# Enabling the Information Transfer between Architecture and Source Code for Security Analysis

Bachelor's Thesis of

Johannes Häring

at the Department of Informatics
Institute of Information Security and Dependability (KASTEL)

| | |
|---|---|
| Reviewer: | Prof. Dr. Ralf H. Reussner |
| Second reviewer: | Prof. Dr.-Ing. Anne Koziolek |
| Advisor: | M. Sc. Frederik Reiche |
| Second advisor: | Dr. rer. nat. Robert Heinrich |

06. June 2021 – 06. October 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

**Karlsruhe, 06.10.2021**

Johannes Häring

(Johannes Häring)

# Abstract

Many software systems have to be designed and developed in a way that specific security requirements are guaranteed. Security can be specified on different views of the software system that contain different kinds of information about the software system. Therefore, a security analysis on one view must assume security properties of other views. A security analysis on another view can be used to verify these assumptions. We provide an approach for enabling the information transfer between a static architecture analysis and a static, lattice-based source code analysis. This approach can be used to reduce the assumptions in a component-based architecture model. In this approach, requirements under which information can be transferred between the two security analyses are provided. We consider the architecture and source code security analysis as black boxes. Therefore, the information transfer between the security analyses is based on a megamodel consisting of the architecture model, the source code model, and the source code analysis results. The feasibility of this approach is evaluated in a case study using Java Object-sensitive ANAlysis and Confidentiality4CBSE. The evaluation shows that information can be transferred between an architecture and a source code analysis. The information transfer reveals new security violations which are not found using only one security analysis.

# Zusammenfassung

Viele Softwaresysteme müssen so konzipiert und entwickelt werden, dass bestimmte Sicherheitsanforderungen gewährleistet sind. Die Sicherheit kann auf verschiedenen Sichten auf das Softwaresystem spezifiziert werden, die unterschiedliche Arten von Informationen über das Softwaresystem enthalten. Daher muss eine Sicherheitsanalyse auf einer Sicht von den Sicherheitseigenschaften auf anderen Sichten ausgehen. Eine Sicherheitsanalyse auf einer anderen Sicht kann zur Überprüfung dieser Annahmen verwendet werden. Wir stellen einen Ansatz vor, der den Informationstransfer zwischen einer statischen Architekturanalyse und einer statischen, gitterbasierten Quellcodeanalyse ermöglicht. Dieser Ansatz kann verwendet werden, um die Annahmen in einem komponentenbasierten Architekturmodell zu reduzieren. In diesem Ansatz werden Voraussetzungen entwickelt, unter denen Informationen zwischen Sicherheitsanalysen ausgetauscht werden können. Wir betrachten die Architektur- und Quellcode-Sicherheitsanalyse als Black Boxes. Daher basiert der Informationstransfer zwischen den Sicherheitsanalysen auf einem Megamodell, das aus dem Architekturmodell, dem Quellcodemodell und den Ergebnissen der Quellcodeanalyse besteht. Die Machbarkeit dieses Ansatzes wird in einer Fallstudie mit Java Object-sensitive ANAlysis und Confidentiality4CBSE evaluiert. Die Auswertung zeigt, dass Informationen zwischen einer Architektur- und einer Quellcodeanalyse übertragen werden können. Der Informationstransfer deckt neue Sicherheitsverletzungen auf, die mit nur einer Sicherheitsanalyse nicht gefunden werden.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Security is a critical property of many software systems. Moreover, the security of software systems has to be guaranteed before the deployment to avoid zero-day-exploits. Guaranteed security is achieved by analyzing the security of a software system. A popular approach for building software is model-driven software development. Model-driven software development enables the developer to build software based on different abstraction views. The developer can specify security on different views each containing other information about the software system. For example, security can be specified on an architecture model, in which the components are defined without the internal behavior. Therefore, the developer has to make assumptions about the internal behavior that may not hold. However, if the developer only specifies security on the source code level, there is, for example, no information about the deployment of the software system. Therefore, using only one view results in an incomplete picture of the security of the software system.

Assumptions in one view can be verified with security analyses on other views. In this bachelor's thesis, an approach to transfer information between an architecture analysis and a source code analysis to reduce the assumptions in the architecture model is proposed. To the best of our knowledge, there is currently no such approach. The architecture analysis, respectively, the source code analysis is based on an architecture model respectively a source code model, both with security specifications. The first goal of this bachelor's thesis is to define requirements for analyses to enable the information transfer between them. The second goal is to develop an approach for exchanging information between the analyses. In this bachelor's thesis, only static security analyses are considered. Furthermore, the domain of the source code analyses is restricted to the domain of lattice-based security analyses.

The expected benefit of this bachelor's thesis is that assumptions of an architecture security analysis can be corrected with a source code analysis by transferring information between them. Additionally, a new analysis can be designed to be ready to exchange information with existing security analyses.

In chapter 2, the foundations for this approach to transfer information between security analyses are introduced. The related work to this bachelor's thesis is presented in chapter 3. Chapter 4 gives an overview of the concept of this approach. Security analyses have to fulfill requirements to be ready to exchange information. These requirements are described in chapter 5. If the analyses fulfill all requirements, the analyses can be connected. In chapter 6, it is presented how security analyses are connected. This approach to transfer information between two security analyses is evaluated in a case study with Confidentiality4CBSE and Java Object-sensitive ANAlysis. The instantiation of this approach with Confidentiality4CBSE and Java Object-sensitive ANAlysis is described in chapter 7. The evaluation is described in chapter 8. The conclusion and future work are presented in chapter 9.

# 2 Foundations

The foundations of this bachelor's thesis are presented in this chapter. The approach to transfer information between security analysis uses models of the software system. Therefore, model-driven software development is introduced in section 2.1. In section 2.2, component-based software engineering is described because this approach considers only component-based software systems. Section 2.3 introduced the Palladio Component Model that is used in the case study for the evaluation. The assume-guarantee approach is presented in section 2.4 since, for example, the architecture analysis guarantees a secure software system assuming a secure implementation of the components. Security analyses examine security properties which can be described as (hyper-)trace properties. Therefore, (hyper-)trace properties are described in section 2.5. In section 2.6, the concept of security lattices is shown because the source code analyses are restricted to the domain of lattice-based security analyses. In this approach, two static security analyses are connected. Therefore, security analyses are presented in section 2.7.

## 2.1 Model Driven Software Development

Model-driven software development (MDSD) is based on models for different phases (e.g. analysis, requirements analysis, design) and domains (e.g. software architect, programmer). Models abstract from the final implementation, which makes them closer to the problem domain than if it is implemented with a programming language [1].

### 2.1.1 Models and Metamodels

Models are used in many disciplines (e.g. physics) to describe a representation of reality. According to Stachowiak's "General Model Theory", models have three properties. The *mapping* property states that a model always represents a reality. The *abstraction* property declares that not all properties of reality are represented but only those properties being relevant to the model domain. The third property, *pragmatism*, declares that the model is created for a specific purpose [2].

In MDSD, these three properties must be supplemented [3]. Models must additionally be *understandable* since only abstracting away details is not sufficient. A software model must be *accurate* concerning the properties being of interest to the system. For example, a model of a decentralized computer system should specify how the different computers communicate. The accurate property is necessary to make predictions about the resulting software. The last must-have property is that it must be *significantly cheaper* to develop such a model than programming the software directly. Furthermore, the goal of MDSD is to use models as first-class development artifacts. For example, models can be used for source code generation or analyses of the software system. Therefore, a fundamental rule system, which is called metamodel [4] is necessary to describe models. Figure 2.1 shows, for example, the relationships between models and metamodels. The class diagram models software from the real world. The class diagram is modeled by a class metamodel.

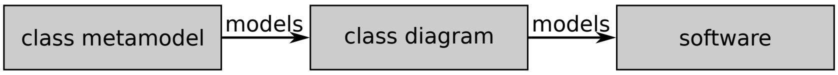


Figure 2.1: Relation between metamodels and models

Source code can also be viewed as a model with the definition of Stachowiak [2]. Source code is an abstract view of a running software system, which is the reality in the context of software development. Furthermore, using source code to model the software system is pragmatic because the source code supplies a better understanding of the software system than, for example, a sequence of binary digits.

### 2.1.2 Model Merging

Often, a modeling task is distributed to different team members each working on a partial view of the overall system. During the development, the different models have to be merged. Model merging is based on related operations, like comparing models, checking their consistency, and finding matches between them. These operators can be characterized as algebraic operators [5].

The *merge* operator is defined by Brunet et al. [5] as:

$$\textbf{merge} : model \times model \times relationship \rightarrow model \tag{2.1}$$

The *relationship* specifies how the models as a whole relate to one another and also defines mappings between the elements of the models, for example, a mapping of stereotypes in Unified Modelling Language (UML) to annotations in the source code model. The *relationship* also represents a connector between two models. It preserves information as well as structure. It maps the source model to the target model and vice versa. A possible implementation of the *merge* operator would be to describe each model as a graph and each mapping in the relationship as a binary relation between two sets of model elements. The two model graphs and the binary relation are forming an interconnection diagram $D = \langle M_1, M_2, R\rangle$. The algorithm of *merge* then unifies the elements in $M_1$ and $M_2$ which are in the same equivalence group induced by $R$. If an element of one model is not in $R$, it falls in its equivalence group.

The *merge* operator is supported by the *match* and the *diff* operation:

$$\textbf{match} : model \times model \rightarrow relationship \tag{2.2}$$

$$\textbf{diff} : model \times model \rightarrow transformation \tag{2.3}$$

The *match* operation produces the relationship which can be used for the merging. It finds commonalities between the two models. The result of *match* may be under-determined because there are maybe multiple possible relationships between models [6]. The *diff* operation does not provide a relationship but a possible transformation from one model to the other. A transformation is a sequence of edit actions [5].

### 2.1.3 Model Transformation

In section 2.1.2, a model transformation is described as a sequence of edit actions. In this section, the description is explained in more detail. Model transformations are relations between models in the same or different modeling languages [7]. Therefore, a model transformation from model A to model B can also be defined as a sequence of edit actions [5]. A model transformation is correct if it fulfills the three requirements *representation*, *syntactic correctness* and *semantic correctness* [7]. A model transformation can be classified as *endogenous* or *exogenous* [8]. An endogenous transformation is between two models with the same metamodel. Whereas, an exogenous transformation is between two models with different metamodels. For example, if class names in an UML class diagram are changed, then the model transformation between the unchanged and changed UML class diagram is endogenous because both are based on the UML metamodel. Whereas, if C# source code is generated from the UML class diagram, it is an exogenous transformation because one model is based on the UML metamodel and the other on the C# metamodel. In the case of an exogenous model transformation, a relationship between the two metamodels is needed [9].

A subset of model transformations is the set of bi-directional model transformations. A bi-directional model transformation is a transformation from model A to model B that is reversible without specifying an extra transformation from model B to model A [10]. A bi-directional model transformation can be described by a triple graph grammar [11].

### 2.1.4 Model Sewing and Waving

The *merge* operation on two models presented in section 2.1.2 can be divided into different kinds of merging, for example, *weaving* and *sewing* [12]. Weaving two models provide tight integration of the two models. Weaving merges the two metamodels based on a weaving specification. For example, if both metamodels have one common metamodel element, an example weaving specification defines that the models are woven based on the common element. The woven metamodel has all metamodel elements from both metamodels, but only once the common metamodel element. The woven model is based on the woven metamodel. The two models are also woven together based on the weaving specification. Sewing provides a loose coupling of the two models based on *mediators*. A mediator supplies a description of how the two models are sewed. An example mediator between two models updates certain model elements in the second model if one model element in the first model is changed. The mediator is based on the two metamodels and is initialized with the two models.

### 2.1.5 Triple Graph Grammars

Triple Graph Grammars are a technique for defining the correspondence between two different types of models in a declarative way [13]. A triple graph grammar consists of three graphs. Two graphs describe the two domain models, one graph for each domain model. There are no direct connections between the two domain graphs. The relation between the two domain graphs is established by the third graph, a correspondence graph. A node of the correspondence graph connects sets of nodes of the two domain graphs. A node of the correspondence graph represents a rule under which nodes of the two domain models can be connected [14]. If two nodes of the correspondence graph are dependent on each other, then there is an edge between the two nodes.

## 2.2 Component-Based Software Engineering

A definition of a software component is stated in [15] as follows:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Thus, a component specifies dependencies and interfaces on which the component can be called. However, what is part of the context is vague [16]. Furthermore, a component has not to specify any internal structure.

Multiple components can be combined into a component-based software system, where the different components are exchanging information over the defined interfaces of the components [17]. The message exchange between two components can be, for example, realized by a method call or an HTTP request. The source code metamodel of a programming language supporting component-based software development, like Java, C# or Haskell, consists at least of classes and methods where methods are always part of classes [18]. A method has a signature and a body. The source code skeleton of a component-based software system consists of the defined interfaces to exchange information between the components. An interface bundles a set of methods [15].

The primary objective of component-based software engineering is to develop a software system as an assembly of parts (components). Those parts then can be easily reused, maintained, and upgraded [17].

## 2.3 Palladio Component Model

The Palladio Component Model (PCM) is a software architecture simulation approach [19]. The two key features of the PCM are the parameterised component quality of system specification and the developer role concept [20, p. 12]. The PCM provides different views of the software system, like the *allocation*, *repository*, *resource environment*, and *system view*.

In this bachelor's thesis, the repository view is used. The repository view is used to specify the component architecture of a software system [20, p. 46]. In the repository model the components, interfaces (incl. operation signatures), and data types are specified. The components are modeled with BasicComponents, the interfaces with OperationInterface, and the operations with OperationSignature. The data types can be realized with BasicDataTypes, CollectionDataTypes, and CompositeDataTypes. Additionally, the interfaces are connected to components with ProvidedRole and RequiredRole.

## 2.4 Assume-Guarantee Approach

Formal verification is important for safety-critical software systems. The problem for concurrent software of industrial complexity is the exponentially increasing number of reachable states in the software system. Two approaches to solve this issue is abstraction [21] and compositional reasoning [22, 23]. The main concept of compositional reasoning is the assume-guarantee approach [22]. The idea is to verify each component independently by assuming its environment and then discharge the assumption on the collection of components.

There are a variety of assume-guarantee proof-rules, for example, the non circular rule AG-NC:

$$\frac{M_1||M_a \preccurlyeq S \qquad M_2 \preccurlyeq M_A}{M_1||M_2 \preccurlyeq S} \tag{2.4}$$

where $M_1||M_2$ is the concurrent system to be verified. The verification is based on the specification $S$, and $\preccurlyeq$ is a notion of conformance between the system and the specification. The rule states, if $M_2$ satisfies the assumption $M_A$ and the combination $M_1||M_A$ conforms $S$, then also $M_1||M_2$ conforms $S$. The main drawback of these rules is that the assumption has to be manually crafted [24].

The assume-guarantee approach is also used for verification of architecture models [25]. The architecture model includes interfaces, interconnections, and specifications for components but not their implementation. When reasoning over the architecture model, $M_A$ is the assumption about a correct implementation of the components. The model $M_1$ is the architecture model, which can be verified independently from components implementation. For a complete verification of the software system, the implementation of the different components has to be verified.

## 2.5 (Hyper-)Traceproperties

Security properties of a software system can be unified with trace properties. A security property either holds or does not hold. Therefore, if the security property holds, the trace property holds respectively. If the security property does not hold, the trace property does also not hold [26].

The software system is modeled as a labelled transition system (lts) model. A lts is a triple $(S, T, \rightarrow)$, where $S$ is a set of states, $T$ is a set of labels (actions) and $\rightarrow$ is a set of labeled transitions. A transition from state $S_1$ to $S_2$ by the action $\alpha$ is written as $(S_1, \alpha, S_2) \in\rightarrow$ or equivalently $S_1 \xrightarrow{\alpha} S_2$ [27]. A trace is defined as a sequence of abstract states. For instance, the abstract states can encode the initialization of fields with methods [26]. The set of abstract states are denoted by $\Sigma$. Traces may be finite or infinite traces over $\Sigma$, categorized into sets:

$$\Psi_{fin} = \Sigma^* \tag{2.5}$$

$$\Psi_{inf} = \Sigma^\omega \tag{2.6}$$

$$\Psi = \Psi_{fin} \cup \Psi_{inf} \tag{2.7}$$

The set $\Sigma^*$ is the set of all finite sequences over $\Sigma$, and respectively $\Sigma^\omega$ is the set of all infinite sequences over $\Sigma$. Each finite trace can be represented by an infinite trace by infinitely adding the final state at the end of the finite trace. For a trace $t = s_0 s_1 ...$ with $s_i \in \Sigma$ and an index $i \in \mathbb{N}$, the indexing notation is defined as:

$$\begin{aligned} t[i] &= s_i \\ t[..i] &= s_0 s_1 ... s_i \\ t[i..] &= s_i s_{i+1} .. \end{aligned}$$

A trace property is a set of infinite traces and the set of all trace properties is $Prop = \mathcal{P}(\Psi_{inf})$ where $\mathcal{P}$ is the power set. A set of traces $T$ satisfies the trace property $P$, denoted by $T \models P$ if all traces of $T$ are in P:

$$T \models P \Leftrightarrow T \subset P$$

Hyperproperties are defined accordingly. Hyperproperties are the intersection of hyperliveness and hypersafety properties. A hyperproperty is a set of sets of infinite traces. Therefore, the set of all hyperproperties is:

$$HP = \mathcal{P}(\mathcal{P}(\Psi_{inf})) = \mathcal{P}(Prop) \tag{2.8}$$

Analogue to the a trace property, a set of traces $T$ satisfies the hyperproperty H, if $T$ is a subset of $H$:

$$T \models H \Leftrightarrow T \subset H \tag{2.9}$$

## 2.6 Lattice-Based Security Property

A lattice-based security property is a security property that is based on a security lattice. A security lattice $\langle S, \rightarrow, \oplus \rangle$ is a finite partially ordered set of security classes [28]. $\rightarrow$ is a flow relation between pairs of security classes. The relation is transitive and reflexive. The security classes are usually assigned to objects. In the context of software engineering, such objects can be, for example, classes, methods, or fields. If information flows from object $x$ to an object $y$, then there is also an information flow from the security class of $x$ to the security class of $y$ [29]. If there is a relation from security class A to security class B, then information is allowed to flow from the objects with security class A to the objects with security class B. However, information is not allowed from security class A

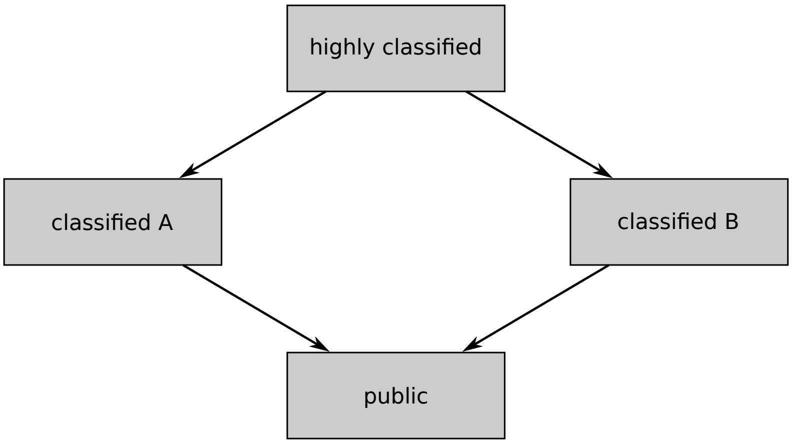


Figure 2.2: Example lattice with the four security classes highly classified, classified A, classified B and public

to security class B. The least-upper-bound operator ⊕ [28] combines two classes into the least upper bound class of the two classes. In the small example of the security classes A and B, the least upper bound of A and B would be A because the relation can be expressed as $B \leq A$ and A is therefore above B.

A lattice can be displayed as a directed graph without cycles. An example lattice in the form of a directed graph is shown in figure 2.2. There are four security classes highly classified, classified A, classified B and public. Someone with the security level highly classified is allowed to read all information. A person with security level classified A can only read information with the security class classified A or public, but not information with security class classified B. The least-upper-bound of classified A and classified B is classified A ⊕ classified B = highly classified, because information can flow from highly classified to classified A as well as to classified B.

If a lattice-based security property holds for a software system, then there are only information flows that are contained in the lattice [30].

## 2.7 Static Security Analyses

In MDSD, the models are not only used for exchanging information about the software but can also be used for static analyses of the software system in different states of development. The analyses can be run on different views on the software system. For example, UMLSec is a security analysis based on an architecture view on the software [31], whereas the KeY framework can be used for analyzing a source code model [32].

Static analysis tools examine the model of the used software. If the source code view is used, the source code model can be compiled but not executed. As an example, the security analysis Java Object Sensitive ANAlysis (JOANA) uses a source code model in its compiled form [33]. The tools search for a fixed set of patterns or rules [34]. For instance, a static analysis based on the source code can search for private API keys with grep. Static security analyses can be opposed to dynamic security analyses that are analyses used in production to, for example, recognize an attacker in the network. The advantage of static analyses is these can be run before deploying the software or be part of a CI/CD Pipeline. Furthermore, a static security analysis examines a security property. A security property can be modeled as (hyper-)trace property [35].

Static security analyses for component-based applications are often using an assume-guarantee approach [36]. The analysis is run for each component independently, and then an analysis using only the components interfaces is run. One could argue assuming component A and component B have no information leaks, and there is no insecure information flow between both components, it is guaranteed that the software is free of information leaks. In architecture analyses, the assumption about the components has to be made, which may be wrong, like seen in chapter 1. Such wrong assumptions are leading to false reports [37].

Static security analysis covers many areas of security. Possible areas of static analyses are looking for private information in the source code, like an API key or checking for reported vulnerabilities in dependencies. Another important static analysis is the information flow control (IFC). IFC guarantees that no information flows from high input to low output [28].

Examples for static security analyses on the architecture view are Access Analysis on PCM [38] and UMLSec on UML [31]. On the source code view, there are JOANA [39], KeY [32] or Reactive Information Flow Control for Java (JRIF) [40]. There are also some analyses which combine an architecture and a source code view like IFlow [41]. Access Analysis and its underlying analysis model is presented in section 2.7.1. JOANA is introduced in section 2.7.2.

### 2.7.1 Access Analysis and Confidentiality4CBSE

Access Analysis is an architecture security analysis that uses Confidentiality4CBSE. The Confidentiality4CBSE project aims to specify and analyze confidentiality for component-based software [42]. Confidentiality4CBSE extends the PCM, as introduced in section 2.3, with security specifications.

A Confidentiality4CBSE security model of a software system is based on two models, the confidentiality model and the adversary model [38]. In the confidentiality model, DataSets are defined. A DataSet is a group of data combined into one data class. A DataSet can represent a security level. The ParametersAndDataPair groups DataSets and assigns them to a parameter. Furthermore, locations and tamper protections for the component-based software system can be specified in the confidentiality model. Those two are specified in the same way as the security level. In the adversary model, the Adversaries are defined. Adversaries are actors who can interact with the system. An individual Adversary consists of the DataSets that the Adversary is allowed to know and its name.

The confidentiality model elements are assigned to model elements of PCM models with stereotypes. For example, in the PCM repository model, methods of interfaces are assigned the stereotype InformationFlow. In this stereotype, the corresponding ParametersAndDataPairs are specified, indicating the security level of the method.

Access Analysis uses a set of Prolog rules to determine the security of the software system. The highest rules is `isInSecureWithRespectTo` which has other rules as condition. The modeled software system with its security model is transformed into Prolog code on which the defined rules are applied. The result of Access Analysis is a proof tree. All determined proof trees are displayed in text form that is printed in text form. The number of proof trees can be determined with the keyword isInSecureWithRespectTo(.

### 2.7.2 Java Object-sensitive ANAlysis

JOANA is a non-interference analysis of Java source code. JOANA uses the model elements Source, Sink, and EntryPoint to examine the source code. These model elements are connected to the Java source code. A possible mechanism to assign the model elements to the Java source code is through annotations. A method or its parameters can be annotated as Source, Sink, or EntryPoint. An EntryPoint is an entry point into the software from which the software is analyzed. A Source and Sink consists of a tag and a security level. An EntryPoint consists of a tag, security levels and a security lattice, as introduced in section 2.6. The lattice of an EntryPoint consists of a list of may flow annotations. JOANA has also a declassification mechanism.

JOANA is run from an EntryPoint. A call tree is built starting with the method with EntryPoint. JOANA searches in the call tree for invalid information flows. JOANA is designed to annotate the main method of a program with an EntryPoint. However, other methods can be an EntryPoint too. The problem is that using an arbitrary method as EntryPoint does not guarantee that every field and parameter is initialized. Therefore, JOANA provides an uninitialized fields helper for this case.

In figure 2.3a, an excerpt of an annotated Java source code is shown. The method generateTicket is annotated with an EntryPoint. The EntryPoint contains the lattice *High* $\longrightarrow$ *Low* and the tag 1. The method String getTicket() is annotated as Source, which assigns *String getTicket()* the security level high. The method *printTicket(String)* is annotated as Sink, which assigns printTicket(String) the security level Low. If JOANA is run on this source code excerpt, an illegal information flow is found. The result of the JOANA analysis is presented in figure 2.3b. The result contains the EntryPoint which is used to find the illegal information flow. The result contains all detected invalid flows. In the presented example, only one invalid flow is found. An invalid flow has a source and a sink which both are specified in the result. The structure of a JOANA result is shown in figure 2.4. The result contains an entry for the EntryPoint from which the analysis is run. The tag entry specifies the tag of the EntryPoint and the examined Sinks and Sources. If the analysis finds invalid flows the entry found_flows is set to true otherwise to false. JOANA can find direct and indirect information flows. The entry only_direct_flow specifies which kind of flows are searched. The found flows are divided by hyphens. A flow consists of four entries. The first entry type indicates the type of the flow, for example, illegal. The second entry is attacker_level. An invalid flow in JOANA is between a Source and a Sink. Therefore, the flow contains entries for source and sink. If the Source or Sink are annotated to a parameter, the type is edu.kit.joana.api.sdg.sdgformalparameter otherwise it is edu.kit.joana.api.sdg.sdgmethod. The source and sink entry contains the entries class, name, selector, return, parameters, and source_level or sink_level. The annotated method is specified by name, parameter, return, and selector entries. If the annotation is of the type edu.kit.joana.api.sdg.sdgformalparameter, then it has also the entry index which indicates the parameter index of the annotated parameter.

```java
public class TicketHandling {
    @EntryPoint(tag = "1", levels = {"Low","High"},
        lattice = {@MayFlow(from ="High", to = "Low")})
    public void generateTicket() {
        String ticket = getTicket();
        printTicket(ticket);
    }
    @Source(tags = {"1"}, level = "high")
    private String getTicket() {
        ...
        return ticket;
    }
    @Sink(tags = {"1"}, level = "low")
    private void printTicket(String ticket) {
        ...
    }
}
```

(a) Example for the JOANA annotations in a Java source code excerpt

```
entry_point_method: void TicketHandling.generateTicket()
tag: 1
found_flows: true
only_direct_flow: false
flow:
  -
    type: illegal
    attacker_level: Low
    source:
      kind: edu.kit.joana.api.sdg.sdgformalparameter
      method:
        class: TicketHandling
        name: getTicket
        selector: getTicket()Ljava/lang/String
        return: String
        parameters:
      index: 1
      type: java.lang.Object
      source_level: High
    sink:
      kind: edu.kit.joana.api.sdg.sdgmethod
      class: TicketHandling
      name: printTicket
      selector: printTicket(S)V;
      return: void
      parameters:
        - String
      sink_level: Low
  -
```

(b) JOANA result of the source code excerpt

Figure 2.3: Annotated source code with corresponding JOANA result

```
entry_point_method:
tag:
found_flows:
only_direct_flow:
flow:
  -
    type:
    attacker_level:
    source:
      kind: edu.kit.joana.api.sdg.sdgmethod
      class:
      name:
      selector:
      return:
      parameters:
        -
      source_level:
    sink:
      kind: edu.kit.joana.api.sdg.sdgformalparameter
      method:
        class:
        name:
        selector:
        return:
        parameters:
          -
      index:
      type:
      sink_level:
```

Figure 2.4: Structure of a JOANA result

# 3 Related Work

In this chapter, the work related to this bachelor's thesis is presented. We are not aware of any approach for exchanging information between security analyses on an architecture view and a source code view. In section 3.1, an approach to generate source code with confidentiality annotations is presented. In section 3.2, the IFlow analysis is presented, which combines a formal and a source code model. In section 3.3, the iObserve approach is introduced. In section 3.4, the VITRUV approach is described. section 3.5 presents an approach to connect an architecture model with a source code model. In section 3.6, an approach for a transition from a model to a formalism to a result is presented.

## 3.1 Palladio Component Model with Confidentiality Model to Java with JML Proof Obligations for KeY

Yurchenko et al. [43] propose a framework to generate Java source code with Java Modelling Language (JML) proof obligations. The JML proof obligations can be verified with the KeY framework [32]. They use PCM with Confidentiality4CBSE, as introduced in section 2.7.1. The Java method stubs are generated from the PCM instance with a generator based on Xtend. The generator produces Java annotations from the confidentiality specification. Those annotations are translated to JML proof obligations. The method stubs are manually completed.

The JML proof obligations are verified with KeY. The source code and the model are held consistent with the VITRUVIUS framework.

In comparison to the approach of Yurchenko et al. [43], this bachelor's thesis presents a more general approach because our approach is not only restricted to PCM with confidentiality model and KeY. Furthermore, Yurchenko et al. [43] do not consider the results of KeY. Whereas, in this bachelor's thesis, the architecture model is adapted based on the results of the source code analysis.

## 3.2 IFlow

IFlow is a security analysis specialized on IFC [44]. IFlow is using a model-driven approach, which is shown in figure 3.1. The starting point is an abstract UML model. It contains a static view, capturing the software structure using class diagrams and also a dynamic view to specify the communication between the components. Both views have security annotations. The annotations restrict the information flow and can be automatically checked with formal verification.

