.0	0.75	1.0	1.0	1.0
ADL CF	1.0	1.0	0.75	1.0	1.0	1.0
ADL DF	1.0	1.0	0.75	1.0	1.0	1.0

Table 8.8.: Overview on values of weighted ratio metrics for expressiveness of DFDs and ADLs using control flows (CF) or data flows (DF).

8.6.3. Result Discussion

We structure the discussion of the results by the corresponding validation questions.

Expressiveness of ADL (VQ8). The goal of the validation question is to show that the ADL, on which the integration guidelines have been applied, is not less expressive than the DFDs with respect to describing systems for conducting analysis for violations of confidentiality requirements. We can show this by comparing the expressiveness metrics for the ADL (VM8.1–VM8.6) with the expressiveness metrics for the DFD syntax (VM1.1, VM1.2, VM2.1) and DFD-based analyses (VM5.1–VM5.3). Table 8.8 summarizes the corresponding metric values. As can be seen from the table, the extended ADL does not have lower expressiveness compared to DFDs for any confidentiality mechanism or communication paradigm (control flow or data flow). The extended ADL shares the limitation of the DFDs regarding expressing tenants in systems, which lowers the metric value of the expressiveness of the ADL syntax to 0.75. However, this does not affect the validation results negatively because the metric value of the expressiveness of the DFD syntax is also only 0.75. Therefore, we can conclude that the expressiveness of the ADL is not lower compared to DFDs with respect to the syntax for expressing systems as well as for covering confidentiality requirements.

Correctness of Analysis Results (VQ9). The goal of the validation question is to show that the analysis results of the analysis framework of the ADL, on which the integration guidelines have been applied, are not less correct than the analysis results of analyses conducted in the DFDs semantics. Less correct means that there are systems, which have not been correctly classified as containing or not containing violations of confidentiality requirements. We can answer the validation question by comparing the presented values for the metrics VM9.1–VM9.4 with the corresponding metric values for DFDs (VM6.1 and VM6.2). The true positive fraction TPF is 1.0 for the DFD-based analysis results (VM6.1) as well as for the analysis results in ADLs using control flows (VM9.1) or data flows (VM9.3). The true negative fraction TNF is also 1.0 for the DFD-based analysis results (VM6.2) as well as for the analysis results in ADLs using control flows (VM9.2) or data flows (VM9.4). Because the metric values are identical, we could not show that the correctness of the ADL-based analysis results is lower than the correctness of the DFD-based analysis results.

Automation of Analyses (VQ10). The goal of the validation question was to show that the degree of automation is not lower for ADL-based analyses compared to DFD-based analyses. Because there are no purposes of analysis activities, which are no longer automated for ADL-based analyses, we can conclude that the degree of automation is not lower than for DFD-based analyses. In fact, all activities to be done in an analysis are automated for DFDs as well as ADLs.

Effort for Introducing Confidentiality Mechanisms (VQ11) All coefficients shown in Table 8.5 are greater than 0, which means that it was never necessary to recreate the whole model from scratch for introducing a confidentiality mechanism to an existing software architecture. This is a benefit compared to state-of-the-art approaches, which do not provide an integration into existing ADLs. The coefficients of the case study systems, which represent the same system and use the same type of communication, are close to each other. For instance, the coefficients of the pair CS1 and CS10, the pair CS2 and CS11 as well as the pair CS3 and CS12 only differ by 0.03 at most. This indicates that introducing information flow control using hierarchical lattices and introducing RBAC requires roughly the same amount of new model elements. The coefficient of CS17 has a greater differences to the coefficients of CS1 and CS10 (up to 0.19 for systems using control flows and up to 0.10 for systems using data flows) because we had to introduce explicit validation activities in the software architecture to support the taint analyses required by CS17. This means, we also had to adjust the structure of the system, which means additional new model elements compared to the other case study systems. We cannot quantify, how much additional modeling effort is required in this case based on the metric value because the modeling effort for creating two different types of model elements can be completely different. However, the goal of the validation was not to quantify the modeling effort but to show that software architects can reuse parts of the modeled software architectures. Because creating model elements usually requires more than no effort, we could show that our approach saves modeling effort compared to approaches that require remodeling the whole software architecture because of missing ADL integrations.

Effort for Switching Confidentiality Mechanisms (VQ12) Every coefficient shown in Table 8.6 is greater than 0, which means that it was never necessary to recreate the whole model for switching confidentiality mechanisms in a modeled software architecture. This is a benefit compared to state-of-the-art approaches, which only support one particular confidentiality mechanism and require the software architect to remodel the software architecture in another modeling language in order to use another confidentiality mechanism. The coefficients show that the reuse of existing models works particularly well when switching between information flow control using hierarchical lattices and RBAC. All corresponding coefficients are equal or greater than 0.75, which indicates a high degree of reuse. When switching to the combination of RBAC and taint analyses (CS17), the coefficients drop to a range from 0.50 up to 0.71. This is comprehensible because this new mechanism requires structural changes such as for introducing explicit validation of input data, which increases the amount of new model elements. The coefficients for case study systems using data flows (VM11.2) are higher than the coefficients for case study systems using control flows (VM12.1). This is also comprehensible because case study systems using data flows require roughly one hundred model elements more than the case study systems using control flows. The ratio of newly introduced model elements compared to the existing model elements is lower for systems using data flows than for systems using control flows. Therefore, the influence of the newly introduced model elements is lower for systems using data flows. We cannot quantify the modeling effort based on the metric value because the modeling effort for creating two different types of model elements can be completely different. However, the goal of the validation was not to quantify the modeling effort but to show that software architects can reuse parts of the modeled software architectures. Because creating model elements usually requires more than no effort, we could show that our approach saves modeling effort compared to approaches that require remodeling the whole software architecture when switching confidentiality mechanisms because of missing support of the new confidentiality mechanisms.

Availability of Information (VQ13) The validation question aims to validate that users of the ADL have access to the information required to use it. As discussed in Section 8.6.2, the values of VM13.1 as well as of VM13.2 are 0, which means that all required information is available to the software architect and the security expert while creating the architecture. This is true for ADLs using control flows as well as ADLs using data flows.

8.6.4. Threats to Validity

Because the major part of the validation took place as a case study, we structure the discussion of threats to validity according to the guidelines of Runeson and Höst [RH09, pp. 153] for discussing the validity of a case study. The categories to be discussed are also suitable for other types of validation designs, such as the discussion of the degree of automation or the availability of knowledge.

Internal Validity is concerned with how well a taken measure supports a cause-effect relationship and especially whether there are alternative explanations for the effect. In the context of VG4, we expect various parts of the ADL integration to be the cause of an effect. The effects are prohibiting degraded expressiveness, correctness and automation, reduced modeling effort for adding and switching confidentiality mechanisms as well as availability of information. We discuss potential alternative explanations of these effects, i.e. other possible influencing factors, in the following.

In the validation of the *expressiveness* (VM8.1–VM8.6), we expect the ADL syntax to be the cause of an effect. The expected effect is that the expressiveness is not degraded compared to the DFD syntax. We measure the amount of systems, which we could express in the extended ADL syntax, to determine expressiveness and compare it with the expressiveness of DFDs. However, there are other potential explanations of this effect, i.e. other possible influencing factors: The selection of case study systems and the analysis definitions for the systems influence the expressiveness because the more systems or analysis definitions, which cannot be expressed, are added to the selection, the worse the expressiveness gets. Because we focus on a comparison with the expressiveness of DFDs and their analysis definitions, we reuse the selection of systems and analysis definitions from the DFD validations (VQ1, VQ2 and VQ5). Therefore, the influence of the selection of systems and analysis definitions has the same effect on the expressiveness of the ADL as for DFDs, which means the effect can be ignored for the comparison and for answering the validation question. The skill in using the ADL is another influencing factor because missing skills can impede expressing systems, which leads to low expressiveness. We can exclude this factor because the person, who modeled the software architecture, is a maintainer of the Palladio ADL and also the author of the ADL extension. The skill of the developer of the ADL extension in using the ADL extension is most probably higher than the skill of an average software architect or security expert. However, this does not invalidate the results because we are interested in the upper bound of expressiveness. The chosen level of abstraction is another influencing factor on the expressiveness. If the level of abstraction is too high, a complex system can become expressible even if it misses important details. We can neglect this factor because the analysis results based on the modeled software architectures were correct as can be seen from the metric values VM9.1–VM9.4 and it is unlikely that the results are correct if the system omits important aspects, which are relevant for reasoning about confidentiality. Therefore, we expect the system models to represent all important aspects in enough detail.

In the validation of *correctness* (VM9.1–VM9.4), we expect the DFD mapping to be the cause of an effect. The expected effect is that the correctness of analysis results is not degraded compared to DFD-based analyses. We measure the true positive fraction of case study systems, which contain an issue and for which only valid violations are reported, as well as the true negative fraction of case study systems, which do not contain an issue and for which no violations are reported. However, there are other potential explanations of this effect, i.e. other possible influencing factors: The analysis framework for DFD-based analyses affects the analysis results because we use this framework after applying the mapping from a software architecture given in the ADL to a DFD. Therefore, the framework can also affect the correctness of the analysis results. However, we are only interested in comparing the correctness of analysis results between ADL-based analyses and DFD-based analyses. If the analysis framework for DFD-based analyses would yield incorrect results, the analysis results for the ADL-based analyses as well as for the DFD-based analyses would be affected in the same way. Therefore, we can exclude the analysis framework for DFDs as alternative explanation of the effect. The modeled software architectures are another potential influencing factor because they are the input of the mapping and therefore also a transitive input of the DFD-based analyses. We use the software architectures, which we modeled for answering VQ8. These software architectures are not fully equivalent to the DFDs because modeling them in Palladio requires adjustments. Differences between DFDs and software architectures given in an ADL potentially affect the analysis results. We cannot completely mitigate this factor but we do not expect a significant impact on the analysis results because we ensure that every reported violation is within the expected violations. If the modeled software architectures would omit an aspect, which is important to reason about confidentiality, the results would certainly be affected but it would be unlikely that all reported violations are still within the expected violations. Because all reported violations were within the expected violations, we assume that the software architectures cover all important aspects and do not influence the correctness by significant simplifications. The classification guidelines are another potential influencing factor because they decide whether the results for a case study system are classified as true positive or true negative. If the classification guidelines are wrong or not correctly applied, the metric values will be wrong. As discussed in Section 8.5.4, the classification guidelines are based on expected violations, which we motivated in Section 8.2.2. We apply the same classification guidelines as for DFDs. Even in presence of faulty classification guidelines, the results would still be comparable because the classification error happens on both types of results. Therefore, the classification guidelines are no alternative explanation of the validation results.

In the validation of the *automation* (VM10.1 and VM10.2), we expect the DFD mapping to be the cause of an effect. The expected effect is that the degree of automation of an analysis is not degraded compared to DFD-based analyses. We measure the purposes of analysis steps, which are no longer automated compared to DFD-based analyses. We already discussed factors, which influence the automation for DFDs in the discussion of threats to validity of VG3 in Section 8.5.4. The mitigation strategies also apply to automated analyses for ADLs. Even if one of these factors would not have been mitigated sufficiently, the factor would affect the DFD-based analyses and the ADL-based analyses in the same way. Therefore, the effect on the automated steps and the automated purposes would be the same, so a comparison would still be valid.

In the validations of the *modeling effort* (VM11.1–VM12.2), we expect the ADL syntax to be the cause of an effect. The expected effect is that the modeling effort is lower compared to state-of-the-art approaches, for which we assume that recreating software architectures from scratch is necessary. We measure the amount of model elements, which can be reused when adding (VM11.1 and VM11.2) or switching (VM12.1 and VM12.2) confidentiality mechanisms. However, there are other potential explanations of this effect, i.e. other possible influencing factors: The amount of model elements to be created depends on the person as well as the skill of the person, who models the software architecture. The amount of model elements varies because different persons may use different levels of abstraction, interpret the case study systems differently or express the same things in different ways. However, the validation neither expects nor requires that a model contains as less elements as possible because the validation only requires that at least one model element can be reused, i.e. that the metric value is greater than 0. It is unlikely that a person cannot use any single element just because of his/her skills in modeling systems. Therefore, the person, who models the software architecture, influences the metric values but not the conclusions drawn from the metric values. The method for creating the baseline models for VQ11 also affects the number of model elements, which have to be added, because the less elements are part of the baseline model, the more elements have to be added later. We consider keeping the model elements, which are commonly used in software architectures using Palladio, as realistic. The resulting software architectures are still valid Palladio architectures but without any information for predicting quality properties or violations of confidentiality requirements. In addition, the validation does not demand exact numbers because the validation only requires that at least one model element can be reused, i.e. that the metric value is greater than 0. Therefore, the creation of the baseline model influences the metric values but not the conclusions drawn from the metric values. The comparison procedure for two modeled software architectures also affects the number of changed model elements because every model element, which is found to be different, is added to the changed model elements. We use the established comparison procedure of EMF Compare [BP08] to avoid wrong comparison results. The matching of elements by identifiers is a common procedure and reasonable in the validation scenarios because the software architectures of the case study systems have been modeled by copying and adjusting existing models, which represent the same system. Therefore, the models, which we compare, really share identifiers. Nevertheless, the matching is not necessarily perfect because we cannot be completely sure that the adjustments of existing models have been done by a minimal number of changes. However, we already discussed that the validation does not depend on exact numbers, so a non-perfect comparison does not invalidate the conclusions drawn from the metrics.

In the validation of the *information availability* (VM13.1 and VM13.2), we expect the ADL syntax to be the cause of an effect. The expected effect is that all information required to create a software architecture including a confidentiality mechanism is available to the users of the ADL syntax. We discuss the required information and measure the amount of unknown information. However, there are other potential explanations of this effect, i.e. other possible influencing factors: The set of required information is essential for identifying missing information. If information is missing from the set of required information, the metric value can look more positive, i.e. lower, than it actually is. We avoid forgetting information by systematically collecting the required information to create or use for every single newly introduced type of model element in the ADL. Therefore, we can consider this potential threat as mitigated. The classification of required information as known or unknown also affects the metric value. To avoid a biased decision, we discuss the availability of every information and motivate its availability by references from literature or by demonstrating that there is a corresponding model element in DFDs, for which we already showed its availability as part of answering VQ4. Therefore, we do not expect the classification to be invalid.

External Validity is concerned with the generalization of results to other contexts. We used case studies and discussions to answer the validation questions of VG4. The results of discussions can be generalized to other contexts as we will discuss later. The results of case studies cannot be generalize to arbitrary other contexts because a case study does not use a representative sample but a limited set of cases. However, the results can be generalized to other contexts with comparable characteristics. We discuss these characteristics in the following.

The result of the *expressiveness* validation (VM8.1–VM8.6) is that the extended ADL does not limit the expressiveness compared to the expressiveness of DFDs. This finding is valid in other contexts if the following conditions hold in these contexts: First of all, the systems to be expressed have to use the same confidentiality mechanisms and the same features of the confidentiality mechanisms as the case study systems in the case study. We cannot draw conclusions for other mechanisms and features because we do not know whether the syntax can express the relevant parts of the system structure or behavior and we do not know whether the analysis definition can describe the relevant properties or whether it can express violations in terms of a label comparison function. We do not see the restriction of the confidentiality mechanisms and features as too limiting because we covered a broad range of commonly used confidentiality mechanisms in our case study. Second, the systems to be expressed either have to exchange data via parameters and return values in control flows or have to exchange data via data flows. If data flows are used, the order of the data flows has to be given by data dependencies and not by additional control flow instructions like suggested by Ward and Mellor [WM85] for designing and analyzing real-time systems. The restrictions of the data exchange is necessary because the ADL cannot express other types of data exchange. This is problematic if the part, which is not expressible, is relevant for reasoning about confidentiality. Third, the usage of the system has to be given by calls to the system, which exchange parameters and return values. The ADL does not support communication based on data flows between the user and the system. This is problematic if the part, which is not expressible, is relevant for reasoning about confidentiality. The restrictions of the data exchange inside systems and between systems and users is not too limiting because many ADLs use calls, which transmit parameters and receive return values, to specify the software architecture.

The result of the *correctness* validation (VM9.1–VM9.4) is that the ADL-based analyses do not yield results with lower correctness than the results of DFD-based analyses. This finding is valid in other contexts if the following conditions hold in these contexts: First of all, the software architecture has to be expressible in the ADL and the confidentiality requirements have to be expressible in a Prolog query. Otherwise, the inputs for an analysis are not available. Second, the same confidentiality mechanisms and features of the confidentiality mechanisms have to be used. We cannot draw conclusions for other mechanisms and features because we did not consider them in the case study. However, we do not see the restriction of the confidentiality mechanisms and features as too limiting because we covered a broad range of commonly used confidentiality mechanisms in our case study.

