Consistency Management for Security Annotations for Continuous Verification

Frederik Reiche
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
frederik.reiche@kit.edu

Thomas Weber
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
thomas.weber@kit.edu

Simeon Becker
Karlsruhe Institute of Technology
Karlsruhe, Germany
simeon.becker@student.kit.edu

Sebastian Weber
FZI Research Center for Information Technology
Karlsruhe, Germany
sebastian.weber@fzi.de

Robert Heinrich
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

Erik Burger
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
erik.burger@kit.edu

ABSTRACT

Analyses on the architecture of systems can yield valuable insights into a system even before it is built. The applicability of the results of these design time analyses to the system requires the system to be built according to its specification, i.e., to not violate constraints defined on the architecture. The conformance of the results of static code analyses and design time analyses ensures the system is built according to its specification. The first step for conforming results of these analyses is to ensure that the system and its specification is represented consistently in the input of the design time analysis and static code analysis, i.e., they comprise corresponding system elements and specifications for them. To achieve conforming inputs, we used consistency specifications between architecture and code models and implemented them between annotation models that enrich the architecture description with security annotations on the architecture level, as well as security annotations on the code level. This allows the continuous conformance checking during implementation and later during evolution of the system. We implemented the consistency specifications in the Vitruvius framework for an ADL and Java and tested it on case studies.

CCS CONCEPTS

• **Computing methodologies → Model verification and validation**; *Simulation environments*; • **Software and its engineering** → *Development frameworks and environments*; **Software maintenance tools**.

KEYWORDS

Model-based Analysis, Consistency Management, Development Framework, Early and Continuous Verification

ACM Reference Format:
Frederik Reiche, Thomas Weber, Simeon Becker, Sebastian Weber, Robert Heinrich, and Erik Burger. 2024. Consistency Management for Security Annotations for Continuous Verification. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3687821

1 INTRODUCTION

The security of software-intensive systems is of great importance, especially because the systems may handle personal or otherwise confidential data [18]. The development of secure software is guided by the Security Development Lifecycle (SDL) [17, 20]. The first step in the SSDL is to gather, among others, security requirements for the system to ensure, for instance, the confidentiality or integrity of the data it processes. From all requirements, a software architecture is envisioned to describe and connect the building blocks of the system, i.e., its components [22]. In Figure 1, the architecture is illustrated as a rounded rectangle on the left. To ensure the system realizes security requirements, they are documented, linked to architectural elements, and checked throughout the development process [2]. These security requirements are used to specify requirements for the system behavior. To link the security requirements, they can be annotated to, e.g., the services a component provides that have to fulfill them. Thus, the security requirements are also linked to the implementing components, through their link to the architecture. This annotation is illustrated in the middle of Figure 1. Static analyses are used to check the fulfillment of security requirements, e.g., the absence of illicit data flows, illustrated on the right of Figure 1 with system state three.

If the analyses conclude, e.g., that there are no illicit data flows, the systems' security fulfills the security requirements [21]. This statement contains a very relevant but hidden assumption: the system is implemented respecting the security requirements [9]. This assumption has to be checked against the implementation, e.g., by applying static source code analyses [15], illustrated in the right of Figure 2.

The static source code analyses need more information than just the code, illustrated in system state 2 in Figure 2, e.g., to decide if a data flow is illicit. This information is partly present in the code annotations, which are related to the architecture security