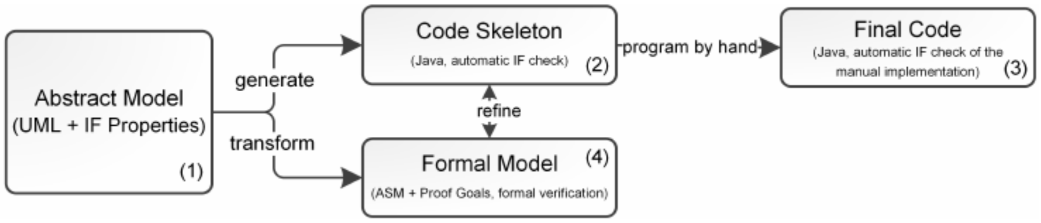

Abstract Model
(UML + IF Properties)
(1)
generate
transform
Code Skeleton
(Java, automatic IF check)
(2)
program by hand
Final Code
(Java, automatic IF check of the manual implementation)
(3)
refine
Formal Model
(4)
(ASM + Proof Goals, formal verification)

Figure 3.1: The IFlow approach [44]

In the next step, Java code is generated from the architecture models. The component structure, as well as their communication and calls to manual methods, are generated in this step [45]. IFlow provides a Java framework to which the code skeleton is linked. The frameworks abstracts from system-specific Application Programming Interfaces (APIs) enabling the developer to use IFC tools like JOANA. The code skeleton is manually implemented. The final code is examined with IFC tools to verify a correct implementation.

The UML model is transformed into a formal model based on an abstract state machine (ASM) used for the interactive theorem prover KIV. The developer can verify information flow (IF) properties beyond simple noninterference, such as constraints on data declassification [46].

If the code skeleton is implemented following rules, the results of the interactive theorem prover and the IFC on the source code have a refine relationship [47].

To distinguish this work from this bachelor's thesis, IFlow does not consider the transfer of results of the supplier analysis back to an analysis on the architecture view. In IFlow, there are no analyses on the architecture view at all.

## 3.3 iObserve

The iObserve approach of Heinrich et al. [48] has the target to combine operation-level adaption and development-level evolution by establishing a correspondence model between the architecture and run-time models. A megamodel integrates design-time models, code generation, monitoring, and run-time model updates [49]. The run-time models are updated with the monitoring data.

The run-time architecture correspondence model (RAC) defines a language to describe the relationship between a measurement meta-model and a run-time prediction element. One relation is a unidirectional mapping between one or more monitoring records and a certain PCM element. New monitoring data triggers a change event which updates the PCM element [50].

The iObserve approach projects the results of monitoring data of the deployed software system back into the architecture analysis model. Whereas, in this bachelor's thesis, static security analyses are considered. Furthermore, a general approach to exchange information between security analyses is developed.

## 3.4 VITRUVIUS Approach

The VITRUVIUS approach developed by Klare et al. [51] is an approach for keeping different views on the software system consistent. The VITRUVIUS approach is based on the orthographic software modelling (OSM) paradigm. They propose a concept for a virtual single underlying model (VSUM) metamodel, which has an internal structure of connected metamodels. However, if it is viewed externally, it appears as one monolithic metamodel. The different views are created from an instance of the VSUM.

The VITRUVIUS approach achieves consistency preservation between different views by introducing two new consistency preservation languages based on an incremental and delta-based approach. The languages rely on a correspondence model between the instances of the individual metamodels of the VSUM. The correspondence model contains a list of correspondences. A correspondence is between two lists of model elements and has a tag to identify the correspondence. Furthermore, the correspondence can depend on other correspondences. If one view is changed, the instance of the individual metamodel from which the view is created is changed. Those changes are propagated to other dependent metamodels in the VSUM with the help of the correspondence model.

The VITRUVIUS approach proposes an approach for preserving consistency between different views while the changes are made by a developer. However, in this bachelor's thesis, there must not be any changes in the source code view that could be propagated back to the architecture view. Only the results of the analysis on the source code view are propagated to the architecture view.

## 3.5 Connection of Architecture and Source Code Model

Approaches to connect an architecture model with a source code model exist. Konersmann [52] presents an approach for a concrete mapping between an architecture and a source code model. Konermann introduces metamodel notations to define formal mappings between architecture and source code model elements. Tooling to create bidirectional transformations between an architecture and source code model based on the metamodel notations is implemented. In his evaluation, the connection between the PCM and Java source code model is shown.

Kramer et al. [53] have a similar approach to connect an architecture model with contracts and a source code model with a bidirectional model transformation which uses a concrete mapping between the two meta models. Their objective is to use the bidirectional model transformation in combination with a change detection system to propagate changes between the two models. They have also shown that the approach is feasible for PCM and Java source code model.

The approaches of Konersmann and Kramer et al. do not consider analyses that are run on the different models. They consider only mechanisms to preserving consistency between different models. Therefore, the approaches do not consider the projection of the source code analysis results back into the architecture model.

## 3.6 Transition from Model to Formalism to Result

Information flow analyses are based on a formalism, for example, Prolog. Hahner et al. [54] proposed a domain-specific language to define data flow constraints in an architecture model. The domain-specific language abstracts from the underlying formalism. Their work provides a mapping between the domain-specific language and formalism. The mapping is used to transform the domain-specific language to a formalism. A formalism can be used to analyze the data flow of the architecture. They also provide a transformation from the analysis results in the formalism analysis to the domain-specific language to present the results to the developer.

Hahner et al. develop an approach to transform data flow constraints in a domain-specific language into a formalism, and the results back into domain-specific language [55]. In this bachelor's thesis, an architecture model with security specification is transformed to a source code model with a security model instead of a formalism. Furthermore, Hahner et al. transform the results of the analysis back into the domain-specific language only to display them. Whereas, in this bachelor's thesis, the source code analysis results are used to adapt the security specification of the architecture model.

# 4 Overview of the Concept to Enable Information Transfer between Analyses

In this chapter, the overview of the approach proposed in this bachelor's thesis is presented. The chapter is divided into two sections. In section 4.1, the concept of the approach to transfer information between two security analyses is described. In section 4.2, a running example for a software system with an architecture and a source code model, both with security specifications, is presented. This example software system is used through the whole bachelor's thesis as a running example.

## 4.1 Approach to Transfer Information between Static Security Analyses

The goal of this bachelor's thesis is to enable the information transfer between security analyses on architecture models and source code models. In this section, the approach is presented from a high-level view.

Component-based software systems are considered. In section 2.2, a component is described as a unit of composition with interfaces and explicit context dependencies. In this bachelor's thesis, the components in the architecture model are modeled as black boxes. Black-box components are components with known interfaces but unknown internal implementation. Therefore, only the component, its interfaces, and the relations between the components are modeled in the architecture model. In this bachelor's thesis, source code is also seen as a model because source code fulfills the six model properties from section 2.1. The source code model contains information about the internal behavior of the components. Consequently, the source code model has more information about the internal behavior of the components than the architecture model [56].

Security analyses are run on a software model, for example, an architecture or a source code model, with a security model. A structural software model with a security model is called a *analysis model*. In this bachelor's thesis, the security model assigns a security specification to structural software model elements. Therefore, the structural software model and the security model are woven together based on the common structural software model elements, as introduced in section 2.1.4. In this approach for information transfer between architecture and source code, there are two analysis models. One is based on the architecture model, the other one on the source code model. The security analysis on the architecture model has to make assumptions about the components' behaviors since the components are black boxes. Therefore, it also assumes the security aspects concerning the internal behavior of the components. These assumptions are verified with the results of a matching security analysis on the source code model. Since the source-code security analysis verifies the assumptions of the architecture security analysis, the source-code security analysis is called *supplier analysis*. Respectively, the architecture security analysis is called *client analysis*. Therefore, the architecture analysis model is called the *client analysis model*. The source code analysis model is called the *supplier analysis model* because the analysis run on the supplier analysis model provides results with which the client analysis model is verified. This approach to connect security analyses treats the analyses as black boxes. Only the input, the analysis models, are given, and the output of the supplier analysis is used to adapt the client analysis model. Therefore, no changes in the analyses themselves have to be made.

The architecture and the source code model are not providing run-time information about the software system because neither of both models is deployed [57]. As soon as the source code model is deployed on a running system and the model is supplemented with run-time information, it is transformed into a run-time model. In this bachelor's thesis, the source code model is not deployed. Therefore, no dynamic security analyses can be used, and only static security analyses are considered.

In figure 4.1, the general concept of the approach to exchange information between static security analyses is shown. The supplier analysis model skeleton is generated from the client analysis model. The supplier analysis model skeleton is a source code skeleton with corresponding security information. Therefore, not the complete source code model can be generated from the client analysis model, especially not the implementation of the interface methods. Only a source code skeleton is generated from the client analysis model because it contains the component interfaces with its method signatures but not the internal behavior. Furthermore, the client analysis model contains the security specification for the interfaces and the interface methods. Therefore, the security specification of the source code skeleton can be generated. Since the source code skeleton and the security specification of it can be generated from the client analysis model, the supplier analysis model skeleton can be generated from the client analysis model.

The generation produces a correspondence model which maps the model elements of the client analysis model to the model elements in the supplier analysis model skeleton [58]. The correspondences exist between the architecture model and the source code model and between the two security models. The client analysis model skeleton is manually completed because the client analysis model contains no behavioral information about the software system. It must be noted that the generation process is only necessary for the creation of the correspondence model. If the supplier analysis model already exists, the correspondence model can also be manually created.

The analysis of the supplier analysis model produces supplier analysis results. The replacement of one or more of the assumptions in the client analysis model is achieved by projecting back the results of the supplier analysis into the client analysis model. The *back-projection* adapts the client analysis model based on the results of the supplier analysis [59]. The supplier analysis model is seen as the ground truth in this bachelor's thesis. Otherwise, an automatic back-projection is not possible. The back-projection is comparable with the iObserve approach presented in section 3.3. The workload specification in the design model is adapted based on monitoring data from a run time model. The back-projection requires a correspondence model between the client and supplier analysis model elements.

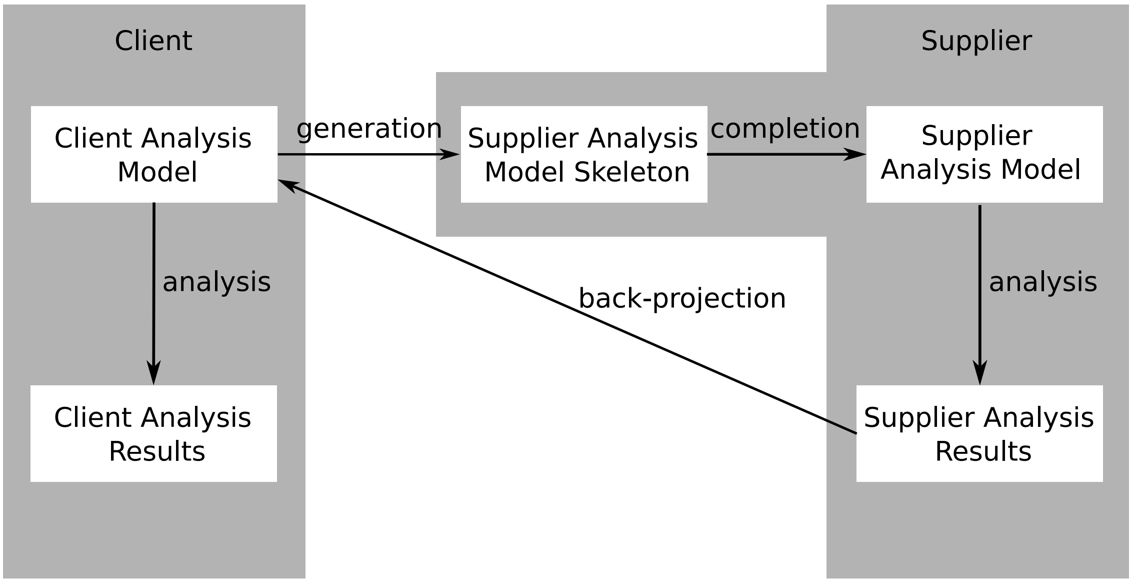


Figure 4.1: Schema of connecting a client analysis with a supplier analysis

The supplier analysis results cannot be automatically linked to the client analysis model elements without a correspondence model. A client analysis and a supplier analysis are called *coupled* if a correspondence model between the analysis models of the two analyses is initiated and a back-projection of the supplier analysis results is possible.

The back-projection first determines the correct security specification on the interface method in the supplier analysis model and then adapts the corresponding security specification in the client analysis model. The process to determine the correct security specification is specific to the domain of the analyzed security property. In this bachelor's thesis, this approach to couple security analyses is shown for the domain of lattice-based security properties, which are introduced in section 2.6.

## 4.2 Running Example

In this section, a running example is introduced to illustrate the approach to couple two security analyses. The software system, used in this example, provides the possibility to exchange information between different parties. The three parties are schools, clinics, and persons. The three parties can interact with the software system over the components School, Clinic and PersonUI. The PersonUI component provides the three interfaces EducationalDataExchange, CreditCardExchange and MedicalDataExchange. The component PersonUI requires the interface SchoolEducation which is provided by School. The component School requires the interface SchoolEducationExchange. The component Clinic requires the interfaces CreditCardExchange and MedicalDataExchange and provides the interface InternalLogistics.

The client analysis examines if the software system complies with a role-based access control schema. The client analysis model assumes that each interface method has a

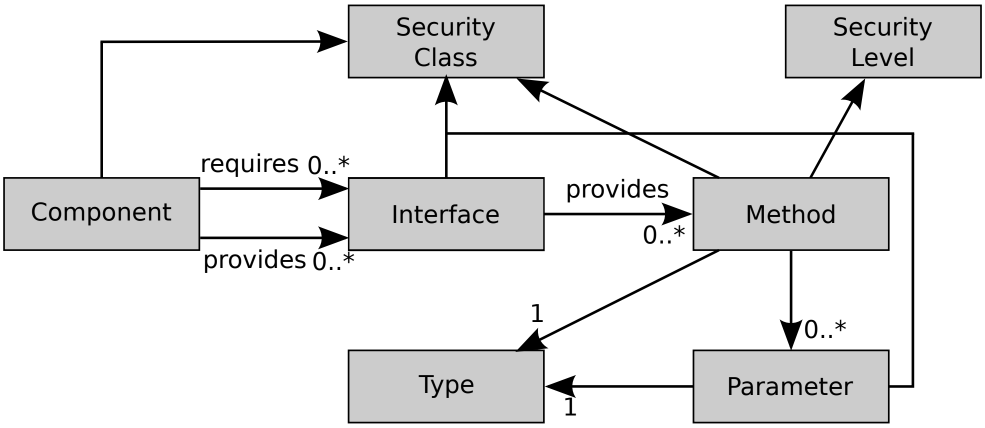


Figure 4.2: Client analysis metamodel for the running example

certain security level. The supplier analysis checks whether every information flow is in the lattice $L$ or not [60]. Therefore, the analysis reports an invalid information flow if information flows from object A with security level x to object B with security level y and $x \rightarrow y$ is not in the lattice $L$. The architecture model contains only components, which provide and require Interfaces. The provided Interfaces can consist of Methods, which have a return type and parameters. The client security model only consists of SecurityClasses and SecurityLevels. The SecurityClass provides general information about the system, like the kind of encryption used. Each architecture model element except the model element Type can have a SecurityClass. The SecurityLevel of a model element determines between which objects information can flow. Only Methods of Interfaces can have a SecurityLevel in this running example. The architecture model combined with the client security specification forms the client analysis model. The metamodel of the client analysis model is depicted in figure 4.2.

The used source code model has Classes, Interfaces, Methods, Parameters, and Variables. The Methods, the Parameters, and the Variables can have a Type. A Class can implement multiple Interfaces. The security specification of the source code has the form of annotations. An annotation identifies a method with a security level. The metamodel of the supplier analysis model is displayed in figure 4.3.

The supplier analysis produces results. The supplier analysis result metamodel is shown in figure 4.4. Each Result consists of the invalid flows and the number of invalid flows. Each Flow consists of a sequence of FlowStates. The sequence is at least 2 FlowStates long because a Flow is between at least two elements. A FlowState consists of the Method which is called, the Class of the Method and SecurityLevel of the Method.

The described software system can be modeled with the client analysis metamodel. The software system model is shown in figure 4.5. The security levels of the client analysis model are partly not fitting to show the back-projection for these incorrect security levels. For example, the security level of getCreditCardInfo of the interface CreditCardDataExchange is contact. The better fitting security level would be creditcard. However, contact is used to better show the advantage of the back-projection in section 6.4.

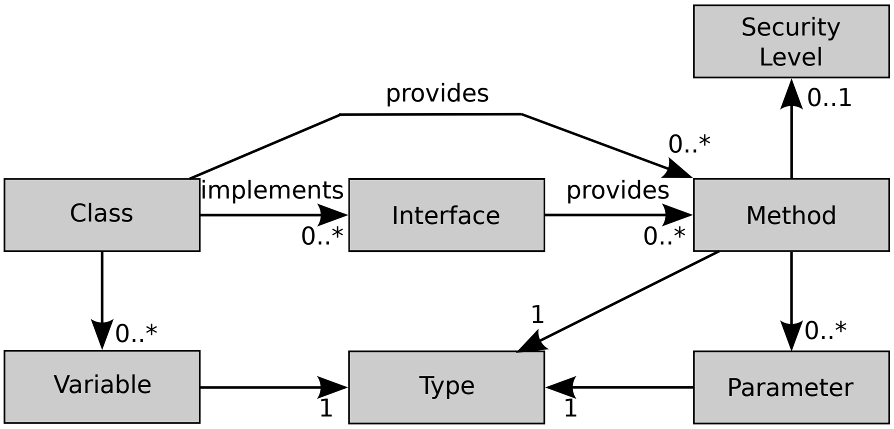


Figure 4.3: Supplier analysis metamodel for the running example

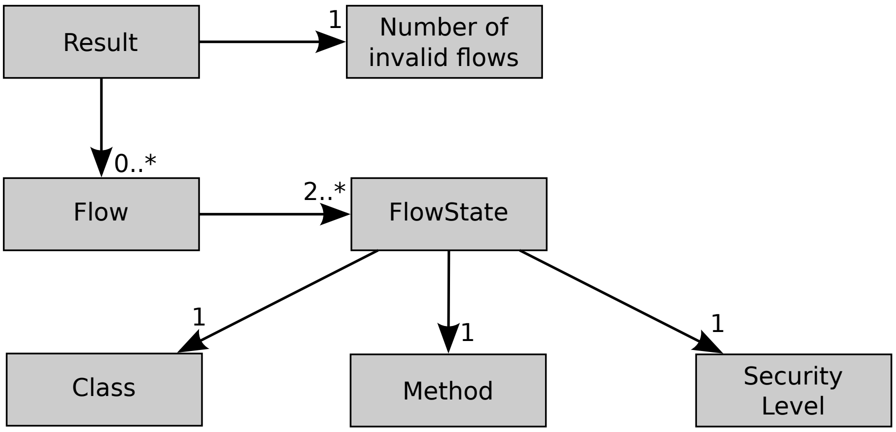


Figure 4.4: Result metamodel for the running example

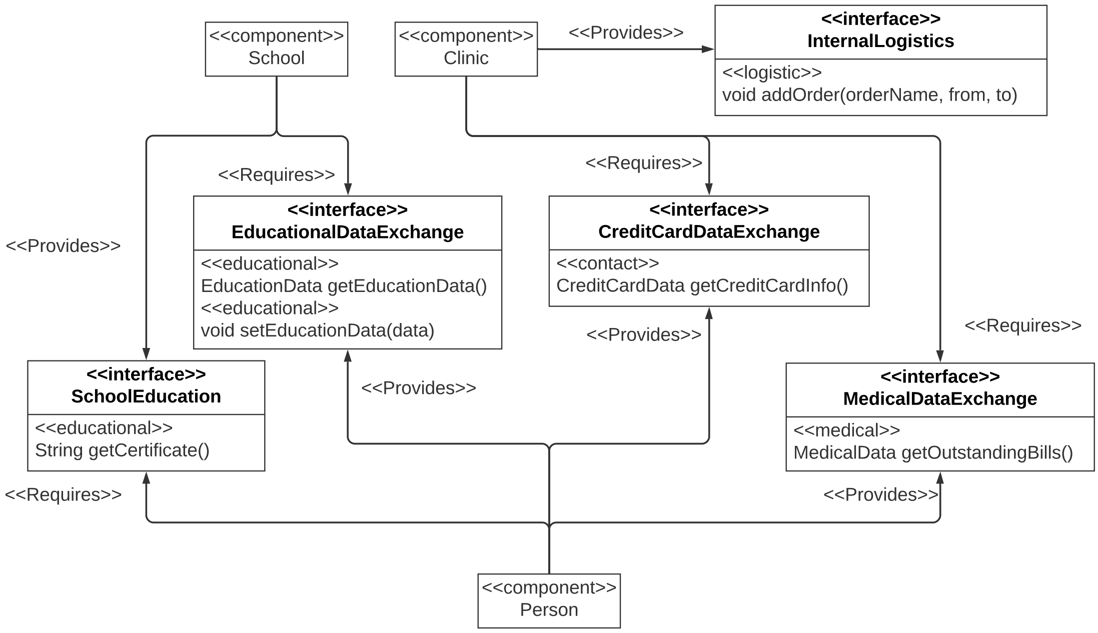


Figure 4.5: Client analysis model for the running example

The corresponding supplier analysis model skeleton is shown in figure 4.6. The figure shows the supplier analysis model without any concrete syntax. Furthermore, the implementation of the methods is missing because the client analysis model does not contain information about the internal implementation.

An excerpt of the manual implementation is shown as a sequence diagram in figure 4.7. The sequence diagram shows three interactions with the software system. In the first interaction, a curriculum vitae is created. A certificate is needed for the curriculum vitae, which is retrieved from School. The component School generates the certificates and returns it to PersonUI. The methods createCV and generateCerticate are internal methods of the components. Both have the security level {educational, contact}. In the second interaction, the credit card score should be determined. The outstanding bills of the person are used to determine how much the person owes to the clinic. This calculation is used to determine the credit card score of the person. The method getCreditCardScore has the security level creditcard. In the third interaction, the person is examined by the clinic. Therefore, the clinic needs the credit card information of the person. The method getCreditCardData accesses information about the tuition loan of the person. The method examinePerson has the security level {educational, contact}. The method getTuitionLoan has the security level creditcard. The supplier analysis is based on a security lattice. The security level is displayed in figure 4.8. For each element having an arrow to another element, information can flow from the first element to the second element.

```
Class School:
    Implements: SchoolEducation
    Variables: EducationalDataExchange educationalDataExchange
    Methods: <<educational>> String getCertificate()
Class Clinic:
    Implements: InternalLogistics
    Variables: CreditCardExchange creditCardExchange, MedicalDataExchange
        medicalDataExchange
    Methods: <<logistic>> void addOrder(ordnerName, from, to)
Class Person:
    Implements: EducationalDataExchnage, CredittCardDataExchange, MedicalDataExchange
    Variables: SchoolEducation schoolEducation
    Methods:
        <<educational>> EducationData getEducationData()
        <<educational>> void setEducationData(data)
        <<contact>> CreditCardData getCreditCardInfo()
        <<medical>> MedicalData getOutstandingBills()
Interface SchoolEducation:
    Methods: String getCertificate()
Interface EducationalDataExchange:
    Methods:
        EducationData getEducationData()
        void setEducationData(data)
Interface CreditCardDataExchange:
    Methods: CreditCardData getCreditCardInfo()
Interface MedicalDataExchange:
    Methods: MedicalData getOutstandingBills()
Interface InternalLogistics:
    Methods: void addOrder(ordnerName, from, to)
```

Figure 4.6: Supplier analysis model skeleton for the running example

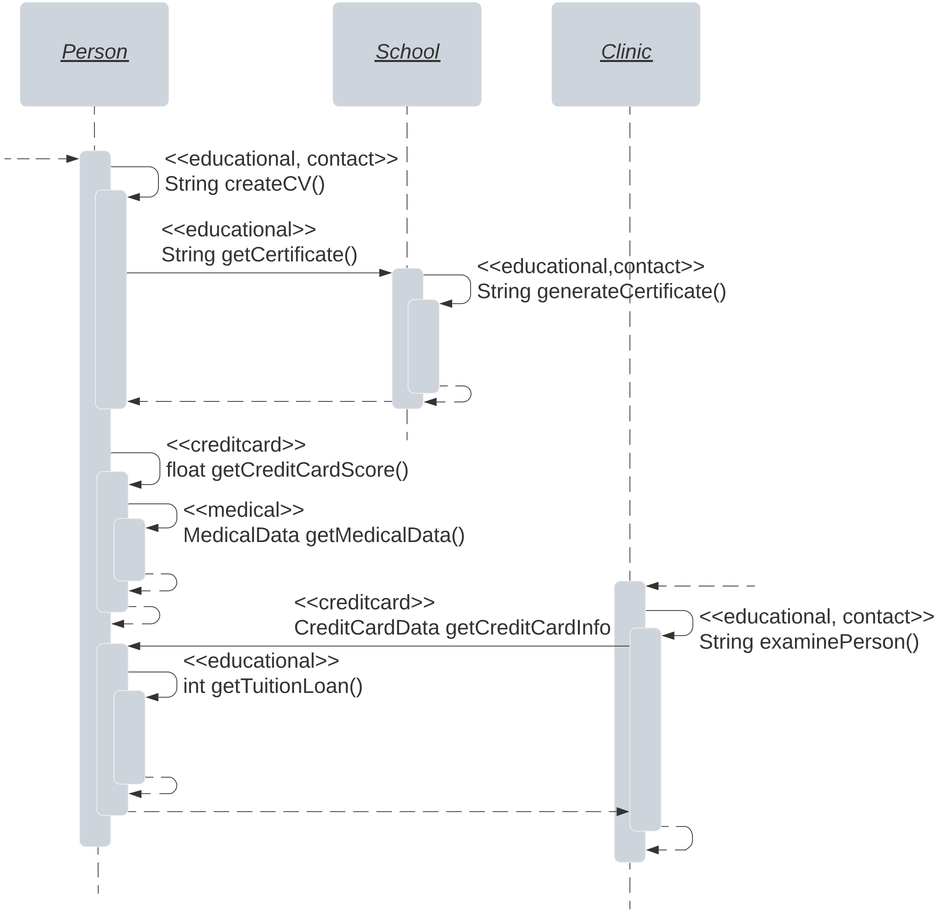


Figure 4.7: Excerpt of the implementation of the supplier analysis model as sequence diagram

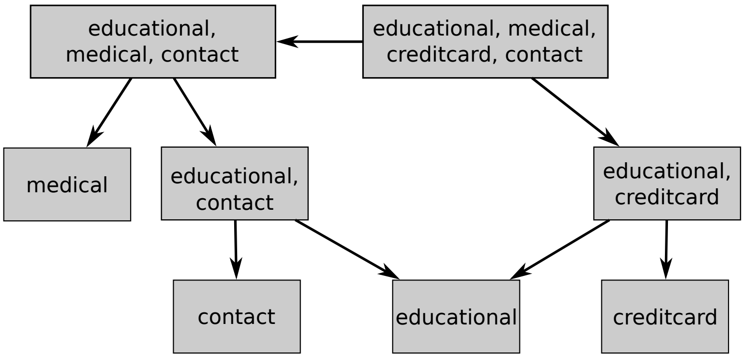


Figure 4.8: Lattice for the supplier analysis in the running example

# 5 Requirements for the Coupling of Analyses

In chapter 4, the overview of the approach developed in this bachelor's thesis is shown. To couple an analysis on the architecture view with an analysis on the source code view, the two analyses have to be compatible. These requirements under which a client analysis model is compatible with a supplier analysis model are described in this chapter. The requirements can be split into three groups. Requirements under which a correspondence model can be created are the first group. These requirements are introduced in section 5.1. In section 5.2, the requirements for compatible security properties are described. In section 5.3, the third group of requirements for the result model of the supplier analysis is described.

## 5.1 Existence of a Structural Correspondence Model

The connection between the client and supplier analysis model requires a structural correspondence model as described in section 4.1. Therefore, the first requirement is *requirement of a structural correspondence model*. The structural correspondence model contains mappings between the model elements specifying the components in the architecture model and the model elements specifying the source code skeleton. The structural correspondence model can be created if it is possible to generate the supplier analysis model skeleton from the client analysis model. In the transformation process, a group of client analysis model elements is used to generate a supplier analysis model element, as described in section 2.1.3. The group of model elements in the client analysis model and the model element in the supplier analysis model are forming a pair. All pairs from the supplier analysis model skeleton generation are bundled into a structural correspondence model.

In section 5.1.1, the requirements under which the generation of the supplier analysis model skeleton is possible are described. The supplier analysis has to be run on a completed analysis model. Therefore, the supplier analysis model skeleton has to be manually completed. When completing the skeleton, the developer has to follow rules that are described in section 5.1.2. These rules also have to be followed when extending the client analysis model.

### 5.1.1 Structural Information

In this section, the *requirement of structural information* is described. If the requirement is fulfilled, the supplier analysis model skeleton can be generated from the client analysis model. As introduced in section 4.1, the analyses are run on analysis models. An analysis model is a composition of two models. An analysis model is formed by the structural model and the security model of the software system [61, 62]. The security model holds the security information about the software system. The structural model contains information about the structure of the software system. As introduced in section 2.1.4, the model operation weaving combines the security model and the structural model into one analysis model. The weaving specification is based on the common metamodel elements concerning the structure of the software system. For instance, UMLSec is an extension of the UML metamodel [31], therefore uses UML elements in its metamodel. SecurityCodeScan is based on the programming languages C# [63] and therefore uses the elements of C# in its metamodel.