The result of the validation of *modeling effort* (VM11.1–VM12.2) is that the modeling effort is reduced compared to state-of-the-art approaches, which require full remodeling of the software architecture when adding or switching confidentiality mechanisms. We already discussed that the extended ADL reuses many parts for describing the structure, deployment, usage and behavior of a software architecture from the underlying, non-extended ADL. Therefore, a reuse is usually given by design. In addition, we demonstrated that reuse was possible for particular systems. These findings are applicable to other contexts as long as the software architectures can be expressed using the extended ADL and as long as the used confidentiality mechanisms are the same as used in the case study. We expect a reduced modeling effort for other confidentiality mechanisms as well but there might be confidentiality mechanisms, which require to restructure the whole system. For instance, the taint analysis used in CS17 requires structural changes because the software architecture has to integrate validation of incoming data, which requires additional actions or components. It is possible that other confidentiality mechanisms require many changes, which effectively forces a software architect to completely restructure the software architecture. However, we consider such a scenario unlikely for non-trivial software architectures because such a confidentiality mechanism would be hard to integrate and would most probably suffer from low acceptance in practice.

The result of the *automation* validation (VM10.1 and VM10.2) is that analyses based on the extended ADL are not less automated than DFD-based analyses. Because the validation took place by a general applicable discussion and the results do not depend on particular systems or application contexts, the results can be generalized to any application context of the extended ADL.

The result of the *information availability* validation (VM13.1 and VM13.2) is that software architects have all required information to use the extended ADL while creating and analyzing software architectures. Because the validation took place by a general applicable discussion and the results do not depend on particular systems or application contexts, the results can be generalized to any application context of the extended ADL.

Construct Validity is concerned with the appropriateness of taken measures to make statements about the research objective.

The validation questions VQ8 and VQ9 aim for comparing the expressiveness of the extended ADL and the correctness of the ADL-based analysis results with their counterparts for DFDs. The statements to be made are that the expressiveness and correctness are not worse than for DFDs. Because the validation questions are about comparisons with other validation results, it is necessary to use the same metrics as for rating expressiveness and correctness for DFDs. Therefore, we use the weighted ratio metric for the DFD metrics VM1.1–VM2.1 and VM5.1–VM5.3 as well as for the ADL metrics VM8.1–VM8.6. These metrics are appropriate to validate expressiveness as we already discussed in Section 8.3.4 and Section 8.4.4.

The validation question VQ10 aims for comparing the degree of automation of ADL-based analyses with the degree of automation of DFD-based analyses. The statement to be made is that the degree of automation of ADL-based analyses is not worse than the degree of automation of DFD-based analyses. We use the sum of no longer automated purposes (VM10.1 and VM10.2) for comparing the degree of automation. The metric is different to the metric used for rating the degree of automation of DFDs because simply comparing the number of not automated analysis activities is not appropriate to answer the validation question: For instance, the new analysis procedure might split analysis activities without introducing more work to be done to better fit the existing process for creating a software architecture. Just comparing the number of analysis steps would falsely report a lower degree of automation. In contrast, comparing purposes is more adequate because it allows matching activities between DFD-based analyses and ADL-based analyses. This matching allows identifying purposes, which are automated for DFD-based analyses but no longer for ADL-based analyses, which is what the validation question is actually about. The metric reports exactly these purposes. Therefore, the metric is appropriate to answer the validation question.

The validation questions VQ11 and VQ12 aim for rating modeling effort. The statement to be made is that adding or switching a confidentiality mechanism requires less modeling effort than doing the same for a state-of-the-art approach, which requires remodeling the whole software architecture. The Jaccard Coefficient is the metric used to capture the amount of model elements, which can be reused, i.e. which do not have to be created or changed. The metric is a commonly used metric for rating the similarity of two sets. Because we can interpret a model as a set of model elements, using the metric is valid. We explain this interpretation in the descriptions of the validation questions in Section 8.1.4. The amount of model elements to be created or changed can be used to reason about the modeling effort because creating a model element requires either no or more than no effort. However, it is more likely that creating a model element requires at least some effort. We discuss the relation between model elements and effort in Section 8.6.1 in more detail. Because the baseline for the metric is 0, i.e. the model has to be recreated from scratch, any value above 0 supports the validation statement mentioned above. Therefore, the particular effort implied by a model element is not important for answering the validation question. To conclude, the Jaccard Coefficient is appropriate to answer the validation question.

The validation question VQ13 aims for identifying information, which is required to create a software architecture using the extended ADL but which is not available. The statement to be made is that all information is available. The unknown information metric measures the sum of required information, which cannot be known by software architects and security experts. The metric provides the information to answer the validation question because it gives the sum of unknown information, which can be compared to the expected value 0.

Reliability is concerned with the dependency between the collected data and the conducting researcher. Best reliability is achieved if the collected data as well as the conclusions are completely independent of the conducting researcher.

We already discussed the reliability for most of the validation design and the metrics in previous validations. Therefore, we only briefly recap the most important aspects and refer to the previous explanations.

In the *expressiveness* validation (VQ8), it is not crucial that other researchers produce the same models of software architectures as we did because we aim for an upper bound of expressiveness. However, we provide all models in our data set [Sei22], so other researchers can check whether we really expressed the case study systems. Based on the provided data, other researchers can come to the same metric values. A more detailed explanation is available in the reliability discussion for the metrics VM1.1–VM2.1 in Section 8.3.4.

The results of the *correctness* validation (VQ9) are completely reproducible. We automated the execution of analyses as well as the classification of the reported violations. Other researchers can reproduce the results and check the classification criteria in the source code of our data set [Sei22]. A more detailed explanation is available in the reliability discussion for the metrics VM6.2 and VM6.1 in Section 8.5.4.

The results of the *information availability* validation (VQ13) are completely reproducible. We provide references to literature or provide the counterparts of ADL elements in DFDs, for which we already have shown that the required information is available. Other researchers can check our explanations and can then come to the same metric values. A more detailed explanation is available in the reliability discussion for the metric VM4.1 in Section 8.3.4.

The results of the *automation* validation (VQ10) are completely reproducible. We describe all analysis activities and provide the source code, which automates these activities in our data set [Sei22]. Therefore, other researchers can come to the same empty list of not automated analysis activities. Even if other researchers find other purposes, to which they can map the analysis activities, the final metric value of no longer automated purposes will still be the same because we automated all analysis activities. A more detailed explanation regarding the reliability of the identification of automated analysis activities is available in the reliability discussion for the metric VM7.1 in Section 8.5.4.

The results of the *modeling effort* validations (VQ11 and VQ12) are reproducible. We provide all models, which we use in the comparisons for calculating the Jaccard Coefficient, in our data set [Sei22]. Therefore, researchers can check that the used models correctly express the case study systems and that the baseline models for VQ11 are reasonable. For the validation, it is not important that other researchers produce the exactly same baseline models because we aim for an upper bound of modeling effort, which can be reduced. This means, other researchers only have to be able to check and understand the used models. The comparison of models as well as the calculation of the Jaccard Coefficient is fully automated by the validation application in our data set [Sei22].

8.7. Summary

In this chapter, we presented the validation of our contributions according to the GQM plan described in Section 8.1. A major part of our validation took place in case studies to get insights into the application of our contributions in particular contexts. Such a context always includes particular systems, i.e. software architectures. To sufficiently answer the validation questions in our case studies, we formulated requirements on the selection of case study systems and presented the selected systems in Section 8.2. In the following, we summarize the results of our validations. We structure the summary by the validation goals.

8.7.1. Validation Goal 1: Validate DFD Syntax

The validation of the DFD syntax aims to ensure that the syntax sufficiently answers the research questions, which it shall address. RQ1 and RQ2 ask for the necessary information in order to reason about information flow control and access control requirements in DFDs. Besides the information, which can already be represented in DFDs, we found properties of nodes and data as well as label propagation functions to be necessary. RQ3 asks what modeling primitives, i.e. syntax, is capable of expressing this required information. The DFD syntax provides the means to express the necessary information.

The validation to answer VQ1 as well as VQ2 demonstrated that the expressiveness of the DFD syntax is sufficient to model all seventeen case study systems except for one system. The case study systems stem from related approaches or at least represent common application scenarios for the used confidentiality mechanisms. The system, which the syntax could not express, requires modeling behaviors of individual users. However, considering individual users is out of scope of ADLs because it is at least questionable whether software architects have such detailed information about users while creating a software architecture. In addition, the related approach, which provides this system, uses analyses based on source code to identify violations. Therefore, we do not consider this case study system as a representative example of a software architecture, which shall be modeled and analyzed in the architectural design time. As already said, the syntax could represent all other case study systems. Because this remaining set of case study systems covers existing systems as well as the most commonly used confidentiality mechanisms, we conclude that the syntax is sufficient to express systems using commonly used confidentiality mechanisms.

The validation to answer VQ3 demonstrated that the elements of the DFD syntax are commonly used to model the case study systems. This means that the elements of the syntax are not only sufficient to express systems but also necessary. The only exception that we found stems from an intended degree of freedom in formulating expressions: We do not only provide the minimal functional complete set of logical operations, so it is possible to formulate equivalent expressions and avoid a logical operation completely. This does not indicate a useless concept because it is always possible to formulate the expression in a different way in order to use the previously unused operation.

The validation to answer VQ4 demonstrated that the information required to use the DFD syntax is available while creating the software architecture. We discussed the availability of information for every element of the syntax and supported the discussion by literature. The availability of information shows that the level of abstraction is not too low, i.e. there are not too many details required to use the syntax. On the other side, the level of abstraction is not too high because the syntax could express all case study systems, which are appropriate software architectures.

8.7.2. Validation Goal 2: Validate Analysis Definitions

The validation of the analysis definitions aims to ensure that the definitions sufficiently answer the research questions RQ5 and RQ6, which they shall address. RQ5 and RQ6 ask for ways to formalize access control and information flow control analyses based on the DFD syntax and semantics. We presented label comparison functions based on Prolog queries and optional additional clauses to represent specific confidentiality requirements as solution. We provided analysis definitions for commonly used confidentiality mechanisms, which consist of characteristic types, behaviors of nodes, the label comparison function given as Prolog query and optional additional Prolog clauses to cover specific confidentiality requirements.

The validation to answer VQ5 demonstrated that the analysis definitions can express all confidentiality requirements of all case study systems, which we can express using the DFD syntax. The analysis definitions were often sufficient without modifications. The only modifications, which were necessary, are changes of the number of inputs and outputs for behaviors and the creation of additional Prolog clauses. Changing the number of inputs and outputs does not imply limitations of the analysis definitions because the label propagation logic remains the same. For instance, finding the highest label of all inputs essentially works the same for any number of inputs. The additional Prolog clauses always followed the suggested structure in the analysis definitions. Because these clauses cover specific confidentiality requirements and this is intended by the analysis definition, this also does not imply a limitation. We conclude that the analysis definitions are expressive enough to cover the confidentiality requirements of the modeled case study systems.

The validation to answer VQ6 demonstrated that the analysis definitions can correctly identify systems, which violate the given confidentiality requirements. We executed analyses using the analysis definitions on variants of the case study systems, which do not contain an issue, and on variants, which contain an issue. The analyses could successfully identify all variants, which contain an issue, and did not report violations for variants, which do not contain an issue. We conclude that the analysis definitions provide a formalization, which is sufficient for automated analyses and is sufficient for correctly identifying violations.

8.7.3. Validation Goal 3: Validate DFD Semantics

The validation of the DFD semantics aims to ensure that the semantics are sufficient to answer the research question RQ4, which they shall address. RQ4 asks for semantics of the DFD syntax, which allow detecting violations of confidentiality requirements. We defined semantics, which describe the meaning of the DFD syntax in terms of a label propagation network.

The validation to answer VQ6 demonstrated that the semantics support analyses, which correctly identify systems that violate the given confidentiality requirements. We executed the analyses for all expressible case study systems and could correctly classify each variant of a case study system as either violating or not violating confidentiality requirements. We conclude that defining properties of nodes and data as labels and propagating these labels are good semantics to identify violations.

The validation to answer VQ7 demonstrated that the semantics are sufficient for driving automated analyses. We discussed the automation of every activity in an analysis and showed that the activities can be automated in our prototypical implementation. For the semantics, this means that they provide unambiguous definitions, which do not require human intervention. We conclude that the semantics do not limit the automation of analyses.

8.7.4. Validation Goal 4: Validate ADL Integration Guidelines

The validation of the ADL integration guidelines aims to ensure that the guidelines are sufficient to answer the research questions, which they shall address. RQ7 asks how to integration the DFD-based analyses in ADLs focusing on control flows. RQ7 asks the same for ADLs focusing on data flows. We provided guidelines on how to integrate the DFD-based analyses for both types of ADLs.

We validated particular applications of the integration guidelines instead of the guidelines themselves because validating guidelines involves many human factors, which impede objective validations. Instead, we applied the guidelines to the existing Palladio ADL, validated the quality of the integration into the Palladio ADL and showed that the guidelines can produce usable integrations. This is no verification of the quality of the integration guidelines but a failed attempt to falsify the hypothesis that the integration guidelines do not produce a usable integration. In the following, we summarize the validations of the integration results, i.e. the integrations into Palladio. We refer to the integration into the part of Palladio, which focuses on control flows, as a dedicated integration and to the integration into the part of Palladio, which focuses on data flows, as a dedicated integration.

The validation to answer VQ8 demonstrated that the integrations are as expressive as DFDs. We attempted to model all seventeen case study systems including the used confidentiality mechanisms in both integrations. We could successfully model all but one case study system in the integrations. We failed to model the same case study system in the integrations as we already failed in DFDs for the same reasons. We conclude that the integrations do not limit the expressiveness compared to DFDs.

The validation to answer VQ9 demonstrated that the analyses within the integrations deliver results as correctly as DFD-based analyses. We executed the same analyses as for DFDs on two variants for every expressible case study system: one variant contains an issue and the other variant contains no issue. The analyses correctly reported all variants, which contain violations of confidentiality requirements, and correctly classified all variants, which do not contain violations, as not containing violations. The analysis results were as correct as the results of the DFD-based analyses. We conclude that the integrations do not limit the correctness of results compared to DFDs.

The validation to answer VQ10 demonstrated that the integrations support fully automated analyses. We collected all analysis activities and showed that they can be automated by a prototypical implementation of the integrations. We conclude that the integrations do not limit the automation of analyses compared to DFDs.

The validation to answer VQ11 demonstrated that adding a confidentiality mechanism to an existing software architecture is possible without remodeling the whole software architecture. For all systems, which are represented by multiple case study systems, we selected all corresponding case study systems and built a variant, which does not contain a confidentiality mechanism. We then compared the case study systems with their variant to identify the amount of model elements, which are shared, i.e. which have been reused. For all case study systems, a considerable amount of model elements could be reused. In contrast, dedicated analysis approaches, which are not integrated into existing ADLs, require remodeling the whole architecture in a new approach. Because creating a model element usually requires at least some effort, we conclude that the integrations save modeling effort when adding confidentiality mechanisms to existing architectures.

The validation to answer VQ12 demonstrated that switching to another confidentiality mechanism in an existing architecture is possible without remodeling the whole software architecture. For all systems, which are represented by multiple case study systems, we selected all corresponding case study systems and compared these systems pairwise. Every comparison detected a considerable amount of shared model elements, which means these model elements can be reused. In contrast, analysis approaches or ADLs, which only support a single confidentiality mechanism, require remodeling the whole architecture when switching the confidentiality mechanism. Because creating a model element usually requires at least some effort, we conclude that the integrations save modeling effort when switching confidentiality mechanisms in existing architectures.

The validation to answer VQ13 demonstrated that the information required to use the integrations is available while creating the software architecture. We discussed the availability of information for every element of the integrated syntax by either referring to literature or demonstrating the equivalence to DFD elements, for which we already have shown the availability of information. The availability of information shows that the level of abstraction is not too low, i.e. there are not too many details required to use the syntax. On the other side, the level of abstraction is not too high because the syntax is expressive as shown in VQ8.

9. Related Work

Developing secure software is a broad topic because security has to be considered in every development phase [McG06, p. 110], which means many different development activities influence the security of a software system. There is a lot of research to improve certain or all development phases by processes, analyses, mechanisms, and so on. In addition, multiple security objectives such as confidentiality or integrity have to be achieved to secure a software system and such objectives require different measures to be taken. Consequently, giving a complete overview on the topic is unfeasible.

We focus on research on achieving confidentiality in software architectures or software designs because the approach presented in this thesis also aims for confidentiality and it belongs to these phases. Despite our focus on confidentiality, we use to the more general term *security* in the following if using *confidentiality* would be uncommon. For instance, we refer to *security patterns* instead of *confidentiality patterns* in the following because the former term is widely known and accepted. However, we still focus on how the approaches can improve confidentiality instead of discussing the effect on further security objectives.

The most prominent directions of related research are integrating confidentiality and rating confidentiality. Integrating confidentiality describes approaches to represent confidentiality mechanisms in software architectures or software designs but also approaches to use such information to support the integration of confidentiality in following phases. These approaches are related to our approach because they often use confidentiality analyses like ours to rate the effectiveness of integrated confidentiality measures. We discuss approaches for integrating confidentiality in Section 9.1.

Rating confidentiality means that approaches analyze software architectures or software designs for threats or violation of requirements and report on the results. The results provide software architects with the information to decide about necessary changes and the implications of these changes. We discuss approaches for analyzing confidentiality in Section 9.2.