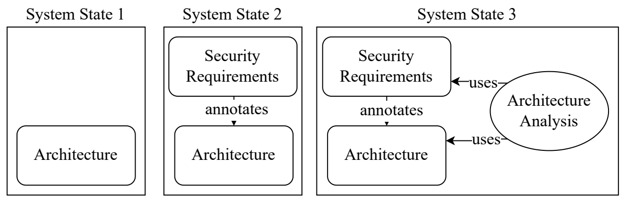


Figure 1: Overview over the development steps from an architectural perspective

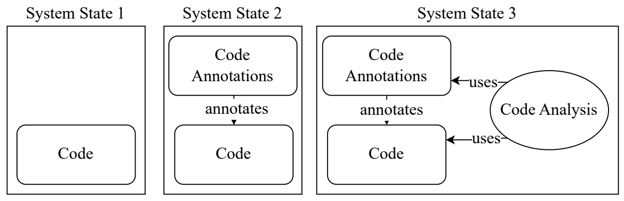


Figure 2: Overview over the development steps from the source code perspective

requirements, represented as architectural annotations. This information from the architecture can be propagated to the source code. If this propagation is not possible, the security requirements cannot be checked on the source code level, because they have no correspondence to the source code. For example, because the security requirements are related to external services used, and their validation requires an audit of the external company, which is not possible with static source code analysis.

During the development of the system and during its evolution, the system and its architecture change. To be able to handle that change, we view the propagation from the architecture annotations to the source code annotations as a bidirectional consistency problem. Changes to architecture annotations should be reflected in the code and the other way around. The need for bidirectionality stems from the different roles evolving the system. An architect may add further security requirements, due to, e.g., a change in legal obligations, while a developer may change method calls and thus produces new or changed information flows. Keeping the inputs of the analyses used consistent, requires both architecture and code to be consistent. This is achieved through the bidirectional consistency relation, which enables incremental verification throughout model evolution. Keeping this consistency relation unidirectional would reduce its applicability, because then either the architecture or the code could be adapted and kept consistent, and not both.

We derive two challenges for our approach. The first challenge, **Ch1: Provision of Specifications**, is in the domain of analysis and verification of source code and lies in the difficulty to obtain the specifications for source code in form of annotations [30]. We address this challenge by using consistency preservation and providing the architectural models and their annotations as initial knowledge source for specifications of security-relevant properties for a source code analysis that reflect security requirements. For this purpose, a domain expert must specify how annotations in the architectural model can be related to annotations used by the source code analysis.

The second challenge, **Ch2: Evolution of Specifications**, is the change of expected security properties in the evolution of the system. The changes may be due to, e.g., different legal obligations through new laws that affect the security requirements of the architecture, or bugs in libraries, which makes the code no longer comply with security standards. In cases the specifications are adapted to capture the expected security properties on either level, this change must be reflected in the other artifacts as well. Otherwise, statements on the views of the same system do not conform anymore. We address this challenge with the use of consistency preservation, containing rules for the modification or addition of new elements of the architectural models or code or their annotations. We improve the state of the art by using consistency preservation, instead of traceability techniques. This allows the automatic consistency preservation of the architecture and code models as well as their annotation models, which improves the evolution process. Additionally, we look not only on artifacts but also on specifications for them.

We apply the concept of consistency preservation to a model quadruple, illustrated in Figure 3, consisting of an architecture model, an annotation model of a static architecture analysis, as well as a source code model, and an annotation model of a static source code analysis. Conceptually, the chosen analyses are arbitrary, but the consistency preservation rules have to use the appropriate specifications. The consistency preservation between code and architecture and code annotations and architecture annotations are bidirectional, as changes in both should be reflected in the other. In contrast, the consistency preservation between code and its annotations and architecture and its annotations is unidirectional. We chose to keep this consistency preservation unidirectional because changes in the annotation model are not supposed to have an influence on the underlying model, e.g., the deletion of a security requirement should not result in the deletion of the system component that had implemented it. There are use cases, where a deletion of an annotation should result in the deletion of an architectural element, e.g., if the encryption of a component is removed, but all components of a system have to be encrypted. In this case, the consistency preservation would delete the component, the code annotation and the code parts of the component, to keep the models consistent. Four our use case, unidirectional consistency preservation between annoations and architecture or code is sufficient. We implemented the consistency preservation in the consistency preservation framework Vitruvius [13] in order to discuss the feasibility of the approach.