In this approach to transfer information between the client and supplier analysis model, the client structural model and the supplier structural model are connected by a model transformation, as shown in figure 5.1. As specified in section 4.1, only component-based software systems are considered. Therefore, the source code model can be divided into the source code skeleton and the internal component implementation. The source code skeleton specifies the component interfaces and skeletons that are all the interface methods of the component without the internal implementation. The internal component implementation is, for example, the implementation of the interface methods or the internal classes of the component. The client respectively supplier structural models are based on a client respectively supplier structural metamodel. The two structural metamodels are connected with a metamodel transformation.

In the client analysis model, the specification of the components is contained in the structural model, which is mapped to the source code skeleton in the supplier analysis model. For a formal definition, let $M$ be an analysis model, $SM$ is the structural model of $M$, and $MM_M$ is the metamodel of $M$.

The client analysis model is denoted by $M_{\text{client}}$. It has the structural model $SM_{\text{client}}$ holding the structural information of the software system in the architectue view. The supplier analysis model is denoted by $M_{\text{supplier}}$. It builds upon the source code model $SM_{\text{supplier}}$. We require $\{class, method, parameter\} \in MM_{SM_{\text{supplier}}}$ because the used source code model must be able to model a component-based software system [15]. The source code model parts defining the source code skeleton $I_{\text{supplier}} \subset SM_{\text{supplier}}$ are a subset of the complete source code model. For example, the internal implementation of the source code skeleton is in $SM_{\text{supplier}}$ but not in $I_{\text{supplier}}$. The model elements $I_{\text{supplier}}$ are based on the same metamodel as $SM_{\text{supplier}}$ but do not have to use the complete metamodel.

The source code skeleton generation from the client structural model is possible if there is a model transformation $T_{\text{structural}}$ with:

$$T_{\text{structural}} : SM_{\text{client}} \longrightarrow I_{\text{supplier}} \tag{5.1}$$

If the transformation exists, the component classes, interfaces, and method definitions of the supplier structural model can be created from the structural client model. Therefore, if the transformation exists, the source code skeleton generation is possible.

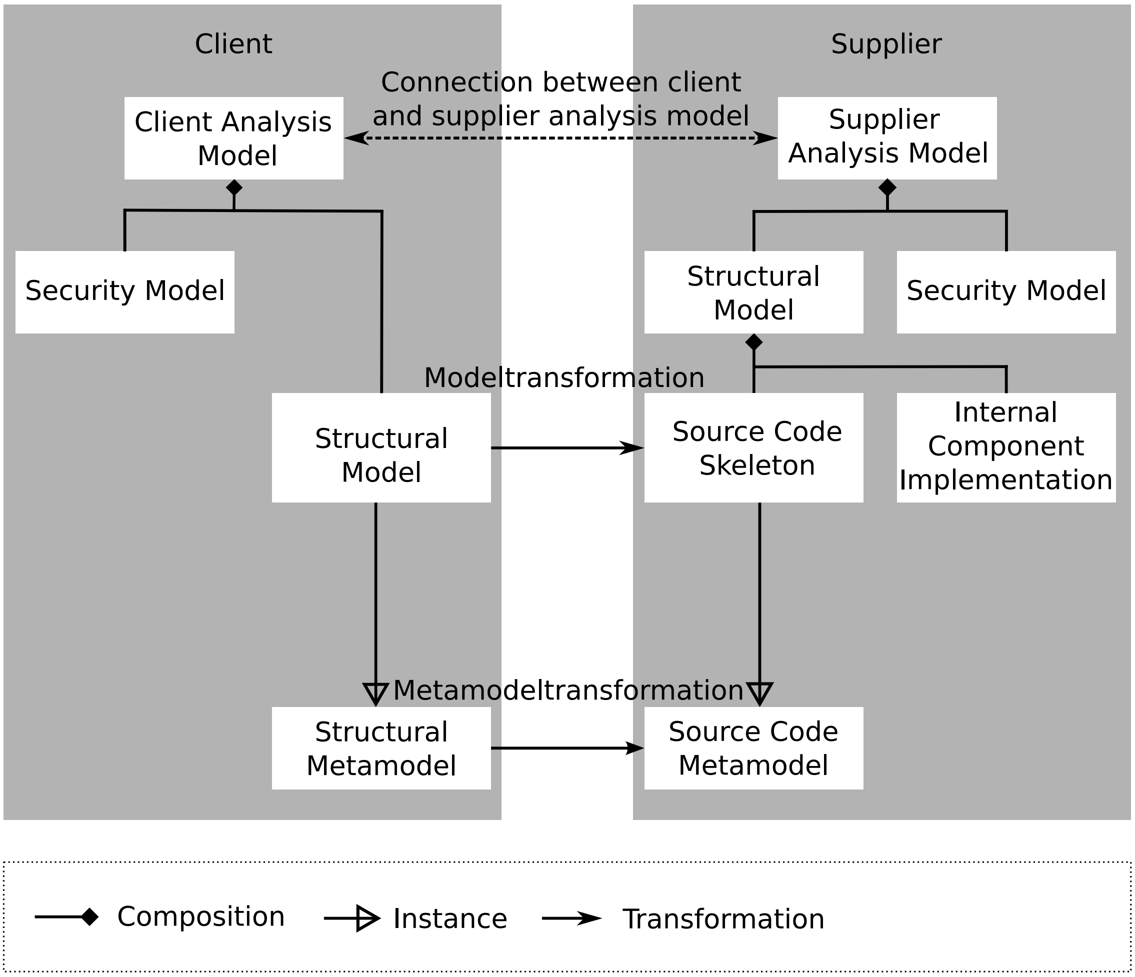


Figure 5.1: Composition of client and supplier analysis models, and the connection between the structural models. The connection between the two security models is not shown for the sake of simplicity.

A model transformation, as introduced in section 2.1.3, requires a relation between the metamodels of the structural models, which can also be expressed as a model transformation:

$$T_{\text{structural,meta}} : MM_{SM_{\text{client}}} \longrightarrow MM_{I_{\text{supplier}}} \tag{5.2}$$

There are three cases for the mapping between elements of both metamodels. The first case is a one-to-one mapping between model elements. In this case, the model transformation uses a direct mapping between elements of both metamodels. For example, if the structural client model is an UML class diagram, there is a one-to-one mapping between the classes in the client analysis model and the classes in the supplier analysis model [15, 64]. The second case is a *n*-to-one mapping. In the second case, multiple client metamodel elements must be combined to create one supplier analysis model element. For example, in the client analysis model, an UML component diagram [65] is used to specify which interfaces a component has and an UML sequence diagram [66] on the component level holds information about the existing methods of the components. The source code skeleton in the supplier analysis model can only be generated if both diagrams are present. The third case is a $m \times n$-to-one mapping. In this case, the underlying structural model contains multiple groups of metamodel elements from which a supplier analysis model can be generated. For example, if the structural model contains UML class, component, and sequence diagrams, the source code skeleton can be generated from the class diagram or the pair of component and sequence diagrams. In this case, the structural correspondence model has to contain all possible model element pairs used for the source code generation. Otherwise, the back-projection cannot adapt the security specifications of all the elements in the client analysis model that are affected by the violation of the security property.

It is not required that $T_{\text{structural,meta}}$ uses all metamodel elements of $MM_{SM_{\text{client}}}$. Therefore, let $U_{\text{client}}$ be the set of metamodel element groups of $MM_{SM_{\text{client}}}$ which are used in $T_{\text{structural,meta}}$. For example, if the metamodel of the source code is C# (Version before 8.0) and the structural client model is an UML class diagram, $U_{\text{client}}$ would be $\{\{\mathit{class}, \mathit{operation}, \mathit{parameter}\}\}$. $SM_{\text{client}}$ must instantiate at least one element of $U_{\text{client}}$ because otherwise $T_{\text{structural}}$ is not possible besides the existence of $T_{\text{structural,meta}}$. If no element of $U_{\text{client}}$ is instantiated, $SM_{\text{client}}$ does not model the necessary architecture information for generating $I_{\text{supplier}}$. If the definition of $T_{\text{structural}}$ is not possible, the source code skeleton cannot be generated from the client structural model. Therefore, there is no mapping between the client structural model and the source code skeleton. If there is no mapping, a structural correspondence model between the client analysis model and the supplier analysis model cannot exist. If there is no structural correspondence model between the client and the supplier analysis model, the back-projection is not possible, because the back-projection requires a mapping between the two analysis models. Therefore, if $T_{\text{structural}}$ does not exist, the client and the supplier analysis model cannot be coupled. Otherwise, if $T_{\text{structural}}$ exists, a structural correspondence model between the client structural model and the source code skeleton can be created.

In case of the running example, which is described in section 4.2, the structural model of the client analysis model is the set of client analysis model elements excluding the SecurityClasses and SecurityLevels. A client Component is transformed into a supplier Class.

A client Interface is transformed into a supplier Interface. A client Method is transformed into a supplier Method. A client Parameter is transformed into a supplier Parameter. A client Type is transformed into a supplier Type. Furthermore, the connections between the model elements have to be transformed. For example, the supplier Component provides and requires Interfaces. The requires relationship is transformed into a Variable of the supplier Class. The provided relationship is transformed into an implements relationship in the supplier analysis model. The Methods defined in the provided Interface are transformed into Methods of the supplier Class. Furthermore, based on the structural metamodel transformation a structural model transformation can be constructed for the structural models. For example, the client component School is transformed into the supplier class School. The client interface SchoolEducation is transformed into the supplier interface SchoolEducation. The client method String getCertificate() is transformed into the supplier method String getCertificate(). The structural correspondence model created during $T_{\text{structural}}$ contains the mappings:

- { Component School } - { Class School }
- { Interface SchoolEducation } - { Interface SchoolEducation }
- { Method String getCertificate() } - { Method String getCertificate() }

Therefore, a transformation between the two metamodels exists, and $\{class, method, parameter\} \in MM_{SM_{\text{supplier}}}$ holds. Furthermore, a structural correspondence model can be initiated between the two structural models. Therefore, the requirement of structural information in the supplier analysis model holds.

### 5.1.2 Consistency between Client and Supplier Analysis Model

In this section, the *requirement of consistency* is described. The source code skeleton is generated from the client structural model. The source code skeleton then has to be completed with the internal implementation of the components. The source code generation establishes a connection between the client and supplier analysis model with the created structural correspondence model [58]. During the completion of the source code skeleton, the model elements contained in the structural correspondence model cannot be changed without adapting the structural correspondence model. Otherwise, if, for example, the signature of one interface method in the supplier analysis model is changed without adapting the structural correspondence model, the structural correspondence model contains an invalid mapping between model elements. If the back-projection cannot find any corresponding model elements in the client analysis model, the back-projection is no longer possible. If the back-projection is no longer possible, the two analysis models are no longer coupled. Furthermore, if a new interface method is added during the completion, the structural correspondence model does not contain the new interface method. Therefore, the structural correspondence model is again in an invalid state, in which not all source code skeleton elements are contained in the structural correspondence model. On the other side, changes to model elements of the client or supplier analysis model, which are not involved in $T_{\text{structural}}$, are allowed because those elements are not contained in the structural correspondence model. Therefore, changes to the elements not involved in $T_{\text{structural}}$ do not invalidate the structural correspondence model.

Now, the allowed changes to the client and the supplier analysis model after creating the structural correspondence model are generalized. Let $SM_{\text{client}}$, $SM_{\text{supplier}}$, $I_{\text{supplier}}$ and $T_{\text{structural}}$ be defined as in section 5.1.1. $T_{\text{structural}}$ produces the structural correspondence model $C_{\text{structural}}$ during the transformation as described in section 4.1. Changes to models are found with the *diff* operation which takes two models and returns a sequence of edit actions to transform the first model into the second model as introduced in section 2.1.2. To generalize the allowed changes to the client and the supplier analysis model, let $\widetilde{SM}_{\text{client}}$ respectively $\widetilde{SM}_{\text{supplier}}$ be structural model instances of the client analysis model respectively supplier analysis model, after some changes to $SM_{\text{client}}$ and $SM_{\text{supplier}}$. Therefore, $C_{\text{structural}}$ already exists. Let $\mathit{diff}(SM_{\text{client}}, \widetilde{SM}_{\text{client}}) \neq \emptyset$ respectively $\mathit{diff}(SM_{\text{supplier}}, \widetilde{SM}_{\text{supplier}}) \neq \emptyset$. $I_{\text{supplier}}$ are the interface model elements of $SM_{\text{supplier}}$. The transformation $T_{\text{structural}}$ produces $\tilde{C}_{\text{structural}}$ when transforming $\widetilde{SM}_{\text{client}}$ in $\tilde{I}_{\text{supplier}}$. The changes to $SM_{\text{client}}$ are allowed, if the following two equations hold:

$$\mathit{diff}(I_{\text{supplier}}, \tilde{I}_{\text{supplier}}) = \emptyset \tag{5.3}$$

$$\mathit{diff}(C_{\text{structural}}, \tilde{C}_{\text{structural}}) = \emptyset \tag{5.4}$$

Equation (5.3) holds if no new interface method is added to the supplier analysis model. Equation (5.4) holds if the unadapted structural correspondence model and the adapted structural correspondence model are the same. This is the case if no new structural model element is added to the client analysis model.

In the running example, the method setCreditCardInfo is added to the interface EducationalDataExchange of the supplier analysis model in the running example introduced in section 4.2 forming $\widetilde{SM}_{\text{supplier}}$. The addition of setCreditCardInfo is a change in the interface of a component, therefore also $I_{\text{supplier}}$ is changed, implying $\tilde{I}_{\text{supplier}}$. The two models are no longer consistent because the setCreditCardInfo is only present in the supplier analysis model but not in the client analysis model and the structural correspondence model. Therefore, $\mathit{diff}(I_{\text{supplier}}, \tilde{I}_{\text{supplier}}) \neq \emptyset$ applies. The class TeachersRoom is added to the component School forming $\widetilde{SM}_{\text{supplier}}$. TeachersRoom does only communicate with the interfaces provided by School and other internal classes of School. Therefore, $I_{\text{supplier}}$ is not changed, therefore $\mathit{diff}(I_{\text{supplier}}, \tilde{I}_{\text{supplier}}) = \emptyset$ applies.

## 5.2 Connection of Security Properties

The goal of this bachelor's thesis is to connect the two analyses by establishing a correspondence model between the client and the supplier analysis model and by projecting back the results of the supplier analysis into the client analysis, as described in section 4.1. The back-projection adapts client analysis model elements that conflict with the supplier analysis results. Therefore, the security property analyzed by the supplier analysis has to be part of the security property analyzed or assumed of the client analysis. If an analysis assumes a security property, there are analysis model elements specifying the security property in the analysis model [67]. For example, if an analysis examines whether the software only outputs encrypted information, the analysis needs an analysis model with a specification which methods return or require encrypted data [68]. If the client analysis model does not assume any security property analyzed by the supplier analysis, the client analysis model contains no information about the security property analyzed by the supplier analysis. If the analysis model has no model elements specifying the security property of the supplier analysis, the back-projection cannot calculate the correct security specification. Therefore, the back-projection cannot adapt any elements of the client security model. If the back-projection fails, then the information transfer between the analysis models fails. Therefore, the two analyses cannot be coupled. For example, the client analysis examines the security property observational determinism [69], for what it assumes non-interference [70, 71]. The supplier analysis covers non-interference. Then, the assumptions of the client analysis model regarding the specifications for non-interference can be verified by the results of the supplier analysis. However, the assumptions that the observational determinism property holds for the internal structure cannot be verified. The reason for this is that the supplier analysis reveals nothing about observational determinism.

Static security analyses, as introduced in section 2.7, can be described as testing whether a set of (hyper-)trace properties holds or not. As introduced in section 2.5, a trace property defines a set of traces that fulfill a given set of rules. Transferred to a software system, a trace consists of system states (states) and method calls (actions) which combined form an instance of an abstract state. If all possible execution traces through the software system are in the defined trace property, the system fulfills the security property induced by the trace property [72]. The client analysis tests the execution traces through the client analysis models, and the supplier analysis tests the execution traces through the supplier analysis model.

### 5.2.1 Composition of Multiple (Hyper-)Trace Properties

One security analysis can test multiple security properties. Each security property can be modeled as a trace property. Therefore, it is shown that multiple trace properties can be encoded into one trace property which can then be tested by one analysis. It is also shown that a corresponding decoding exists.

A trace property is defined as a set of traces and a hyperproperty as a set of sets of traces that fulfill certain rules, as introduced in section 2.5. Therefore, the set-builder notation for expressing sets can be used [73]

$$M = \{x \in G \mid A(x)\} \tag{5.5}$$

where $M$ is the defined set, $G$ is the domain of the set, and $A(x)$ is a logical predicate specifying the rules by which $x$ is chosen. The domain for trace properties is $\Psi_{inf}$ and for hyperproperties *Prop* as introduced in section 2.5. For instance, the property guaranteed service (*GS*) can be modeled with the set-builder notation [26]:

$$GS = \{t \in \Psi_{inf} \mid \forall i \in \mathbb{N} : isReq(t[i]) \Rightarrow (\exists j > i : isRespToReq(t[j], t[i]))\} \tag{5.6}$$

The predicate $isReq(s)$ identifies all trace states, which are a request, whereas the predicate $isRespToReq(s', s)$ identifies, if $s'$ completes the request initiated in $s$.

Now, the composition of two trace properties can be defined as the cut of two sets. Let $H_1$ and $H_2$ be trace properties, defined by:

$$H_1 = \{T \in \Psi_{inf} \mid A_1(T)\} \tag{5.7}$$
$$H_2 = \{T \in \Psi_{inf} \mid A_2(T)\} \tag{5.8}$$

Then, two trace properties are encoded into one by:

$$H = H_1 \cap H_2 = \{T \in \Psi_{inf} \mid A_1(T) \land A_2(T)\} \tag{5.9}$$

The cut of two trace properties results in a set of traces which are in both hyperproperties. For every other trace $t' \notin H$, $H_1$ or $H_2$ does not hold. Each trace in $H$ has to fulfill the logical predicates of both trace properties. Generalized for $n$ trace properties

$$H = \bigcap_{i=1}^{n} H_i = \{T \in \Psi_{inf} \mid \bigwedge_{i=1}^{n} A_i(T)\} \tag{5.10}$$

where $H[i]$ is the $i$-th trace property. The encoding for $n$ hyperproperties is achieved accordingly:

$$H = enc(H) = \bigcap_{i=1}^{n} H_i = \{T \in Prop \mid \bigwedge_{i=1}^{n} A_i(T)\} \tag{5.11}$$

A possible security analysis can cover multiple security properties, some of which are trace properties and some of which are hyperproperties. For example, an analysis based on UMLSec [31] can be constructed to analyze information flow and access control. To formulate one hyperproperty for the complete security analysis, a composition of hyperproperties and trace properties must exist. The domain of trace properties is $\Psi_{inf}$ and the domain of hyperproperties is $Prop = \mathcal{P}(\Psi_{inf})$. Therefore, an element of $Prop$ fulfills a trace property $P$, if $P$ holds for all elements of $T \in Prop$. The encoding of the trace property $P$ as hyperproperty $hyper(P)$ is achieved with:

$$hyper(P) = \{T \in Prop \mid \forall t \in T : t \models P\} \tag{5.12}$$
$$= \{T \in Prop \mid \forall t \in T : A(t)\} \tag{5.13}$$

The trace properties have to be first encoded into hyperproperties and then combined with equation (5.11) to combine trace properties and hyperproperties into one hyperproperty. The decoding $dec(P)$ of a trace property into a set of trace properties is achieved by splitting the logical predicate at the top-level conjunctions. The reason for this is that the (hyper-)trace properties are combined by the conjunction of the (hyper-)trace properties. For example, if the hyperproperty $H$ contains traces which fulfill non-interference ($H_{\text{NI}}$) and observational determinism ($H_{\text{OD}}$), then the decoding of $H$ is the set $\{H_{\text{NI}}, H_{\text{OD}}\}$.

### 5.2.2 Overlap of Hyperproperties

The traces through the two models have to be comparable to compare the hyperproperty of the client analysis and the supplier analysis. A trace through the software system consists of abstract states as introduced in section 2.5. In this subsection, the abstract state contains at least the software state and the executed action, where $\mathit{action}(t[i])$ is the action performed in the $i$-th state of the trace and $\mathit{state}(t[i])$ is the software state of the $i$-th state of the trace.

The client analysis model for a component-based software system only possesses information about the interfaces and communication between the components from which the source code skeleton can be generated as described in section 4.1. The client analysis model views a component as a black box as explained in section 4.1. The state of the software system in the client analysis model can be expressed in more detail than in the supplier analysis model because the internal structure of the components is additionally known. However, if the source code skeleton is generated from the client analysis model and implemented according to the requirements from section 5.1.2, the information contained in the software state of the client analysis model can be generated from the software state in the supplier analysis results. The set of possible actions in the client analysis model consists of the interface methods of the components. We call them visible actions. In the supplier analysis model, the set of possible actions is extended with the manually implemented methods. The manually implemented methods are called invisible actions because they can not be triggered directly from the outside of the component if the requirement of consistency, described in section 5.1.2, is fulfilled. The execution of an internal component method can only be a result of the execution of visible actions. Therefore, a trace through the supplier analysis model can be expressed as a trace through the client analysis model by removing all states with invisible actions. The reduction to component interface methods is necessary because only those are in the correspondence model and can be projected back into the client analysis model. The client analysis model does not know the internal structure of the components. Therefore, the client analysis model has no information about the invisible actions.

Let $S_{\text{client}}$ be the set of software states of the client analysis model with the actions $A_{\text{vis,client}}$. Accordingly, for the supplier analysis model, let $S_{\text{supplier}}$ be the software states of the supplier analysis model. The supplier analysis model has visible actions $A_{\text{vis,supplier}}$ (the interface methods) and invisible actions $A_{\text{in,supplier}}$ [27]. Therefore, all possible actions in the supplier analysis are $\alpha := A_{\text{vis,supplier}} \cup A_{\text{in,supplier}}$. Furthermore, let $g$ be a transformation from the software state of the supplier analysis model to the software state of the client analysis model:

$$g : S_{\text{supplier}} \longrightarrow S_{\text{client}} \tag{5.14}$$

The transformation $g$ is based on the structural correspondence model introduced in section 5.1. The trace $t$ is a trace through the supplier analysis model. The action of $t[p]$ is visible, the next visible action is performed in $t[q]$. Therefore, every state between $p$ and $q$ has an invisible action. We define $I_{\text{vis,supplier}}$ as the set of indices of states with visible actions, which can be generated with $f$ from the trace $t$:

$$f(t) = I_{\text{vis,supplier}} = \{i \in \{1..|t|\} \mid \mathit{action}(t[i]) \in A_{\text{vis,supplier}}\} \tag{5.15}$$

With the index set, a trace through the supplier analysis can be expressed as a trace through the client analysis with the trace transformation $h$ with

$$t_{\text{client}} = h(t_{\text{supplier}}) = \sum_{i \in I_{\text{vis,supplier}}} (g(\mathit{state}(t_{\text{supplier}}[i])),\ \mathit{action}(t_{\text{supplier}}[i])) \tag{5.16}$$

where $\sum$ is the concatenation of the trace states. If $t_{\text{supplier}}$ is not a single trace, but a set of traces, $h$ transforms every trace independently in the set. Therefore, $h$ transforms a set of supplier traces into a set of client traces. We can now compare the hyperproperties of the supplier analysis ($H_{\text{supplier}}$) with those of the client analysis ($H_{\text{client}}$) by representing the hyperproperty of the supplier analysis as a hyperproperty of the client analysis. The hyperproperty $H_{\text{supplier}}$ can be presented as a hyperproperty in the client analysis $H_{\text{supplier in client}}$ by:

$$H_{\text{supplier in client}} = h(H_{\text{supplier}}) = \{h(T) \mid T \in \mathit{Prop} \ \wedge\ A(T)\} \tag{5.17}$$

$H_{\text{client}}$ and $H_{\text{supplier in client}}$ can be compared, because their traces are trough the same model. Two security analyses can be coupled if they are partly covering the same security properties. The requirements can be formalized, as the intersection of the covered hyperproperties:

$$\mathit{dec}(H_{\text{client}}) \cap \mathit{dec}(H_{\text{supplier in client}}) =: H_{\text{both}} \neq \emptyset \tag{5.18}$$

In this approach, this requirement is called the *requirement of overlapping (hyper-)trace properties*. If this requirement is fulfilled, the assumptions for $H_{\text{both}}$ in the client analysis can be verified with the results of the supplier analysis.

### 5.2.3 Transformation of the Metamodels

A hyperproperty requires extra information about the states of a trace. For example, the trace property for guaranteed service, shown in equation (5.6), requires the information on whether an action is a request or response. Such information is added to the client and supplier analysis by supplementing the underlying model with a security model forming an analysis model as described in section 5.1.1. The requirement of overlapping hyperproperties has to be extended to connectable security models for the generation of the supplier analysis model from the client analysis model. Therefore, the *requirement of a security correspondence model* is introduced in this section.

A security model for a security analysis can be divided into model element sets. The division is parallel to the division of the tested hyperproperty into multiple hyperproperties. A model element can be part of multiple model element sets. If the two analyses are both covering the hyperproperty $H_{\text{both}}$ and the corresponding model elements are $M_{\text{supplier,both}}$ and $M_{\text{client,both}}$, then the model transformation:

$$T_{\text{security}} : M_{\text{client,both}} \longrightarrow M_{\text{supplier,both}} \tag{5.19}$$

has to exist. $T_{\text{security}}$ exists if a transformation between the two security metamodels exists. The security metamodel transformation is defined as:

$$T_{\text{security,meta}} : MM_{M_{\text{client,both}}} \longrightarrow MM_{M_{\text{supplier,both}}} \tag{5.20}$$

If $T_{\text{security,meta}}$ does not exist, then the supplier security model cannot be generated from the client security model. Additionally, $T_{\text{security}}$ is necessary for the back-projection because during the supplier security model generation a security correspondence model is created, that is similar to the structural correspondence model, described in section 5.1. The security correspondence model is, like the structural correspondence model, essential for the back-projection of the supplier analysis results.

For example, the client analysis in the running example introduced in section 4.2 is based on a client security model with a security class and a security level. The hyperproperty of the client and the supplier analysis is not known. However, the concrete hyperproperty for the client analysis can be divided into the hyperproperty concerned with the security level (e.g. IFC) and the hyperproperty concerned with the security class. The supplier analysis hyperproperty is IFC. Therefore, there is an overlap between the two hyperproperties. Furthermore, the security levels from the client analysis model can be directly transformed into the security levels of the supplier analysis model. Since there is an overlap of hyperproperties and a transformation between the security metamodels, the security properties can be connected.

## 5.3 Requirements for the Supplier Analysis Results

In this bachelor's thesis, the client analysis model is adapted based on the supplier analysis results. The supplier analysis can have two results. The first case is that the analysis does not detect any traces which are violating the security property. In this case, the results do not have to be projected back into the client analysis because the assumptions in the supplier analysis are correct. The second case is that the results contain that the security property does not hold. In this case, the results have to be projected back because there is at least one assumption in the client analysis model which is not correct.

The supplier analysis results have to fulfill requirements on the results structure, described in section 5.3.1. Furthermore, the supplier analysis results have to contain public structural elements. This requirement is presented in section 5.3.2.

### 5.3.1 Required Structure of the Supplier Analysis Results

The first requirement is the *requirement of result structure* which is described in this section. The supplier analysis examines a security property that can be represented as (hyper-)trace property, as described in section 2.5. The supplier analysis results can be projected back if the supplier analysis results contain information about the traces, which are violating the security property. Therefore, a supplier analysis result is required to contain the traces violating the security property. Information about the invalid traces is required because client analysis model elements have to be adapted based on the supplier analysis results. The back-projection needs information about the reasons why the security property does not hold. This information is not present if the results only contain the information that the security property does not hold for the supplier analysis model.

Client analysis model elements are adapted in the back-projection. As described in section 4.1, the client and supplier analysis models contain a structural and a security model.

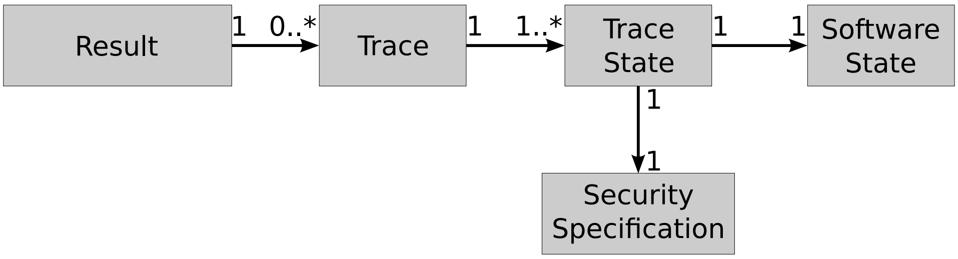


Figure 5.2: Minimum required metamodel of the supplier analysis results

The back-projection adapts single client analysis model elements containing a structural and a security model element. Therefore, the detected invalid flows have to contain structural and security information about the supplier analysis model elements.

A trace consists of a sequence of trace states modeling the software state. All trace states have to identify a structural model element that is part of the invalid trace. The kind of structural model element is specific to the two analyses. Furthermore, the security specification of a client analysis model element is adapted based on the supplier analysis results. Therefore, it is required that a security specification can be assigned to each abstract state. If no such assign-function exists, the analysis cannot be executed because the information concerning the security property is missing.