In addition to the discussion of full-fledged approaches for improving the confidentiality in software architectures or software designs, we also discuss related work for individual parts of our approach. There have been various attempts to define DFD semantics in order to address shortcomings or to make DFDs applicable to new problem domains or analyses. We discuss such previous attempts and relate them to the DFD semantics presented in this thesis in Section 9.3.

Besides dedicated analysis approaches, there is related work that integrates confidentiality analyses into existing ADLs. Because we also provide an ADL integration, we discuss such approaches and compare their integration approach with ours in Section 9.4.

9.1. Approaches to Integrate Confidentiality

In this thesis, we focus on identifying violations of confidentiality requirements by the structure, behavior, deployment or usage of a specified software architecture. For instance, our approach shall detect that a software architecture requires users to transmit confidential data to non-trustworthy system parts in order to provide certain functionality. This means, the software architecture violates confidentiality requirements by its intended usage. Based on this knowledge, software architects can adjust the software architecture to provide the functionality but also meet the confidentiality requirements. In contrast, approaches to integrate confidentiality into software architectures focus on the integration of enforcement mechanisms for confidentiality requirements or countermeasures for attacks against confidentiality. However, the approaches for analyzing confidentiality do not completely ignore the part of enforcing confidentiality and vice versa. The approaches for integrating confidentiality usually also analyze the effectiveness of the integrated mechanisms and the approaches for analyzing confidentiality usually consider the effect of integrated confidentiality mechanisms in their analysis results. In the following, we discuss the relations of approaches to enforce confidentiality to our analysis approach. The most prominent approaches to consider the enforcement of confidentiality in software architectures are security patterns, which we briefly discuss in Section 9.1.1, and code generation, which we briefly discuss in Section 9.1.2.

9.1.1. Security Patterns

Security patterns provide generic solutions for recurring security problems in certain contexts according to Schumacher et al. [Sch+06, p. 31]. For instance, *Secure Channels* [Sch+06, p. 79] is a security pattern, which protects the confidentiality of information, which is transmitted over public networks, by encrypting the information. The security patterns are meant to help software architects with and without security expertise to avoid common security problems. Many research in security patterns is about building catalogs of these patterns and to improve their quality [Lav+06].

Our approach does not aim for providing such patterns or for general applicable solutions of integrating security in general or confidentiality in particular into software architectures. However, our approach considers the effect of integrated security patterns on meeting the confidentiality requirements. For instance, encryption can lower the classification of exchanged data and can avoid a node getting data classified higher than its own clearance. Ignoring such effects would lead to falsely reported violations.

Besides the research on patterns, there is also research on ensuring that patterns are correctly applied and effective. Taspolatoglu and Heinrich [TH16] suggest to formalize the prerequisites for applying a pattern, to link these prerequisites to the software architecture and to check whether all prerequisites are met. Heyman, Scandariato, and Joosen [HSJ12] verify that the patterns work as expected by verifying that defined preconditions and postconditions hold. It is, especially, useful to verify the effectiveness of patterns in the context of software evolution because prerequisites might no longer be met because of changes in the architecture or the context of the architecture. SecVolution [Bür+18] provides an approach to systematically capture such changes, identify implications on the security and address them by adaptions in the software architecture.

The previously described approaches all use analyses to identify degraded effectiveness of the integrated mechanisms. Our approach can be used as one of these analyses: For instance, an analysis of violated confidentiality requirements can reveal an issue in applying a security pattern for improving confidentiality. Such a combination is, especially, useful to complement coarse-grained analyses such as the analysis suggested by Taspolatoglu and Heinrich [TH16] because their analysis only identifies if the pattern is applied correctly but not if it effectively protects confidentiality.

9.1.2. Generation of Development Artifacts

The generation of development artifacts such as code or configuration files based on models of software architectures or software designs is frequently used. In model-driven software development, code generation is actually essential according to Stahl and Völter [SV06, Sec. 2.3]. Generating code has many benefits but the most important benefits in the context of enforcing confidentiality are that the generated code matches the software architecture [SB12] and it can contain additional information for introspection [SV06, Sec. 9.1]. Matching the architecture is beneficial because the considerations for protecting confidentiality, which have been made in the software architecture, could be invalid in the implementation, otherwise. The information for introspection is beneficial because information, which is only available in the software architecture but which is also important for protecting confidentiality, can become part of the implementation. In the following, we give examples on approaches, which use code generation, and how they use the benefits of code generation for protecting confidentiality. We also discuss the relation to the approach presented in this thesis.

SecureUML [LBD02] provides a UML profile to annotate roles and access rights specified in RBAC requirements to software designs given in UML models. The major purpose of the annotations is to generate a RBAC policy, i.e. enforceable RBAC requirements, and an enforcement platform for the policy. The expected benefits are increased productivity and quality of systems but also better consistency between the modeled policy and the actually used policy. In contrast to our approach, SecureUML does not analyze the software design and can, therefore, not identify violations of RBAC requirements in the UML model.

Hoisl, Sobernig, and Strembeck [HSS14] also provide a UML profile to annotate existing UML models but focus on information flow in UML activity diagrams. The objects transferred via object flows can be annotated as sensitive and the nodes can be marked as preserving confidentiality. An analysis propagates the objects through object flows and ensures that no sensitive object traverses a node not marked as preserving confidentiality. The code generation produces security configurations for web services specified in the Web Services Description Language (WSDL). In contrast to our approach, Hoisl, Sobernig, and Strembeck prescribe the behaviors of node types, which is sufficient for taint analyses but is not capable of describing more complex behavior such as the declassifications in our running example. Consequently, the approach only supports information flow control using a linear ordered lattice containing two levels.

The *Component Information Flow* (CIF) toolkit [Abd+11] allows to annotate connectors and attributes of components in a software architecture with classification levels. The confidentiality requirements are given in terms of non-interference, which means that information with a certain classification level must only flow to a connector or attribute with at least this classification level. CIF ensures that the requirement is not violated by wrong connectors in the software architecture. CIF generates Java code based on the software architecture, which includes code to perform information flow control, i.e. enforce the confidentiality requirements. In contrast to our approach, CIF requires source code to identify violations caused by the data processing within components and only focuses on information flow control.

iFlow [Kat+13] aims for generating deployable source code for apps and web services, which adhere to information flow requirements. They specify the software architecture by UML models and the operations by a DSL. Annotated UML models represent the information flow requirements, which are given as non-transitive non-interference, which essentially is non-interference using declassifications to allow certain information flows. Analyses of the UML models propagate data and can detect violations of the information flow requirements. In addition, verification of the generated source code is possible. In contrast to our approach, iFlow relies on communication based on call and return and focuses only on information flow control. In addition, it operates on a different level of abstraction because operation specifications in the DSL have a considerable higher complexity than our specifications of the label propagation. Consequently, it is at least questionable whether such detailed information is available while creating a software architecture.

To summarize, there are various approaches, which generate policies, source code stubs or complete applications. Most of these approaches perform analyses on the used input models to identify potential confidentiality violations before generating artifacts. Because these analyses often are only available to support the code generation, they are limited with respect to the supported confidentiality mechanisms and the complexity of confidentiality requirements. In contrast, our approach does not focus on code generation but on analyzing the modeled software architectures. Therefore, we support more confidentiality mechanisms and more complex confidentiality requirements.

9.2. Approaches to Analyze Confidentiality

There is a wide range of approaches to analyze the confidentiality of software architectures or software designs. We see three groups of approaches, which are related to the approach presented in this thesis. Each group has a different focus and purpose.

The first group focuses on identifying threats, i.e. a vulnerability and an attacker, who exploits the vulnerability. The goal of approaches in this group is to identify threats, estimate the impact of the threat and decide how to mitigate the threat. We discuss this first group in Section 9.2.1.

The second group focuses on identifying violations in the software architecture. This group is different to the first group because it does not require an active attacker but aims to reveal problems in the software architecture itself. For instance, if the architecture prescribes sending sensitive data to a non-trustworthy node in order to provide functionality, this is an issue of the software architecture independent of any attackers. We discuss this second group in Section 9.2.2.

The third group focuses on quantifying confidentiality by metrics, which are called *security metrics*. This group often aggregates results from the previous two groups and combines them in a metric. A metric is useful in trade-off decisions to quantify a benefit for confidentiality and relate this benefit compared to other factors such as required investments. We discuss this third group in Section 9.2.3.

9.2.1. Identification of Threats

The identification of threats is a common security-related activity on software designs and software architectures. *Threat Modeling* [Sho14] is a commonly used term summarizing many different approaches to identify and potentially also mitigate threats. According to Bedi et al. [Bed+13], "threat modeling provides a structured way to secure software design by allowing security designers to accurately estimate the attacker's capabilities in respect of known threats". In a survey, Xiong and Lagerström [XL19] found that many activities of threat modeling are still done manually. Contributions such as STRIDE by Microsoft [Her+06] support this manual work by guidelines on how to identify threats but there are also approaches (partially) automating the threat modeling process.

The focus of threat modeling approaches is different compared to our approach: Threat modeling aims to identify threats and to harden the system in order to mitigate attacks. Instead, our approach focuses on revealing design issues, which lead to violations of confidentiality requirements by the system or usage of the system itself, i.e. not by a malicious user or attacker. Nevertheless, such identified violations can indicate vulnerabilities that attackers can use to compromise the system. This means, our approach provides one piece of information to identify threats.

In the following, we give an overview on threat modeling approaches and relate these approaches to the approach presented in this thesis.

Manual Analysis of DFDs Threat modeling does not prescribe a certain way of identifying potential threats nor does it prescribe particular types of models to be used. However, DFDs are commonly used because they are intuitive and recognizing threats in data flows is often better than in control flows [Sho14, pp. 43]. Inspecting such a DFD manually allows to flexibly incorporate additional knowledge from other sources or allows to address missing information by reasonable assumptions. In contrast to our approach, the results of manual threat modeling on non-extended DFDs heavily depend on the expertise of a threat modeler as well as on the availability of additional information, which is not part of the DFDs.

Manual Analysis of Extended DFDs The disadvantage of the simple DFD syntax is that much information has to be stored outside of DFDs because it is not expressible within DFDs. Many threat modeling approaches extend the syntax of DFDs to cover additional information. Yampolskiy et al. [Yam+12] extended DFDs to fit the need of threat modeling for cyber-physical systems. The extensions include distinctions between physical and virtual elements as well as elements to specify details of the physical and logical communication. Deng et al. [Den+11] also extend DFDs but focus on capturing threats and mitigations regarding privacy. Both extended DFDs do not come with automated analyses in contrast to our approach.

Automated Analysis of Extended DFDs DFD extensions not only allow representing additional information but also enable automated threat analyses. Berger, Sohr, and Koschke [BSK16] introduce data channels, trust areas and typed data flows. They can identify threats stemming from the databases Common Weaknesses Enumeration (CWE)[1] or Common Attack Pattern Enumeration and Classification (CAPEC)[2] by an automated analysis. Abi-Antoun, Wang, and Torr [AWT07] also extend DFDs and provide an automated analysis but focusing on threats according to STRIDE [Her+06]. *SPARTA* [Sio+18a; Sio+18b] extends DFDs by means to make integrated security or privacy solutions explicit. The provided automated analyses identify threats and calculate risks. Frydman et al. [Fry+14] extend the DFDs by attributes such as the asset value and use a catalog of attack patterns to identify threats in an automated analysis. In addition, they calculate risk values. All mentioned automated analyses on the extended DFDs do not consider the behavior of system parts to derive properties of exchanged data. In consequence, modelers have to define the properties for all exchanged data manually. This means, the automated analyses are restricted to pattern matching. In contrast, our approach propagates such data properties, which lowers the amount of required specifications of data properties significantly.

[1] `https://web.archive.org/web/20220217013953/https://cwe.mitre.org/`
[2] `https://web.archive.org/web/20220217013947/https://capec.mitre.org/`

9.2.2. Identification of Violated Confidentiality Requirements

Confidentiality requirements specify who is allowed to know which information. This includes users of a system as well as the parts of the system itself. During the implementation and runtime of a system, confidentiality mechanisms like access control enforce such requirements by allowing or denying access at enforcement points such as the interface to a system. This is necessary because there can be many different types of users, which also includes users, which have not been foreseen when designing the system. While creating the software architecture or software design, the situation is different. Software architects specify how the system provides its services and how users use these services. Therefore, the software architecture only contains allowed access requests to information. Software architects have to avoid violations of confidentiality requirements by the structure, behavior, deployment or usage of the software architecture because the underlying issues would be implemented in later stages. Consequently, identifying confidentiality violations in a software architecture means identifying violations caused by issues in the software architecture. In contrast to threat modeling, this does not require an explicit attacker model.

Various approaches [Ngu+15] to identify violated confidentiality requirements in software architectures or software designs are available. Because of the sheer amount of available approaches, we only consider closely related approaches, which support propagating properties of exchanged data, in the following. We distinguish approaches analyzing control flows and approaches analyzing data flows.

9.2.2.1. Control Flow Analyses

Control flow analyses analyze software architectures using call-and-return communication. Such approaches require representations of software architectures, which are more detailed than DFDs. Instead of just describing the required data and its processing, call-and-return communication requires software architects to specify execution orders of activities. The execution order and control flow specifications enable the detection of implicit information flows at the cost of additional complexity. Our approach aims for lower complexity as well as simple, data-oriented specifications of confidentiality requirements. Besides this fundamental difference, there are additional differences to the analysis approaches using control flows, which we describe in the following.

iFlow [Kat+13] is an approach to verify UML models and generate application code based on the models. An UML profile allows to annotate components, which describe the structure of the system, as well as elements of sequence diagrams, which describe the communication between components. The behavior of components is specified in a DSL. The confidentiality requirements are given by a lattice. iFlow can identify violations of non-interference requirements, i.e. the lattice, by transforming the UML models into input for the theorem prover KIV. To identify violations, the analysis determines the classification level of exchanged information based on the processing in the components. Our approach also propagates classification levels through the system but uses behavior descriptions given by assignments, which have lower complexity than the DSL used in iFlow, to derive the effect of processing. In addition, our approach also supports access control besides information flow control.

Gerking and Schubert [GS19] present an approach to verify that models given in MechatronicUML [SW10], which is a UML profile for representing mechatronic systems, do not violate information flow requirements given in terms of non-interference. The approach provides refinement relations between components, i.e. how to break down a big component into smaller components, in order to support modular verification of the components. Software architects specify the behavior of components as timed automata. By simulating the behavior, the approach can verify that publicly visible behavior does not change when private behavior changes. This simulation also involves propagating data through the system and keeping track of its classification. In contrast to our approach, the behavior specifications are more complex because they are stateful and consider time. Specifying behavior in this detail is at least challenging while creating a software architecture because it requires a deep understanding of the behavior of a component. In addition, our approach supports analyzing access control, which the approach of Gerking and Schubert does not.

UMLSec [Jür05] is one of the most commonly known approaches for analyzing software designs for security issues. UMLSec extends UML by a profile that introduces properties, such as the criticality of system parts and calls, for reasoning about security in general but also confidentiality in particular. The approach is capable of identifying violations of non-interference requirements using a high/low lattice as well as violations of requirements given in terms of RBAC. The analyses take place in UML Machines, which are transition systems with states using algebraic structures. Using CARiSMA [Ahm+17], users can define own analyses and extend the existing capabilities. In contrast to our approach, UMLSec has not been shown to support a range of access control mechanisms, which is as wide as the supported access control mechanisms of our approach. In addition, UMLSec defines the RBAC requirements based on actions instead of data, which intertwines requirements with the system design and, therefore, makes changing the system design more complex. In contrast, our approach supports specifying the confidentiality requirements independently of the system design and only requires the assignment of roles to users or groups of components.

Hoisl, Sobernig, and Strembeck [HSS14] describe an approach to systematically consider confidentiality in Service-Oriented Architectures (SOAs). The starting point is a UML model extended by a UML profile. The profile allows marking object flows in activity diagrams as containing sensitive data and nodes in such a diagram as preserving confidentiality. The behavior of the involved actions is predefined and essentially follows the logic of a taint analysis. Nodes affecting the control flow, such as decisions or forks, are an essential part of the analysis. A node marks an outgoing object flow as containing sensitive data if one of the incoming object flows contains sensitive data. The corresponding analysis is specified in OCL and reports a violation if a node, which does not preserve confidentiality, receives an object flow containing sensitive data. This means, the analysis supports non-interference requirements given by a lattice consisting of two levels. In contrast, our approach supports more sophisticated lattices and uses customizable definitions of node behaviors to support more complex analyses of non-interference as well as access control.

9.2.2.2. Data Flow Analyses

Data flow analyses detect violations of confidentiality requirements in software architectures based on exchanged data. The analyses use networks of processing steps, which each have inputs and outputs as well as a behavior. The propagation and comparison of data properties is a core element of these analyses. The approaches providing such analyses are closely related to our approach. In the following, we describe the most important approaches and describe the differences to our approach.