The remainder of this paper is structured as follows. First, we discuss the background of this paper, i.e., consistency management and static analysis in section 2. Afterward, we introduce a running example to illustrate the annotations and the consistency relations in section 3. We elaborate on our general approach in section 4 and discuss our empirical evaluation in section 5. After the discussion of the threats to validity in section 6, we present related work in section 7 and finish this paper with concluding remarks in section 8.

2 BACKGROUND

Our approach combines static analyses with consistency management. We thus first introduce the general idea of consistency management and describe the framework we used to implement the consistency rules. Afterward, we explain static analyses and the analyses we used in our prototype.

2.1 Consistency Management

The development of complex systems involves several artifacts of different types that are created by developers to describe aspects of the system. The artifacts are often models, and for the remainder of the paper we will refer to them as models. As the models contain information about the same system and the developers do not work independent of each other, the models share certain parts. These shared parts may become inconsistent, if one model is changed and the other is not kept consistent [13]. To build a correct system, these inconsistencies need to be avoided, as consistency is a heuristic for correctness [6].

Consistency management deals with managing the inconsistencies, e.g., by avoiding or fixing them. One consistency management approach is consistency preservation, where a consistent system is changed by a list of changes. This list is the input for the consistency preservation, which computes consequential changes, that make the system consistent again, i.e., fixing inconsistencies the initial changes caused. Both the initial and the consequential changes are then applied together and transform the system into a new, consistent state [13].

When two models are consistent with each other depends on the models and the information they can represent, defined by their metamodels [13]. Each metamodel pair requires its own definition of consistency, which can be achieved with the help of *consistency rules*. These consistency rules specify what must happen in a model when a change is made in another model. Consistency rules are unidirectional, i.e., consistency rules must be defined for both directions between two models.

Vitruvius. We use the consistency management Vitruvius [13], which defines a *virtual single underlying model* (V-SUM) as a state-of-the-art tool for consistency management through consistency preservation. A V-SUM contains the models describing the system as well as consistency specifications as *consistency preservation rules* (CPRs) [4, 19]. The CPRs are formulated using the *Reactions* language [13, 14]. The concept itself does not depend on any specific consistency management tool, but on the existing consistency preservation rules between architecture and code models. These consistency preservation rules are available for Vitruvius, thus we used it to implement our prototype.

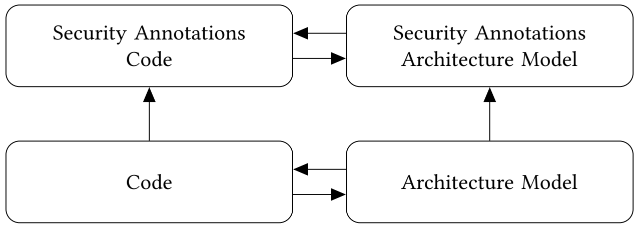

Security Annotations Code
Security Annotations Architecture Model
Code
Architecture Model

Figure 3: The four input models for the security analyses. The arrows represent a consistency preservation between the models in the corresponding direction.

Listing 1: An example of the Reactions language. The `umlToJavaClass` Reactions file reacts to changes in `uml` by changing a `java` model. The `CreateUmlClass` Reaction is triggered after a UML class is inserted into a UML package as an `OwnedElement` and calls the Routine `createJavaClass`, which handles the creation of a corresponding Java class.

```
reactions: umlToJavaClass
in reaction to changes in uml
execute actions in java

reaction CreatedUmlClass {
    after element uml::Class
        inserted in uml::Package[OwnedElements]
    call {
        val umlClass = newValue
        createJavaClass(umlClass)
    }
}

routine createJavaClass(uml:Class umlClass) {
    match { /* retrieve_elements */ }
    create { /* create_elements */ }
    update { /* update_models */ }
}
```

The Reactions language defines unidirectional and change-driven *Reactions* and *Routines*. An example Reactions file is shown in Listing 1, starting with the name of the Reactions defined in the file, used for importing into other files. Afterward, a source metamodel is defined, i.e., the metamodel whose instances are modified by the developer. These changes then make the models inconsistent, i.e., the source metamodel and the target metamodel, defined in line 3 of Listing 1.