For example, the client analysis assumes that the security property guaranteed service [26] holds for all component interface methods. The client analysis should be coupled with a supplier analysis to verify and potentially correct these assumptions. In the supplier analysis model, each method is assigned the security specification request, response, or internal. Internal is, in this context, defined as neither request nor response. Then, it is tested whether every request entails a response in each trace through the supplier analysis model. Therefore, the supplier analysis results have to contain the traces for which service is not guaranteed. The traces consist of trace states which have to identify a supplier structural element and a security specification. In this example, the supplier structural model element which has to be identified is the method. The methods could, for example, be identified with unique IDs or fully qualified names. Furthermore, it must be possible that a security specification can be assigned to each trace state.

The requirements on the structure of a supplier analysis result can be represented as a minimum required supplier analysis result metamodel. The result can be represented as a model $M_{\text{result}}$ since it fulfills the mapping, the abstraction, and pragmatism property defined in section 2.1. An analysis result has a general structure, namely the result metamodel $MM_{M_{\text{result}}}$. The minimal supplier analysis results metamodel is depicted in figure 5.2. A supplier analysis result has to contain information about the invalid traces. The traces consist of trace states with at least the software state and the security specification of the software state. A supplier analysis result fulfills the *requirement of result structure* if a transformation $T_{\text{result}}$ from its metamodel to the minimal required results metamodel exists.

### 5.3.2 Required Public Structural Elements in the Supplier Analysis Results

In this section, the *requirement of public structural element* is described. It is already required, that the result has to have information about the traces violating the security property. As described in section 4.1, the client structural model is a component-based architecture model with components as black boxes. Therefore, each invalid trace has to contain at least one software state, which can be mapped to a supplier structural model element generated from a client structural model element. Otherwise, the invalid trace is based on the internal behavior of the component about which the client analysis model does not know. Therefore, the assumption that the implementation of the supplier analysis model is correct is violated.

If the supplier analysis examines a hyperproperty, then the supplier analysis result is a set of traces, which invalidates the hyperproperty. The result is a set of traces because the domain of a hyperproperty is $Prop = \mathcal{P}$, as introduced in section 2.5. If the result contains a set of traces instead of a single trace, then the correct security specification is not determined based on one trace but on the set of traces. Therefore, it is required that at least one trace of the detected set of traces fulfills the *requirement of public structural element*.

# 6 Connecting two Security Analyses

The information transfer between security analyses is possible if the analyses fulfill the requirements from chapter 5. Therefore, the security analyses are described as coupled. The coupled security analyses form a megamodel, introduced in section 6.1. The central part of the megamodel is the structural correspondence model described in section 6.2. If the supplier analysis model skeleton is not already present, it is generated from the client analysis model. The generation is presented in section 6.3. After the supplier analysis is run, the supplier results are projected back into the client analysis model to complete the information transfer. The back-projection is described in section 6.4.

## 6.1 Modeling the Coupling of Two Analyses as a Megamodel

Two static security analyses can be coupled if the requirements, shown in chapter 5 are fulfilled. The first requirement, *requirement of a structural correspondence model* presented in section 5.1, states that it is possible to construct a correspondence model between the underlying structural models. The *requirement of a security correspondence model*, as described in section 5.2.3, is the second requirement. The third requirement is *requirement of result structure*, introduced in section 5.3.1. The coupled system is described as a megamodel consisting of an architecture view (client analysis) and a source code view (supplier analysis), similar to the megamodel of the iObserve approach, introduced in section 3.3. The client structural model is connected with the supplier structural model by the structural correspondence model $C_{\text{structural}}$. The two security models of the analyses are connected with the security correspondence model $C_{\text{security}}$. Furthermore, the supplier analysis model is connected to the client analysis model by generating supplier analysis results that are projected back into the client analysis model. The proposed megamodel is shown in figure 6.1.

The megamodel is based on the model operations sewing and weaving introduced in section 2.1.4. The client structural model and the client security model are woven together to produce the client analysis model. The supplier structural model and the supplier security model are woven together to produce the supplier analysis model, as described in section 4.1. The client analysis model and the supplier analysis model are sewed together using mediators. The first mediator connects the client structural metamodel and the supplier structural metamodel. The mediator is the structural metamodel transformation structural correspondence metamodel. The initiated mediator between the two structural models is the structural correspondence model. The second mediator connects the client security metamodel and the supplier security metamodel. The mediator is the security correspondence metamodel. The initiated mediator between the two security models is the security correspondence model. Based on the two mediators, the client and supplier

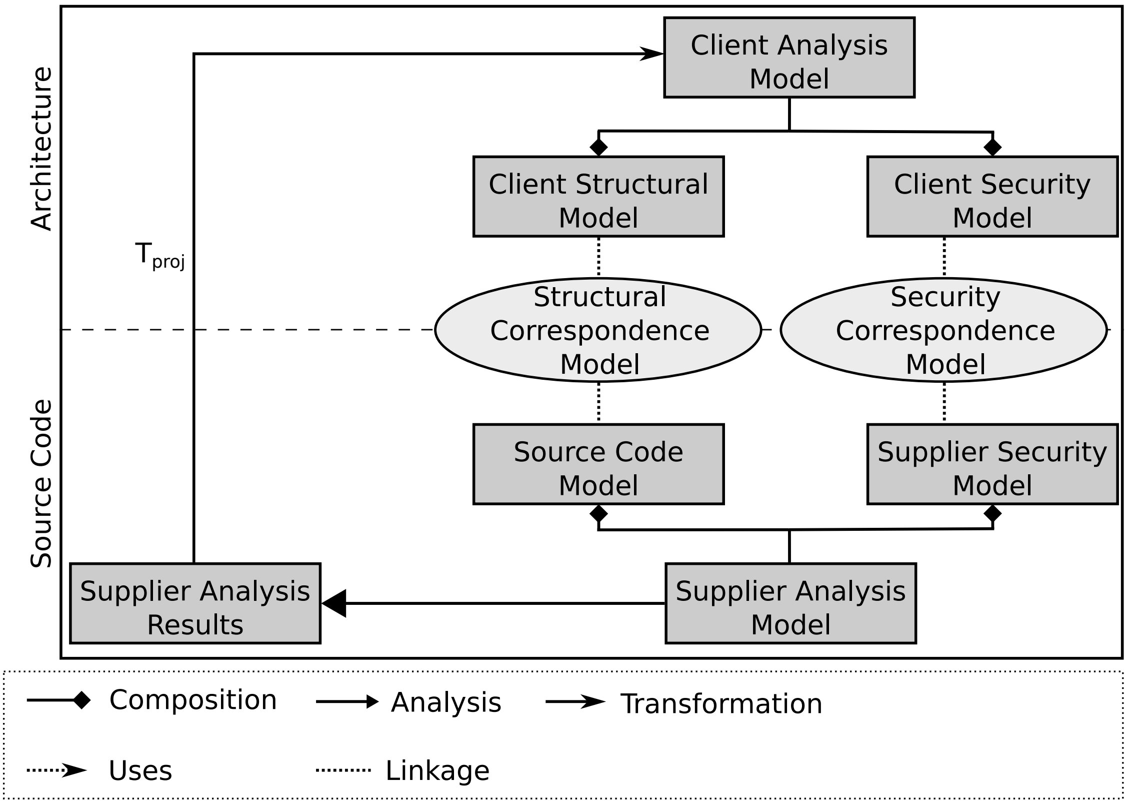


Figure 6.1: Formed megamodel by coupling an client and a supplier security analysis. The transformation $T_{\text{proj}}$ from the supplier analysis results to the client analysis model is the back-projection.

analysis model are sewed into one megamodel. Furthermore, the supplier analysis model produces the supplier analysis results. The produce relation incorporates the supplier analysis results into the megamodel. The back-projection of the results represents a mediator between the supplier analysis results and the client analysis model. Therefore, the client analysis model can be sewed together with the supplier analysis results. The megamodel is formed by the client analysis model, the supplier analysis model, and the supplier analysis results.

## 6.2 Structural and Security Correspondence Model

Two correspondence models are used in this approach to couple two security analyses. The first correspondence model is between the client structural model and the supplier structural model. The second correspondence model is between the client security model and the supplier security model. In section 6.2.1, the structure of the correspondence models is presented. The creation of the correspondence models during the generation process is explained in section 6.2.2.

### 6.2.1 Structure of the Correspondence Models

As in the VITRUVIUS approach, presented in section 3.4, the correspondence model contains correspondences between two groups of model elements. A triple graph grammar, as introduced in section 2.1.5, can be used to describe rules under which a correspondence between two sets of model elements is allowed. The correspondence model is established between the client analysis model and the supplier analysis model. As described in section 6.1, both analysis models consist of a structural model and a security model that are woven together. Therefore, the correspondence model between the two analysis models is divided into two correspondence models to separate concerns. The first correspondence model is defined between the two structural models, as described in section 5.1.1. The second correspondence model is defined between the two security models and is, therefore, called the security correspondence model as described in section 5.2.3. The correspondence models are used in the back-projection of the supplier analysis results.

As introduced in section 2.1.5, a triple graph grammar consists of two domain models and a correspondence model, for example, the two structural models and the structural correspondence model. The metamodel of a triple graph grammar is based on the metamodels of the domain models. Based on a metamodel transformation, a triple graph grammar can be constructed. Two graphs are the two domain metamodels. Each case of the metamodel transformation can be represented as a node of the correspondence graph with edges to all input metamodel elements and all output metamodel elements. In section 5.1.1 and section 5.2.3 the two metamodel transformations $T_{\text{structural,meta}}$ and $T_{\text{security,meta}}$ are introduced. Both metamodel transformations use a set of client analysis metamodel elements as input and a supplier analysis metamodel element as output. Therefore, a triple graph grammar can be constructed based on $T_{\text{structural,meta}}$ and $T_{\text{security,meta}}$.

The triple graph grammar for the two metamodels is used as a metamodel for the correspondence model. The correspondence nodes relate to all client analysis model elements, which are used for the generation of a supplier analysis model element and the generated supplier analysis model element.

A possible triple graph grammar for the structural metamodels of the running example based on the structural metamodel transformation, defined in section 5.1.1, is shown in figure 6.2. Connections between model elements also have to be preserved during the transformation. For example, the *required* relation between components and interfaces is transformed into an *implements* relation between classes and interfaces. If a relationship between two model elements exists and both model elements have an edge to a correspondence node, then there is an edge between these two correspondence models.

The triple graph grammar for the two structural metamodels is used as a correspondence metamodel for the two structural models. An excerpt of the correspondence model initiated for the two structural models is depicted in figure 6.3. The excerpt only shows the correspondence for the component instance `School` with its provided interface instance `SchoolEducation`.

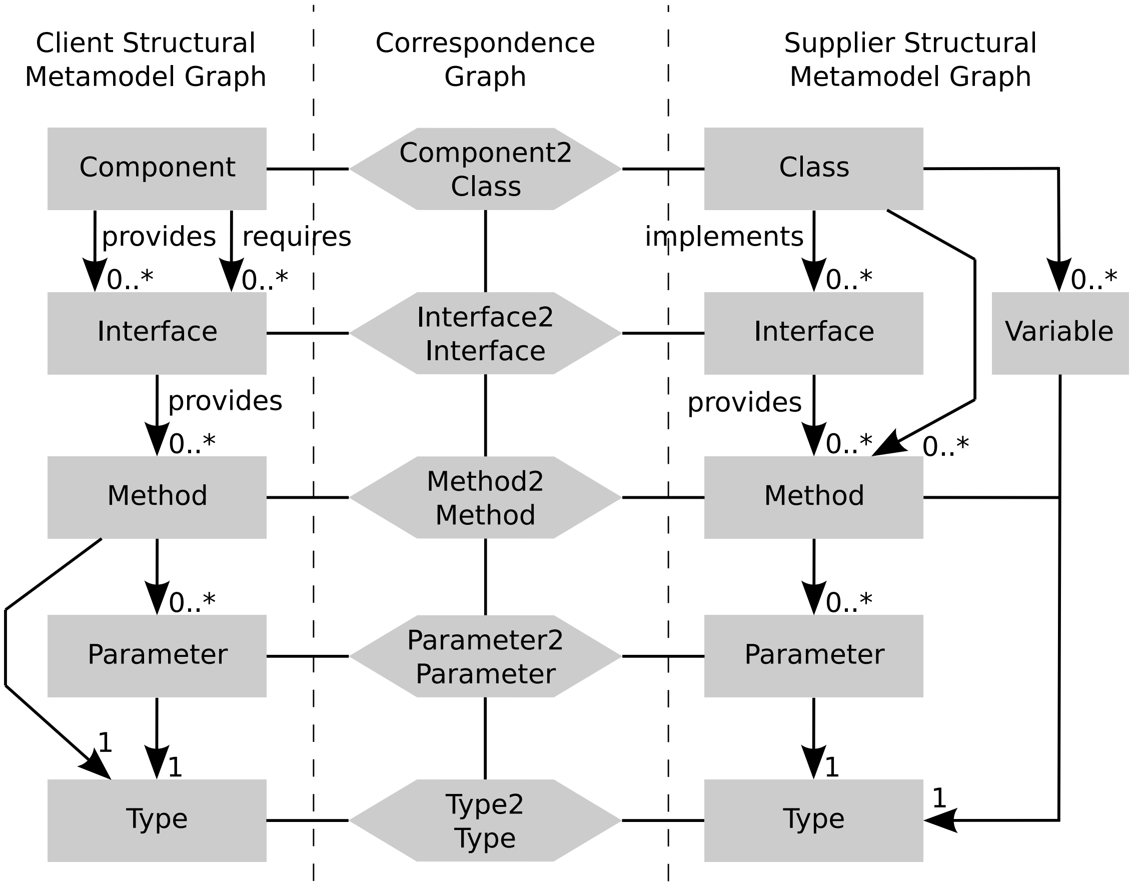


Figure 6.2: Possible triple graph grammar for the structural metamodel of the running example with the structural metamodel transformation

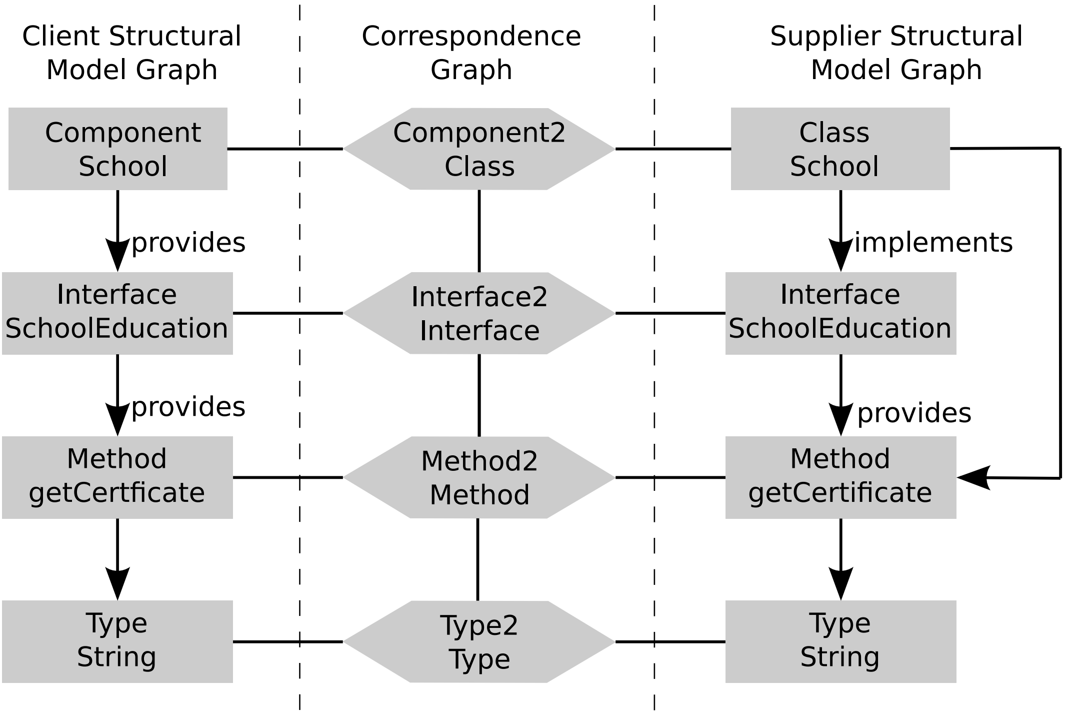


Figure 6.3: Excerpt of the correspondence model between the client and supplier structural model of the running example

### 6.2.2 Creation of the Correspondence Models during the Supplier Analysis Model Skeleton Generation

Both correspondence models are created during the generation of the supplier analysis model skeleton. If both analysis models already exist and the *requirement of consistency*, as described in section 5.1.2, holds, the correspondence models are generated during the supplier-analysis-model-skeleton generation. However, the generated supplier analysis model skeleton is not used after generation because the supplier analysis model already exists.

At first, only the structural correspondence model and the structural transformation $T_{\text{structural}}$ are considered. During the transformation, an exact matching between a set of client structural model elements $E_{\text{client}}$ and one supplier structural model element $e_{\text{supplier}}$ takes place. The matching is based on the structural metamodel transformation $T_{\text{structural,meta}}$. As described in section 6.2.1, the structural correspondence metamodel is based on $T_{\text{structural,meta}}$. Therefore, the matching which takes place during $T_{\text{structural}}$ is added to the structural correspondence model. Let $MM_{E_{\text{client}}}$ be the set of the metamodel elements which are used in $E_{\text{client}}$ and $MM_{e_{\text{supplier}}}$ the metamodel element of $e_{\text{supplier}}$. The set of $MM_{E_{\text{client}}}$ has an edge to a correspondence node which has also an edge to $MM_{e_{\text{supplier}}}$. Therefore, the elements for $MM_{E_{\text{client}}}$ respectively $MM_{e_{\text{supplier}}}$ in the structural correspondence model are initiated with $E_{\text{client}}$ respectively $e_{\text{supplier}}$. Their correspondences are added to the structural correspondence model. The same process is applicable to the creation process of the security correspondence model during the supplier security model generation. The reason for this is that the security correspondence metamodel is based on $T_{security,meta}$ which is used for $T_{security}$.

During the generation of the supplier analysis model skeleton, multiple correspondences are created and added to one of the correspondence models. If a client analysis model element is involved more than once in $T_{\text{structural}}$ or $T_{\text{security}}$, it has multiple edges to different correspondence nodes. If the same supplier analysis model element can be generated from different sets of client analysis model elements, it has edges to different correspondence nodes. Whenever $T_{\text{structural}}$ is used to generate a new supplier analysis model element, the structural correspondence model is updated, and whenever $T_{\text{security}}$ is used, the security correspondence model is updated.

## 6.3 Supplier Analysis Model Skeleton Generation

If the supplier analysis model does not already exist, the supplier analysis model skeleton is generated from the client analysis model. In this process also the structural and the security correspondence model is created. The supplier analysis model consists of a structural model and a security model as described in section 4.1. As stated in section 4.1, the client analysis model does not have the complete information about the internal implementation of the components. Therefore, it is not possible to generate a complete supplier analysis model from the client analysis model. The supplier analysis model skeleton consists of the supplier structural model skeleton containing the structural data existing in the client analysis model. Furthermore, the supplier security model skeleton only contains the security information which can be generated from the client analysis model. A supplier analysis model skeleton, for instance, only contains the interfaces and the components implementing the interfaces. However, the implemented methods have empty bodies because the exact internal implementation of the components is unknown.

To couple two security analyses, a structural metamodel transformation $T_{\text{structural,meta}}$ and a security metamodel transformation $T_{\text{security,meta}}$ are required, as introduced in section 5.1.1 respectively section 5.2.3. The corresponding model transformation is constructed based on the triple graph grammar, implied by a metamodel transformation [74]. The client and the supplier analysis model can be represented as a directed graph, with the model elements as nodes and their edges to each other. In the running example from section 4.2, the interface SchoolEducation has an edge to the method getCertificate(), which has an edge to its type and to its security level. SchoolEducation has also incoming edges from the components School and Person. The graph of the client analysis mode is the input of the generation process with the supplier analysis graph as output. The graph of the client analysis model can be traversed. In the graph traversal, it is checked for each node if $T_{\text{structural}}$ or $T_{\text{security}}$ can be applied. If one of the two transformations can be applied, a new supplier model element is created in the output graph [75]. However, $T_{\text{structural,meta}}$ or $T_{\text{security,meta}}$ can use multiple client analysis model elements as input to create one supplier analysis model element. In simple graph traversal, only one-to-one matching is possible. Therefore, a graph traversal is not sufficient to generate the client analysis model. One possible solution for this case is pattern matching [76]. The patterns are provided by the metamodel transformations $T_{\text{structural,meta}}$ and $T_{\text{security,meta}}$. Each metamodel transformation defines all sets of model elements from which a supplier analysis model element can be generated. Those input sets of the transformations are patterns that are searched in the client analysis model. The patterns can also be viewed as graphs such as the client and supplier analysis model. Therefore, the search for a pattern graph in the client analysis model graph is a subgraph isomorphism problem [77]. For each pattern found in the client analysis model, the corresponding supplier analysis model element is generated with $T_{\text{structural}}$ or $T_{\text{security}}$. Each time $T_{\text{structural}}$ or $T_{\text{security}}$ is applied, it is checked whether the supplier analysis model element that would be generated is already in the generated supplier analysis model skeleton or not. If it is already in the supplier analysis model skeleton, it is not added to the supplier analysis model skeleton. Otherwise, there are duplicates in the generated supplier analysis model skeleton. Each time $T_{\text{structural}}$ or $T_{\text{security}}$ is used, the structural or the security correspondence model has to be updated with the newfound correspondence. If the generated supplier analysis model element already exists in the supplier analysis model skeleton, it is not added again, but the newly found correspondence has to be added to the fitting correspondence model.

## 6.4 Back-Projection of the Supplier Analysis Results

After the supplier analysis model skeleton is supplemented with the internal structure of the component, the supplier analysis is run to produce supplier analysis results. The supplier results are projected back into the client analysis model for the information transfer back to the client analysis model. In this approach, the back-projection $T_{\text{proj}}$ is defined as:

$$T_{\text{proj}} : M_{\text{result}} \times C_{\text{structural}} \times C_{\text{security}} \rightarrow M_{\text{client}} \tag{6.1}$$

The back-projection $T_{\text{proj}}$ uses the supplier result model $M_{\text{result}}$, the structural correspondence model $C_{\text{structural}}$, and the security correspondence model $C_{\text{security}}$ to produce new client analysis model elements. The client analysis model elements are bundled by $M_{\text{client}}$. The client analysis model is updated with $M_{\text{client}}$.

The back-projection is split into two steps. The first step is to determine the adapted security specification on the public method of the result trace. The second step is to adapt the client analysis model. If the supplier analysis does not find any security violation, the assumptions of the client analysis examined by the supplier analysis hold, and nothing has to be projected back. However, if at least one invalid trace through the supplier analysis is detected, the result is projected back.

In section 5.3, the transformation $T_{\text{result}}$ is introduced. It transforms any result which contains information about the invalid traces into a simplified result. The metamodel of the simplified result is depicted in figure 5.2. The simplified result consists of a list of invalid traces. Each trace consists of trace states. Each trace state contains information about the executed action and its security specification.

In this approach, as introduced in section 4.1, the ground truth is the implementation. The client analysis model only assumes the security properties which are examined by the supplier analysis. The security model of the supplier analysis model skeleton is generated from the client analysis model. Therefore, the security model of the supplier analysis model

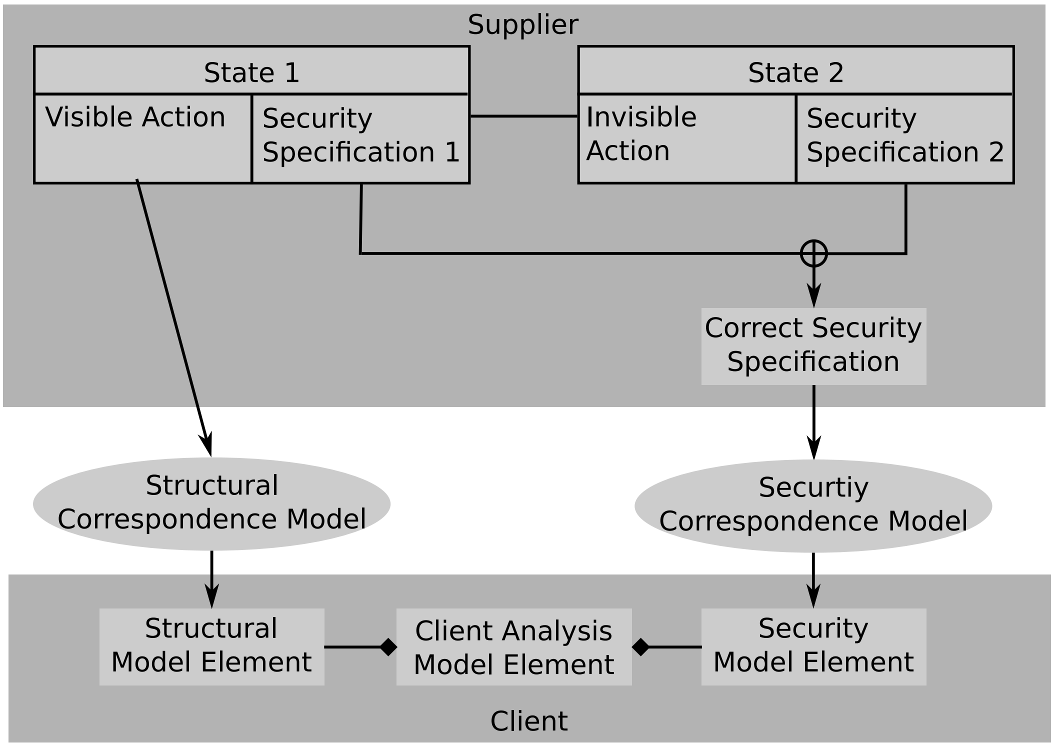


Figure 6.4: A result trace has to contain at least one public software state, in this case, the visible action. Furthermore, each trace has to contain a security specification. The security specifications are combined into the correct security specification of the visible action. The visible action is converted to a client structural model element with the structural correspondence model. The correct security specification is transformed into a client security model element with the security correspondence model.

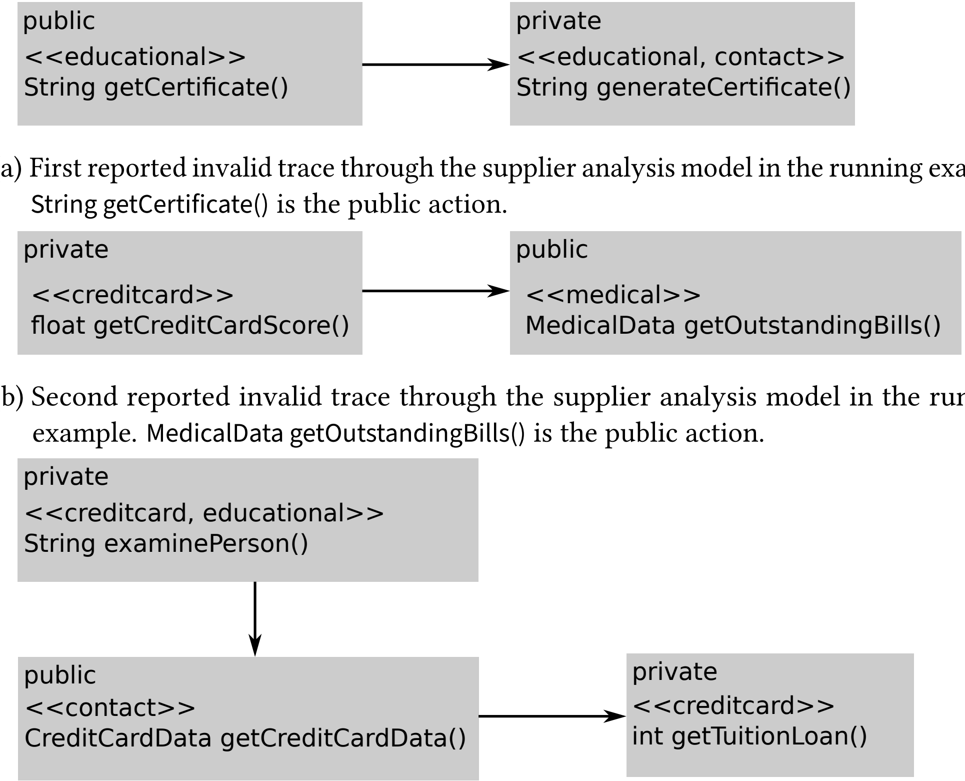
skeleton also contains the assumptions of the client analysis model. The assumptions are verified with the supplier analysis. If the supplier analysis finds invalid traces through the supplier analysis model and there are trace states with public actions in the invalid traces, the assumption of the security specification of this public action is wrong. The security specifications of the public actions are wrong because the internal implementation of the components is assumed to be correct. For each public action that is not contained in any invalid trace, the security specification is correct.

The schema of the back-projection of one invalid trace is shown in figure 6.4. First, the back-projection has to determine the adapted security specification on the visible actions of the result trace. These adapted security specifications are used to adapt the client analysis model. The back-projection is comparable with the approach of iObserve, as introduced in section 3.3. The monitoring events in the run-time model are used as ground truth on which the workload specification in the architecture model is adapted.

As described in section 4.1, the security domain of the supplier analyses is the domain of lattice-based security properties, introduced in section 2.6. At first, it is assumed that the supplier analysis only examines one lattice-based security property. If the supplier analysis examines multiple lattice-based security properties, the supplier analysis result is split into single results for the different security properties. The single results for each analyzed security property are projected back independently.

### 6.4.1 Determining the Adapted Security Specification for One Invalid Trace

The supplier analysis model skeleton of the running example, introduced in section 4.2, is manually implemented. An excerpt of the implementation is described in section 4.2. If the supplier analysis is run on this implementation excerpt, three invalid traces are detected, shown in figure 6.5.



(a) First reported invalid trace through the supplier analysis model in the running example. String getCertificate() is the public action.

(b) Second reported invalid trace through the supplier analysis model in the running example. MedicalData getOutstandingBills() is the public action.