AuthUML [AW03] and FlowUML [AFW06] originate from the same authors. Because both approaches share many aspects, they can be considered as one approach. The approaches aim to report violations of confidentiality requirements given in terms of access control as well as in information flow control for software designs given in UML. In a first step, the approach extracts activities of users from use case diagrams and data flows between actors or system parts from sequence diagrams. This information is formalized in a logic program given in Prolog. The user of the approach has to specify sensitive information and the confidentiality policy, which is added to the logic program in order to identify violations. The approaches report on supporting access control using DAC, MAC and RBAC as well as information flow control. The approaches have many similarities with our approach such as the use of logic programming to identify violations or the use of annotated software models as inputs. However, neither AuthUML nor FlowUML report on a validation, which shows that any of the approaches supports systems other than the small running examples. An implementation is not available. In addition, the support of access control mechanisms is only described vaguely: For MAC, only the military access model is considered. For DAC, the delegation of rights is missing. The handling of declassification, which is essential to support realistic systems according to Zdancewic [Zda04], is not mentioned. Because of the missing validation and the shortcomings in representing the confidentiality mechanisms, we do not consider the approaches to have shown a successful combination of access control and information flow control.

SecDFD [TSB19] uses extended DFDs to identify violations of non-interference requirements. The extensions of the DFDs are an additional property to indicate if a node is within a trusted zone or an attack zone as well as a predefined set of nodes including a behavior. The behavior is given in terms of a label propagation function that takes incoming labels and maps them to outgoing labels. Apart from the fixed node types, the approach follows the same analysis logic as our approach. Both approaches [TSB19; SHR19] have been developed independently and have been published at the same venue in 2019. The major difference is the support of confidentiality requirements. SecDFD supports non-interference with a high/low lattice and can consider keyless encryption.

In addition, our approach supports non-interference using arbitrary lattices, encryption using key pairs and various access control mechanisms.

Berghe et al. [Ber+17] represent systems by a network of predefined processing operations, which propagate data and its properties, in order to identify illegal access to data. A processing operation receives data, calculates attributes of outgoing data based on the attributes of incoming data and sends the data. Essentially, this is label propagation. The behavior is specified in the language of the theorem prover Coq. Consequently, the detection of violations also is done in Coq. The formalization is more powerful than the formalization presented in this thesis because it supports stateful modeling and uses linear-time temporal logic. While this increases the expressiveness, it also increases the complexity of the specification. It is at least questionable if the detailed information for creating such detailed specifications is actually available while creating the software architecture. As already discussed in Section 8.3.3, we favor simple specifications over expressiveness of all possible cases. The approach of Berghe et al. demonstrates the support for a simple form of non-interference, which is comparable with a taint analysis. In contrast, our approach supports non-interference using sophisticated lattices as well as various access control mechanisms.

9.2.3. Calculation of Security Metrics

There is a high demand for security metrics, which quantify the security of software systems [Fla18]. These metrics shall help answering trade-off questions such as trade-offs between gained security and budget to be spent [PC10]. In our approach, we do not aim for quantifying security because creating a reliable metric that satisfies the expectations on these metrics is a different field of research. However, our approach can provide one of many possible inputs for calculating security metrics.

In the following, we give examples of how our approach could provide inputs for two approaches providing security metrics for software architectures. For a more exhaustive list of security metrics, we refer to surveys on security metrics [MFP10; Pen+16].

Busch, Strittmatter, and Koziolek [BSK15] suggest the *mean time to security failure* as a metric for trade-off decisions in software architectures. The metric considers factors like the skills of an attacker, the time spent in securing a component or security interferences between components. The approach is probabilistic and requires estimations of factors and probabilities. To improve these estimations, the results of the approach presented in this thesis could be used. For instance, our approach can provide information about security interferences between components by tracing, which information is transitively exchanged between components.

Almorsy, Grundy, and Ibrahim [AGI13] provide an extensible framework to calculate security metrics. They use system descriptions and corresponding security specifications as input. Success conditions of various attacks are given in OCL. Metric calculations can include arbitrary information but also the results of testing the success conditions by building weighted sums. There are examples of metrics such as the attack surface or the compartmentalization but the authors themselves stress that these are not complete and have to be extended. Because of the high flexibility in calculating metrics and considering inputs, it is possible to consider our analysis results in the metric calculations. The analysis results of our approach indicate structural problems of the software architecture, which can be considered in the attack surface, for instance.

9.3. DFD Semantics

The major benefit of DFDs is their simplicity. A DFD only consists of three types of nodes (external actors, processes and stores) and one type of edge (data flow). DeMarco [DeM79] describes the semantics of these syntactical elements in an intuitive way. This simple description of semantics is not really problematic for discussing diagrams with stakeholders. However, weak semantics impede automated processing of DFDs, which usually relies on clear meanings for every syntactical element as well as combinations of these elements. The semantics are a commonly identified weakness of DFDs. As a survey [Jil+08] on formal semantics for DFDs shows, there are many different approaches to address the shortcomings of the DFD semantics. These approaches often do not only extend or redefine the semantics but also extend the DFD syntax. This is reasonable because additional information can be necessary to solve ambiguities in the semantics and this additional information has to be expressed in the DFD syntax. The semantics are not particularly tailored to confidentiality or security but still point out and address important shortcomings, which affect many domains including confidentiality. In the following, we report on the four most commonly mentioned shortcomings, motivate their importance on the running example, discuss suggested solutions and relate these solutions to the solution, which we used for defining our DFD semantics. The explanations are based on one of our previous publications [Sei+22].

Node Properties DeMarco [DeM79] sees the importance of additional properties and suggests adding them to a data dictionary. However, he does not prescribe a specific format or typing for these properties, so automated processing of the properties is challenging. Representing properties of nodes is important in our running example in order to capture the clearance levels of processes, stores and actors. The clearance level is necessary to identify violations of the non-interference requirement that nodes can only access data if their clearance level is at least the classification level of the data. Some execution semantics [Fra92] [Pet+94], i.e. semantics that specify the execution order of processes, describe properties of nodes as part of a global execution state. Consequently, the properties of nodes can change dynamically over time. Instead of these dynamically changing node properties, we use statically assigned node properties to represent information required to detect violations of confidentiality requirements. We favor these static properties over dynamic properties because dynamic properties need a definition of state transitions and this introduces additional complexity to the modeling process. However, we have demonstrated that dynamic properties are not necessary to express the information required to identify violations of commonly used confidentiality requirements.

Multiple Inputs DeMarco [DeM79] is aware of potential different meanings of multiple incoming data flows but either refers to an intuitive understanding based on the name of a process or suggests describing the process in a data dictionary. However, he does not prescribe a specific format for this description, which makes automated processing is challenging. Our running example also includes situations, in which multiple incoming data flows have different meanings: The meaning of the two inputs into the *book flight* process in Figure 3.2 on page 23 is that both inputs are required for the booking. The meaning of the two inputs *ccd* and *declassifiedCcd* to the user is that either of the two is required, i.e. the inputs are alternative flows. It is important to distinguish these meanings because the DFD shown in Figure 3.2 would always violate the non-interference requirements by always picking both inputs of the credit card data instead of only violating it when selecting the wrong input. Tracing back the issue becomes harder if the violation is reported in more situations than necessary. Related semantics address this shortcoming differently. A simple approach is to always assume that all incoming data flows to a node are mandatory. The benefit of such semantics [Fen+93; LPT94; Pet+94; Xio+17] is that they do not require extensions of the DFD syntax. However, such semantics do not allow expressing our running example. Other semantics [Fra92; LT91; PKP91; WBL93; Lea+96; LWB99] provide a more flexible approach by supporting definitions of valid combinations of incoming data flows via preconditions or sets. Such semantics can express our running example. However, the specification via preconditions or sets implies considerable complexity if an additional, mandatory incoming data flow shall be added. Adding such a mandatory data flow is not unlikely because it is comparable with adding an additional parameter to an operation signature, which occurs quite frequently when designing systems or writing source code. Instead, we aim to simplify such definitions by the notion of pins. Pins are already known from UML [Obj17, p. 444], in which they define inputs and outputs of nodes. The pins define mandatory input data, which means that there has to be an incoming data flow to every input pin. If multiple incoming data flows are available, these data flows are alternatives but at least one of these alternatives always has to be used. Adding another mandatory input only requires adding a new input pin and does not require to change other pins or data flows.

Behavior of Nodes DeMarco [DeM79] sees the need to describe the behavior of nodes and uses entries in a data dictionary to cover this information. However, he does not prescribe a structure or specific typing of the information, so processing this information automatically is hard. The behavior of nodes is necessary to reason about data processing and its effect. Knowing the effect of data processing is not only important for our running example but it is essential for the propagation of data properties. Without knowing the processing effect, we would have to manually classify every exchanged data flow in the running example to know the classification of arriving data and to compare it with the clearance of nodes. Semantics [KBB86; BW89; Pet+94; Xio+17], which do not require the propagation of properties or only use very simple properties, do not describe the behavior of nodes but use the semantics for handling multiple inputs to derive necessary properties. This approach is not capable of expressing the propagation of the classification level in our running example. Semantics [PKP91; Fra92; Fen+93; LPT94] mainly focusing on execution semantics support defining trigger conditions. These conditions can evaluate properties of incoming data such as how many data items are available and decide to run a process or wait for more or other incoming data. However, these semantics cannot describe the effect of data processing on data properties such as that the classification level of outgoing data is the highest classification level of all incoming data. In contrast, semantics [LT91; WBL93; Lea+96; LWB99] that support describing the values of outputs based on inputs support the propagation of data properties. These semantics use general purpose languages to describe the calculation of the output values. General purpose languages allow to describe much more than just the propagation of properties, so they introduce a considerable amount of complexity. Therefore, we favor the use of a DSL instead of a general purpose language to describe the behavior of nodes. The DSL only provides a limited set of operations to derive properties of outgoing data based on incoming properties of data. We expect the complexity of using the DSL to be lower compared to a general purpose language.

Behavior of Actors DeMarco [DeM79] does not consider the behavior of external actors because he wants to focus on the system and external actors are, per definition, not part of the system. However, reducing actors to providing input to the system and consuming outputs from the system is too restrictive. In our running example, it is important to know that the actor uses the credit card details, which he/she has received from the declassification operation, for booking a flight in the system. This does not violate the non-interference requirements. In contrast, if the user uses the credit card details received from the credit card center, a violation of the non-interference requirement occurs because the data has not been declassified before sending it to the airline. By making the behavior of the user explicit, we can avoid wrongly reported violations. There are basically two approaches used by existing semantics: One set of approaches [KBB86; LT91; BW89; Fen+93] ignores the behavior of users because it is not necessary to serve the corresponding purpose of the semantics. The other set of approaches [PKP91; Fra92; LPT94; Lea+96; LWB99] simply uses the same means to describe the behavior of actors as already used for describing the behavior of other nodes. We also see the benefit of using the same means because this provides a streamlined solution. Therefore, we use the same means to describe the behavior of actors as for describing the behavior of other nodes.

9.4. ADL Integrations

The integration of confidentiality analyses into existing ADLs is beneficial to lower the initial learning effort for software architects and to make use of existing software architectures represented in the ADL. An integration usually consists of two parts: An extension of the ADL syntax and an integration of an analysis, which uses inputs given in the extended ADL. We briefly discuss both parts and explain the reason for choosing a particular integration approach in this thesis.

Extension of ADL Syntax. Analyses usually need additional information, which is not present in the software architecture yet. ADLs are used to represent software architectures. The ADL is often specified by a metamodel and the software architecture is a model, i.e. an instance of that metamodel. To represent the additional information, this metamodel has to be extended. Heinrich, Strittmatter, and Reussner [HSR21] distinguish intrusive and non-intrusive extensions. Intrusive extensions change the metamodel, which means that it becomes cluttered in case of many extensions and that existing models, i.e. instances of that metamodel, potentially become incompatible depending on the particular change. In contrast, non-intrusive extensions do not break existing models. If a metamodel already provides means for extensions, these extensions should be used. Therefore, many previously presented approaches [Jür05; LBD02; HSS14; Kat+13], which use UML to represent the software architecture, use UML profiles to extend the UML by additional information. If the metamodel does not provide means for extensions, it is still possible to use annotation models, which point to the software architecture. Almorsy, Grundy, and Ibrahim [AGI13] make use of such an annotation model. Using inheritance as part of additional metamodels is another option, which Heinrich, Strittmatter, and Reussner [HSR21] mention and which we also use. In general, such additional metamodels can be seen as a fork of the original metamodel. In case of EMF, which Palladio uses to specify its metamodel, the subtypes specified in the additional metamodels are integrated into the existing metamodel by the tooling[3]. Therefore, the use of inheritance is non-invasive in our approach. In fact, we make use of all three mentioned non-intrusive ADL extension approaches. Using inheritance is often the most comprehensible solution because the concept of inheritance is known from object-oriented programming. However, using inheritance is not always possible because of the structure of the metamodel. Therefore, we also use profiles in combination with annotation models. The annotation models contain the additional information and the profiles link the information to the software architecture.

Integration of Analysis. There are usually two options to integrate analyses in ADLs: model queries and transformations. Model queries require formulating the analysis as a query to the model of the software architecture. This works well for matching patterns in the software architecture or for checking well-formedness rules. The approaches for identifying threats mentioned in Section 9.2.1 use this approach. Transformations map a software architecture given in an ADL to another representation tailored for analyses. AuthUML and FlowUML [AW03; AFW06], iFlow [Kat+13], SecDFD [TSB19], UMLSec [Jür05] and the approach of Gerking and Schubert [GS19] use this integration approach. The analyses take place on the dedicated analysis artifact and the results are reported

[3] `https://web.archive.org/web/20211108130758/https://ed-merks.blogspot.com/2008/01/creating-children-you-didnt-know.html`

back to the software architect. The benefit of the transformation approach is that software architects can make use of existing, powerful analysis tools, which allows for more sophisticated analyses. We also use the transformation approach to provide such sophisticated analyses.

10. Conclusions

We conclude this thesis in this chapter. We summarize the contributions, their validation as well as the differences to state-of-the-art approaches in Section 10.1. The benefits of the contributions for various stakeholders is part of Section 10.2. We recap the most important assumptions and limitations in Section 10.3. Eventually, we discuss future work in Section 10.4.

10.1. Summary

In this thesis, we presented a data-oriented approach to identify violations of confidentiality requirements in software architectures. We presented four validated contributions, which answer the research questions described in Section 1.4. In the following, we summarize the contributions (C), the corresponding research questions (RQ) and the validation covered by the corresponding validation questions (VQ).

DFD Syntax (C1) The extended DFD syntax captures additional information, which is required to identify violations of confidentiality requirements as well as to avoid ambiguities implied by the original DFD syntax and semantics of DeMarco. The additional information is structured in three viewpoints. The functional viewpoint introduces actor processes, which are necessary to represent non-trivial activities of external actors, and pins, which are necessary to solve ambiguities caused by multiple incoming data flows. The confidentiality primitives viewpoint introduces characteristic types and characteristics, which represent relevant properties for reasoning about confidentiality violations, as well as behaviors, which represent the effect of data processing on properties. The confidentiality viewpoint relates the behaviors and properties to nodes in the DFD. The benefit of the separation into viewpoints is that the responsibility of involved roles are clear: A security expert creates potentially reusable confidentiality primitives and assists the software architect in using them. A software architect represents the system as well as its usage and binds the security primitives to the actual architecture.

The representation of this additional information in an extended DFD metamodel answers the research question about sufficient modeling primitives for reasoning about confidentiality (RQ3). In a validation (VQ1 and VQ2) based on a case study involving seventeen systems, which define confidentiality requirements in terms of various access control and information flow control mechanisms, we demonstrated the expressiveness of the extended DFD syntax. In contrast to the state of the art, the extended DFD syntax is capable of expressing access control and information flow control individually as well as combinations of these mechanisms within the same syntax.

We found that the additional information in the DFD syntax is necessary in order to identify violations of confidentiality requirements given in terms of access control (RQ1) as well as information flow control (RQ2). The validation of the usage frequency of the syntax elements (VQ3) for modeling the case study systems supports this statement by showing that the elements of the DFD syntax are frequently used. In contrast to many state-of-the-art approaches, the syntax does not include special elements, which are only useful for reasoning about certain confidentiality mechanisms.

DFD Semantics (C2) The semantics for the extended DFD syntax define the meaning of DFD elements in terms of label propagation. All nodes become nodes in a label propagation network. The data flows become connections between the nodes. Characteristics become labels. Behaviors of nodes become the propagation logic of a node. A mapping from the extended DFD syntax to clauses of first-order logic given in Prolog assigns the previously described meaning. The Prolog program resulting from the mapping of a particular DFD identifies all labels at all places in the DFD. Analyses compare these labels with expected labels to identify violations of confidentiality requirements.

The validation of the correctness of detected violations (VQ6) on sixteen case study systems shows that the semantics can correctly identify violations in systems. Therefore, the semantics provide a sufficient answer to RQ4, which asks for appropriate semantics for identifying violations. In contrast to many state-of-the-art approaches, the semantics are sufficient to analyze the propagation of data through the system. Many other approaches are limited to well-formedness constraints or pattern matching. In addition, the semantics are flexible enough to cover a wide range of different confidentiality requirements. Many other approaches are limited to one particular type of analysis.