A Reaction itself also has a name and a trigger, which defines the change that has to happen in order to execute the Reaction. In the example in Listing 1 the trigger is the insertion of a UML class into a UML package as part of the owned elements of a UML package. The Reaction calls a Routine, consisting of three blocks. The `match` block allows retrieving objects, i.e., finding objects corresponding to other objects, e.g., because they have been created to react to the creation of each other. The `create` block can be used to create new objects. The `update` block contains arbitrary Xtend[1] code to execute operations necessary to preserve consistency.

2.2 Static Analysis

Static analysis is a process to assess (code) artifacts without executing them [5]. We use two static analysis, one on our architecture and one on our code. Architectural security analyses are used to detect vulnerabilities in the system when no current implementation is available, e.g., during early design or when the architecture is evolved. Architectural security analyses are also used to detect vulnerabilities based on information not available in the implementation, such as the deployment of a component. Early detection and

[1]https://eclipse.dev/Xtext/xtend/, accessed 03.07.2024

fixing of vulnerabilities is beneficial as the effort of fixing vulnerabilities increases the later the fix is performed [24]. The architectural analysis, however, does not free from considering security in the development phases, e.g., implementation, because it does not contain a complete view of the system [25].

This is why static analyses for source code had been developed to detect security weaknesses in the implementation. For detecting weaknesses, different approaches exist, such as inspecting the realized information flows [8] or the detection of patterns in the source code that are expected to be insecure, like the usage of weak hashing algorithms in an encryption [16]. One type of static security analysis detects weaknesses by determining whether the implementation conforms to specifications that describe the expected properties of the system. With this type of analysis, required security properties in the architectural model can be verified on the implementation level. For the analysis of the security properties defined in the architecture with source code analyses, the information captured in the architectural model and the specifications of the static analyses must be in the same state to represent the same system, i.e., if something is changed in the architecture model, this change must also be made in the other models so that the results of the analyses are conforming.

Architecture — Confidentiality Analysis. We use the security Domain Specific Language (DSL) *Confidentiality4CBSE* (C4C) [15] as one representative for the description of security specifications for software architectures that are then used by an architectural analysis. This DSL uses the *Palladio Component Model* (PCM) [22] for the description of the software architecture. C4C describes *DataSets*, representing some form of information a data-carrying element in the PCM, such as a parameter or return, is expected to contain. *DataSets* can, for instance, express roles the data belongs to or the type of data it contains, such as flight data or credit card data.

The architectural models are used as prescriptive or descriptive descriptions, i.e., they specify required properties of the system, either during its development or during its evolution. The consistent models can then be used in static architecture analyses [15, 25].

Code — Java Object-sensitive ANAlysis. We use the extended *Java Model Parser and Printer* (JaMoPP) [3] to model the Java source code of our application. To include security annotations, we use annotation metamodels used for the *Java Object-sensitive Analysis* (Joana) [8]. Similar to *DataSets* in C4C, annotations in Joana describe expected properties of the data by *levels* contained in parameters or fields in the source code. Joana also receives a lattice [7] specifying allowed flows between data of the specified *levels*. Based on these annotations, Joana calculates the information flows in the system and reports flows between the annotated elements that are not allowed, given the provided lattice.

3 RUNNING EXAMPLE

For our running example, we chose an excerpt of the travel planner case study [11]. The excerpt consists of a *Booking Agency* that communicates with an *Airline* to provide services, illustrated in Figure 4.