(c) Third reported invalid trace through the supplier analysis model in the running example. CreditCardData getCreditCardData() is the public action.

Figure 6.5: Invalid traces through the supplier analysis model reported by the supplier analysis

In the following, a single invalid trace with $n$ states but only one state $s$ with a public action is considered. Then, the security specification of $s$ has to be adapted so that the new security specification resolves the invalid trace. The trace state with the public action can be determined with the structural correspondence model. A trace state is a public trace state if its action is present in the structural correspondence model. In this approach to couple security analyses, the adapted security specification is calculated based on the supplier analysis results. To calculate the adapted security specification of $s$, only the previous trace state and the next trace state of $s$ in the invalid trace have to be considered.

The reason for this is that the manual implementation is viewed as ground truth, as described in section 4.1. Therefore, each internal information flow to the previous trace state or from the next trace state in the invalid trace must be correct. The previous trace state of $s$ is called the predecessor of $s$ denoted with $pred_s$. The next trace state of $s$ is called the successor of $s$ denoted with $succ_s$. The security level of $s$ is denoted by $sc_s$. The adapted security level of $s$ is denoted by $\widetilde{sc}_s$. The notation $sc_x \longrightarrow sc_y$ means that information may flow from the security level of state $x$ to the security level of state $y$. Accordingly, the notation $sc_x \nrightarrow sc_y$ means that information is not allowed to flow from the security level of state $x$ to the security level of state $y$.

The trace state $s$ can have three different positions in the invalid trace. The three positions and possible strategies to determine the adapted security specifications for each position are examined in sections 6.4.1.1 to 6.4.1.3 respectively.

#### 6.4.1.1 The Public Trace State at the Beginning of the Invalid Trace

The first position is at the beginning of an invalid trace. Therefore, only hidden states of the software are entered from $s$ in the invalid trace. In this case, the predecessor of $s$ does not exist. An example for this case is depicted in figure 6.5a. There is an invalid flow from educational to educational, contact which is not in the lattice of the supplier analysis model, introduced in section 4.2. In section 2.6, the least-upper-bound operator is introduced as an operation that outputs the first security level that relates to both input security levels. The adapted security specification for $s$ on the first position can be determined with the least-upper-bound operator of the lattice. The least-upper-bound operator is used because the invalid trace is resolved. Furthermore, all information flows to any software state accessible from $s$ remain valid because $\widetilde{sc}_s \longrightarrow sc_s$ is in the lattice, which is a property of the least-upper-bound operator. The input of the least-upper-bound operator is the security level of the public action itself and the security level of its successor:

$$\widetilde{sc}_s = sc_s \oplus sc_{succ_s} \tag{6.2}$$

For the invalid trace seen in figure 6.5a, the adapted security specification of String getCertificate() is

$$\widetilde{sc}_{String\,getCertificate()} = educational \oplus educational, contact = educational, contact \tag{6.3}$$

However, this strategy does not always work. The first case in which this strategy does not work is a missing definition for the least-upper-bound for the two security specifications. That is not possible for the security lattice of the running example. To build a counterexample, consider a security lattice with the following relations:

$$A \longrightarrow B, \qquad C \longrightarrow B \tag{6.4}$$

If the public state in the invalid trace has security level $A$ and the successor has security level $C$, then there is no security level that relates to $A$ and $C$. In this case, the least-upper-bound for $A$ and $C$ is not defined. The fallback strategy for this case is to use the security level of the successor as the adapted security level of $s$:

$$\widetilde{sc}_s = sc_{succ_s} \tag{6.5}$$

This strategy to determine the adapted security specification for $s$ has problems when another software state $y$, from which $s$ is entered, exists and is not in the invalid trace. The reason is that the adapted security level of $s$ is higher than the old security level of $s$. Therefore, it is possible that $sc_y \longrightarrow \widetilde{sc}_s$ is not in the security lattice. This is the case, if $sc_y = sc_s$. This problem also occurs with the fallback strategy. We view the manual implementation of the supplier analysis as the ground truth in this approach to couple security analyses. Therefore, a new invalid trace is reported after the adaption of $sc_s$. If the supplier analysis is re-run, the manual implementation contains errors, which violates the correctness requirement of the manual implementation.

#### 6.4.1.2 The Public Trace State at the End of the Invalid Trace

The second possible position of $s$ is at the end of the trace. Therefore, $s$ is only entered from hidden states. In this case, the successor of $s$ does not exist. An example for this case is shown in figure 6.5b. There is an invalid flow from creditcard to medical which is not in the lattice of the supplier analysis model, introduced in section 4.2. The strategy for determining the adapted security specification, in this case, is to determine a security specification $\widetilde{sc}_s$ with:

$$sc_{pred_s} \longrightarrow \widetilde{sc}_s \tag{6.6}$$

$$sc_s \longrightarrow \widetilde{sc}_s \tag{6.7}$$

Therefore, information can flow from $pred_s$ to $s$ with the adapted security level $\widetilde{sc}_s$. Furthermore, information may flow from each software state, which is not in the invalid trace from which $s$ can be entered, to $s$. However, this strategy does also have some shortcomings. The first shortcoming is revealed in the second invalid trace of the running example, shown in figure figure 6.5b. $\widetilde{sc}_s$ should fulfill the following two equations:

$$creditcard \longrightarrow \widetilde{sc}_s \tag{6.8}$$

$$medical \longrightarrow \widetilde{sc}_s \tag{6.9}$$

There is no security level which fulfills these two equations. The fallback strategy for this case is to use $\widetilde{sc}_s = sc_{pred_s}$. Furthermore, if $s$ enters another software state $y$, it can be, that $\widetilde{sc}_s \longrightarrow sc_y$ is not in the lattice. The fallback stratey cannot solve this case. An intuitive approach is to require, that also the equation $\widetilde{sc}_s \longrightarrow sc_s$ holds for a adapted $\widetilde{sc}_s$. However, if $sc_{pred_s} \longrightarrow \widetilde{sc}_s$ and $\widetilde{sc}_s \longrightarrow sc_s$ can be fulfilled, then the invalid information flow from $pred_s$ to $s$ is not reported in the first place. The reason for this is, that the relations in a lattice are transitive and $sc_{pred_s} \longrightarrow sc_s$ is also in the lattice. As mentioned in section 6.4.1.1, this case must not appear in the supplier analysis results, because the internal implementation is assumed to be correct.

#### 6.4.1.3 The Public Trace State in the Middle of the Invalid Trace

The third position of $s$ in an invalid trace is at the position $i$ with $0 < i < n$. Therefore, $s$ has a successor as well as a predecessor. The strategy for this case is to calculate a security level to which information can flow from $sc_{pred_s}$ and from which information can flow to $sc_{succ_s}$:

$$sc_{pred_s} \rightarrow \widetilde{sc}_s \tag{6.10}$$

$$\widetilde{sc}_s \rightarrow sc_{succ_s} \tag{6.11}$$

The reason for this strategy is that the information flow $sc_{pred_s} \rightarrow \widetilde{sc}_s \rightarrow sc_{succ_s}$ is valid if $\widetilde{sc}_s$ holds the equations (6.10) and (6.11). However, a valid information flow from every other state which can enter $s$ or to every state $s$ can enter is no longer ensured. To ensure this, additionally $\widetilde{sc}_s \rightarrow sc_s$ and $sc_s \rightarrow \widetilde{sc}_s$ has to be fulfilled, whereby a bidirectional relation would exist between the two security levels. If $\widetilde{sc}_s \rightarrow sc_s$ could be fulfilled, the initial information flow would not have been reported because $sc_{pred_s} \rightarrow \widetilde{sc_s} \rightarrow sc_s$ is a valid flow. Therefore, only $sc_s \rightarrow sc_{succ_s}$ would have been reported. If $sc_s \rightarrow \widetilde{sc}_s$ could be fulfilled, then also $sc_s \rightarrow \widetilde{sc}_s \rightarrow sc_{succ_s}$ is a valid flow. Therefore, only $sc_{pred_s} \rightarrow sc_s$ would have been reported. In figure 6.5c, an example for this case is shown. The invalid information flow is:

$$educational, contact \rightarrow creditcard \rightarrow educational \tag{6.12}$$

Therefore, $\widetilde{sc}_s$ has to fulfill the following equations:

$$educational, contact \rightarrow \widetilde{sc}_s \tag{6.13}$$

$$\widetilde{sc}_s \rightarrow educational \tag{6.14}$$

A $\widetilde{sc_s}$ which fulfills the two equations is, for instance, {educational, contact} and educational.

It is possible that for an invalid information flow no $\widetilde{sc}_s$ can be found which corrects the invalid information flow. For example:

$$medical \rightarrow contact \rightarrow educational \tag{6.15}$$

contact cannot be replaced with any other security level, so that the invalid trace is corrected.

#### 6.4.1.4 Determining the Adapted Security Specification for a Trace with Multiple Public Actions

The invalid trace could have multiple trace states with different public actions. The set of trace states with public actions in the invalid trace is called $S$. The trace state $s_i$ is at the $i$-th position of the invalid trace. To determine the adapted security levels for the trace states, it is assumed that each $s_i \in S$ has a different public action from all other trace states. Only the successor and the predecessor are necessary to determine the adapted security specification of a trace state. Therefore, there are two possibilities for $s_i$. First, $s_i$ has no trace states with public actions as neighbors, formalized, $succ_{s_i} \notin S$ and $pred_{s_i} \notin S$ hold. In this case, to determine the adapted security level for $s_i$ the strategies from sections 6.4.1.1 to 6.4.1.3 can be used, because the security level of the neighbors are not adapted. The second case is that $s_i$ has at least one neighbor that contains a public action. Let $I = \{j, \dots, k\}$ be the set of indices of the first sequence of trace states with public actions, whereby, $s_j \in S$, $s_{j+1} \in S, \ldots, s_k \in S$. This sequence of trace states is only reported if $sc_{s_i} \nrightarrow sc_{s_{i+1}}$ applies $\forall i \in I, i < j$. If $j > 1$, then also $sc_{pred_{s_j}} \nrightarrow sc_{s_j}$ holds. If $k < n$ holds, then also $sc_{s_k} \nrightarrow sc_{succ_{s_k}}$ holds. Otherwise, multiple invalid traces are reported because there must be an invalid information flow from each trace state to the next trace state. To determine the adapted security specification for each trace state in the sequence, the adapted security level is determined step by step. First, the adapted security level for $s_j$ is determined, then for $s_{j+1}$ and so on. The adapted security level for $s_j$ is determined, by viewing $s_{j+1}$ as a trace state with an internal action. Then, one of the three before-mentioned strategies is used to determine the adapted security specification for $s_j$. To determine the adapted security level for $s_{j+1}$, the adapted security level $\widetilde{sc}_{s_j}$ of $s_j$ is seen as correct. This is repeated for each trace state in the sequence.

If the first sequence of trace states with public actions in the invalid trace is resolved, the next sequence is resolved with the same strategy. This strategy can have cases in which it does not work because it is based on the three strategies from sections 6.4.1.1 to 6.4.1.3, which have edge cases, for which the correct security level cannot be determined. If this approach to determine the security levels based on the supplier analysis results cannot resolve all invalid traces, the user of this approach is notified.

#### 6.4.1.5 Determining the Adapted Security Specification for Multiple Traces with Multiple Public Actions

The supplier analysis can find multiple invalid traces with multiple trace states with public actions in each invalid trace. Each trace state with a public action can be in multiple invalid traces. To determine the adapted security specification for each public action in the supplier result, an equation system is built. The trace state $s_{m,i}$ is the $i$-th state in the $m$-th invalid trace. Let $I$ be the set of all $(m, i)$ for which the trace state $s_{m,i}$ has a public action. For each $s_{m,i}$ with $(m, i) \in I$ the following equations have to hold:

$$sc_{pred_{s_{m,i}}} \longrightarrow \widetilde{sc}_{s_{m,i}} \tag{6.16}$$

$$\widetilde{sc}_{s_{m,i}} \longrightarrow sc_{succ_{s_{m,i}}} \tag{6.17}$$

The equations are based on the third strategy described in section 6.4.1.3. If the public action of $s_{m,i}$ is only used once and $s_{m,i}$ is not part of a sequence, as described in section 6.4.1.4, and $i = 1$ respectively $i = n_m$, then the strategy from section 6.4.1.1 respectively section 6.4.1.2 is used. For all other trace states $s_{m,i}$ with $(m, i) \in I$ the equation system $E$ can be built:

$$E = \{sc_{pred_{s_{m,i}}} \longrightarrow \widetilde{sc}_{s_{m,i}}, \widetilde{sc}_{s_{m,i}} \longrightarrow sc_{succ_{s_{m,i}}} \,|(m, i) \in I\} \tag{6.18}$$

The equation system consists of two equations for each public trace state in the results. The first equation for a public trace state is, that information may flow from the security level of the predecessor to the adapted security level of the public trace state. The second equation for a public trace state is, that information may flow from the adapted security level of the public trace state to the security level of the successor. Then, the set $M = \{\widetilde{sc_{s_{m,i}}} \,|(m, i) \in I\}$ solves the most equations of $E$.

### 6.4.2 Adapting the Client Analysis Model

A client analysis model element consists of a structural and a security model, as described in section 6.1. The security model of the client analysis model is adapted with the adapted security specification if the supplier analysis finds a security violation. The adaption of the security model of the client analysis model is a model update [78]. A model update consists of the model elements which are updated and the updated model element. The old model element is replaced with the new one.

If the supplier analysis does not find any invalid traces, the client analysis model is not adapted. However, if the supplier analysis result contains invalid traces, then the client analysis model is based on assumptions verified to be wrong with the supplier analysis. Therefore, the client analysis model has to be adapted with the determined client analysis model elements based on the supplier analysis results. The set of adapted security levels $M$ is used to adapt the client analysis model. For each adapted security level in the trace $m$ at the position $i$ $\widetilde{sc}_{s_{m,i}}$, the corresponding client structural model elements $SM_{\text{client}}$ to the action of the state $s_{m,i}$ are found with the structural correspondence model. The corresponding client security model elements $M_{\text{client}}$ for $\widetilde{sc}_{s_{m,i}}$ are found with the security correspondence model. The old security specification of $SM_{\text{client}}$ is updated with $M_{\text{client}}$. The model update of the client analysis model can then be triggered with the pair of the old corresponding client analysis model elements and the new, adapted client analysis model elements.

In the running example, the three invalid flows seen in figure 6.5 are reported. In section 6.4.1.1, the adapted security level for the public trace state with the method getCertificate is determined. The adapted security level for getCertificate is {educational, contact}. The structural correspondence model is searched for the Method2Method correspondence with getCertificate as supplier structural model element. The found Method2Method correspondence contains the corresponding client structural element for getCertificate. The annotation of the client structural model element getCertificate is resolved by searching the client analysis model. The old client analysis model element is created by using the client structural model element getCertificate and the found annotation of getCertificate. The new adapted security level of the supplier structural element getCertificate is transformed to the corresponding client security model element with the security structural correspondence model. The new client analysis model element is created by combining the client structural model element getCertificate with the corresponding client security model element to the determined security level. A model update operation on the client analysis model is triggered with the pair of the old and new client analysis model elements.

# 7 Implementation in a Case Study

The approach to couple two security analyses is instantiated with two concrete security analyses. The initiation is used to evaluate the approach. We use the Access Analysis of Kramer et al. [38] as client analysis. The information flow analysis JOANA is used as supplier analysis. The PCM repository model [79] is used as the client structural model. Confidentiality4CBSE is used as the client security model, as introduced in section 2.7.1. The used supplier structural metamodel is a Java source code metamodel. The supplier security metamodel is JOANA, described in section 2.7.2. JOANA uses annotations of Java methods.

In section 7.1, the overview of the architecture of the implementation is presented. In the following sections, it is shown that Confidentiality4CBSE and JOANA can be coupled. Furthermore, the used transformations to form the megamodel, described in section 6.1, are presented. The generation of annotated Java source code is a two-step process. The first step is to generate a reduced Java source code model, described in section 7.2, and a JOANA security model, described in section 7.3. The second step is to generate annotated source code from the two models, described in section 7.4. The advantage of this two-step approach is that the generated models can be used in other model-driven approaches like the back-projection. The last transformation used to form the megamodel is the back-projection. The implementation of the back-projection is explained in section 7.5.

## 7.1 Architecture Overview

The approach is implemented as two eclipse plugins. One plugin contains the generation capabilities for the annotated source code and the other for the back-projection. The plugins are written in Java and Xtend [80]. The plugins can be found in the package `edu.kit.kastel.scbs.pcm2java4joana` [81]. The architecture is shown in figure 7.1. There are three main packages on which these two plugins are based. The first package includes the handlers, which orchestrate the plugin executions. The handler for the annotated source code generation is in the class `PCM2Java4JoanaHandler.java`, and the handler for the back-projection is in `BackprojectionHandler.java`. A user of the plugin triggers the handlers. The second package consists of the model generators for generating the supplier analysis model. The third package is concerned with the back-projection. The supplier results are first parsed with the `JoanaResultParser`. Then, the adapted security level is determined with the `CorrectSecurityLevelFinder`. The class `ClientAnalysisModelUpdater` uses the found security levels to adapt the client analysis model.

In this approach, models for Java source code, JOANA, structural and security correspondences, and the supplier results are used besides the required client supplier analysis model. The models are based on the Ecore model from the eclipse modeling framework [82]. The models can be found in the packages `edu.kit.kastel.scbs.pcm2java4joana.sourcecode` [83], `edu.kit.kastel.scbs.pcm2java4joana.correspondencemodel` [84], and `edu.kit.kastel.scbs.pcm2java4joana.joana` [85]. Besides the models, three helper packages are used. First, the helper package `edu.kit.kastel.scbs.pcm2java4joana.models` is used for aggregated models. An example aggregated model is the abstract supplier analysis model, which combines a source code model with a JOANA model and the correspondence models. Secondly, the helper package `edu.kit.kastel.scbs.pcm2java4joana.utils` is used, where utilities for the different models are located. The third helper package is the exceptions package `edu.kit.kastel.scbs.pcm2java4joana.excceptions`, which bundles self-defined exceptions for the implementation.

## 7.2 Structural Transformation and Structural Correspondence Model

In this section, the connection between the client structural model and the supplier structural model in the initiation with the analyses Access Analysis and JOANA are described. The connection between the two structural models is based on the work of Konersmann, described in section 3.5. The client analysis metamodel for the Access Analysis is the PCM repository model extended with Confidentiality4CBSE, as introduced in section 2.7.1. Therefore, the client analysis metamodel contains structural information about the software system. As described in section 2.3, the repository metamodel consists of among other elements BasicComponent, OperationInterface and CompositeDataType. A BasicComponent can provide or require OperationInterfaces. Therefore, the components have concrete interfaces. An OperationInterface can have OperationSignatures, which define methods of the interface. Therefore, component-based software systems can be modeled with the repository metamodel. A repository model which initializes at least one BasicComponent contains structural information about the software system and fulfills the *requirement of structural information*, as introduced in section 5.1.1.

The supplier structural model is a Java model which uses the Java programming language as metamodel. The Java programming language contains model elements for specifying a component-based software system. Facade classes that implement the provided interfaces of a component are used to specify components in the Java model [86]. The Java interfaces are used as provided interfaces of the components [87]. The used Java source code metamodel contains a subset of the Java programming language because not every element of the Java programming language is required. For example, inheritance of interfaces from other interfaces is not required in this bachelor's thesis. The Java source code metamodel is used for the concrete transformation in this bachelor's thesis. The Java source code metamodel is described in section 7.2.1. A correspondence model is created between the client structural model and the supplier structural model, described in section 7.2.3. The structural correspondence model is based on the structural metamodel transformation, introduced in section 7.2.2. The concrete structural transformation is described in section 7.2.4.

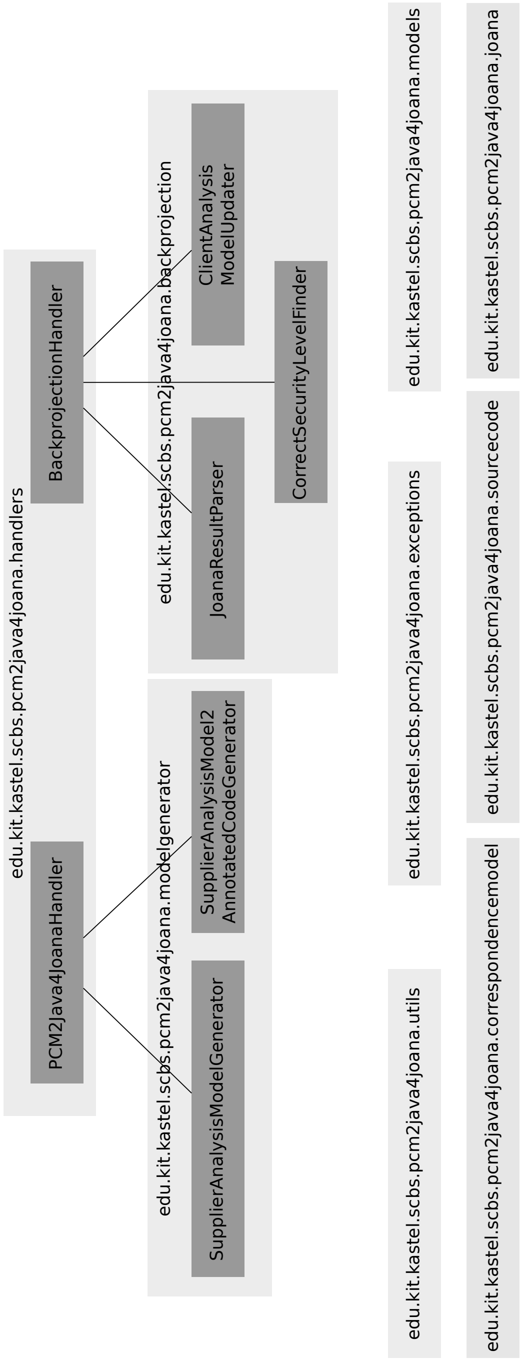

edu.kit.kastel.scbs.pcm2java4joana.handlers
BackprojectionHandler
PCM2Java4JoanaHandler
edu.kit.kastel.scbs.pcm2java4joana.backprojection
ClientAnalysis
ModelUpdater
CorrectSecurityLevelFinder
JoanaResultParser
edu.kit.kastel.scbs.pcm2java4joana.modelgenerator
SupplierAnalysisModel2
AnnotatedCodeGenerator
SupplierAnalysisModelGenerator
edu.kit.kastel.scbs.pcm2java4joana.models
edu.kit.kastel.scbs.pcm2java4joana.exceptions
edu.kit.kastel.scbs.pcm2java4joana.utils
edu.kit.kastel.scbs.pcm2java4joana.joana
edu.kit.kastel.scbs.pcm2java4joana.sourcecode
edu.kit.kastel.scbs.pcm2java4joana.correspondencemodel

Figure 7.1: Architecture overview of the implemented approach

### 7.2.1 Reduced Java Source Code Model

A Java metamodel based on the Ecore model is used because Ecore-based models can easily be added to toolchains for model-driven software development. For simplicity, a reduced Java metamodel is used because not all Java metamodel elements are necessary for the transformation from a repository model to a Java model.

The reduced Java source code metamodel is illustrated in figure 7.2. The reduced Java source code metamodel can be found in the project `edu.kit.kastel.scbs.pcm2java4joana.sourcecode`. The source code model can contain TopLevelTypes like Classes and Interfaces. The Interfaces and Classes can contain Methods. A Method can have Parameters and a (return) Type. The Type is the return type of the Method. If the Type is null, then the return type is `void`. A Type can be a BuiltInType, like `int`, `String`, or `float`, a ReferenceType, which contains a Class, Interface, or a CollectionType, which has an InnerType. A Class can also contain Variables.

### 7.2.2 Structural Metamodel Transformation

The structural metamodel transformation generates supplier structural metamodel elements from client structural metamodel elements. A PCM repository BasicComponent is transformed into a Class. A PCM repository Interface is transformed into a Java Interface. The Methods, Parameters, and Types of the repository model are transformed into Java Methods, Parameters, and Types. Java Methods are assigned to Interfaces and Classes. A CompositeDataType is transformed into a Class with the inner declarations of CompositeDataType as Variables. The ProvidedRoles between BasicComponents and Interfaces are transformed into implements relations in the Java metamodel. The methods of a provided Interface are added to the class corresponding to the BasicComponent of the ProvidedRole. The required Interfaces of BasicComponents are transformed into instance Variables of the Classes which are generated from the BasicComponent.

### 7.2.3 PCM-Java Correspondence Model

As described in section 6.2.1, the structural metamodel transformation $T_{\text{structural,meta}}$ implies a triple graph grammar that is shown in figure 7.3. A structural correspondence metamodel can be derived from the triple graph grammar. As described in section 6.4, only the component methods and their parameters are relevant for the back-projection because only to these the stereotype InformationFlow can be applied. Therefore, the correspondence between CompositeDataType and Class as well as between DataType and Type are irrelevant for projecting the supplier analysis results back. Therefore, the structural correspondence model does not have to contain these correspondences.

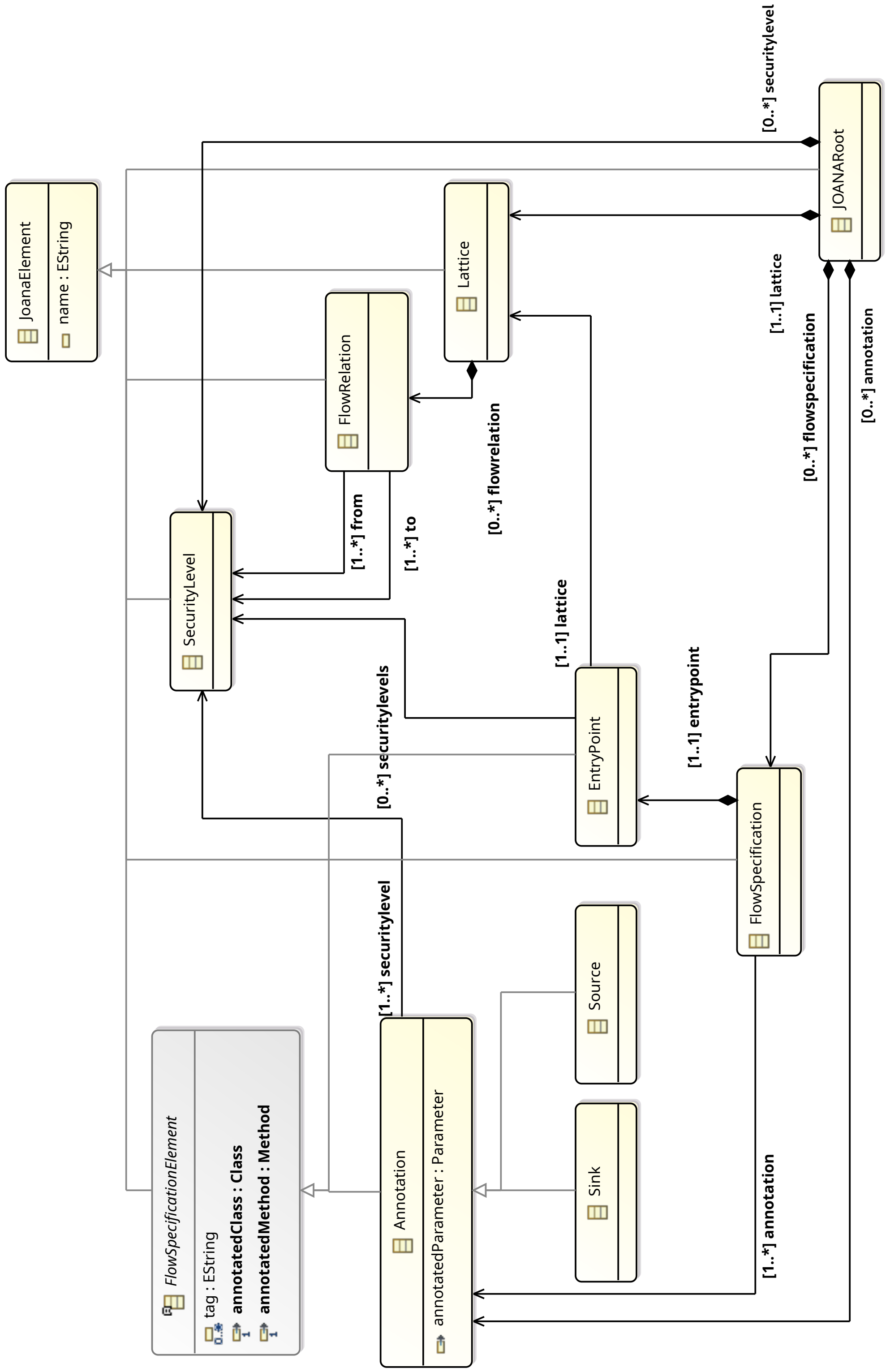


Figure 7.2: Metamodel of the reduced Java source code model as Ecore model

The used structural correspondence metamodel between the repository and Java metamodel is depicted in figure 7.4 and is located in the project `edu.kit.kastel.scbs.pcm2java4joana.correspondencemodel`. The correspondence between components and classes can have multiple Interface2Interface correspondences because a component can require and provide multiple interfaces. An interface can also contain multiple methods. Therefore, Interface2Interface can also have multiple Operation2Method correspondences. Each method can have multiple parameters, thus Operation2Method can have multiple Parameter2Parameter correspondences. All PCM repository model elements are identified with their ID and name.

A structural transformation can be defined between the PCM repository model and the Java source code model [78]. Therefore, a structural correspondence model between the client and supplier structural model exists. Furthermore, the *requirement of a structural correspondence model*, described in section 5.1, is fulfilled. If the generated supplier analysis model skeleton is implemented consistent with the client analysis model, as described in section 5.1.2, then also the *requirement of consistency* is fulfilled.