DFD-based Analyses (C3) The analysis definitions for DFDs provided in this thesis cover information flow control with non-interference requirements as well as the four most commonly used access control mechanisms DAC, MAC, RBAC and ABAC. In addition, a partial analysis definition for encryption is available, which can be combined with the previously mentioned analysis definitions. An analysis definition always consists of a set of characteristic types, characteristics, behaviors and a label comparison function. The label comparison function is given as a Prolog query, which compares the propagated labels on data with expected labels to identify a violation. For specific confidentiality requirements, we provide templates for Prolog clauses, which cover the additional requirements. A security expert creates the analysis definitions because creating them requires security expertise. Software architects can reuse these definitions in multiple systems or define custom analyses using a DSL for creating a label comparison function.

The set of analysis definitions, which considers information flow control, answers research question RQ5, which asks how an information flow analysis can be formalized using the extended DFD syntax and the corresponding semantics. The set of analysis definitions, which considers access control, answers research question RQ6, which asks about the formalization of access control analyses. The validation of the expressiveness of the analysis definitions (VQ5) shows that the confidentiality requirements of sixteen case study systems can be expressed in terms of the analysis definitions. The set of systems contains systems using information flow control as well as systems using access control. In contrast to many other state-of-the-art approaches, there are multiple analysis definitions, which software architects can use within the same architecture. Many state-of-the-art approaches only provide one particular type of analysis and do not provide means to specify additional analyses. In contrast, software architects can use the DSL provided in this thesis to formulate custom analyses. The validation of the correctness of detected violations (VQ6) shows that confidentiality requirements are not only expressible but that the analyses based on the analysis definitions can correctly identify violations.

ADL Integration Guidelines (C4) The guidelines for integrating the analysis capabilities of extended DFDs into existing ADLs support ADLs focusing on control flows as well as ADLs focusing on data flows. The core idea of the integration guidelines is to extend the syntax of the ADL by necessary but missing means to express information relevant for confidentiality and then to map this extended ADL to an extended DFD. Afterwards, software architects can use the existing DFD-based analyses to identify violations of confidentiality requirements. To show the applicability of the guidelines, we apply the integration guidelines to the Palladio ADL, which supports control flows as well as data flows. This results in two integrations: one integration into the Palladio subset that uses control flows and one integration into the Palladio subset that uses data flows. Because applying the integration guidelines worked for Palladio, we can see the integration guidelines as answer to RQ7 and RQ8, which ask for a way of using the DFD-based analyses in ADLs using control flows as well as data flows. In contrast to the state of the art, the software architect can still use the existing ADL and only has to learn the newly introduced elements of the syntax. The state of the art either requires dedicated analysis models or prescribes the ADL, which is often the UML.

The Palladio integrations are as expressive as the extended DFDs and the analysis results are also correct. In a validation of the expressiveness (VQ8) based on seventeen case study systems for each integration, we could show that we could express the same case study systems by the integrations as by DFDs. In a validation of the correctness of the analysis results (VQ9) based on the expressed case study systems, we could show that all reported violations were correct.

A major benefit of the ADL integration is that already modeled software architectures can usually be extended by the necessary information to detect violations of confidentiality requirements. In a validation of the amount of model elements to be changed when adding such information to an existing software architecture (VQ11), we could show that a considerable part of the software architecture does not have to be changed. state-of-the-art approaches, which do not provide ADL integrations, require the software architect to remodel the whole software architecture in a new ADL. Because our ADL integrations support various confidentiality mechanisms, switching the confidentiality mechanism also does not require remodeling the whole software architecture. In a validation of the amount of model elements to be changed when switching confidentiality mechanisms, we could show that a considerable amount of model elements can be reused. state-of-the-art approaches, which only support one particular confidentiality mechanism, require the software architect to remodel the whole software architecture in another analysis model or ADL.

10.2. Benefits

With the contributions presented in this thesis, confidentiality requirements can be considered systematically while defining software architectures. In the following, we discuss how tool engineers, security experts, software architects and organizations benefit from these contributions.

Tool Engineers are responsible for creating and maintaining the tools to specify software architectures. By using the extended DFD syntax, its semantics as well as the ADL integration guidelines, tool engineers can build new tools or integrate the analysis capabilities presented in this thesis into existing tools. Because the engineers do not have to define and validate the propagation logic on their own, they can provide tools faster than by implementing the tools from scratch. They can also use the analysis definitions to provide a broad range of analyses right from the beginning, which means the tools provide many features without the need to spend time on implementing these features.

Security Experts are responsible for supporting the software architect in meeting the confidentiality requirements. By defining analysis definitions, the security expert provides the building blocks to represent confidentiality in software architectures. Because these analysis definitions are often reusable for multiple systems, the security expert does not have to assist the software architect for every system, which gives him/her time for other tasks. When considering a situation without analysis capabilities, which the software architect can use on his/her own, the security expert would even have to do the analysis of the software architecture, which implies considerable effort.

Software Architects are responsible for defining the software architecture in a way that it meets the requirements on the software system. By using the ADL integration, software architects can systematically consider confidentiality requirements while defining the software architecture. Because the analysis capabilities are built into the ADLs, which the software architects already use, they only have to learn the new elements of the extended ADL. This lowers the initial learning and training effort. Because the existing ADL is only extended and not replaced, the software architects can add information to consider confidentiality requirements to the existing software architectures instead of recreating the software architectures from scratch in a dedicated analysis tool. This lowers the modeling effort. Because the reusable analysis definitions provided by the security expert already specify the important aspects for analyzing the confidentiality requirements, the software architect can use these analysis definitions without the need of having high security expertise. Instead, a short introduction by the security expert should be sufficient.

Organizations hire employees for creating software systems, which can be sold for profit. The previously mentioned benefits, especially the benefits for the security expert and the software architect, contribute to the overall goal of reducing the cost for creating a software system. Security experts as well as software architects are highly qualified people, who usually receive high compensations. Reducing the effort spent by these people to create a software system reduces the overall cost because less people are required to create the software system. As explained before, the security expert does not have to be involved in the analysis of every system anymore, which reduces his/her effort for a particular software system heavily. The software architects have some initial effort for getting their head around the extended ADL and the analysis definitions. However, this one-time effort pays off because the effort spent in modeling and reasoning about confidentiality requirements is reduced for every system. By systematically considering confidentiality in the software architecture, the probability of detecting confidentiality violations already in the software architecture increases and the probability that the underlying issue becomes implemented decreases. Because it requires less effort to fix an issue early, this also reduces the total cost of creating a software system. Because extensive changes of the implementation caused by undetected issues in the software architecture are reduced, there is a good chance that the software system can be sold earlier.

10.3. Assumptions and Limitations

We already discussed the assumptions and limitations of our contributions together with the contributions. The assumptions and limitations regarding the DFD syntax and semantics are discussed in Section 5.3. Section 6.7 does the same for the DFD-based analyses and Section 7.4 covers the assumptions and limitations regarding the ADL integration guidelines. In the following, we recap the most important assumptions and limitations and justify why the assumptions are reasonable and the limitations are not too restrictive.

Properties as Discrete Values. The DFD syntax as well as the semantics limit the value range of properties to sets of discrete values. This means a property of a node or data cannot be an arbitrary numerical value or arbitrary string, for instance. We do not see this limitation as too restrictive because properties, which are relevant for confidentiality, are often discrete values. As we demonstrated by the analysis definitions and their validation, we can express the aspects, which are relevant in commonly used confidentiality mechanisms. Out of the most commonly used confidentiality mechanisms, only ABAC potentially requires the representation of continuous values. However, we still argue that the limitation to sets of discrete values is not too limiting here because usually a particular value in a continuous range of values only has a meaning because it is within a certain interval. Representing the available intervals as discrete values is possible and, therefore, the limitation does not impede analyses here.

Explicit Flows. The DFD semantics only consider explicit information flows, i.e. data flows, in analyses. This means, an analysis only detects a violation of a confidentiality requirement if the violation occurs because of an explicit information flow. We do not consider implicit information flows, which are caused by timing-dependencies, for instance. This limitation is the result of a trade-off decision: Detecting implicit information flows requires detailed descriptions of the execution order or even timing information. The necessary information might not be available while creating the software architecture and would certainly complicate the representation of the software architecture. We decided to focus on explicit flows because we can assume the information about explicit flows to be available. Consequently, the ADL integration neither does consider such implicit flows.

No Information About State, Time and Instances. The DFD-based analyses do not consider state, time or instance-level information. No information about time means that the analyses do not consider whether a data processing happens before another as long as there is no data dependency. State cannot be supported without a notion of time, so there is no information about the state of a system. No information about instances means that the analyses do not consider individual users but only types of users. We do not consider any of these limitations as inappropriate for analyses of software architectures as we explain in the following. To represent information about state and time, software architects have to specify the structure and behavior in a high level of detail. This increases expressiveness but also complicates the specification of software architectures and requires detailed specifications, which restrict the freedom in finding appropriate solutions for the architectural specifications in later phases. To avoid increasing the effort for specifying a software architecture significantly, we decided to not consider state and time. We decided to not consider instance information such as individual users of a system because there is not enough information about individual users while creating the software architecture. Instead, we only represent types of users.

ADL Elements Without Effect. In the ADL integrations of Palladio, we did not explicitly specify a meaning for every element of the Palladio syntax but implicitly specified that all elements without explicitly specified meaning have no meaning for identifying violations of confidentiality requirements. We consider the assumption that not all ADL elements affect confidentiality reasonable because Palladio is an ADL, which supports various types of quality predictions. Existing analyses realized in Palladio also do not consider every element. For instance, the resource demand of a certain action in the system is not important for reasoning about confidentiality.

10.4. Future Work

We identified five topics of future work, which we describe in the following.

Further Security Objectives. In this thesis, we focused on the security objective *confidentiality*. However, only meeting one security objective is usually not sufficient to secure a software system. Therefore, we would like to investigate whether the DFD syntax and the semantics can also represent the required information to identify violations of other security objectives. Especially, researching the representation of aspects relevant for integrity and formulating analyses to detect violations of integrity are of high interest because information flow control as well as access control are also mechanisms to protect the integrity of information.

Consideration of Uncertainty. The most fundamental assumption of model-based analyses is that the model correctly represents the subject under investigation. In case of a software architecture, this means that developers realize the software architecture correctly and that the execution context of the system matches the execution context specified in the software architecture. However, there might be aspects in the software architecture, which the software architect cannot know yet or which might change dynamically during runtime. For instance, a component might not be deployed on the expected node but on another because of a disruption at a cloud provider. This means, the software architect is uncertain about aspects of the software architecture. The topic of the ongoing research project FluidTrust[1] is to capture this uncertainty and identify the influence of uncertainty on confidentiality. There is already initial work on considering uncertainty in the context [BWH20] and within the structure and behavior [Hah21] of a software architecture within the approach described in this thesis. However, the work is limited with respect to the architectural elements affected by uncertainty and the validation. Therefore, further research is necessary.

Usage of catalogs. We already discussed, which parts of the analysis definitions depend on particular systems and which parts software architects can reuse for specifying other systems. In the context of security, catalogs are frequently used to provide reusable artifacts. However, we see the need for further research on how an entry of such a catalog has to look like and how software architects can make use of such a catalog. For instance, we have to define mechanisms to override parts of an analysis definition or provide partial analysis definitions if the definition cannot be reused as a whole. A formalized approach to combine such partial analysis definitions could also help security experts to integrate encryption into other analysis definitions, for instance.

[1] `https://web.archive.org/web/20200808112444/https://fluidtrust.ipd.kit.edu/home/`

Validation of Analysis DSL. We presented a DSL for formulating custom analyses in this thesis. The DSL is meant to be used by software architects, so they can formulate label comparison functions without expertise in logic programming. We provided a concrete syntax for illustration purposes only. However, we think that software architects can also use this concrete syntax because we designed it according to common best practices. In another work [Hah+21], we already validated the DSL according to objective criteria. But to support the statement about usability, we have to conduct a user study together with potential users. We would like to validate the usability and compare it to the usability of Prolog queries.

Extended Validation of ADL Integration. In this thesis, we applied the integration guidelines to the Palladio ADL to show their applicability and to perform further validations on the resulting integrations for the communication paradigms *call-and-return* as well as for data flows. It would be interesting to apply the integration guidelines to further ADLs, which use different communication paradigms, in order to validate the applicability of the guidelines.

A. Predicate exactCharacteristicValues/5

The predicate `exactCharacteristicValues/5` tests if the available labels of a certain characteristic type are exactly the requested labels. The predicate defined in line 1 of Listing A.1 takes a node identifier `N`, a pin `PIN`, a characteristic type `CT`, a list of values `VALS` and a flow tree `S`. The predicate succeeds if the values, which belong to the given characteristic type and which are available at the given pin at the given node for the given flow tree, are exactly the given values. To answer a query to this predicate, it is necessary to collect all values, which are available at the pin, for the given characteristic type and test the resulting set for equality with the given set of values.

The `allCharacteristicValues/5` predicate finds all available values for a given characteristic type. The rule in line 5 first finds a flow tree for the given node and pin because the flow tree is necessary to determine labels. The rule then queries the `allCharacteristicValues/6` predicate, which has the same signature except for an additional argument for remembering already considered values of a characteristic type. Two rules realize the

Listing A.1: Clauses providing the exactCharacteristicValues/5 predicate.

```
 1 exactCharacteristicValues(N, PIN, CT, VALS, S) :-
 2   allCharacteristicValues(N, PIN, CT, V, S),
 3   sort(VALS, V).
 4
 5 allCharacteristicValues(N, PIN, CT, VALS, S) :-
 6   flowTree(N, PIN, S),
 7   allCharacteristicValues(N, PIN, CT, S, [], VALS).
 8
 9 allCharacteristicValues(N, PIN, CT, S, VISITED, RESULT) :-
10   characteristic(N, PIN, CT, V, S),
11   intersection(VISITED, [V], []),
12   (
13     VISITED = [];
14     nth0(0, VISITED, FIRSTV),
15     V @< FIRSTV
16   ),
17   allCharacteristicValues(N, PIN, CT, S, [V | VISITED], RESULT).
18
19 allCharacteristicValues(N, PIN, CT, S, RESULT, RESULT) :-
20   \+ (
21     characteristic(N, PIN, CT, V, S),
22     intersection(RESULT, [V], [])
23   ).
```

`allCharacteristicValues/6` predicate. The rule shown in line 9 identifies a characteristic value `V`, which is available at the given pin and tests if the value has already been visited by ensuring that the intersection of the visited values and the set consisting of the new value is empty. Afterwards, the rule ensures that the found value is smaller than the first element of the already visited values according to the natural ordering relation. This ensures that the list of visited values is sorted. Afterwards, the rule adds the new value at the beginning of the list of visited values and recurses. The rule in line 19 provides the stop condition for this recursion. The rule succeeds if there are no more characteristic values are available, which have not been visited before. In this case, the list of visited values becomes the result.

B. DFDs of Self-Created Case Study Systems

In the following, we provide the DFDs for the case study systems, which we created completely from scratch. The DFD for CS13 is given in Figure B.1. The DFD for CS14 is given in Figure B.2. The DFD for CS16 is given in Figure B.3. We omit the pins in the visualization to enable a more compact representation. To indicate data flows to the same pin, we let the data flows overlap in the illustration. All visualizations are based on existing visualizations from one of our previous publications [Sei+22].