Users can book flights via the *Booking Agency*, illustrated in Figure 5, by sharing their flight information and their credit card data

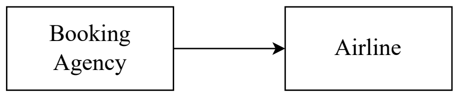


Figure 4: The *Booking Agency* communicates with the *Airline* to be able to provide services, e.g., the booking of flights.

with the *Booking Agency* in order for the *Booking Agency* to book the desired flight at the desired *Airline*. The flight information can be, e.g., the date of the flight, information about the luggage, or seat reservations. At the architecture level, we do not want the credit card data to flow to the *Airline*.

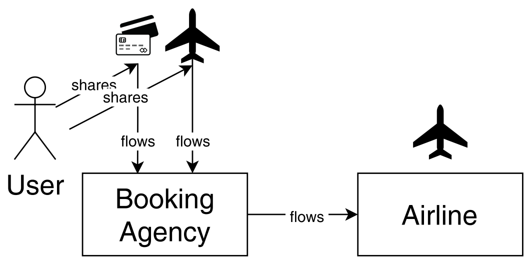


Figure 5: A *User* may share his flight information with the *Booking Agency* for the *Booking Agency* to be able to book a flight for him. Additionally, the *User* shares his credit card information with the *Booking Agency* to pay for the flight booked for him.

The result of the architecture analysis is shown in Figure 6, i.e., the *Booking agency* has access to the flight information and the credit card data. However, only the flight information is shared with the airline. Thus, our security requirement is fulfilled.

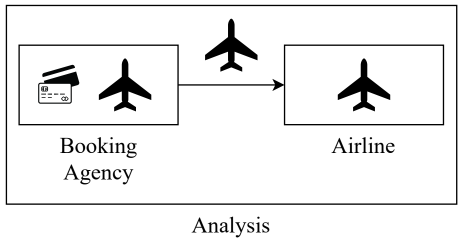
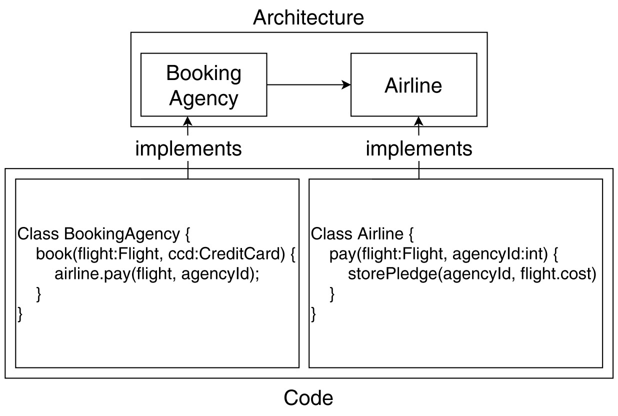


Figure 6: The *Booking Agency* shares the flight information with the *Airline*. The result of the architectural analysis shows, that only the flight information flows to the *Airline*, and thus our security requirement, that no credit card information flows to the *Airline*, is fulfilled.

After the system is implemented, we can implement correspondences between the architecture and the code, illustrated in Figure 7. The class *BookingAgency* implements the architectural element *Booking Agency* and the class *Airline* implements the architectural element *Airline*. The static code analysis can now be executed and detects a flow of flight information and no flow of credit card information to the *Airline*. Thus, our implementation conforms to our architecture with regard to the security requirements, i.e., only flight information flows to the airline.



```
Class BookingAgency {
    book(flight:Flight, ccd:CreditCard) {
        airline.pay(flight, agencyId);
    }
}
```