### 7.2.4 Structural Transformation

The structural transformation $T_{\text{structural}}$ transforms model elements of the repository model into model elements of the Java model. $T_{\text{structural}}$ is based on $T_{\text{structural,meta}}$, introduced in section 7.2.2, and is implemented in the class `SourceCodeModelWithCorrespondenceModelGenerator.java`.

Interfaces and CompositeDataTypes are first class entities because these are not depended on other elements. First, the first class entities are transformed because because Interfaces are needed for transforming ProvidedRoles and RequiredRoles of the BasicComponents. The definition of an interface is, for example, `public interface Example {}`. To transform the first class entities, first, the definitions of the first class entities are generated because the first class entities can reference each other as a datatype. For example, a CompositeDataType or Interface can be the return type of a method. After the definitions of the first class entities are generated, the field definitions of the first class entities are added. In the case of Interfaces, the added fields are Methods. The Interface Methods are generated from the provided OperationSignatures of the Interface. The fields of a class generated from a CompositeDataType are instance Variables which are generated from the InnerDeclarations of the CompositeDataType. The transformation $T_{\text{structural}}$ transforms the BasicComponents into facade classes using the first class entities. A BasicComponent has ProvidedRoles that are referencing Interfaces. The referenced Interfaces are added to the list of implemented Interfaces of the transformed Class. A BasicComponent has RequiredRoles that are referencing Interfaces. The referenced Interfaces are added to the transformed Class as Variables.

Each time a component, interface, method, or parameter is transformed into the corresponding supplier structural element, the correspondence is added to the structural correspondence model, described in section 7.2.3.

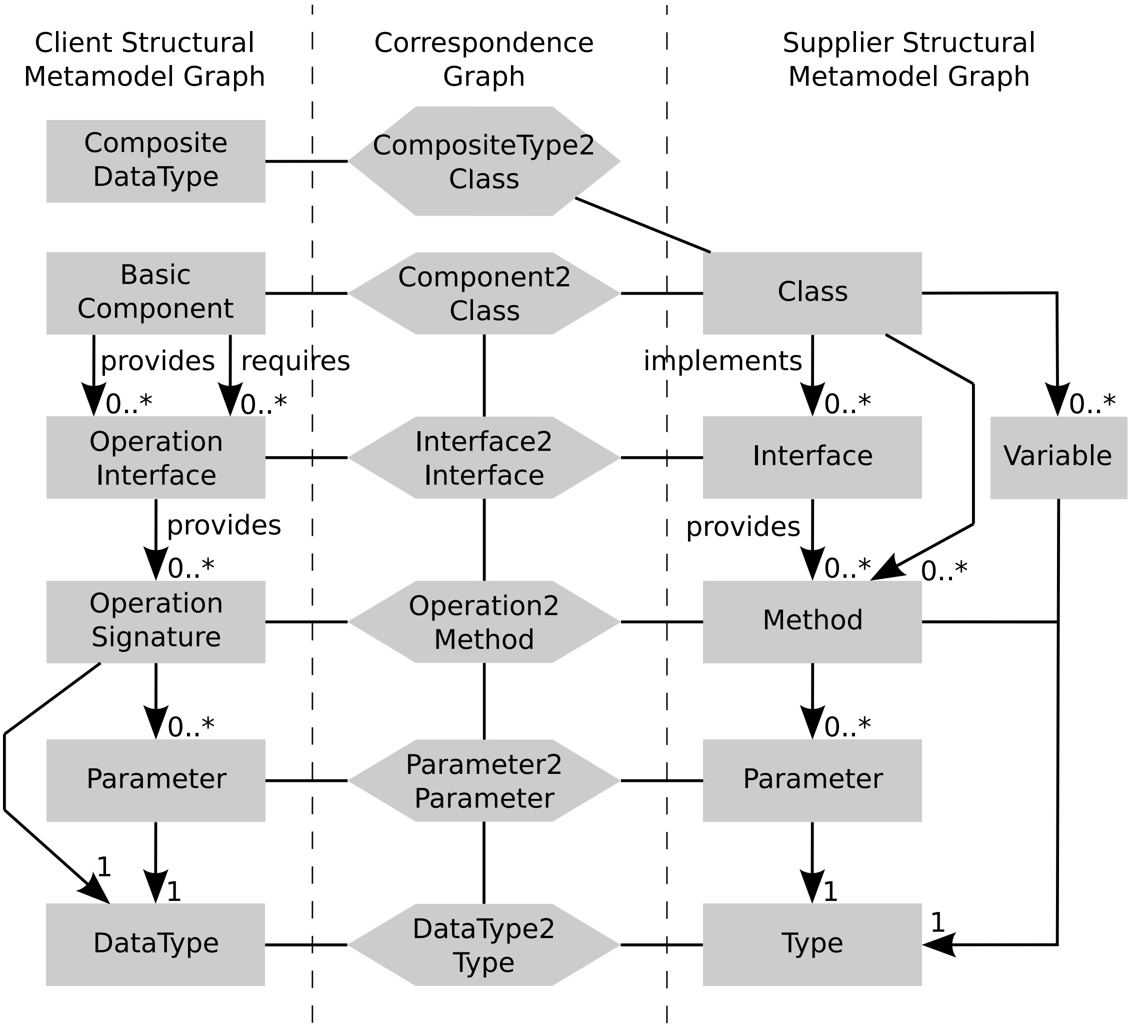


Figure 7.3: Triple graph grammar for the repository and Java model implied by the structural metamodel transformation

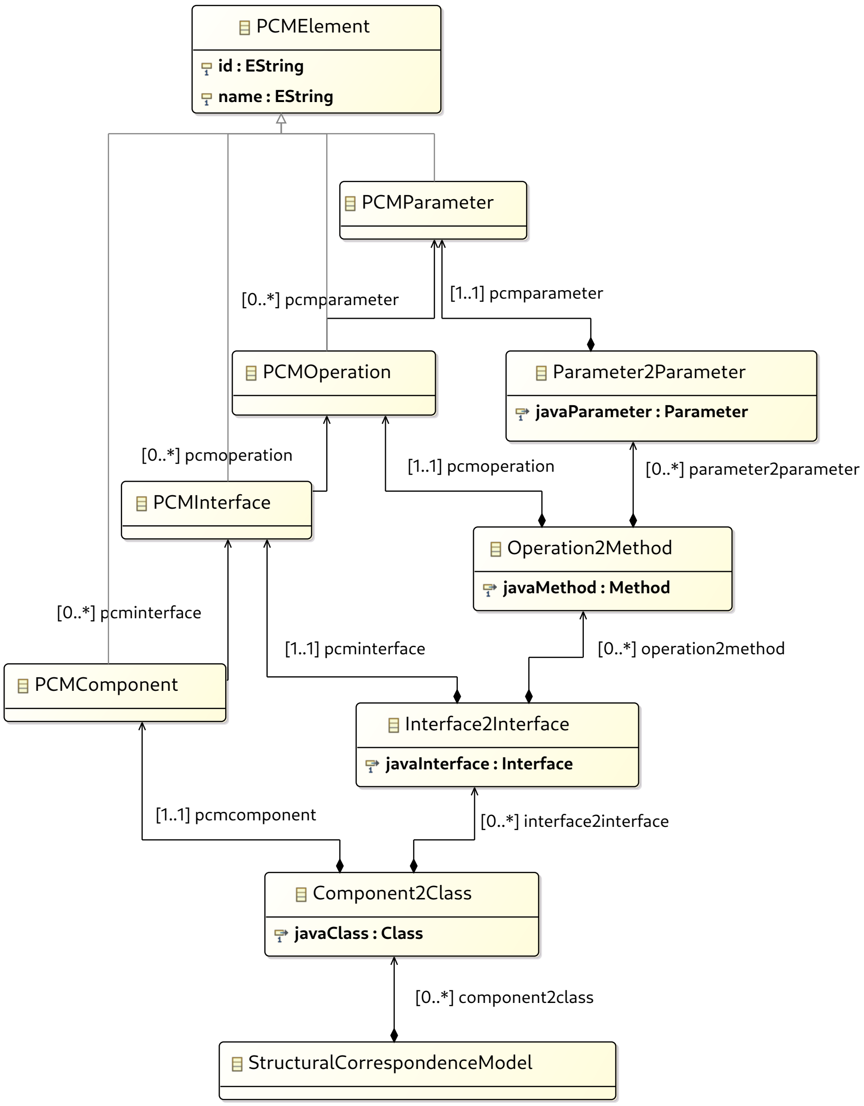


Figure 7.4: Structural correspondence metamodel as Ecore model

## 7.3 Security Transformation and Security Correspondence Model

The used client security metamodel is the metamodel of Confidentiality4CBSE, introduced in section 2.7.1. The metamodel of JOANA is used as the supplier security metamodel. The shared trace property of Confidentiality4CBSE and JOANA is non-interference. Both analyses analyze the software system based on the security level of the methods and parameters. If additionally, a security model transformation exists, then the *requirement of a security correspondence model*, described in section 5.2, is fulfilled. The security metamodel and model transformation are presented in section 7.3.2. The security metamodel transformation is based on an Ecore model for the JOANA metamodel, presented in section 7.3.1. Based on the security metamodel transformation, the security correspondence model between Confidentiality4CBSE and JOANA is created. This process is described in section 7.3.3.

### 7.3.1 JOANA Model

The JOANA metamodel is based on the Ecore model, which is shown in figure 7.5. The JOANA metamodel can be found in `edu.kit.kastel.scbs.pcm2java4joana.joana`. As described in section 2.7.2, JOANA is based on Source, Sink, and EntryPoint elements, which are grouped by a common tag. We abstract from these three model elements for a better management of the model. The model root is JOANARoot. It can have multiple FlowSpecifications. A FlowSpecification consists of an EntryPoint and Annotations with the same tag. The annotation EntryPoint and Annotation are inheriting from FlowSpecificationElement, because both can be assigned to methods of a class. An Annotation can also be assigned to a parameter of a method. The elements Source and Sink are inheriting from Annotation. Each FlowSpecificationElement has a tag which has to be the same for all FlowSpecificationElements in a flow. The element EntryPoint has exactly one Lattice that contains relations between SecurityLevels. A FlowRelation between two sets of SecurityLevels means that information is allowed to flow from the first set of SecurityLevels to the second set. JOANA can only define a flow between two levels. However, it can be beneficial to define a FlowRelation between two sets of SecurityLevels during the model generation. A set of SecurityLevels can be combined into one security level. A set of SecurityLevels is combined into one by sorting the security level names ascending and then merging the names into one `string`. Therefore, FlowRelation defines a set of SecurityLevels as from and the second set of SecurityLevels as to. Therefore, information can flow from a method that has the combined security level of SecurityLevels defined in from to a method that has the combined security level of SecurityLevels defined in to.

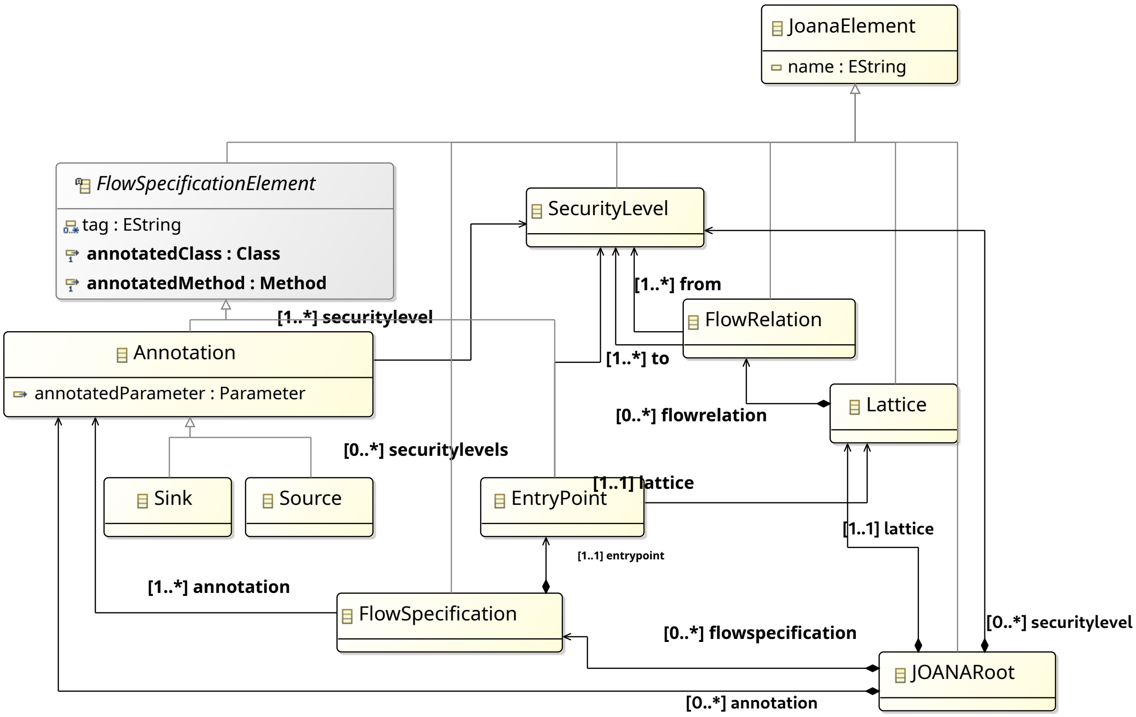


Figure 7.5: JOANA metamodel as Ecore model

### 7.3.2 Security Metamodel Transformation and Security Model Transformation

The JOANA model is generated from the Confidentiality4CBSE model. The security transformation is implemented in the class `AnnodationModelGenerator.java`. As introduced in section 2.7.1, first the DataSets have to be defined to specify confidentiality with Confidentiality4CBSE. The DataSets are assigned to methods and parameters by applying the stereotype InformationFlow. The stereotype InformationFlow assigns a ParametersAndDataPair to a method of an interface. The element ParametersAndDataPair contains DataSets as DataTargets.

The first generated JOANA model elements are the SecurityLevels. A JOANA SecurityLevel is generated from one DataSet. A ParametersAndDataPair can assign a set of DataSets to a method. Each set of DataSets is transformed into a SecurityLevel in the JOANA model.

The second generated JOANA model element is the Lattice. The pseudo-code for the SecurityLevel generation and the Lattice generation is shown in figure 7.6. There is no security lattice defined in the client security model. Therefore, the Lattice is approximated based on the power set of the DataSets. Information may flow from a set of SecurityLevels A to another set of SecurityLevels B if set B is a subset of A. For example, if there are two DataSet instances Medical and CreditCard, then power set is:

$$\{\{\}, \{Medical\}, \{CreditCard\}, \{Medical, CreditCard\}\} \tag{7.1}$$

The empty subset is ignored because it does not contain any SecurityLevel. The other subsets are the SecurityLevels of the JOANA model. As described in section 2.6, each security level has a relation to itself. A method that has the security level $\{Medical, CreditCard\}$ can also access information with the security level $\{Medical\}$ or $\{CreditCard\}$. Therefore, the lattice implied by the security levels in the example is:

$$Medical \longrightarrow Medical \tag{7.2}$$

$$CreditCard \longrightarrow CreditCard \tag{7.3}$$

$$Medical, CreditCard \longrightarrow Medical \tag{7.4}$$

$$Medical, CreditCard \longrightarrow CreditCard \tag{7.5}$$

However, the JOANA Lattice only contains the last two relations because each security level has a relation to itself. The Lattice contains FlowRelations that are not used in the supplier analysis model. Some of the FlowRelations in the Lattice could not be wanted by the developer. Another option to generate the Lattice is to use the data set combinations existing in the client security model for the security lattice. The problem is that it is unknown if those are the only combinations that can be used. The adapted security level could not be possible to determine in the back-projection with the limited set of SecurityLevels.

The next step is to generate the Annotations for the generated supplier structural model from the client security model. In the client security model, the methods and parameters are annotated with the security level of the method or parameter. In the JOANA model, the Annotations are subtyped by Sources or Sinks whereas the client analysis model contains

```
Input: datasets
    securityLevels = new List()
    for dataset in datasets:
        securityLevels.add(generateSecurityLevel(dataset))
    powerSetOfSecurityLevels = generatePowerSet(securityLevels)
    lattice = new Lattice()
    for from in powerSetOfSecurityLevels:
        for to in powerSetOfSecurityLevels:
            if to != {} and from != to and isIn(to, from).
                flowRelation = generateFlowRelation(from, to)
                lattice.addFlowRelation(flowRelation)
```

Figure 7.6: Pseudo code for generating of the JOANA SecurityLevels and of the Lattice

no information about information sources or sinks. Therefore, the assignment of Sources and Sinks to methods and parameters has to be approximated. An Annotation that has neither a tag nor has been initiated as Sink or Source is called a blank annotation. Blank annotations are used as temporary model elements from which later Sinks, Sources, and EntryPoints are generated. In the first step, blank annotations are generated from the client security model because, at this point, it is unknown whether an Annotation is a Source or Sink. In the second step, the blank annotations are converted to Sources and Sinks using an approximation schema.

For each applied stereotype InformationFlow in the client security model, JOANA blank annotations are generated. The stereotype contains information to which method it is applied and which ParametersAndDataPairs it has. A ParametersAndDataPair defines the DataSets of the Method that are later used to generate the SecurityLevels of a Method.

Security information is specified with stereotypes applied to interface methods in the client analysis model. In the supplier analysis model, on the other hand, the security information is specified with annotations on methods of components. Therefore, for each annotated method of an interface in the client analysis model, all components implementing this method in the supplier analysis model have to be annotated. The JOANA Annotations are generated in two steps from an applied InformationFlow stereotype. First, all component methods implementing the method to which the stereotype is applied have to be determined. Then for each component and its method, a JOANA Annotation is generated. The Annotation contains the Class, the Method, and the SecurityLevel. If the ParameterSource is a parameter name, the annotation also contains the Parameter specified in the ParameterSource. The pseudo-code for the generation of the blank annotations is shown in figure 7.7.

After all the blank annotations are generated, the FlowSpecifications with Sources, and Sinks are generated. A FlowSpecification specifies an EntryPoint and a set of Sources and Sinks. Each interface method is an EntryPoint to the software system because the interfaces specify how to interact with the other components. Therefore, each implementation of an interface method is at least once an EntryPoint in a FlowSpecification. The interface methods in the client analysis are not assigned to be Sources or Sinks. Therefore, it is unknown if an implemented interface method in the supplier analysis model is a Source or Sink. Thus, each combination of Sources or Sinks for the Annotations has to be tested. First, a blank

```
Input: informationFlowStereotype
    parametersAndDataPair = findParametersAndDataPair(informationFlowStereotype)
    datasets = parametersAndDataPair.dataTargets
    securityLevels = getJoanaSecurityLevelsForAversaries(datasets)

    pcmOperation = informationFlowStereotype.getAppliedTo()
    pcmInterface = pcmOperation.getInterface()
    javaMethod = getCorrespondingMethod((pcmOperation)
    javaInterface = getCorrespondingInterface(pcmInterface)
    components = findComponentsWhichImplementInterface(javaInterface)

    annotations = []
    for component : components:
        implementedMethod = getImplementedMethod(component, javaMethod)
        if parametersAndDataPair.getParameterSource() != "\return":
            implementedParameter = findParameterWithName(implementedMethod,
                parametersAndDataPair.getParameterSource())
        annotation = generateAnnotation(component, implementedMethod,
            implementedParameter, securityLevels)
        annotations.add(annotation)
```

Figure 7.7: Pseudo code for generating the annotations for applied stereotype InformationFlow

annotation is assigned to be the EntryPoint for the FlowSpecification. The Lattice and the SecurityLevels are required, which are determined before the blank annotation generation to generate an EntryPoint from a blank annotation. Second, all possibilities for a source-sink distribution are generated for the blank annotations. A source-sink distribution contains two sets of blank annotations. The first set contains all blank annotations from which Sources are generated. The second set contains all blank annotations from which Sinks are generated. A FlowSpecification is created with the same EntryPoint for each source-sink distribution. The FlowSpecifications are tagged with consecutive integers. For example, there are two blank annotations A and B, where A is assigned to be the EntryPoint. Then, all possible source-sink distributions for A and B are:

$$A : \text{Source} \qquad B : \text{Source} \tag{7.6}$$

$$A : \text{Source} \qquad B : \text{Sink} \tag{7.7}$$

$$A : \text{Sink} \qquad B : \text{Source} \tag{7.8}$$

$$A : \text{Sink} \qquad B : \text{Sink} \tag{7.9}$$

For each sink-source distribution, a FlowSpecification with the EntryPoint based on A is generated. This process is repeated with each annotation as base for the EntryPoint. The complete pseudo-code is shown in figure 7.8. If the Annotation annotates a Parameter, then this information is dropped because an EntryPoint annotates a complete Method.

```
Input: blankAnnotations, securityLevels, lattice
    flowSpecifications = []
    tag = 0
    for annotation : blankAnnotations:
        sourceDistributions = generatePowerSet({0..(annotationsLeft.length - 1)}
        for sourceDistribution : sourceDistributions:
            entryPoint = generateEntryPoint(annotation, securityLevels, Lattice, tag)
            sources = generateSources(blankAnnotations, sourceDistribution, tag) // Generate sources for all elements which index is in sourceDistribution
            sinks = generateSinks(blankAnnotations, sourceDistribution, tag) // Generate sinks for all elements which index is not in sourceDistribution
            flowSpecification = generateFlowSpecification(entryPoint, sources, sinks)
            flowSpecifications.add(flowSpecification)
            tag++
```

Figure 7.8: Generation of the flow specifications for the blank annotations

### 7.3.3 Confidentiality4CBSE-JOANA Correspondence Metamodel

Based on the underlying security metamodel transformation, a security correspondence metamodel can be created. The security metamodel transformation is defined in section 7.3.2. The implied security correspondence metamodel is depicted in figure 7.9. The security correspondence metamodel is in the project `edu.kit.kastel.scbs.pcm2java4joana.correspondencemodel`. The DataSet2SecurityLevel correspondence contains the DataSets which are converted to a list of SecurityLevels. The ParametersAndDataPair2Annotation correspondence matches the ParametersAndDataPair to the corresponding Annotation. However, because the information about InformationFlow is not needed for the back-projection, the correspondence does not contain the stereotype InformationFlow. The Confidentiality4CBSE model elements are identified by their ID. Each time a DataSet is transformed into a SecurityLevel, a list of DatatSets is transformed into a list of SecurityLevels or a ParametersAndDataPair and InformationFlow are transformed into an Annotation, the correspondence is added to the fitting security correspondence model.

## 7.4 Combining the Structural Transformation and Security Transformation

The structural model transformation is described in section 7.2 and the security model transformation is described in section 7.3. The models introduced in section 7.2 and figure 7.9 are based on ecore models. The structural model transformation produces a Java source code model. The security model transformation produces a JOANA model. These two models are combined into the supplier analysis model by generating annotated Java source code. The generation of the annotated Java source code is implemented in the class `SupplierAnalysisModel2AnnotatedCodeGenerator.xtend`. The generation of the annotated source code is implemented in Xtend [80]. Xtend is chosen for the source code generator because it offers strings with template methods that are useful for source code

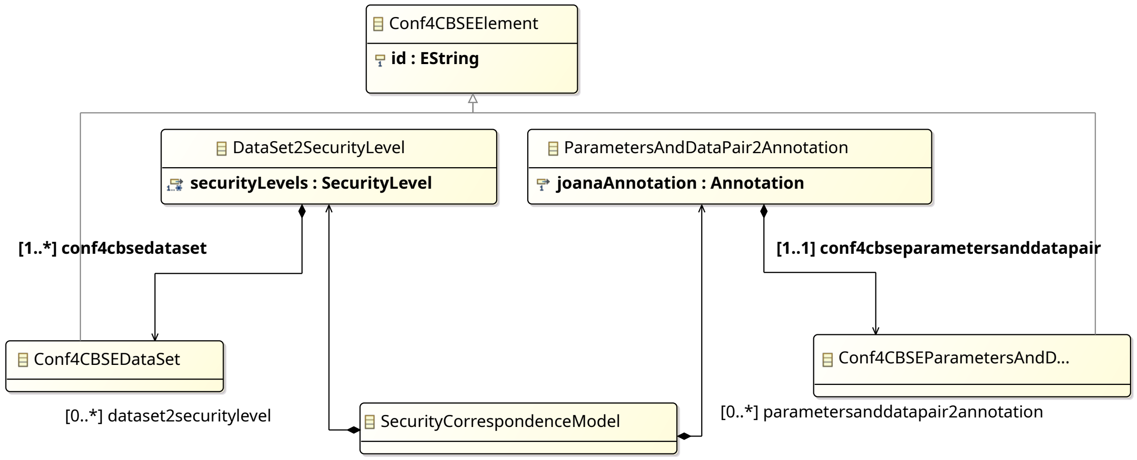

Conf4CBSEElement
id : EString
DataSet2SecurityLevel
securityLevels : SecurityLevel
ParametersAndDataPair2Annotation
joanaAnnotation : Annotation
[1..*] conf4cbsedataset
[1..1] conf4cbseparametersanddatapair
Conf4CBSEDataSet
Conf4CBSEParametersAndD...
[0..*] dataset2securitylevel
SecurityCorrespondenceModel
[0..*] parametersanddatapair2annotation

Figure 7.9: Security correspondence model between Confidentiality4CBSE and JOANA as Ecore model

generation. Template methods can be used to initiate String based on multiple model elements. Additionally, traversing Ecore models with Xtend is more compact than with Java.

The generator iterates over the TopLevelTypes of the source code model. The package of all generated TopLevelTypes is `generated.code`. All referenced TopLevelTypes are imported if they are used as a Type in the children of the TopLevelType. If a child of the TopLevelType uses a CollectionType, `java.util.Collection` is imported too. The Methods of the Interfaces are not annotated with JOANA model elements. If a Class is generated, also `edu.kit.joana.ui.annotations.*` is imported to use the JOANA annotations in the source code. Each generated Class has a constructor with which all class Variables of the Class can be set. For each class Variable getters and setters are generated. If a Class is generated from a BasicComponent, the Methods of the implemented Interfaces are added as method stumps to the Class. The method stubs do not contain any implementation because the internal behavior is unknown in the client analysis model, as introduced in section 4.1. JOANA annotations are generated for all Methods and Parameters for which a FlowSpecificationElement can be found.

## 7.5 Back-Projection of the JOANA Results

The information transfer between two security analyses has two directions. First, information is transferred from the client analysis to the supplier analysis to generate the supplier analysis model skeleton and the correspondence model. Second, information is transferred from the supplier analysis to the client analysis by executing the supplier analysis on the supplier analysis model and projecting the supplier analysis results back into the client analysis model. The generated supplier analysis model skeleton has to be implemented by fulfilling the consistency requirement described in section 5.1.2. Therefore, no new interface methods are added to a component. Then, the supplier analysis, in this case, JOANA, is run on the supplier analysis model. JOANA is run for each entry point, and all results are added to one file with the file ending ".joanaresults". If the supplier analysis detects any invalid traces, the results that are written to the file are projected back into the client analysis model. As described in section 6.4, the results are first transformed into a simplified result model. This result transformation is presented in section 7.5.1. Based on the simplified results, the adapted security level is approximated, which is described in section 7.5.2. The process of adapting the client security model is shown in section 7.5.3.

### 7.5.1 Result Transformation from the JOANA Results to the Simplified Results

A JOANA result contains more information than the needed information for the back-projection. The minimum required information in the supplier analysis is described in section 5.3.1. A trace state has to contain information about the software state and the security level of the software state. The software state is the executed action with its class in JOANA. If a parameter is annotated, the parameter index is part of the software state. A trace state also contains its position in the invalid trace to be able to determine the adapted security specification as described in section 6.4.1. Therefore, the *requirement of result structure* is fulfilled.

The simplified JOANA result model is depicted in figure 7.10 and found in the project `edu.kit.kastel.scbs.pcm2java4joana.joana`. The simplified JOANA result consists of a list of invalid traces. An invalid trace is a list of TraceStates. A TraceState contains its index in the invalid trace, its class name, the security level name, and a parameter index. A TraceState also contains a ResultMethod. The ResultMethod has a name, a return type, and parameter types.

The result transformation from the JOANA result to the simplified results is implemented in the class `JoanaResultsParser.java`. It divides the results into single results for each tag. The first entry of a JOANA result is `entry_point_method`. Therefore, the list of all results can be split at this entry. Then, the flows are extracted from every single result. The flows are in the single result entry with the key `flow`. Each flow element in the flows is transformed into a trace state. The $i$-th transformed flow element has the trace state index $i$. The class name is behind the key `class` in the JOANA trace state. The JOANA results do not have enough information to build a complete Java method as presented in section 7.2.1 because the types are not fully qualified. For example, if a method has the return type `List<String>`, the return type in the JOANA result is `java.util.List` without the InnerType. Therefore, the result metamodel has a method element in which the types are only specified by the type name. If only the method name is saved, the method cannot uniquely be identified inside the class. The method is generated with the name in the result entry `name`. Only the types of the parameters of the method are specified in the entry `parameters`. If the JOANA annotation, with which the invalid trace is detected, is at a parameter, the entry `index` specifies the index of the parameter in the parameters of the method.