We do not provide the DFDs for the other case study systems because the respective sources mentioned in Section 8.2.2 already provide enough information about the cases

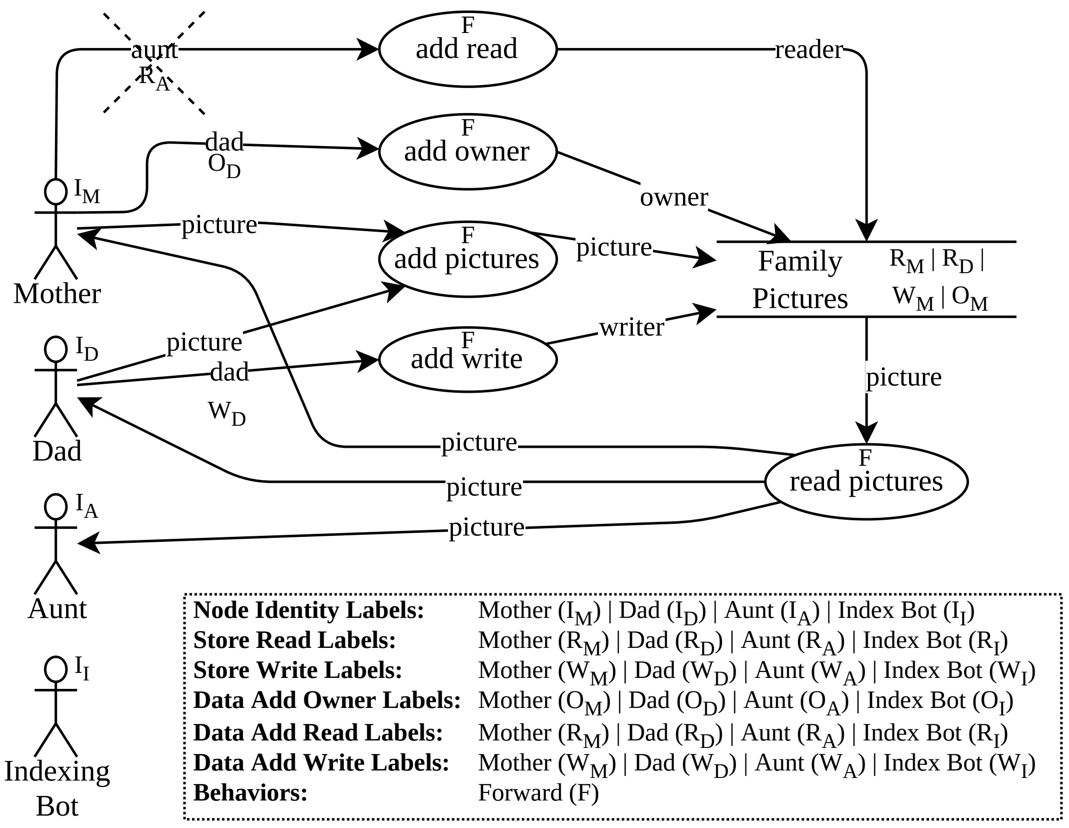

aunt
R_A
F
add read
reader
dad
O_D
F
add owner
owner
I_M
Mother
picture
F
add pictures
picture
Family Pictures
R_M | R_D | W_M | O_M
I_D
picture
dad
W_D
F
add write
writer
picture
Dad
picture
picture
F
read pictures
I_A
picture
Aunt
I_I
Indexing Bot
Node Identity Labels: Mother (I_M) | Dad (I_D) | Aunt (I_A) | Index Bot (I_I)
Store Read Labels: Mother (R_M) | Dad (R_D) | Aunt (R_A) | Index Bot (R_I)
Store Write Labels: Mother (W_M) | Dad (W_D) | Aunt (W_A) | Index Bot (W_I)
Data Add Owner Labels: Mother (O_M) | Dad (O_D) | Aunt (O_A) | Index Bot (O_I)
Data Add Read Labels: Mother (R_M) | Dad (R_D) | Aunt (R_A) | Index Bot (R_I)
Data Add Write Labels: Mother (W_M) | Dad (W_D) | Aunt (W_A) | Index Bot (W_I)
Behaviors: Forward (F)

Figure B.1.: DFD of ImageSharing case study system (CS13) based on visualization in previous publication [Sei+22]. Removing the crossed data flow introduces an issue.

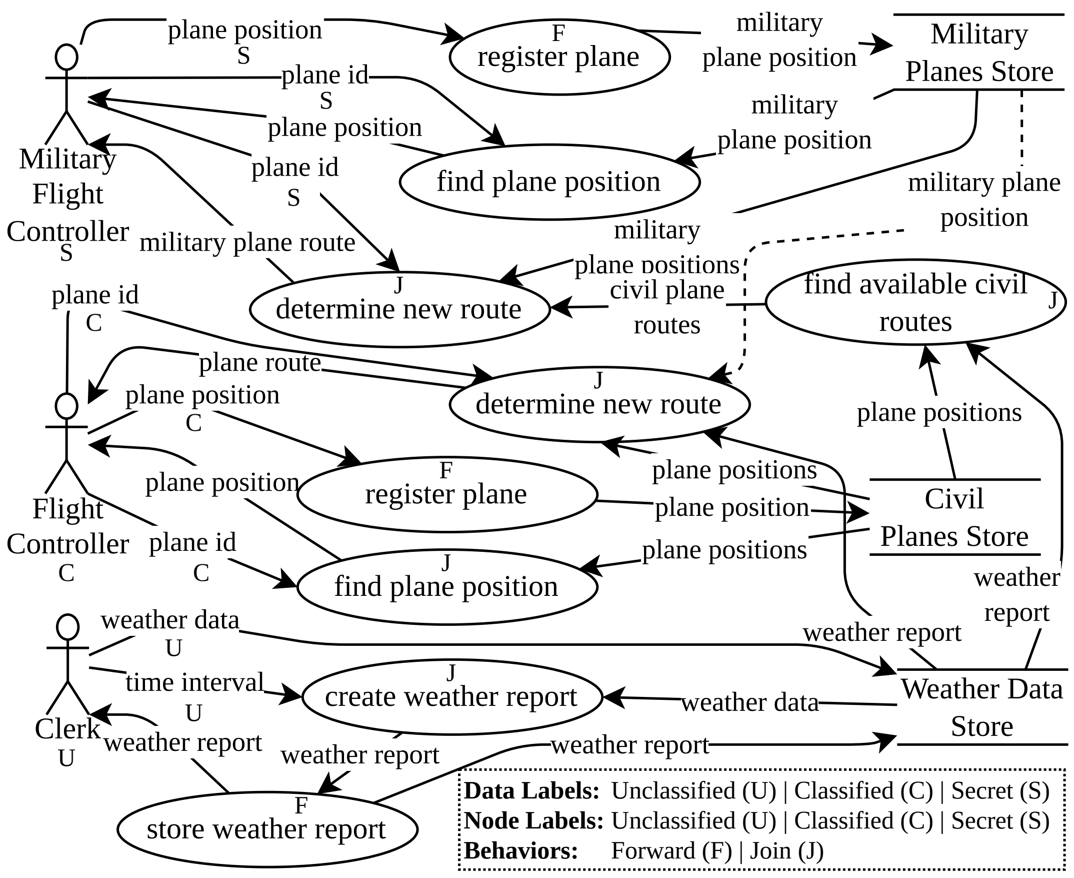


Figure B.2.: DFD of FlightControl case study system (CS14) as visualized in previous publication [Sei+22]. Adding the dashed data flow introduces an issue.

and their complexity. In addition, the data set of one of our previous publications [Sei+22] provides further explanations and visualizations of the cases.

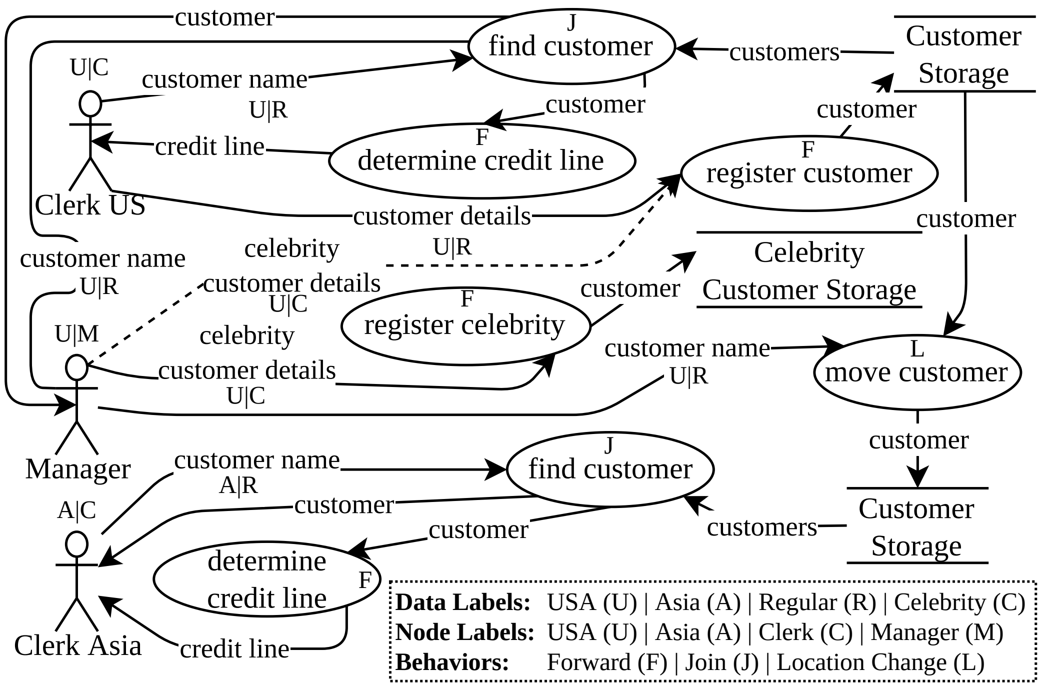


Figure B.3.: DFD of BankBranches case study system (CS16) as visualized in previous publication [Sei+22]. Adding the dashed data flow introduces an issue.

Bibliography

[Abd+11] Takoua Abdellatif et al. “Automating information flow control in component-based distributed systems”. In: *Proceedings of International ACM Sigsoft Symposium on Component-based Software Engineering*. CBSE’11. ACM, 2011, pp. 73–82. DOI: 10.1145/2000229.2000241.

[AC18] Hala Assal and Sonia Chiasson. “Security in the Software Development Lifecycle”. In: *Proceedings of Symposium on Usable Privacy and Security*. SOUPS’18. 2018, pp. 281–296. URL: https://web.archive.org/web/20210613175439/https://www.usenix.org/system/files/conference/soups2018/soups2018-assal.pdf (visited on 02/27/2022).

[AC19] Hala Assal and Sonia Chiasson. “’Think secure from the beginning’: A Survey with Software Developers”. In: *Proceedings of the Conference on Human Factors in Computing Systems*. CHI ’19. ACM, 2019, pp. 1–13. DOI: 10.1145/3290605.3300519.

[AF08] Bandar Alhaqbani and Colin Fidge. “Access Control Requirements for Processing Electronic Health Records”. In: *Business Process Management Workshops*. LNCS. Springer, 2008, pp. 371–382. DOI: 10.1007/978-3-540-78238-4_38.

[AFW06] Khaled Alghathbar, Csilla Farkas, and Duminda Wijesekera. “Securing UML Information Flow using FlowUML”. In: *Journal of Research and Practice in Information Technology* 38.1 (2006), pp. 111–120. URL: https://web.archive.org/web/20211026093035/https://50years.acs.org.au/content/dam/acs/50-years/journals/jrpit/JRPIT38.1.111.pdf (visited on 02/27/2022).

[AGI13] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. “Automated software architecture security risk analysis using formalized signatures”. In: *Proceedings of International Conference on Software Engineering*. ICSE’13. IEEE, 2013, pp. 662–671. DOI: 10.1109/ICSE.2013.6606612.

[AH97] Vijayalakshmi Atluri and Wei-Kuang Huang. “An Extended Petri Net Model for Supporting Workflows in a Multilevel Secure Environment”. In: *Database Security: Status and prospects*. IFIP Advances in Information and Communication Technology. Springer, 1997, pp. 240–258. DOI: 10.1007/978-0-387-35167-4_15.

[Ahm+17] Amir Shayan Ahmadian et al. “Model-based privacy and security analysis with CARiSMA”. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ESEC/FSE’17. ACM, 2017, pp. 989–993. DOI: 10.1145/3106237.3122823.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs.* 2nd ed. Electrical engineering and computer science series. MIT Press, 1996. ISBN: 978-0-262-01153-2.

[AT83] Selim G. Akl and Peter D. Taylor. "Cryptographic solution to a problem of access control in a hierarchy". In: *ACM Transactions on Computer Systems* 1.3 (1983), pp. 239–248. DOI: `10.1145/357369.357372`.

[AW03] Khaled Alghathbar and Duminda Wijesekera. "authUML: a three-phased framework to analyze access control specifications in use cases". In: *Proceedings of ACM Workshop on Formal Methods in Security Engineering.* FMSE '03. ACM, 2003, pp. 77–86. DOI: `10.1145/1035429.1035438`.

[AWT07] Marwan Abi-Antoun, Daniel Wang, and Peter Torr. "Checking threat modeling data flow diagrams for implementation conformance and security". In: *Proceedings of IEEE/ACM International Conference on Automated Software Engineering.* ASE '07. ACM, 2007, pp. 393–396. DOI: `10.1145/1321631.1321692`.

[Bau05a] Friedrich L. Bauer. "Cryptosystem". In: *Encyclopedia of Cryptography and Security.* Springer, 2005, pp. 119–119. DOI: `10.1007/0-387-23483-7_90`.

[Bau05b] Friedrich L. Bauer. "Encryption". In: *Encyclopedia of Cryptography and Security.* Springer, 2005, pp. 202–202. DOI: `10.1007/0-387-23483-7_141`.

[BCR94] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering - 2 Volume Set.* Wiley, 1994, pp. 528–532. DOI: `10.1002/0471028959.sof142`.

[Bed+13] Punam Bedi et al. "Threat-oriented security framework in risk management using multiagent system". In: *Software: Practice and Experience* 43.9 (2013), pp. 1013–1038. DOI: `10.1002/spe.2133`.

[Ber+17] Alexander van den Berghe et al. "A Model for Provably Secure Software Design". In: *Proceedings of IEEE/ACM International FME Workshop on Formal Methods in Software Engineering.* FormaliSE'17. IEEE, 2017, pp. 3–9. DOI: `10.1109/FormaliSE.2017.6`.

[Ber+18] Alexander van den Berghe et al. "A Lingua Franca for Security by Design". In: *Proceedings of IEEE Cybersecurity Development.* SecDev'18. IEEE, 2018, pp. 69–76. DOI: `10.1109/SecDev.2018.00017`.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22. DOI: `10.1016/j.jss.2008.03.066`.

[BP08] Cédric Brun and Alfonso Pierantonio. "Model Differences in the Eclipse Modeling Framework". In: *UPGRADE: The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34. URL: `https://web.archive.org/web/20090205174152/http://upgrade-cepis.org/issues/2008/2/upg9-2Brun.pdf` (visited on 02/27/2022).

[Bra13] Max Bramer. *Logic programming with Prolog.* 2. ed. Springer, 2013. ISBN: 978-1-4471-5487-7. DOI: `10.1007/978-1-4471-5487-7`.

[Bro+12] Franz Brosch et al. “Architecture-Based Reliability Prediction with the Palladio Component Model”. In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1319–1339. DOI: 10.1109/TSE.2011.94.

[Bro+98] Manfred Broy et al. “What characterizes a (software) component?” In: *Software - Concepts & Tools* 19.1 (1998), pp. 49–56. DOI: 10.1007/s003780050007.

[BSK15] Axel Busch, Misha Strittmatter, and Anne Koziolek. “Assessing Security to Compare Architecture Alternatives of Component-Based Systems”. In: *IEEE International Conference on Software Quality, Reliability and Security*. QRS’15. 2015, pp. 99–108. DOI: 10.1109/QRS.2015.24.

[BSK16] Bernhard J. Berger, Karsten Sohr, and Rainer Koschke. “Automatically Extracting Threats from Extended Data Flow Diagrams”. In: *Proceedings of International Symposium on Engineering Secure Software and Systems*. ESSoS’16. Springer, 2016, pp. 56–71. DOI: 10.1007/978-3-319-30806-7_4.

[Bür+18] Jens Bürger et al. “A framework for semi-automated co-evolution of security knowledge and system models”. In: *Journal of Systems and Software* 139 (2018), pp. 142–160. DOI: 10.1016/j.jss.2018.02.003.

[Bus+15] Axel Busch et al. “Automated Workload Characterization for I/O Performance Analysis in Virtualized Environments”. In: *Proceedings of the International Conference on Performance Engineering*. ICPE ’15. ACM, 2015, pp. 265–276. DOI: 10.1145/2668930.2688050.

[BW84] 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. DOI: 10.1109/TSE.1984.5010301.

[BW89] P.D. Brunza and Th.P. van der Weide. “The Semantics of Data Flow Diagrams”. In: *Proceedings of International Conference on Management of Data*. CISMOD’89. McGraw-Hill, 1989, pp. 66–78. URL: https://web.archive.org/web/20181008133048/http://cs.ru.nl/Th.P.vanderWeide/docs/1989-Bruza-DataFlowSem.pdf (visited on 02/27/2022).

[BWH20] Nicolas Boltz, Maximilian Walter, and Robert Heinrich. “Context-Based Confidentiality Analysis for Industrial IoT”. In: *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications*. SEAA’20. 2020, pp. 589–596. DOI: 10.1109/SEAA51224.2020.00096.

[Cam13] Patrizio Campisi, ed. *Security and Privacy in Biometrics*. Springer, 2013. ISBN: 978-1-4471-5230-9. DOI: 10.1007/978-1-4471-5230-9.

[CCR14] J. Cabot, R. Clarisó, and D. Riera. “On the verification of UML/OCL class diagrams using constraint programming”. In: *Journal of Systems and Software* 93 (2014), pp. 1–23. DOI: 10.1016/j.jss.2014.03.023.

[Cis19] Cisco Systems, Inc. *Consumer Privacy Survey*. Technical Report. Cisco Systems, Inc., 2019, p. 14. URL: https://web.archive.org/web/20210701075125/https://www.cisco.com/c/dam/global/en_uk/products/collateral/security/cybersecurity-series-2019-cps.pdf (visited on 02/27/2022).

[Dav+13] Jennifer A. Davis et al. “Study on the Barriers to the Industrial Adoption of Formal Methods”. In: *Formal Methods for Industrial Critical Systems*. LNCS. Springer, 2013, pp. 63–77. DOI: 10.1007/978-3-642-41010-9_5.

[DeM79] Tom DeMarco. *Structured analysis and system specification*. Prentice Hall, 1979. ISBN: 0-13-854380-1.

[Den+11] Mina Deng et al. “A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements”. In: *Requirements Engineering* 16.1 (2011), pp. 3–32. DOI: 10.1007/s00766-010-0115-7.