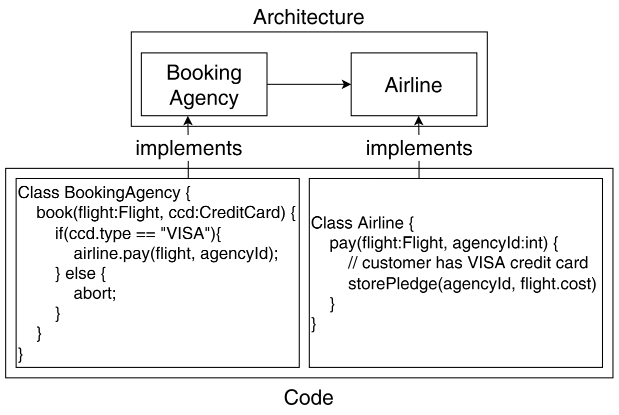
```
Class Airline {
    pay(flight:Flight, agencyId:int) {
        storePledge(agencyId, flight.cost)
    }
}
```

Figure 7: The *Architecture* contains the *Booking Agency* and the *Airline* with their data flows. The *Code* contains possible implementations not violating the security specifications by avoiding the information about the type of the credit card to flow to the *Airline*.

Now consider an evolution scenario for the system, in which the booking agency has a cooperation with the credit card company VISA and because of that cooperation only accepts VISA cards. The change in the code is relatively simple, illustrated in Figure 8. The check about the type of the card is added and booking attempts with different credit cards are aborted. This detail is not documented in the architecture, and the static architecture analysis still does not find a violation. However, the static source code analysis can now find a violation, i.e., an illicit flow of the type of the credit card to the *Airline*, which contradicts our security requirement.

4 APPROACH

We now present our general approach for the consistency management between the annotations of static architecture analyses and static source code analyses, as illustrated in Figure 3. This concept for maintaining consistency was developed based on the properties of the model elements. The input models include the source code, architecture model, security annotations for the source code, and security annotations for the architecture model. Our concept is designed for consistency management systems with unidirectional consistency preservation rules. In order to achieve bidirectionality for the consistency specifications between the architecture annotations and the code annotations, it is necessary to also define a rule for the reverse direction between the annotations for each rule. The direction of the rules between annotations and models they annotate should only allow modifications of annotations. The reason for unidirectionality is that annotations should only provide further information about the annotated element and should not influence or change the element itself.



```
Class BookingAgency {
    book(flight:Flight, ccd:CreditCard) {
        if(ccd.type == "VISA"){
            airline.pay(flight, agencyId);
        } else {
            abort;
        }
    }
}
```

```
Class Airline {
    pay(flight:Flight, agencyId:int) {
        // customer has VISA credit card
        storePledge(agencyId, flight.cost)
    }
}
```

Figure 8: The *Architecture* contains the *Booking Agency* and the *Airline* with their data flows. The *Code* contains possible implementations violating the security specifications by allowing the information about the type of the credit card to flow to the *Airline*. This may be because the *Booking Agency* has a new collaboration with VISA as an evolved system requirement.

The rules for preserving consistency are kept general yet specific enough to develop a functioning consistency preservation for the specific instances of the models from the concept. The rules do not assume any specific consistency management software or models, but metamodels are specified for the models so that the concept can refer to specific elements. The security analyses in this study are limited to annotation-based methods. This is because annotations can be modeled and kept consistent with the source code and architecture models.

4.1 Security Analysis Structure

To describe consistency preservation rules between models at a higher level of abstraction than the metamodels of specific models used, this section presents a specific form of annotation-based security analysis structure, illustrated in fig. 9. For our reference structure, a system consists of system elements. An annotation always refers to one system element and a number of security information elements.

This structure is not a general structure of annotation-based security analyses, but a restriction to annotation-based security analyses to a specific form. The structure is intentionally kept generic, to define the consistency preservation rules, that can be applied to a wide range of annotation-based security analyses. Although the structure does not constitute the metamodels of the security analyses, it abstracts the security analyses syntactically. By defining consistency preservation rules using this structure, they are not tied to specific security analyses but can be used to preserve consistency between any annotation-based security analyses that are based on this structure.