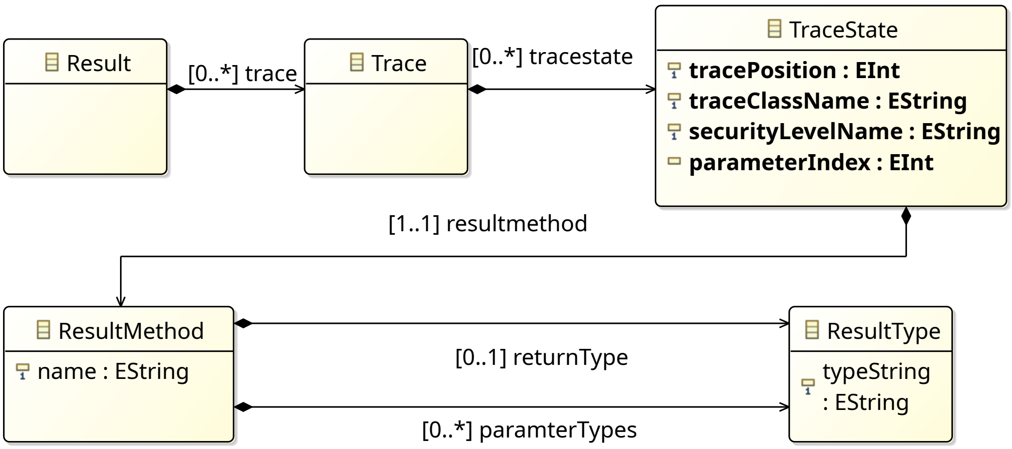


Figure 7.10: Simplified JOANA result model as an Ecore model

### 7.5.2 Determining the Adapted Security Level Based on the JOANA Results

Based on the JOANA results, the adapted security levels have to be determined to resolve the assumptions in the client analysis model. The approach to determine the adapted security level based on the JOANA results is implemented in `CorrectSecurityLevelFinder.java`.

The first step is to determine all public actions in the invalid trace. The public actions of a trace can be determined with the structural correspondence model. For each trace state in the invalid trace, it is tested whether the structural correspondence model contains a Component2Class correspondence, where the class has the name of the trace class name in the trace state. It is checked whether a method signature of the component methods matches the method signature of the trace method. Two method signatures match if both have the same name, the same return type, and the same parameter types in the same order. If there is an invalid trace without a public component method, the back-projection is aborted. The reason for this is that the supplier analysis model has security violations that are not based on the assumptions of the client analysis model.

The second step is to create security level equations for all trace states with a public component method. The pseudo-code for the security level equation creation is presented in figure 7.11. An equation consists of the own aggregated trace state as well as the predecessors and the successors, as described in section 6.4. The aggregated trace state includes a flag to identify the TraceState as a public trace state. If it is a public trace state, it also includes the Interface, which is implemented in the component. The Interface is important because it can be implemented in multiple components, for which also public trace states can exist in the JOANA results. If multiple components implement the same interface method, the same interface method in the client analysis model is adapted with multiple new security levels. Therefore, the security levels must be the same for all methods implementing the same interface method. If two security levels for the same interface method are different, one of the security levels is overwritten by the other during the back-projection. Therefore, the algorithm to determine the adapted security

```
Input: joanaResult, structuralCorrespondenceModel

equationSystem = new EquationSystem()

for flow : joanaResult.traces:
    for traceState : flow.traceStates:
        implementedInterface = getImplementedInterfaceFor(traceState.traceClassName,
            traceState.resultMethode, structuralCorrespondenceModel)
        if implementedInterface == null: // trace state does not contain a public method
            continue
        existingSecurityLevelEquation = equationSystem.getEquationForInterface(
            securityLevelEquation)
        if existingSecurityLevelEquation == null:
            securityLevelEquation = new SecurityLevelEquation()
            securityLevelEquation.owner = traceState
            securityLevelEquation.predecessors.add(getPredecessor(traceState, flow.
                traceStates)
            securityLevelEquation.successors.add(getSuccessors(traceState, flow.
                traceStates)
            equationSystem.addEquation(securityLevelEquation)
        else:
            existingSecurityLevelEquation.predecessors.add(getPredecessor(traceState,
                flow.traceStates)
            existingSecurityLevelEquation.successors.add(getSuccessors(traceState, flow.
                traceStates)
```

Figure 7.11: Creation of the Security Level Equation System

levels has to determine only one new security level for each interface method in the invalid traces. All security level equations are added to a security level equation system. In the equation system, security level equations with the same method signature and implemented interface are combined into one equation by merging their predecessors, respectively, successors. Predecessors of a trace state are states in front of the trace state in the invalid trace. Respectively, successors of a trace state are behind the trace state. As described in section 6.4.1, the direct predecessors and successors are needed to determine the adapted security level for a method.

In the last step to determine the adapted security levels, the security level equation system is solved. The solver for the equation system is implemented in the class `EquationSystem.java`. As introduced in section 7.3.2, the generated lattice is defined on the power set of all security levels. Therefore, the least-upper-bound operator is always defined because the security level that is the combination of all data sets has a relation to all other security levels. If a trace state only has internal successors, the solution is determined by applying the least-upper-bound operator to all security levels of the successors and the own security level. An internal successor is a trace state which has no public component method. If a public trace state A has a public trace state B as successor respectively predecessor, then B has A as predecessor respectively successor. Therefore, if the security level of A is set, the security level of A as successor or predecessor of B is set too. This problem is solved by using references to the public trace states in the equations. A back-

```
Input: equationSystem, securityLevels

solve(0, equationSystem.equations, securityLevels, new Map<int, SecurityLevel), -1, new
    Map<int, SecurityLevel))

function solve(index, securityLevelEquations, securityLevels, currentSolution bestScore,
     bestSolution):
    if index == securityLevelEquations.size:
        score = calculateScore(securityLevelEquations, currentSolution)
        if score > bestScore:
            bestScore = score
            bestSolution = currentSolution
        return bestScore

    for securityLevel : securityLevels:
        currentSolution[index] = securityLevel
        bestScore = solve(index + 1, securityLevelEquations, securityLevels,
            currentSolution, bestScore, bestSolution)

    return bestScore
```

Figure 7.12: Structure of the backtracking algorithm to determine the adapted security levels

tracking algorithm is used for all other equations with public trace states as neighbors. The structure of the backtracking algorithm is presented in figure 7.12. It iterates over all distributions of security levels. The number of predecessors and successors related to the security level of an equation for each distribution and equation is calculated. This number is called the score of a security level distribution. The possibility with the highest score is considered the solution. The reason for this is that the highest score implies the most resolved invalid traces. The determined security level for each equation is saved in the security level equations. As described in section 6.4.1, this is only an approximation. The determined security level for a single equation could lead to multiple new invalid traces. It is also possible that not all invalid traces are resolved. The plugin notifies the user what the score is and what the best score could be. The user can then decide to back project the determined security levels or resolve the trace himself.

### 7.5.3 Adapting the Client Security Model with the Adapted Security Levels

The client analysis model is updated with the solution of the equations system. This update-logic for the client analysis model is implemented in the class `ClientAnalysisModelUpdater.java`. The client analysis model has to be updated for each `SecurityLevelEquation`. The solution of the `EquationSystem` is saved in the `SecurityLevelEquation` as the adapted security level.

One security level can be combined from multiple DataSets as described in section 7.3.2. The DataSet2SecurityLevel correspondences for the determined security level are searched in the security correspondence model. A DataSet2SecurityLevel correspondence contains the ID of the corresponding DataSet. When the corresponding IDs are determined, the client analysis model is traversed to retrieve the data sets with the resolved ID. A method is annotated with an Annotation instance. An Annotation is generated from a ParametersAndDataPair. Therefore, the ParametersAndDataPair2Annotation correspondence for the trace state is determined by comparing the annotated class, the annotated method signature, and the annotated parameter. The client analysis model is traversed to find the ParametersAndDataPair for the ID contained in the ParametersAndDataPair2Annotation correspondence.

The client analysis model is updated by setting the DataSets of the ParametersAndDataPair to the resolved DataSets for the determined security level. The client analysis model update is repeated for each `SecurityLevelEquation`.

# 8 Evaluation

In this chapter, the feasibility of the approach to transfer information between an architecture and a source code analysis is evaluated. The initiation of the approach with JOANA and Confidentiality4CBSE, presented in chapter 7, is used for the evaluation. The evaluation questions are presented in section 8.1. The instantiated approach is applied to a model of an example scenario which is introduced in section 8.2 to answer the evaluation questions. The model is provided by a third party. The results of the evaluation are presented in section 8.3 and discussed in section 8.4. In section 8.5, the threats to validity are presented.

## 8.1 Evaluation Questions

The feasibility of the approach to transfer information between an architecture and a source code analysis is evaluated by answering the following two evaluation questions.

**(Q1)** Does the supplier analysis find the invalid traces which are intentionally injected in the supplier analysis model based on the generated supplier analysis model skeleton?

**(Q2)** Does the client analysis find the security violations which are found with supplier analysis after the back-projection of the supplier analysis results?

In this evaluation, the supplier analysis model skeleton is generated from the client analysis model. The supplier analysis model skeleton is completed with the provided implementation. The first evaluation question item (**Q1**) is answered with yes if the supplier analysis model can be compiled and the supplier analysis can find all the invalid traces through the supplier analysis model which are intentionally placed in the supplier analysis model.

The second evaluation question item (**Q2**) is answered with yes if the supplier analysis results can be projected back into the client analysis model. Furthermore, the client analysis has to find the security violations resulting from removing the assumptions in the client analysis model.

## 8.2 Evaluation Scenario

In this section, the scenario used in the evaluation is presented. In section 8.2.1, the provided client analysis model is described. The provided supplier analysis model is shown in section 8.2.2. Both the client and the supplier analysis models, are provided by a third party.

### 8.2.1 Client Analysis Model of the Scenario

The evaluation scenario is a travel planner application [88]. The scenario models a modern travel planner system and consists of the four parties User, Airline, TravelAgency, and AirlineTechnicalService which interact with the system. The model base of the scenario is the model for Access Analysis of Kramer et al. [89]. The model and the scenario are extended with the party AirlineTechnicalService and the components AirlineLogger and UserInterface. UserInterface is added because the original model of Kramer et al. uses DelegationRoles. However, this approach needs a component for the user interaction to analyze the complete program with JOANA. The used repository model is depicted in figure 8.1. The software system consists of six components. A user wants to book a hotel or flight with the TravelPlanner component. The user interacts with the application via a UserInterface. The user uses a TravelAgency as a broker for the booking. The flights are sold by an Airline. The CreditCardCenter provides credit card data that is needed for the booking. The AirlineLogger provides the possibility to log technical data that can be used by the technical staff of an airline. The communication between the components is realized via provided and required interfaces. Additionally, data types are used for bundling data. The client analysis finds 130 security violations in the unadapted client analysis model provided by a third party.

Each party should only be given the minimal needed amount of information. Only three security levels are used in the original scenario of the travel planner scenario [88]. Kramer et al. [89] use eight DataSets for a more finer granular view of the software system. The eight DataSets are:

- CreditCardInformation
- TravelData
- CreditCardInformationDeclassified
- FlightOffer
- Selection
- AirlineTechnicalInformation
- Comissioning
- UserDetails

The client security model contains the four parties User, Airline, TravelAgency, and AirlineTechnicalService as adversaries. The adversaries and the DataSets which they may know are shown in table 8.1. The stereotype InformationFlow is applied to twelve of the interface methods. InformationFlow assigns ParametersAndDataPairs to a method. The assignment is presented in table 8.2.

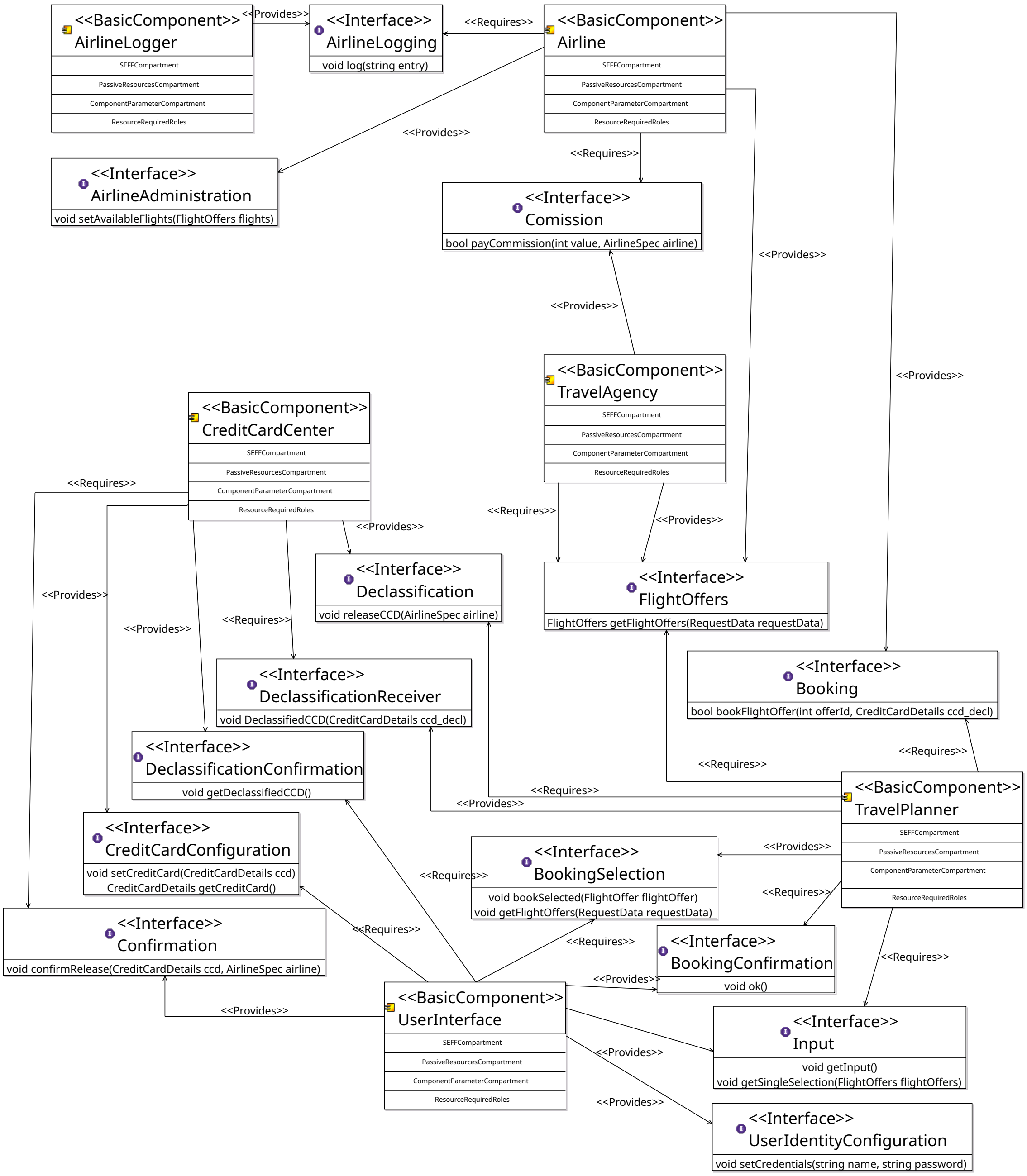


Figure 8.1: Repository model of the scenario

| Adversary | May Know Data |
|---|---|
| User | CreditCardInformation, CreditCardInformationDeclassified, FlightOffer, Selection, TravelData, Comissioning, UserDetails |
| Airline | CreditCardInformationDeclassified, FlightOffer, Selection, TravelData, Comissioning, AirlineTechnicalInformation |
| TravelAgency | FlightOffer, TravelData, Comissioning |
| AirlineTechnicalService | AirlineTechnicalInformation |

Table 8.1: Adversaries and their data sets which the adversaries may know

| Method | Data Sets |
|---|---|
| setAvailableFlights : AirlineAdministration | FlightOffer |
| getSingleSelection : Input | FlightOffer |
| getFlightOffers : BookingSelection | TravelData |
| bookSelected : BookingSelection | Selection, FlightOffer, TravelData |
| setCreditCard : CreditCardConfiguration | CreditCardInformation |
| confirmRelease : Confirmation | FlightOffer, Selection, TravelData, CreditCard-Information |
| bookFlightOffer : Booking | FlightOffer, Selection, CreditCardInformation-Declassified, TravelData |
| getFlightOffers : FlightOffers | TravelData, FlightOffer |
| releaseCCD : Declassification | FlightOffer, Selection, TravelData |
| payCommission : Comission | TravelData |
| log : AirlineLogging | AirlineTechnicalInformation |
| setCredentials : UserIdentityConfiguration | TravelData, UserDetails |

Table 8.2: Methods to which the InformationFlow stereotype is applied with the combined data sets from the ParametersAndDataiPairs of the stereotype InformationFlow. The method is part of the interface which is defined behind the colon.

s

| Number | Type | Method | Security Level |
|---|---|---|---|
| 1 | Source | UserInterface.confirmRelease | CreditCardInformation |
| 1 | Sink | Airline.bookFlightOffer | CreditCardInformationDeclassified |
| 2 | Source | Airline.bookFlightOffer | CreditCardInformationDeclassified |
| 2 | Sink | TravelAgency.payCommission | CreditCardInformation |
| 3 | Source | UserInterface.setCredentials | UserDetails, TravelData |
| 3 | Sink | AirlineLogger.log | AirlineTechnicalInformation |
| 4 | Source | TravelPlanner.getFlightOffers | TravelData |
| 4 | Sink | AirlineLogger.log | AirlineTechnicalInformation |

Table 8.3: Intentionally placed invalid information flows through the supplier analysis model. An invalid information flow is between a Source and a Sink. The method is part of the class which is defined in front of the point.

### 8.2.2 Supplier Analysis Model of the Scenario

The provided supplier analysis model contains the complete implementation of the client analysis model except for the JOANA annotations at the implemented interface methods of the components. The supplier analysis model is implemented obeying the consistency requirement introduced in section 5.1.2. Invalid information flows between the components are placed intentionally in the supplier analysis model. The invalid information flows are only known by the third party. The intentionally placed invalid information flows are shown in table 8.3. The first invalid flow is a requirement from the IFlow Travel Planner [88]. This information flow requires declassification. The second invalid flow is reported because the payment is only successful if the user has a MasterCard or VisaCard. Therefore, TravelAgency can obtain credit card information. The third invalid flow leaks user information to the technicians of Airline because Airline uses a logger for technical data also for other purposes like booking histories. The only difference between the third and fourth invalid flow is the Source of the flow and its SecurityLevel, but the reason is the same. The technical data logger is used for data other than technical data.

## 8.3 Results of the Evaluation Study

The results of the evaluation are presented in this section. The presentation is divided into three parts. First, the results of the supplier analysis model skeleton generation are described in section 8.3.1. Second, the results of the execution of the supplier analysis are presented in section 8.3.2. Third, the results of the back-projection are depicted in section 8.3.3.

### 8.3.1 Results of the Supplier Analysis Model Skeleton Generation

The supplier analysis model skeleton is generated from the provided client analysis model. The generated security levels are:

- CreditCardInformation
- TravelData
- CreditCardInformationDeclassified
- FlightOffer
- Selection
- AirlineTechnicalInformation
- Comissioning
- UserDetails

and all combinations of these (e.g. FlightOfferSelectionTravelData). The generated lattice contains all possible combinations of the security levels corresponding to the data sets of the client analysis model. The generated security lattice is based on the power set of all security levels. The combined security level of all other security levels is at the top of the lattice. Information can flow from the combined security level of all other security levels to every other security level. Information may flow from a combination of security levels to another combination of security levels if the second combination is a subset of the first combination. The generated supplier analysis model skeleton contains 2.228.224 EntryPoints. The combined size of all generated files is 18.2 GB. The complete generation process needed 20 minutes with a Laptop with i7-8550U CPU, 16 GB DDR4 RAM, and the Windows operating system. The eclipse IDE cannot open these files. The files of the component implementation have over 100.000 lines of code.

The generated supplier analysis model skeleton is completed with the provided implementation. The supplier analysis model cannot be compiled with the JOANA IFC analysis framework [90] because there is not enough RAM available to load 18.2 GB of data on the used computer. Therefore, the JOANA analysis cannot be executed. Furthermore, it would need around 2.5 days to execute the JOANA analysis sequentially for each EntryPoint assuming an average analysis run-time of 2 seconds for each EntryPoint which is the time observed during tests with other models.

An alternative generation approach is chosen to allow further evaluation. Not all source-sink distributions for each EntryPoint are generated, as described in section 7.3.2. Only one source-sink distribution is generated for each blank annotation as EntryPoint. The generated source-sink distribution contains the blank annotation for which the EntryPoint annotation is generated as Source and all other blank annotations as Sinks. This source-sink distribution is in the set of all source-sink distribution. The needed time for the generation is reduced to under a minute. The combined size of all generated files is 2.5 MB. The generated supplier analysis model skeleton is completed with the provided implementation. As a result, the supplier analysis model can be compiled.

```
function executeJoana() {
    local joanaPath=$1;
    local projectPath=$2;
    local numberEntryPoints=$3;

    for ((i = 0 ; i < $numberEntryPoints; i++)); do
        java -jar $joanaPath \"classPath $projectPath\" \"sdgOptions
            enableUninitializedFieldTypes\" \"sinks remove .*\" \"sources remove .*\" \"
            run $i\ --out=\'$i.joanaresults\'\"
    done

    cat *.joanaresults > merged.joanaresults
}
```

Figure 8.2: Bash function to automatically run JOANA for all EntryPoints

### 8.3.2 Results of the Supplier Analysis

The JOANA analysis is executed for each EntryPoint of the supplier analysis model. The JOANA IFC analysis framework does not provide the possibility to execute the analysis for all EntryPoints at once. Therefore, the bash function seen in figure 8.2 is used to execute JOANA for all EntryPoints up to the index specified as argument.

JOANA finds 114 invalid traces with many duplicates. A duplicate trace is a trace for which a second trace with the same source and sink exists. In total, 19 invalid traces remain after removing the duplicates. These are shown in table 8.4. Each invalid flow consists of a Source and Sink. Both the Source, and Sink, are specified with their class, their method name, their security level, and the implemented interface of the method. The used notation is `class.method:interface`. All methods in the sources and sinks are public component methods. Traces 7, 14, and 18 are the intentionally placed invalid results. The others are not intentionally placed.

| Number | Type | Method | Security Level |
|---|---|---|---|
| 1 | Source | UserInterface.getSingleSelection: Input | FlightOffer |
| 1 | Sink | Airline.bookFlightOffer: Booking | FlightOffer, Selection, TravelData |
| 2 | Source | UserInterface.getSingleSelection: Input | FlightOffer |
| 2 | Sink | UserInterface.confirmRelease: Confirmation | CreditCardInformation, FlightOffer, Selection, TravelData |
| 3 | Source | UserInterface.getSingleSelection: Input | FlightOffer |
| 3 | Sink | CreditCardCenter.releaseCCD: Declassification | FlightOffer, Selection, TravelData |

| Number | Type | Method | Security Level |
|---|---|---|---|
| 4 | Source | UserInterface.getSingleSelection: Input | FlightOffer |
| 4 | Sink | TravelPlanner.bookSelected: BookingSelection | FlightOffer, Selection, TravelData |
| 5 | Source | UserInterface.getSingleSelection: Input | FlightOffer |
| 5 | Sink | TravelAgency.payCommission: Comission | TravelData |
| 6 | Source | TravelAgency.getFlightOffers: FlightOffers | TravelData |
| 6 | Sink | Airline.getFlightOffers: FlightOffers | FlightOffer, TravelData |
| 7 | Source | TravelAgency.getFlightOffers: FlightOffers | TravelData |
| 7 | Sink | AirlineLogger.log: AirlineLogging | AirlineTechnicalInformation |
| 8 | Source | TravelPlanner.getFlightOffers: BookingSelection | TravelData |
| 8 | Sink | UserInterface.confirmRelease: Confirmation | CreditCardInformation |
| 9 | Source | TravelPlanner.getFlightOffers: BookingSelection | TravelData |
| 9 | Sink | UserInterface.getSingleSelection: Input | FlightOffer |
| 10 | Source | TravelPlanner.getFlightOffers: BookingSelection | TravelData |
| 10 | Sink | Airline.bookFlightOffer: Booking | CreditCardInformationDeclassified |
| 11 | Source | TravelPlanner.getFlightOffers: BookingSelection | TravelData |
| 11 | Sink | TravelPlanner.bookSelected: BookingSelection | FlightOffer, Selection, TravelData |
| 12 | Source | TravelPlanner.bookSelected: BookingSelection | FlightOffer, Selection, TravelData |
| 12 | Sink | Airline.bookFlightOffer: Booking | CreditCardInformationDeclassified |
| 13 | Source | TravelPlanner.bookSelected: BookingSelection | FlightOffer, Selection, TravelData |

| Number | Type | Method | Security Level |
|---|---|---|---|
| 13 | Sink | UserInterface.confirmRelease: Confirmation | CreditCardInformation, FlightOffer, Selection, TravelData |
| 14 | Source | UserInterface.confirmRelease: Confirmation | FlightOffer, Selection, TravelData |
| 14 | Sink | Airline.bookFlightOffer: Booking | CreditCardInformationDeclassified |
| 15 | Source | UserInterface.confirmRelease: Confirmation | FlightOffer, Selection, TravelData |
| 15 | Sink | UserInterface.confirmRelease: Confirmation | CreditCardInformation |
| 16 | Source | UserInterface.confirmRelease: Confirmation | CreditCardInformation |
| 16 | Sink | TravelAgency.payCommission: Comission | TravelData |
| 17 | Source | Airline.bookFlightOffer: Booking | FlightOffer, Selection, TravelData |
| 17 | Sink | Airline.bookFlightOffer: Booking | CreditCardInformationDeclassified |
| 18 | Source | Airline.bookFlightOffer: Booking | CreditCardInformationDeclassified |
| 18 | Sink | TravelAgency.payCommission: Comission | TravelData |
| 19 | Source | TravelPlanner.getFlightOffers: BookingSelection | TravelData |
| 19 | Sink | Airline.getFlightOffers: FlightOffers | FlightOfferTravelData |

Table 8.4: Results of the JOANA analysis. An invalid information flow always consists of a source and a sink. The format of the method of an annotation is `class.method:interface`.

### 8.3.3 Results of the Back-Projection of Supplier Analysis Results

The supplier analysis results only contain trace states with at least one public component method. Therefore, the results can be projected back into the client analysis model. A security level equation system is built based on the 19 invalid traces. The equation system consists of 9 security level equations with 38 invalid relations. The algorithm to determine the correct security levels finds an approximated solution in which 24 of 38 invalid relations are resolved. The solution is projected back into the client analysis model to adapt it. The adapted security levels are presented in table 8.5. For example, the security level of the method log is adapted to be the DataSet TravelData.

If the client analysis is run on the adapted client analysis model, it finds new security violations. The number of security violations in the client analysis results are determined by searching the prettified results for the keyword `isInSecureWithRespectTo()` as described in section 2.7.1. The client analysis finds 130 security violations on the unadapted client analysis model and 140 security violations on the adapted client analysis model. The client analysis does, for example, find the invalid access of travel data by the AirlineTechnicalService. The client analysis output changed from 7.611 to 9.629 lines.

| **Method Name** | **Old Security Level** | **New Security Level** |
|---|---|---|
| getSingleSelection : Input | FlightOffer | CreditCardInformation, TravelData, CreditCard-InformationDeclassified, FlightOffer, Selection, AirlineTechnical-Information, Comissioning, UserDetails |
| bookFlightOffer : Booking | TravelData, FlightOffer, Selection | TravelData, CreditCard-InformationDeclassified FlightOffer, Selection, AirlineTechnical-Information, Comissioning, UserDetails |
| confirmRelease : Confirmation | CreditCardInformation, TravelData, FlightOffer, Selection | CreditCardInformation, TravelData, CreditCard-InformationDeclassified, FlightOffer, Selection, AirlineTechnical-Information, Comissioning, UserDetails |
| releaseCCD : Declassification | TravelData, FlightOffer, Selection | FlightOffer |
| bookSelected : BookingSelection | TravelData, FlightOffer, Selection | CreditCardInformation, TravelData, CreditCard-InformationDeclassified, FlightOffer, Selection, AirlineTechnical-Information, Comissioning, UserDetails |
| payCommission : Comission | TravelData | FlightOffer |

| Method Name | Old Security Level | New Security Level |
|---|---|---|
| getFlightOffers : BookingSelection | TravelData | TravelData, CreditCardInformationDeclassified, FlightOffer, Selection, AirlineTechnicalInformation, Comissioning, UserDetails |
| log : AirlineLogging | AirlineTechnicalInformation | TravelData |
| getFlightOffers : FlightOffers | TravelData | CreditCardInformation, TravelData, FlightOffer, Selection, AirlineTechnicalInformation |

Table 8.5: Changed security levels of a method in the invalid information flows. The interface of a method is defined behind the colon.

## 8.4 Discussion of the Evaluation Results

The evaluation results are discussed in this section. The results concerning the first evaluation question (**Q1**) are discussed in 8.4.1. In section 8.4.2, the discussion of the results concerning the second evaluation question item (**Q2**) is presented. The threats to the validity of our conclusions are described in section 8.5.

### 8.4.1 Discussion of the First Evaluation Question

The supplier analysis model skeleton can be generated from the client analysis model. The approach to generate every source-sink distribution is not scalable because the generated file sizes of the components are between 2 GB and 5 GB. The completed supplier analysis model cannot be compiled on the used computer because the used computer does not have enough RAM to work with 18.2 GB of data at once. When the alternative generation for the supplier analysis model skeleton is used, then the supplier analysis model can be compiled. JOANA runs on the reduced supplier analysis model. If the reduced supplier analysis model can be compiled, also the complete supplier analysis model can be compiled under the requirement of a well enough equipped computer.

JOANA finds 19 invalid traces through the reduced supplier analysis model. Three of the intentionally placed invalid traces through the supplier analysis model are detected. The third intentionally placed invalid information flow is not detected. The reason for this is that JOANA creates a call tree from the used EntryPoint and searches the call tree for invalid information flows as described in section 2.7.2. JOANA is designed to use the main method as EntryPoint which initializes all fields in the supplier analysis model. If a main method is used as EntryPoint, JOANA can also find relations through the value assignment in object instances. The object instances could not be initialized if an arbitrary method is used as EntryPoint. In this approach to couple Confidentiality4CBSE and JOANA, no main method is used because it cannot be generated from the client analysis model. This is the reason why the third information flow is not found because the origin of the invalid information flow is the assignment of data to an object instance.