[Den20a] Elizabeth Denham. *Penatly Notice*. Penatly Notice COM0783542. Information Commissioner’s Office, 2020. URL: https://web.archive.org/web/20210620130131/https://edpb.europa.eu/sites/default/files/article-60-final-decisions/uk_2010-10_data_breach_security_of_processing_decisionpublic_final.pdf (visited on 02/27/2022).

[Den20b] Elizabeth Denham. *Penatly Notice*. Penatly Notice COM0804337. Information Commissioner’s Office, 2020. URL: https://web.archive.org/web/20210802034347/https://edpb.europa.eu/sites/default/files/article-60-final-decisions/uk_2020-10_personal_data_breach_decisionpublic_final.pdf (visited on 02/27/2022).

[Den76] Dorothy E. Denning. “A lattice model of secure information flow”. In: *Communications of the ACM* 19.5 (1976), pp. 236–243. DOI: 10.1145/360051.360056.

[EB11] Wolfgang Ertel and Nathanael Black. *Introduction to artificial intelligence*. Undergraduate topics in computer science. Springer, 2011. ISBN: 978-0-85729-299-5. DOI: 10.1007/978-0-85729-299-5.

[End01] Herbert B. Enderton. *A mathematical introduction to logic*. 2nd ed. Harcourt/Academic Press, 2001. ISBN: 978-0-12-238452-3.

[Eur16] European Union. “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”. In: *Official Journal of the European Union* 59 (2016), pp. 1–88. URL: https://web.archive.org/web/20220224100241/https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 02/27/2022).

[FA03] Thom W. Frühwirth and Slim Abdennadher. *Essentials of constraint programming*. Cognitive Technologies. Springer, 2003. ISBN: 978-3-540-67623-2. DOI: 10.1007/978-3-662-05138-2.

[Fab16] Fabric Project, *Hospital Example* 2016. URL: https://github.com/apl-cornell/fabric/tree/master/examples/hospital, SWHID: ⟨swh:1:dir:34e7f4abcabc2679ae824c58fcd9761ed918c8de;visit=swh:1:snp:0a93db788ab7684b997c93ec236f8d4f24ec5a02;anchor=swh:1:rev:b6bc62eccce6ff0d867efeb5e7f928d54bcef13f;path=/examples/hospital/⟩.

[Fab17] Fabric Project, *FriendMap Example* 2017. URL: https://github.com/apl-cornell/fabric/tree/master/examples/friendmap, SWHID: ⟨swh:1:dir:6265029c039385d7601749396530cc3e6320ead7;visit=swh:1:snp:0a93db788ab7684b997c93ec236f8d4f24ec5a02;anchor=swh:1:rev:b6bc62eccce6ff0d867efeb5e7f928d54bcef13f;path=/examples/friendmap/⟩.

[Fac+13] Michael Factor et al. "Secure Logical Isolation for Multi-tenancy in cloud storage". In: *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*. MSST'13. ISSN: 2160-1968. 2013, pp. 1–5. DOI: 10.1109/MSST.2013.6558424.

[Fen+93] D. Fensel et al. "Giving Structured Analysis Techniques a Formal and Operational Semantics with KARL". In: *Requirements Engineering '93: Prototyping*. Vieweg+Teubner Verlag, 1993, pp. 267–285. DOI: 10.1007/978-3-322-94703-1_18.

[FGL93] David F. Ferraiolo, Dennis M. Gilbert, and Nickilyn Lynch. "An examination of federal and commercial access control policy needs". In: *Proceedings of the National Computer Security Conference*. National Institute of Standards and Technology, 1993, pp. 107–116. URL: https://web.archive.org/web/20211128223200/https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1993/09/20/proceedings-16th-national-computer-security-conference-1993/documents/1993-16th-NCSC-proceedings.pdf (visited on 02/27/2022).

[Fla18] David Flater. "Bad Security Metrics Part 1: Problems". In: *IT Prof.* 20.1 (2018), pp. 64–68. DOI: 10.1109/MITP.2018.011301733.

[FP11] Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley, 2011. ISBN: 978-0-321-71294-3.

[Fra92] R.B. France. "Semantically extended dataflow diagrams: a formal specification tool". In: *IEEE Transactions on Software Engineering* 18.4 (1992), pp. 329–346. DOI: 10.1109/32.129221.

[Fry+14] Maxime Frydman et al. "Automating Risk Analysis of Software Design Models". In: *The Scientific World Journal* 2014 (2014). Publisher: Hindawi, p. 12. DOI: 10.1155/2014/805856.

[Fur08] Steven Furnell, ed. *Securing information and communications systems: principles, technologies, and applications*. Artech House computer security series. Artech House, 2008. ISBN: 978-1-59693-228-9.

[GBB12] Thomas Goldschmidt, Steffen Becker, and Erik Burger. "Towards a Tool-Oriented Taxonomy of View-Based Modelling". In: *Proceedings of Modellierung*. Vol. P-201. LNI. GI, 2012, pp. 59–74. URL: https://web.archive.org/web/20220227110725/https://dl.gi.de/handle/20.500.12116/18148 (visited on 02/27/2022).

[GBP20] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. "The 2020 Expert Survey on Formal Methods". In: *Formal Methods for Industrial Critical Systems*. LNCS. Springer, 2020, pp. 3–69. DOI: 10.1007/978-3-030-58298-2_1.

[GCH03] Ashish Garg, Jeffrey Curtis, and Hilary Halper. "The Financial Impact of IT Security Breaches: What Do Investors Think?" In: *Information Systems Security* 12.1 (2003), pp. 22–33. DOI: 10.1201/1086/43325.12.1.20030301/41478.5.

[Gee10] D. Geer. "Are Companies Actually Using Secure Development Life Cycles?" In: *Computer* 43.6 (2010), pp. 12–16. DOI: 10.1109/MC.2010.159.

[GM82] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *Proceedings of the IEEE Symposium on Security and Privacy*. S&P'82. IEEE, 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[Gol+19] Nazila Gol Mohammadi et al. "Privacy Policy Specification Framework for Addressing End-Users' Privacy Requirements". In: *Trust, Privacy and Security in Digital Business*. LNCS. Springer, 2019, pp. 46–62. DOI: 10.1007/978-3-030-27813-7_4.

[GS19] Christopher Gerking and David Schubert. "Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures". In: *Proceedings of the IEEE International Conference on Software Architecture*. ICSA'19. IEEE, 2019, pp. 61–70. DOI: 10.1109/ICSA.2019.00015.

[Hah+21] Sebastian Hahner et al. "Modeling Data Flow Constraints for Design-Time Confidentiality Analyses". In: *Proceedings of the IEEE International Conference on Software Architecture - Companion*. ICSA-C'21. IEEE, 2021, pp. 15–21. DOI: 10.1109/ICSA-C52384.2021.00009.

[Hah21] Sebastian Hahner. "Architectural Access Control Policy Refinement and Verification under Uncertainty". In: *Companion Proceedings of the European Conference on Software Architecture*. Vol. 2978. CEUR Workshop Proceedings. CEUR-WS.org, 2021. URL: https://web.archive.org/web/20211115044740/http://ceur-ws.org/Vol-2978/ds-paper87.pdf (visited on 02/27/2022).

[HAM06] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. "From Languages to Systems: Understanding Practical Application Development in Security-typed Languages". In: *Proceedings of Computer Security Applications Conference*. ACSAC'06. IEEE, 2006, pp. 153–164. DOI: 10.1109/ACSAC.2006.30.

[Hei+18] Robert Heinrich et al. "Architecture-based change impact analysis in cross-disciplinary automated production systems". In: *Journal of Systems and Software* 146 (2018), pp. 167–185. DOI: 10.1016/j.jss.2018.08.058.

[Hei20] Robert Heinrich. "Architectural runtime models for integrating runtime observations and component-based models". In: *Journal of Systems and Software* 169 (2020), p. 110722. DOI: 10.1016/j.jss.2020.110722.

[Her+06] Shawn Hernan et al. *Uncover Security Design Flaws Using The STRIDE Approach*. 2006. URL: https://web.archive.org/web/20220204210717/https://docs.microsoft.com/en-us/archive/msdn-magazine/2006/november/uncover-security-design-flaws-using-the-stride-approach (visited on 02/27/2022).

[HJ03] Anthony Harrington and Christian Jensen. "Cryptographic access control in a distributed file system". In: *Proceedings of the ACM symposium on access control models and technologies.* SACMAT '03. ACM, 2003, pp. 158–165. DOI: 10.1145/775412.775432.

[HL06] Michael Howard and Steve Lipner. *The Security Development Lifecycle.* Microsoft Press, 2006. ISBN: 978-0-7356-2214-2.

[HS12] Daniel Hedin and Andrei Sabelfeld. "A Perspective on Information Flow Control". In: *Software Safety and Security - Tools for Analysis and Verification.* Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 319–347. DOI: 10.3233/978-1-61499-028-4-319.

[HSJ12] T. Heyman, R. Scandariato, and W. Joosen. "Reusable Formal Models for Secure Software Architectures". In: *Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture.* WICSA/ECSA'12. IEEE, 2012, pp. 41–50. DOI: 10.1109/WICSA-ECSA.212.12.

[HSR21] Robert Heinrich, Misha Strittmatter, and Ralf Reussner. "A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis". In: *IEEE Transactions on Software Engineering* 47.4 (2021), pp. 775–800. DOI: 10.1109/TSE.2019.2903797.

[HSS14] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. "Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach". In: *Software & Systems Modeling* 13.2 (2014), pp. 513–548. DOI: 10.1007/s10270-012-0263-y.

[Hu+14] Vincent C. Hu et al. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* Tech. rep. NIST SP 800-162. National Institute of Standards and Technology, 2014. DOI: 10.6028/NIST.SP.800-162.

[IBM20] IBM Corporation. *Cost of a Data Breach Report 2020.* Report. IBM Security, 2020, p. 82. URL: https://www.ibm.com/security/digital-assets/cost-data-breach-report/ (visited on 06/30/2021).

[IH18] Jim Isaak and Mina J Hanna. "User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection". In: *Computer* 51.8 (2018), pp. 56–59. DOI: 10.1109/MC.2018.3191268.

[Int11] Int. Organization for Standardization. *ISO/IEC/IEEE 42010:2011 Systems and software engineering - Architecture description.* Standard. ISO, 2011. DOI: 10.1109/IEEESTD.2011.6129467.

[Int18] Int. Organization for Standardization. *ISO/IEC 27000:2018(E) Information technology – Security techniques – Information security management systems – Overview and vocabulary.* Standard. ISO, 2018.

[Int96] Int. Organization for Standardization. *ISO/IEC 14977:1996 Information technology - Syntactic metalanguage - Extended BNF.* Standard. ISO, 1996.

[Jil+08] Atif Aftab Ahmed Jilani et al. “Formal Representations of the Data Flow Diagram: A Survey”. In: *Proceedings of Advanced Software Engineering and Its Applications*. ASEA’08. 2008, pp. 153–158. DOI: `10.1109/ASEA.2008.34`.

[JKS12] Xin Jin, Ram Krishnan, and Ravi Sandhu. “A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC”. In: *Data and Applications Security and Privacy XXVI*. Vol. 7371. Springer, 2012, pp. 41–55. DOI: `10.1007/978-3-642-31540-4_4`.

[JR91] Kurt Jensen and Grzegorz Rozenberg, eds. *High-level Petri nets: theory and application*. 1st ed. Springer, 1991. ISBN: 978-3-642-84524-6. DOI: `10.1007/978-3-642-84524-6`.

[JUN11] Atif A. A. Jilani, Muhammad Usman, and Aamer Nadeem. “Comparative Study on DFD to UML Diagrams Transformations”. In: *CoRR* abs/1102.4162 (2011). URL: `http://arxiv.org/abs/1102.4162`.

[Jür05] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005. ISBN: 978-3-540-00701-2. DOI: `10.1007/b137706`.

[Kal05a] Burt Kaliski. “Asymmetric Cryptosystem”. In: *Encyclopedia of Cryptography and Security*. Springer, 2005, pp. 11–11. DOI: `10.1007/0-387-23483-7_12`.

[Kal05b] Burt Kaliski. “Symmetric Cryptosystem”. In: *Encyclopedia of Cryptography and Security*. Springer, 2005, pp. 602–603. DOI: `10.1007/0-387-23483-7_422`.

[Kar+09] Gabor Karsai et al. “Design Guidelines for Domain Specific Languages”. In: *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling*. DSM’09. Helsingin kauppakorkeakoulu, 2009, pp. 7–13. URL: `https://web.archive.org/web/20190819150400/http://epub.lib.aalto.fi/pdf/hseother/b108.pdf` (visited on 02/27/2022).

[Kat+13] K. Katkalov et al. “Model-Driven Development of Information Flow-Secure Systems with IFlow”. In: *Proceedings of International Conference on Social Computing*. SocialCom’13. 2013, pp. 51–56. DOI: `10.1109/SocialCom.2013.14`.

[Kat17] Kuzman Katkalov. “Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme”. German. PhD Thesis. University of Augsburg, 2017. URL: `https://web.archive.org/web/20210926154149/https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/4339` (visited on 02/27/2022).

[KBB86] Kavi, Buckles, and Bhat. “A Formal Definition of Data Flow Graph Models”. In: *IEEE Transactions on Computers* C-35.11 (1986), pp. 940–948. DOI: `10.1109/TC.1986.1676696`.

[Kel+09] Patrick Gage Kelley et al. “A "nutrition label" for privacy”. In: *Proceedings of the Symposium on Usable Privacy and Security*. SOUPS ’09. ACM, 2009, pp. 1–12. DOI: `10.1145/1572532.1572538`.

[Kno00] K. Knorr. “Dynamic access control through Petri net workflows”. In: *Proceedings of the Computer Security Applications Conference*. ACSAC’00. 2000, pp. 159–167. DOI: `10.1109/ACSAC.2000.898869`.

[Kow79] Robert Kowalski. “Algorithm = logic + control”. In: *Communications of the ACM* 22.7 (1979), pp. 424–436. DOI: 10.1145/359131.359136.

[KRK17] R. Kuhn, M. Raunak, and R. Kacker. “It Doesn’t Have to Be Like This: Cybersecurity Vulnerability Trends”. In: *IT Professional* 19.6 (2017), pp. 66–70. DOI: 10.1109/MITP.2017.4241462.

[Lam73] Butler W. Lampson. “A note on the confinement problem”. In: *Communications of the ACM* 16.10 (1973), pp. 613–615. DOI: 10.1145/362375.362389.

[Lan+12] Philip Langer et al. “EMF Profiles: A Lightweight Extension Approach for EMF Models.” In: *The Journal of Object Technology* 11.1 (2012), 8:1. DOI: 10.5381/jot.2012.11.1.a8.

[Lav+06] Marc-André Laverdiere et al. “Security Design Patterns: Survey and Evaluation”. In: *Proceedings of the Canadian Conference on Electrical and Computer Engineering*. CCECE’06. 2006, pp. 1605–1608. DOI: 10.1109/CCECE.2006.277727.

[LBD02] Torsten Lodderstedt, David Basin, and Jürgen Doser. “SecureUML: A UML-based modeling language for model-driven security”. In: *Proceedings of the International Conference on the Unified Modeling Language*. UML’02. Springer, 2002, pp. 426–441. DOI: 10.1007/3-540-45800-X_33.

[LCM06] Christian F.J. Lange, Michel R.V. Chaudron, and Johan Muskens. “In practice: UML software architecture and design description”. In: *IEEE Software* 23.2 (2006), pp. 40–46. DOI: 10.1109/MS.2006.50.

[Lea+96] Gary T. Leavens et al. *An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams.* Technical Report TR93-28c. Iowa State University, 1996, p. 38. URL: https://web.archive.org/web/20200725235500/https://lib.dr.iastate.edu/cs_techreports/101/ (visited on 02/27/2022).

[Liu+20] Cong Liu et al. “Petri Net Based Data-Flow Error Detection and Correction Strategy for Business Processes”. In: *IEEE Access* 8 (2020), pp. 43265–43276. DOI: 10.1109/ACCESS.2020.2976124.

[LPT94] Peter Gorm Larsen, Nico Plat, and Hans Toetenel. “A Formal Semantics of Data Flow Diagrams”. In: *Formal Aspects of Computing* 6.6 (1994), pp. 586–606. DOI: 10.1007/BF03259387.

[LT91] Tong Liu and C. S. Tang. “Semantic specification and verification of data flow diagrams”. In: *Journal of Computer Science and Technology* 6.1 (1991), pp. 21–31. DOI: 10.1007/BF02943404.

[LW71] Michael Levandowsky and David Winter. “Distance between Sets”. In: *Nature* 234.5323 (1971), pp. 34–35. DOI: 10.1038/234034a0.

[LWB99] Gary T. Leavens, Tim Wahls, and Albert L. Baker. “Formal semantics for SA style data flow diagram specification languages”. In: *Proceedings of ACM Symposium on Applied Computing*. SAC’99. ACM, 1999, pp. 526–532. DOI: 10.1145/298151.298433.

[Mal+13] Ivano Malavolta et al. “What Industry Needs from Architectural Languages: A Survey”. In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 869–891. DOI: 10.1109/TSE.2012.74.

[McG06] Gary McGraw. *Software Security - Building Security In.* Addison-Wesley Professional, 2006. ISBN: 0-321-35670-5.