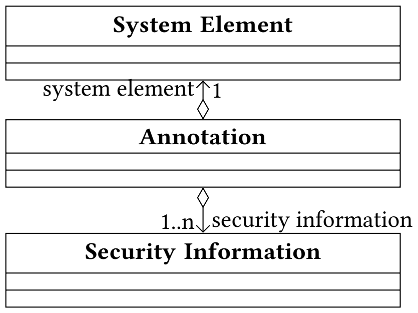


Figure 9: Structure of annotation models for our approach. An *Annotation* refers to a *System Element*, which is part of the system model, and a *Security Information*, which contains security relevant information, e.g., a security level.

4.2 System Model Structure

The system models are based on the system elements, which in Java include methods and parameters. Annotations are used to relate security information to domain elements, but do not carry any information themselves. They reference security information that cannot be inferred from the element. This consistency only applies to specification-based security analyses, which utilize security information and therefore differ from security analyses that do not require any additional information beyond the analyzed model. The reason for this limitation is that an annotation model is only necessary if information is available for it. Security information can include simple details such as security levels, as well as more complex information like grids of security permissions.

4.3 Consistency Preservation Rules

We assume the existence of consistency preservation between the architecture and code model. Thus, the following section focuses on the consistency preservation rules between the annotation models and between the architecture and code and their respective annotation model. The first element to keep consistent are the roots of both annotation models. The consistency preservation rules are defined as reactions to changes in a source model, which construct changes in a target model to preserve their consistency.

4.3.1 Root. We assume metamodels and models to be representable as trees, as, for instance, realized by Steinberg et al. [26]. The root of the model is the element of the tree that transitively contains all other model elements. Whenever a root is created, the system should check if a corresponding root already exists in the target model. If so, a correspondence should be established between the roots. If a root does not exist, a new one is created in the target model and a correspondence is established between the roots. This rule is defined in this form for all models in all directions, and illustrated in Listing 2 for the direction between Java and Joana.

4.3.2 Code Element. Only the annotatable elements are relevant for preserving consistency, as the consistency for other elements will be preserved by the existing consistency preservation rules. When a code element is deleted, the attached annotations must also be deleted. When a code element is created, it is necessary to ensure that the maintainer of the annotation model is consulted to explain their intentions to avoid incorrect assignment of annotations. When creating a code element, it is often unclear how to proceed, as the content of an annotation cannot be determined from the code element alone. If completeness is required for the analysis, default values should be used for annotations, e.g., the lowest available security level as a default security level for new elements.

4.3.3 Architecture Element. The general rules between the architecture model and its annotations are the same as those between the source code and its annotations. However, their implementation may differ, even if they are based on the same rules. This is especially important when the rules are executed *transitively.* Transitive execution allows rules to react to changes made not only by the user but also to changes created by the rule execution through the consistency management themselves, ensuring consistency across more than two models. To prevent an infinite loop, the rules for transitive execution must be adapted, for example, by using suitable queries.

4.3.4 Annotation. The annotations are first class entities, i.e., they are model elements in their own. Annotations can thus be linked with correspondences, which allows their tracing for consistency preservation. In general, it cannot be assumed that there is a one-to-one relationship between annotations. Any number of annotations in the source model can correspond to any number of annotations in the target model. An example in our case study is that an information flow specification is represented as one annotation in the architectural annotation metamodel, while a source code analysis uses two annotations to represent the corresponding information. When an annotation is removed, the corresponding annotations must also be removed. Therefore, a rule can change several elements either in the models of the architectural analysis or the source code analysis. In such cases, only the explicit correspondence, i.e., a one-to-one correspondence, should be removed. Additionally, modifications of the annotated elements, i.e., the source code or architecture elements, may be necessary. This is recommended if the elements do not fulfill any other requirement and are thus a potential security weakness or even vulnerability. If the elements do fulfill other purposes, they should be kept, but the developers should be notified to re-assess the elements and their requirements.