JOANA does not only find the discoverable intentionally placed invalid information flows but also 16 more that are not intentionally placed. The invalid trace 6 is caused by errors in understanding the functionality of JOANA annotations. The statement of a JOANA developer that annotations of a method annotate the return value of a method is wrong. In trace 6, both components implement the interface FlightOffer with its method getFlightOffers. Two ParametersAndDataPairs are applied to the method getFlightOffers. The first ParametersAndDataPair contains the DataSet TravelData and is applied to a parameter. The second contains the DataSets TravelData and FlightOffer and is applied to the return value of the method. The second ParametersAndDataPair is transformed to an annotation of the complete method. JOANA translates an annotation of the method to annotations of all parameters of the method. This contradicts our assumption that JOANA annotates the return value of the method, which is suggested by the semantics of JOANA. The consequence is, if an implementation of the interface method calls itself or another implementation of the interface method, it results in an invalid information flow from TravelData to TravelData, FlightOffer. Furthermore, traces 15 and 17 are traces between the same method in the same class. Both methods have two parameters, both with an applied ParametersAndDataPair. In both cases, information is not allowed to flow from the security level of the first parameter to the security level of the second parameter or the other way around. JOANA reports an invalid information flow if the first ParametersAndDataPair is a Source and the second ParametersAndDataPair is a Sink. If another source-sink distribution is generated, this problem would not have occurred. However, the used client analysis model does not provide any information about a source-sink distribution. The security transformation could be adapted so that the developer could specify a source-sink distribution specific to the client analysis model.

To conclude, the supplier analysis finds three out of four intentionally injected invalid information flows in the supplier analysis. The fourth intentionally injected invalid information flow cannot be discovered because the main method is not generated. Therefore, the first evaluation question item (**Q1**) can be answered with yes under the restriction that all invalid information flows can be discovered without a main method that initializes all fields. The 16 other information flows could be false positives or true positives, but the developer can filter these invalid information flows. The quality of the found information flows like traces 6, 15, and 17 depends on the used generation schema.

### 8.4.2 Discussion of the Second Evaluation Question

All found invalid information flows contain at least one public component method. Therefore, the back-projection can be executed. The back-projection finds an approximated solution for the supplier analysis run on the reduced supplier analysis model skeleton. The client analysis model can be adapted with the approximated solution. The client analysis finds new invalid flows. For example, the security level of the method `log` of `AirlineLogger` is changed from AirlineTechnicalInformation to TravelData. The log method is intended to only log technical information for the airline. The client analysis finds this invalid access. Therefore, the second evaluation question is answered with yes. However, the back-projection does not guarantee a good solution because the used algorithm does not know which of the original security levels are correct. For example, the log method does not contain the DatSet AirlineTechnicalInformation anymore which is why the information is lost. A better adaption would be, for example, the combination of the two DataSets AirlineTechnicalInformation and TravelData. The increase of the result size by 27 % is also explained this way. The approximation algorithm to find the correct security level based on the supplier analysis results sometimes overapproximate the correct security level. For example, the security level of the method bookFlightOffer is changed to the set of all possible security levels except CreditCardInformation. However, bookFlightOffer does not need the security level AirlineTechnicalInformation because there is no invalid trace between bookFlightOffer and log. The method log is the only method with the security level AirlineTechnicalInformation in the provided client analysis model. An algorithm that would produce a better solution would need more information. If the algorithm had, for example, the information that only the method log must have the security level AirlineTechnicalInformation, the solution would be improved. However, this information is not available in the client analysis model.

If the supplier analysis model is adapted with the new security levels and JOANA is run on the adapted supplier analysis model, it would find new invalid information flows with the method `log`. The newfound invalid information flow would not be the one from TravelData to AirlineTechnicalInformation but the inverted flow. However, the client analysis finds a security violation that the airline technicians can access users' travel data. Therefore, the developer can manually fix this invalid information flow and can adapt the supplier analysis model.

## 8.5 Threats to the Validity

In this section, the threats to the internal and external validity of the conclusions are discussed.

Internal validity measures how the drawn conclusions are in a relationship to the observations taken during the study. A threat to internal validity is if the conclusions are affected by an unknown factor. The instantiation of the approach itself is a threat to the internal validity of our conclusion because it is dependent on the developer. For example, our instantiation does not scale with bigger client analysis models. If only one source-sink distribution is generated, the approach scales with bigger client analysis models. Furthermore, the third party, which provides the model for the scenario, extended the models and included information flow errors. The third party knows the approach of this bachelor's thesis that could have influenced the extension of the base models of Kramer et al. [89]. Another threat to the internal validity is that the provided client analysis model already contains security violations. The already existing security violations could cause security violations in the supplier analysis model that are not intentionally injected.

External validity measures how the conclusions can be generalized to other entities. The approach is only evaluated with one instantiation and one scenario. The instantiation with Confidentiality4CBSE and JOANA shows that the approach is feasible. The feasibility of an instantiation highly depends on the concrete implementation. While arguments are given, why the approach can transfer information between two security analyses, it is not shown that the approach can work for different security models in different scenarios. A bigger case study can provide more certainty. Therefore, the external validity is also threatened.

# 9 Conclusion and Future Work

In this bachelor's thesis, an approach to transfer information between two security analyses is developed. Static security analyses and component-based software systems are considered. Furthermore, the supplier analyses are restricted to the domain of lattice-based security analyses. The client analysis model is on the architecture view with components as black boxes. The supplier analysis model is on the source code view. The client security model is based on assumptions about the internal implementation of the components. Two analyses that can exchange information with each other are called coupled. Two analyses can be coupled if the following requirements are fulfilled:

1. A structural transformation from the client structural model to the supplier structural model exists. The existence of a structural transformation implies a structural correspondence model.
2. The generated supplier analysis model skeleton is implemented consistent with the structural correspondence model. During the implementation, the developer must not add a new method to an interface of a component.
3. The two analyses are concerned with a security property. A security property can be modeled as (hyper-)trace property. The (hyper-)trace properties of the client and supplier analysis overlap each other.
4. A security transformation between the client security model and the supplier security model exists. The security transformation implies the existence of a security correspondence model.
5. The supplier analysis results contain information about the found invalid traces. An invalid trace consists of trace states. Each trace state has information about the executed method and the security classes of the software state. Each invalid trace has at least one trace state with a public component method.

If these requirements are fulfilled, the two analyses can be coupled. Two analyses are coupled by combining the client analysis model, the supplier analysis model, and the supplier analysis results into a megamodel. The client and the supplier structural model are connected by a structural correspondence model. Furthermore, the client and the supplier security model are connected by a security correspondence model. The two correspondence models are created by generating the supplier analysis model skeleton from the client analysis model. The supplier analysis model is connected to the supplier analysis results by executing the supplier analysis on the supplier analysis model. The supplier analysis results are used to verify the assumptions in the client analysis model. If the supplier analysis does not find any invalid traces, then the assumptions in the client analysis model are correct. If the supplier analysis finds any invalid traces, then these traces are projected back into the client analysis model. Based on the detected invalid traces, the correct security levels of the public actions in the results are approximate. The client analysis model is updated with the determined security levels by using the structural and security correspondence model.

The approach is evaluated by instantiating the approach in a case study with Access Analysis and JOANA. The evaluation shows the feasibility of the approach. The evaluation shows that the instantiation of the approach with Confidentiality4CBSE and JOANA is not scalable. However, the supplier analysis finds three of four intentionally injected invalid information flows through the supplier analysis model. The fourth invalid trace cannot be discovered because no main method is generated from the client analysis model. The supplier analysis finds 16 invalid information more than intentionally injected. The back-projection of the invalid traces is successful because the client analysis model is adapted. The client analysis finds new security violations on the adapted client analysis model. The internal validity and the external validity of the evaluation are threatened. The internal validity is threatened because the feasibility of this approach is dependent on the instantiation of the approach with two concrete security analyses. The external validity is threatened because the approach is only evaluated in one instantiation with one scenario.

Due to the limitations of a bachelor's thesis, not all possible options and considerations could be inspected and analyzed for this approach. Therefore, the possible future work is presented in the following.

Only static, lattice-based security analyses are considered for the supplier analysis. The approach could be extended to cover different static security analyses. An algorithm for the determination of the correct security class has to be developed for other security properties to support more static security analyses. Furthermore, only the architecture and the source code view are considered. The approach could be extended to consider automatic proof systems like KIV [91]. If more domains of security analysis on more views on the software system are used, more assumptions of the client analysis model can be verified.

The feasibility is evaluated in one instantiation with one scenario. The approach should be instantiated with more security analyses, e.g. UMLSec [31] or Paragon [92]. If the feasibility of other instantiations is evaluated, the external validity increases. The instantiation with Confidentiality4CBSE and JOANA could also be evaluated in other scenarios that would increase the external validity. Furthermore, one could also evaluate the practicability. The practicability of this approach can be measured by comparing the effort between a manual and an automatic approach.

The generation of the JOANA security model can be improved by using the JOANA CLI commands instead of the annotations because then the return type of a method can be annotated. Furthermore, the generation schema of the source-sink distributions could be improved. An improved source-sink distribution generation would increase the quality of the supplier analysis results. It could also make the instantiation with Confidentiality4CBSE and JOANA more scalable. The back-projection algorithm of this instantiation could be improved to give the developer the possibility to provide additional information. Additional information could be the security levels of the client analysis model that should stay the same, be increased, or decreased. An improved back-projection algorithm would improve the quality of the adapted client analysis model.

# Bibliography

[1] B. Selic. “Model-driven development: its essence and opportunities”. In: *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’06)*. Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’06). ISSN: 2375-5261. Apr. 2006, 7 pp.–-. DOI: 10.1109/ISORC.2006.54.

[2] Herbert Stachowiak. *General model theory*. Springer, Wien, New York, 1973.

[3] B Selic. “The pragmatics of model-driven development”. In: *IEEE SOFTWARE* 20.5 (2003), pp. 19–25.

[4] Franck Fleurey et al. “A Generic Approach for Automatic Model Composition”. In: *Models in Software Engineering*. Ed. by Holger Giese. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 7–15. ISBN: 978-3-540-69073-3.

[5] Greg Brunet et al. “A manifesto for model merging”. In: *Proceedings of the 2006 international workshop on Global integrated model management - GaMMa ’06*. the 2006 international workshop. Shanghai, China: ACM Press, 2006, p. 5. ISBN: 978-1-59593-410-9. DOI: 10.1145/1138304.1138307.

[6] Mehrdad Sabetzadeh et al. “Consistency Checking of Conceptual Models via Model Merging”. In: *15th IEEE International Requirements Engineering Conference (RE 2007)*. 15th IEEE International Requirements Engineering Conference (RE 2007). ISSN: 2332-6441. Oct. 2007, pp. 221–230. DOI: 10.1109/RE.2007.18.

[7] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. “Model-driven development of model transformations”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2011, pp. 47–61.

[8] Tom Mens and Pieter Van Gorp. “A Taxonomy of Model Transformation”. In: *Electronic Notes in Theoretical Computer Science* 152 (Mar. 2006), pp. 125–142. ISSN: 15710661. DOI: 10.1016/j.entcs.2005.10.021.

[9] Artur Boronat. “Exogenous model merging by means of model management operators”. In: *Electronic Communications of the EASST* 3 (2007).

[10] Soichiro Hidaka et al. “A compositional approach to bidirectional model transformation”. In: *2009 31st International Conference on Software Engineering - Companion Volume*. 2009 31st International Conference on Software Engineering - Companion Volume. May 2009, pp. 235–238. DOI: 10.1109/ICSE-COMPANION.2009.5070990.

[11] Hartmut Ehrig et al. “Information preserving bidirectional model transformations”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2007, pp. 72–86.

[12] Th Reiter et al. “Model integration through mega operations”. In: *Workshop on Model-driven Web Engineering*. Vol. 20. 2005.

[13] Ekkart Kindler and Robert Wagner. *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios.* Technical Report tr-ri-07-284, University of Paderborn, June 2007, p. 80.

[14] Andy Schürr and Felix Klar. “15 Years of Triple Graph Grammars”. In: *Graph Transformations.* Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 411–425. ISBN: 978-3-540-87405-8.

[15] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[16] Steffen Becker, Jens Happe, and Heiko Koziolek. “Putting Components into Context: Supporting QoS-Predictions with an explicit Context Model”. In: *Proc. 11th International Workshop on Component Oriented Programming (WCOP'06)* (June 2006), pp. 1–6.

[17] Ivica Crnkovic. “Component-based software engineering ? new challenges in software development”. In: *Software Focus* 2.4 (2001), pp. 127–133. ISSN: 1529-7942, 1529-7950. DOI: `10.1002/swf.45`.

[18] Florian Heidenreich et al. “Closing the Gap between Modelling and Java”. In: *Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383. ISBN: 978-3-642-12107-4.

[19] Software Design, Karlsruhe Institute of Technology (kit) Quality Group, and Fzi Forschungszentrum Informatik. *Palladio Software Architecture Simulator: Home.* [Online; accessed 27. Sep. 2021]. Sept. 2021. URL: `https://www.palladio-simulator.com/home`.

[20] Steffen Becker. “The palladio component model”. In: *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering - WOSP/SIPEW '10.* the first joint WOSP/SIPEW international conference. San Jose, California, USA: ACM Press, 2010, p. 257. ISBN: 978-1-60558-563-5. DOI: `10.1145/1712605.1712651`.

[21] Edmund Clarke et al. “Counterexample-guided abstraction refinement”. In: *International Conference on Computer Aided Verification.* Springer. 2000, pp. 154–169.

[22] Amir Pnueli. “In transition from global to modular temporal reasoning about programs”. In: *Logics and models of concurrent systems.* Springer, 1985, pp. 123–144.

[23] E.M. Clarke, D.E. Long, and K.L. McMillan. “Compositional model checking”. In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science.* [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. Pacific Grove, CA, USA: IEEE Comput. Soc. Press, 1989, pp. 353–362. ISBN: 978-0-8186-1954-0. DOI: `10.1109/LICS.1989.39190`.

[24] Sagar Chaki et al. “Automated assume-guarantee reasoning for simulation conformance”. In: *International Conference on Computer Aided Verification.* Springer. 2005, pp. 534–547.

[25] Darren Cofer et al. “Compositional Verification of Architectural Models”. In: *NASA Formal Methods*. Ed. by Alwyn E. Goodloe and Suzette Person. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–140. ISBN: 978-3-642-28891-3.

[26] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (Sept. 20, 2010). Ed. by Andrei Sabelfeld, pp. 1157–1210. ISSN: 18758924, 0926227X. DOI: 10.3233/JCS-2009-0393.

[27] R. Focardi and R. Gorrieri. “A taxonomy of trace-based security properties for CCS”. In: *Proceedings The Computer Security Foundations Workshop VII*. The Computer Security Foundations Workshop VII. Franconia, NH, USA: IEEE Comput. Soc. Press, 1994, pp. 126–136. ISBN: 978-0-8186-6230-0. DOI: 10.1109/CSFW.1994.315941.

[28] Dorothy E. Denning. “A lattice model of secure information flow”. In: *Communications of the ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360051.360056.

[29] R. S. Sandhu. “Lattice-based access control models”. In: *Computer* 26.11 (Nov. 1993). Conference Name: Computer, pp. 9–19. ISSN: 1558-0814. DOI: 10.1109/2.241422.

[30] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. “A lattice-based approach to mashup security”. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’10*. the 5th ACM Symposium. Beijing, China: ACM Press, 2010, p. 15. ISBN: 978-1-60558-936-7. DOI: 10.1145/1755688.1755691.

[31] Jan Jürjens. “UMLsec: Extending UML for Secure Systems Development”. In: *«UML»002 — The Unified Modeling Language*. Ed. by Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 412–425. ISBN: 978-3-540-45800-5.

[32] Wolfgang Ahrendt et al., eds. *Deductive Software Verification - The KeY Book: From Theory to Practice*. 1st ed. 2016. Programming and Software Engineering 10001. Cham: Springer International Publishing : Imprint: Springer, 2016. 1 p. ISBN: 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6.

[33] Christian Hammer and Gregor Snelting. “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs”. In: *International Journal of Information Security* 8.6 (Dec. 2009), pp. 399–422. ISSN: 1615-5262, 1615-5270. DOI: 10.1007/s10207-009-0086-1.

[34] B. Chess and G. McGraw. “Static analysis for security”. In: *IEEE Security Privacy* 2.6 (Nov. 2004). Conference Name: IEEE Security Privacy, pp. 76–79. ISSN: 1558-4046. DOI: 10.1109/MSP.2004.111.

[35] Benedikt Schmidt et al. “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”. In: *2012 IEEE 25th Computer Security Foundations Symposium*. 2012 IEEE 25th Computer Security Foundations Symposium. ISSN: 2377-5459. June 2012, pp. 78–94. DOI: 10.1109/CSF.2012.25.

[36] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. "You assume, we guarantee: Methodology and case studies". In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1427. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 440–451. DOI: `10.1007/BFb0028765`.

[37] Omer Tripp et al. "ALETHEIA: Improving the Usability of Static Security Analysis". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS'14: 2014 ACM SIGSAC Conference on Computer and Communications Security. Scottsdale Arizona USA: ACM, Nov. 3, 2014, pp. 762–774. ISBN: 978-1-4503-2957-6. DOI: `10.1145/2660267.2660339`.

[38] Max E Kramer, Martin Hecker, and Simon Greiner. *Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems*. TechnicalReport 2017,12. Karlsruhe: Department of Informatics, Karlsruhe Institute of Technology, Dec. 2017, p. 34.

[39] Gregor Snelting et al. "Checking probabilistic noninterference using JOANA". In: *it - Information Technology* 56.6 (Dec. 28, 2014), pp. 280–287. ISSN: 1611-2776, 2196-7032. DOI: `10.1515/itit-2014-1051`.

[40] Elisavet Kozyri et al. "JRIF: Reactive Information Flow Control for Java". In: *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*. Ed. by Joshua D. Guttman et al. Cham: Springer International Publishing, 2019, pp. 70–88. ISBN: 978-3-030-19052-1. DOI: `10.1007/978-3-030-19052-1_7`. URL: `https://doi.org/10.1007/978-3-030-19052-1_7`.

[41] Kuzman Katkalov et al. *Model-Driven Code Generation forInformation Flow Secure Systems withIFlow*. TechnicalReport 2012-04. Augsburg: Fakultät für Angewandte Informatik der Universität Augsburg, 2012, p. 33.

[42] Kastel-Scbs. *Confidentiality4CBSE*. [Online; accessed 25. Aug. 2021]. Aug. 2021. URL: `https://github.com/KASTEL-SCBS/Confidentiality4CBSE`.

[43] Kateryna Yurchenko et al. "Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems". In: *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, September 17-22, 2017. Ed.: L. Burgueño*. Vol. 2019. CEUR Workshop Proceedings. ISSN: 1613-0073. CEUR, 2017, pp. 321–323.

[44] Kuzman Katkalov et al. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *2013 International Conference on Social Computing*. 2013 International Conference on Social Computing (SocialCom). Alexandria, VA, USA: IEEE, Sept. 2013, pp. 51–56. ISBN: 978-0-7695-5137-1. DOI: `10.1109/SocialCom.2013.14`.

[45] Kuzman Katkalov et al. "Evaluation of Jif and Joana as Information Flow Analyzers in a Model-Driven Approach". In: *Data Privacy Management and Autonomous Spontaneous Security*. Ed. by Roberto Di Pietro et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 174–186. ISBN: 978-3-642-35890-6.

[46] Peter Fischer et al. *Formal Verification of Information Flow Secure Systems with IFlow.* TechnicalReport 2012-05. Augsburg: Fakultät für Angewandte Informatik der Universität Augsburg, 2012.

[47] Kuzman Katkalov. "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme". PhD thesis. Augsburg: Universität Augsburg, 2017. 223 pp.

[48] Robert Heinrich et al. "An Architectural Model-Based Approach to Quality-Aware DevOps in Cloud Applicationsc * *This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution and the MWK (Ministry of Science, Research and the Arts Baden-Württemberg) in the funding line Research Seed Capital (RiSC)." In: *Software Architecture for Big Data and the Cloud.* Elsevier, 2017, pp. 69–89. ISBN: 978-0-12-805467-3. DOI: 10.1016/B978-0-12-805467-3.00005-3.

[49] Robert Heinrich. "Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications". In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (Feb. 25, 2016), pp. 13–22. ISSN: 0163-5999. DOI: 10.1145/2897356.2897359.

[50] Robert Heinrich et al. "Integrating Run-Time Observations and Design Component Models for Cloud System Analysis". In: *CEUR* (2014), p. 6.

[51] Heiko Klare et al. "Enabling consistency in view-based system development — The Vitruvius approach". In: *Journal of Systems and Software* 171 (Jan. 2021), p. 110815. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110815.

[52] Marco Konersmann. "Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code". PhD thesis. Essen: Universität Duisburg-Essen, May 9, 2018. 322 pp.

[53] Max E. Kramer et al. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE '15. New York, NY, USA: Association for Computing Machinery, May 4, 2015, pp. 21–26. ISBN: 978-1-4503-3471-6. DOI: 10.1145/2737166.2737177.

[54] Sebastian Hahner. "Domain-specific Language for Data-driven Design Time Analyses and Result Mappings for Logic Programs". Masterthesis. Karlsruhe: Karlsruher Institut für Technologie (KIT), Aug. 17, 2020. 123 pp.

[55] Sebastian Hahner et al. "Modeling Data Flow Constraints for Design-Time Confidentiality Analyses". In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C).* 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C). Stuttgart, Germany: IEEE, Mar. 2021, pp. 15–21. ISBN: 978-1-66543-910-7. DOI: 10.1109/ICSA-C52384.2021.00009.

[56] Roel Wuyts and Stéphane Ducasse. "Composition Languages for Black-Box Components". In: *First OOPSLA Workshop on Language Mechanisms for Programming Software Components* (2001), p. 4.

[57] Jin Shao et al. “A Runtime Model Based Monitoring Approach for Cloud”. In: *2010 IEEE 3rd International Conference on Cloud Computing*. 2010 IEEE 3rd International Conference on Cloud Computing. ISSN: 2159-6190. July 2010, pp. 313–320. DOI: 10.1109/CLOUD.2010.31.

[58] Anantha Narayanan and Gabor Karsai. “Verifying Model Transformations by Structural Correspondence”. In: *Electronic Communication European Association Software Scientific Technology* 10 (2008), p. 15.

[59] El Hadji Bassirou Toure et al. “Megamodel-based Management of Dynamic Tool Integration in Complex Software Systems”. In: *Position Papers of the 2016 Federated Conference on Computer Science and Information Systems*. 2016 Federated Conference on Computer Science and Information Systems. Vol. 9. Annals of Computer Science and Information Systems. Gdansk, Poland, Oct. 2, 2016, pp. 211–218. DOI: 10.15439/2016F585.

[60] Andrew C. Myers. “JFlow: practical mostly-static information flow control”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’99. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1999, pp. 228–241. ISBN: 978-1-58113-095-9. DOI: 10.1145/292540.292561.

[61] 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 (Apr. 1, 2021), pp. 775–800. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2019.2903797.

[62] István Nágy. “On the design of aspect-oriented composition models for software evolution”. ISBN: 9789036523684 OCLC: 71728964. PhD thesis. S.l.: s.n.], 2006.

[63] *security-code-scan/security-code-scan*. original-date: 2017-12-31T09:38:54Z. July 8, 2021. URL: https://github.com/security-code-scan/security-code-scan (visited on 07/08/2021).

[64] Manny Rayner et al. *OMG Unified Modeling Language*. Version 2.5. Jan. 3, 2005.

[65] Donald Bell. *UML basics: The component diagram*. Apr. 15, 2004.

[66] Donald Bell. *UML basics: The sequence diagram*. Feb. 16, 2004.

[67] Thomas Colcombet and Pascal Fradet. “Enforcing trace properties by program transformation”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’00*. the 27th ACM SIGPLAN-SIGACT symposium. Boston, MA, USA: ACM Press, 2000, pp. 54–66. ISBN: 978-1-58113-125-3. DOI: 10.1145/325694.325703.

[68] Jan Jurjens. “Security Analysis of Crypto-based Java Programs using Automated Theorem Provers”. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06). ISSN: 1938-4300. Sept. 2006, pp. 167–176. DOI: 10.1109/ASE.2006.60.

[69] Tachio Terauchi. "A Type System for Observational Determinism". In: *2008 21st IEEE Computer Security Foundations Symposium*. 2008 21st IEEE Computer Security Foundations Symposium. ISSN: 2377-5459. June 2008, pp. 287–300. DOI: 10.1109/CSF.2008.9.

[70] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. 1982 IEEE Symposium on Security and Privacy. ISSN: 1540-7993. Apr. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[71] Daryl McCullough. "Specifications for Multi-Level Security and a Hook-Up". In: *1987 IEEE Symposium on Security and Privacy*. 1987 IEEE Symposium on Security and Privacy. ISSN: 1540-7993. Apr. 1987, pp. 161–161. DOI: 10.1109/SP.1987.10009.

[72] A. Zakinthinos and E.S. Lee. "A general theory of security properties". In: *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*. Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097). ISSN: 1081-6011. May 1997, pp. 94–102. DOI: 10.1109/SECPRI.1997.601322.

[73] Gabriele Link. "Vorlesung 3: Logik und Mengenlehre". In: *Lineare Algebra 1, KIT* (Oct. 28, 2013), p. 32.

[74] Alexander Konigs. *Model Transformation with Triple Graph Grammars*. TechnicalReport 2005-09. Darmstadt: Real-Time Systems Lab University of Technology Darmstadt, Sept. 23, 2005, p. 16.

[75] Tihamér Levendovszky et al. "A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS". In: *Electronic Notes in Theoretical Computer Science* 127.1 (Mar. 2005), pp. 65–75. ISSN: 15710661. DOI: 10.1016/j.entcs.2004.12.040.

[76] Jean Bézivin, Frédéric Jouault, and Jean Paliès. "Towards model transformation design patterns". In: *Proceedings of the First European Workshop on Model Transformations (EWMT 2005)*. 2005.

[77] Amine Benelallam et al. "Distributed model-to-model transformation with ATL on MapReduce". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. SLE '15: Software Language Engineering. Pittsburgh PA USA: ACM, Oct. 26, 2015, pp. 37–48. ISBN: 978-1-4503-3686-4. DOI: 10.1145/2814251.2814258.

[78] Max Emanuel Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284.

[79] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (Jan. 2009), pp. 3–22. ISSN: 01641212. DOI: 10.1016/j.jss.2008.03.066.

[80] Sven Efftinge and Miro Spoenemann. *Xtend - Documentation*. [Online; accessed 9. Aug. 2021]. Nov. 2020. URL: https://www.eclipse.org/xtend/documentation/index.html.

[81] *Plugin Implementation Project.* [Online; accessed 28. Sep. 2021]. Sept. 2021. URL: https://git.scc.kit.edu/i43/stud/abschlussarbeiten/bachelorarbeiten/johanneshaering/-/tree/master/PCM2Java4JOANA/bundles/edu.kit.kastel.sbcs.pcm2java4joana.

[82] *org.eclipse.emf.ecore (EMF Documentation).* [Online; accessed 25. Aug. 2021]. Jan. 2015. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html.

[83] *Source Code Model Project.* [Online; accessed 28. Sep. 2021]. Sept. 2021. URL: https://git.scc.kit.edu/i43/stud/abschlussarbeiten/bachelorarbeiten/johanneshaering/-/tree/master/PCM2Java4JOANA/bundles/edu.kit.kastel.scbs.pcm2java4joana.sourcecode.

[84] *Correspondence Model Implementation Project.* [Online; accessed 28. Sep. 2021]. Sept. 2021. URL: https://git.scc.kit.edu/i43/stud/abschlussarbeiten/bachelorarbeiten/johanneshaering/-/tree/master/PCM2Java4JOANA/bundles/edu.kit.kastel.scbs.pcm2java4joana.correspondencemodel.

[85] *Models concerning JOANA Project.* [Online; accessed 28. Sep. 2021]. Sept. 2021. URL: https://git.scc.kit.edu/i43/stud/abschlussarbeiten/bachelorarbeiten/johanneshaering/-/tree/master/PCM2Java4JOANA/bundles/edu.kit.kastel.scbs.pcm2java4joana.joana.

[86] *Facade Design Pattern in Java | Baeldung.* [Online; accessed 25. Aug. 2021]. Dec. 2019. URL: https://www.baeldung.com/java-facade-pattern.

[87] *Interfaces (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance).* [Online; accessed 25. Aug. 2021]. July 2021. URL: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html.

[88] *Modeling the Travel Planner Application with IFlow.* [Online; accessed 3. Sep. 2021]. May 2018. URL: https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html.

[89] Kastel-Scbs. *Examples4SCBS.* [Online; accessed 10. Sep. 2021]. Sept. 2021. URL: https://github.com/KASTEL-SCBS/Examples4SCBS/tree/master/bundles/edu.kit.kastel.scbs.iflowexample.

[90] joana-team. *joana.* [Online; accessed 3. Sep. 2021]. Sept. 2021. URL: https://github.com/joana-team/joana.

[91] D. Fensel and A. Schnogge. "Using KIV to specify and verify architectures of knowledge-based systems". In: *Proceedings 12th IEEE International Conference Automated Software Engineering.* 1997, pp. 71–80. DOI: 10.1109/ASE.1997.632826.

[92] Niklas Broberg, Bart van Delft, and David Sands. "Paragon for Practical Programming with Information-Flow Control". In: *Programming Languages and Systems.* Ed. by Chung-chieh Shan. Red. by David Hutchison et al. Vol. 8301. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2013, pp. 217–232. DOI: 10.1007/978-3-319-03542-0_16.