[Met78] Charles E. Metz. “Basic principles of ROC analysis”. In: *Seminars in Nuclear Medicine* 8.4 (1978), pp. 283–298. DOI: 10.1016/S0001-2998(78)80014-2.

[MFP10] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. “A comparison of software design security metrics”. In: *Proceedings of the European Conference on Software Architecture: Companion Volume.* ECSA ’10. ACM, 2010, pp. 236–242. DOI: 10.1145/1842752.1842797.

[Mic21] Microsoft Corporation. *Microsoft Security Development Lifecycle Threat Modelling.* 2021. URL: https://web.archive.org/web/20210929041026/https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling (visited on 02/27/2022).

[Mon+21] David Monschein et al. “Enabling Consistency between Software Artefacts for Software Adaption and Evolution”. In: *Proceedings of the IEEE International Conference on Software Architecture.* ICSA’21. IEEE, 2021, pp. 1–12. DOI: 10.1109/ICSA51549.2021.00009.

[Moz20] Mozilla Foundation, *WebRTC* 2020. URL: https://github.com/mozilla/libwebrtc, SWHID: ⟨swh:1:snp:b316d8bdb3ba6f190b648e758843918ab840580e⟩.

[MS14] Philipp Merkle and Christian Stier. “Modelling database lock-contention in architecture-level performance simulation”. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering.* ICPE’14. ACM, 2014, pp. 285–288. DOI: 10.1145/2568088.2576762.

[Nat22] National Institute of Standards and Technology. *National Vulnerability Database - Statistics.* 2022. URL: https://web.archive.org/web/20220223063247/https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all&isCpeNameSearch=false (visited on 02/27/2022).

[Ngu+15] Phu H. Nguyen et al. “An extensive systematic review on the Model-Driven Development of secure systems”. In: *Information and Software Technology* 68 (2015), pp. 62–81. DOI: 10.1016/j.infsof.2015.08.006.

[Obj17] Object Management Group (OMG). *Unified Modeling Language 2.5.1.* Specification formal/17-12-05. 2017. URL: https://web.archive.org/web/20210225061032/https://www.omg.org/spec/UML/2.5.1/PDF (visited on 02/27/2022).

[Obj19] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification.* Specification formal/2019-10-01. 2019. URL: https://web.archive.org/web/20220121014702/https://www.omg.org/spec/MOF/2.5.1/PDF (visited on 02/27/2022).

[Obj20] Object Management Group (OMG). *Unified Architecture Framework*. Specification formal/19-11-07. 2020. URL: https://web.archive.org/web/20220227132703/https://www.omg.org/spec/UAF (visited on 02/27/2022).

[OKe90] Richard A. O'Keefe. *The Craft of Prolog*. Logic Programming. MIT Press, 1990. ISBN: 0-262-15039-5.

[Ozk18] Mert Ozkaya. "The analysis of architectural languages for the needs of practitioners". In: *Software: Practice and Experience* 48.5 (2018), pp. 985–1018. DOI: 10.1002/spe.2561.

[Pal21a] Palladio Simulator, *Indirections Metamodel* 2021. URL: https://github.com/PalladioSimulator/Palladio-Addons-Indirections/blob/releases/5.1.0/bundles/org.palladiosimulator.indirections/model/indirections.ecore, SWHID: ⟨swh:1:cnt:7e45b9492e213bf5cebc5f702af795c2beae41ce;origin=https://github.com/PalladioSimulator/Palladio-Addons-Indirections;visit=swh:1:snp:2edc40beb976093d5ddfc1d478ace6bc82393751;anchor=swh:1:rev:1a485e0a70de4c7be66165fa52b7f9f4a5facec7;path=/bundles/org.palladiosimulator.indirections/model/indirections.ecore⟩.

[Pal21b] Palladio Simulator, *Palladio Metamodel* 2021. URL: https://github.com/PalladioSimulator/Palladio-Core-PCM/blob/releases/5.1.0/bundles/org.palladiosimulator.pcm/model/pcm.ecore, SWHID: ⟨swh:1:cnt:f8b38ca863127ff4950ae9174d25f010b7a95743;origin=https://github.com/PalladioSimulator/Palladio-Core-PCM;visit=swh:1:snp:8db0d3ebb919c088c20ec89f572c2831df2d555c;anchor=swh:1:rev:c3e33fa9e47792214aa97714a1b8ab5ba4e3018f;path=/bundles/org.palladiosimulator.pcm/model/pcm.ecore⟩.

[PC10] Shari Pfleeger and Robert Cunningham. "Why Measuring Security Is Hard". In: *IEEE Security Privacy* 8.4 (2010), pp. 46–54. DOI: 10.1109/MSP.2010.60.

[Pen+16] Marcus Pendleton et al. "A Survey on Systems Security Metrics". In: *ACM Computing Surveys* 49.4 (2016), 62:1–62:35. DOI: 10.1145/3005714.

[Pet+94] Carsta Petersohn et al. "Formal Semantics for Ward & Mellor's Transformation Schemas". In: *Proceedings of Refinement Workshop*. Springer, 1994, pp. 14–41. DOI: 10.1007/978-1-4471-3240-0_2.

[Pet62] Carl Adam Petri. "Kommunikation mit Automaten". German. PhD Thesis. University of Bonn, 1962. URL: https://web.archive.org/web/20220128231247/https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/ (visited on 02/27/2022).

[PKP91] Nico Plat, Jan Katwijk, and Kees Pronk. "A case for structured analysis/formal design". In: *VDM'91 Formal Software Development Methods*. Vol. 551. Springer, 1991, pp. 81–105. DOI: 10.1007/3-540-54834-3_8.

[POB00] Richard F. Paige, Jonathan s. Ostroff, and Phillip J. Brooke. "Principles for modeling language design". In: *Information and Software Technology* 42.10 (2000), pp. 665–675. DOI: 10.1016/S0950-5849(00)00109-9.

[PR15] Klaus Pohl and Chris Rupp. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam, foundation level, IREB compliant.* 2nd ed. Rocky Nook, 2015. ISBN: 978-1-937538-77-4.

[Reg+15] Gianna Reggio et al. “What Are the Used UML Diagram Constructs? A Document and Tool Analysis Study Covering Activity and Use Case Diagrams”. In: *Model-Driven Engineering and Software Development.* Communications in Computer and Information Science. Springer, 2015, pp. 66–83. DOI: 10.1007/978-3-319-25156-1_5.

[Reu+11] Ralf Reussner et al. *The Palladio Component Model.* Tech. rep. 14. Karlsruhe Institute of Technology (KIT), 2011, p. 193. DOI: 10.5445/IR/1000022503.

[Reu+16] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures - The Palladio Approach.* MIT Press, 2016. ISBN: 978-0-262-03476-0.

[RH09] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[Ris+17] Erik Rissanen et al. *eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01.* OASIS Standard. 2017, p. 154. URL: https://web.archive.org/web/20200203104743/http://docs.oasis-open.org/xacml/3.0/errata01/os/xacml-3.0-core-spec-errata01-os-complete.html (visited on 02/27/2022).

[RW05] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional, 2005. ISBN: 978-0-321-11229-3.

[SB12] Lakshitha de Silva and Dharini Balasubramaniam. “Controlling software architecture erosion: A survey”. In: *Journal of Systems and Software.* Dynamic Analysis and Testing of Embedded Software 85.1 (2012), pp. 132–151. DOI: 10.1016/j.jss.2011.07.036.

[Sch+06] Markus Schumacher et al. *Security Patterns - Integrating Security and Systems Engineering.* Wiley, 2006. ISBN: 978-0-470-85884-4.

[Sei+21] Stephan Seifermann et al. “A Unified Model to Detect Information Flow and Access Control Violations in Software Architectures”. In: *Proceedings of International Conference on Security and Cryptography.* SECRYPT’21. SCITEPRESS, 2021, pp. 26–37. DOI: 10.5220/0010515300260037.

[Sei+22] Stephan Seifermann et al. “Detecting violations of access control and information flow policies in data flow diagrams”. In: *Journal of Systems and Software* 184 (2022), p. 111138. DOI: 10.1016/j.jss.2021.111138.

[Sei22] Stephan Seifermann. *Data Set of Thesis on Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements.* 2022. DOI: 10.5281/zenodo.6299921.

[Sho14] Adam Shostack. *Threat modeling: designing for security.* Wiley, 2014. ISBN: 978-1-118-80999-0.

[SHR19] Stephan Seifermann, Robert Heinrich, and Ralf Reussner. “Data-Driven Software Architecture for Analyzing Confidentiality”. In: *Proceedings of International Conference on Software Architecture*. ICSA. IEEE, 2019, pp. 1–10. DOI: `10.1109/ICSA.2019.00009`.

[Shu+02] F. Shull et al. “What we have learned about fighting defects”. In: *Proceedings of the IEEE Symposium on Software Metrics*. METRIC’02. ISSN: 1530-1435. 2002, pp. 249–258. DOI: `10.1109/METRIC.2002.1011343`.

[Sio+18a] Laurens Sion et al. “Solution-aware data flow diagrams for security threat modeling”. In: *Proceedings of the ACM Symposium on Applied Computing*. SAC ’18. ACM, 2018, pp. 1425–1432. DOI: `10.1145/3167132.3167285`.

[Sio+18b] Laurens Sion et al. “SPARTA: Security & Privacy Architecture Through Risk-Driven Threat Assessment”. In: *Proceedings of the IEEE International Conference on Software Architecture Companion*. ICSA-C’18. IEEE, 2018, pp. 89–92. DOI: `10.1109/ICSA-C.2018.00032`.

[Sio+20] Laurens Sion et al. “Security Threat Modeling: Are Data Flow Diagrams Enough?” In: *Proceedings of the IEEE/ACM International Conference on Software Engineering Workshops*. ICSEW’20. 2020, pp. 254–257.

[SM03] A. Sabelfeld and A.C. Myers. “Language-based information-flow security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: `10.1109/JSAC.2002.806121`.

[Smi+15] Justin Smith et al. “Questions developers ask while diagnosing potential security vulnerabilities with static analysis”. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ESEC/FSE’15. ACM, 2015, pp. 248–259. DOI: `10.1145/2786805.2786812`.

[SS09] Andrei Sabelfeld and David Sands. “Declassification: Dimensions and principles”. In: *Journal of Computer Security* 17.5 (2009), pp. 517–548. DOI: `10.3233/JCS-2009-0352`.

[SS94] R. S. Sandhu and P. Samarati. “Access control: principle and practice”. In: *IEEE Communications Magazine* 32.9 (1994), pp. 40–48. DOI: `10.1109/35.312842`.

[Sta+19] Cristian-Alexandru Staicu et al. “An Empirical Study of Information Flows in Real-World JavaScript”. In: *Proceedings of ACM SIGSAC Workshop on Programming Languages and Analysis for Security*. PLAS’19. ACM, 2019, pp. 45–59. DOI: `10.1145/3338504.3357339`.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. 1st ed. Springer, 1973. ISBN: 3-211-81106-0.

[Ste+16] Kurt Stenzel et al. “Declassification of Information with Complex Filter Functions:” in: *Proceedings of the International Conference on Information Systems Security and Privacy*. ICISSP’16. SCITEPRESS, 2016, pp. 490–497. DOI: `10.5220/0005782904900497`.

[Ste09] Dave Steinberg, ed. *EMF: Eclipse Modeling Framework*. 2nd ed., Rev. and updated. The eclipse series. Addison-Wesley, 2009. ISBN: 978-0-321-33188-5.

[Sto79] Robert R. Stoll. *Set theory and logic.* Dover Publications, 1979. ISBN: 978-0-486-63829-4.

[Sub99] V.S. Subrahmanian. “Nonmonotonic logic programming”. In: *IEEE Transactions on Knowledge and Data Engineering* 11.1 (1999), pp. 143–152. DOI: 10.1109/69.755623.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development - Technology, Engineering, Management.* 1st ed. Wiley, 2006. ISBN: 978-0-470-02570-3.

[SW10] Wilhelm Schäfer and Heike Wehrheim. “Model-Driven Development with Mechatronic UML”. In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday.* LNCS. Springer, 2010, pp. 533–554. DOI: 10.1007/978-3-642-17322-6_23.

[TBS20] Weslley Torres, Mark G. J. van den Brand, and Alexander Serebrenik. “A systematic literature review of cross-domain model consistency checking by model management tools”. In: *Software and Systems Modeling* (2020). DOI: 10.1007/s10270-020-00834-1.

[TCS18] K. Tuma, G. Calikli, and R. Scandariato. “Threat analysis of software systems: A systematic literature review”. In: *Journal of Systems and Software* 144 (2018), pp. 275–294. DOI: 10.1016/j.jss.2018.06.073.

[TH16] Emre Taspolatoglu and Robert Heinrich. “Context-Based Architectural Security Analysis”. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture.* WICSA’16. 2016, pp. 281–282. DOI: 10.1109/WICSA.2016.55.

[Tor05] P. Torr. “Demystifying the threat modeling process”. In: *IEEE Security Privacy* 3.5 (2005), pp. 66–70. DOI: 10.1109/MSP.2005.119.

[TSB19] K. Tuma, R. Scandariato, and M. Balliu. “Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis”. In: *Proceedings of IEEE International Conference on Software Architecture.* ICSA’19. IEEE, 2019, pp. 191–200. DOI: 10.1109/ICSA.2019.00028.

[Val+16] Tassio Vale et al. “Twenty-eight years of component-based software engineering”. In: *Journal of Systems and Software* 111 (2016), pp. 128–148. DOI: 10.1016/j.jss.2015.09.019.

[Voo20] David P. Voorhees. “Software Design and Security”. In: *Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models.* Texts in Computer Science. Springer, 2020, pp. 351–360. DOI: 10.1007/978-3-030-28501-2_24.

[Wan+09] Lei Wang et al. “TMAC: Taint-Based Memory Protection via Access Control”. In: *Proceedings of International Conference on Dependability.* DEPEND’09. 2009, pp. 19–27. DOI: 10.1109/DEPEND.2009.33.

[WBL93] Tim Wahls, Albert L Baker, and Gary T Leavens. *An Executable Semantics for a Formalized Data Flow Diagram Specification Language.* Technical Report TR93-27. Iowa State University, 1993, p. 29. URL: `https://web.archive.org/web/20200601123325/https://lib.dr.iastate.edu/cs_techreports/160/` (visited on 02/27/2022).

[Wei18] Herb Weisbaum. *Trust in Facebook has dropped by 66 percent since the Cambridge Analytica scandal.* 2018. URL: `https://web.archive.org/web/20210820004535/https://www.nbcnews.com/business/consumer/trust-facebook-has-dropped-51-percent-cambridge-analytica-scandal-n867011` (visited on 02/27/2022).

[WM85] Paul T. Ward and Stephan J. Mellor. *Structured Development for Real-Time Systems.* Vol. 2: Essential modeling techniques. Yourdon, 1985. ISBN: 0-13-854795-5.

[WSK20] Dominik Werle, Stephan Seifermann, and Anne Koziolek. "Data Stream Operations as First-Class Entities in Component-Based Performance Models". In: *Software Architecture.* LNCS. Springer, 2020, pp. 148–164. DOI: `10.1007/978-3-030-58923-3_10`.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks". In: *Proceedings of USENIX Security Symposium.* USENIX Association, 2006, pp. 121–136. URL: `https://web.archive.org/web/20170811171351/https://www.usenix.org/legacy/events/sec06/tech/full_papers/xu/xu.pdf` (visited on 02/27/2022).

[Xio+17] Hao Xiong et al. "DFDVis: A Visual Analytics System for Understanding the Semantics of Data Flow Diagram". In: *Proceedings of International Conference of Pioneering Computer Scientists, Engineers and Educators.* ICPCSEE'17. Springer, 2017, pp. 660–673. DOI: `10.1007/978-981-10-6385-5_55`.

[XL19] Wenjun Xiong and Robert Lagerström. "Threat modeling - A systematic literature review". In: *Comput. Secur.* 84 (2019), pp. 53–69. DOI: `10.1016/j.cose.2019.03.010`.

[Yam+12] Mark Yampolskiy et al. "Systematic analysis of cyber-attacks on CPS-evaluating applicability of DFD-based approach". In: *Proceedings of International Symposium on Resilient Control Systems.* ISRCS'12. 2012, pp. 55–62. DOI: `10.1109/ISRCS.2012.6309293`.

[Zda04] Steve Zdancewic. "Challenges for Information-flow Security". In: *Proceedings of the International Workshop on Programming Language Interference and Dependence.* PLID'04. invited paper. 2004, p. 5. URL: `https://web.archive.org/web/20210730102056/http://www.cis.upenn.edu/~stevez/papers/Zda04.pdf` (visited on 02/27/2022).

[ZG02] Xiaojin Zhu and Zoubin Ghahramani. *Learning from Labeled and Unlabeled Data with Label Propagation.* Technical Report CMU-CALD-02-107. Carnegie Mellon University, USA, 2002. URL: `https://web.archive.org/web/20220101183440/http://mlg.eng.cam.ac.uk/zoubin/papers/CMU-CALD-02-107.pdf` (visited on 02/27/2022).