When creating an annotation, it may be necessary to create several annotations and edit existing ones. This can either be a direct reaction to the creation of a new annotation, e.g., the creation of two new security levels. Additionally, the use of lattices may necessitate the creation of multiple lattice elements, which in turn creates new security annotations through consistency preservation. It is important to check whether suitable annotations already exist and, if necessary, adapt them instead of creating duplicates, e.g., for security levels.

It is also important to ensure that the consistency preservation does not invalidate existing annotations. In case of a deletion of a security information, it is necessary to ensure that it is no longer needed by another annotation. An example of such a case might be the security level used by the two annotations in the source code analysis, which corresponds to another annotated security information in the architectural annotation model. The deletion of the security level in the architectural annotation model would

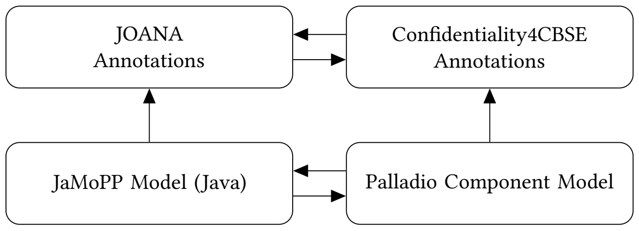


Figure 10: The specific models used for the Vitruvius implementation of the consistency preservation.

invalidate the source code annotations, as they point to a non-existing element.

4.3.5 Multiplicities. When comparing two security annotation models, it is important to verify if any security information elements represent the same information. It is possible that a security information syntactically represented by one element in the source model is represented by multiple elements in the target model or vice versa. An example is a security level consisting of several basic security levels. This security level can be represented as a single syntactical element in one model, while it is represented by using the syntactical elements of the basic security levels in the other models.

4.3.6 Complex Mappings. The structure of security requirements and the connection between the security annotations on the architecture and on the code level may become quite complex. We currently do not see any limitations for our approach in regard to the complexity of the consistency preservation, but we did not implement other, more complex consistency preservation rules, and therefore cannot make a statement about that. Note, that we do not consider the complexity of the mapping between architecture and code itself here, as this a prerequisite of our approach and thus out of scope for this paper.

4.4 Prototypical Implementation

We implemented the consistency preservation rules between security annotations as a prototypical implementation. The purpose of the sample implementation was to evaluate the correctness of the developed concept. The implementation adhered to the concept specifications during development and was checked for consistency with the input models.

The specific models used in the implementation, illustrated in Figure 10, are as follows: The *Palladio Component Model* (PCM) [22] as the architecture model, *Confidentiality4CBSE* (C4C) [15] for static architecture analysis, the extended *Java Model Parser and Printer* (JaMoPP) [3] for the source code model with Java as the source code, and the *Java Object-sensitive Analysis* (JOANA) [8] for static analysis of the source code. The Vitruvius [13] framework was used to implement consistency preservation using transitive consistency preservation rules, which means that a change caused by a consistency preservation rule can trigger the execution of another consistency rule. The consistency rules are written in Vitruvius’ Reaction language and are unidirectional. This means that the consistency rules must be implemented in both directions between the two annotation models.

Listing 2: Reaction for the insertion of a java package as a root into a model, which is the application of the rules described in subsubsection 4.3.1. The Routines called search for a corresponding joana root and, if none exists, create a new one and add the correspondence to the given java package.

```
reaction JavaRootCreated{
 after element java::Package inserted as root
 with newValue.name === null || (!newValue.name.contains("
      contracts") && !newValue.name.contains("datatypes"))
 call {
  searchForJoanaRoot(newValue)
 }
}

routine searchForJoanaRoot(java::Package javaRoot){
 match{
  require absence of joana::JOANARoot corresponding to javaRoot
  val joanaRoot = retrieve optional joana::JOANARoot corresponding
        to ContainersPackage.Literals.PACKAGE
 }
 update{
  if(joanaRoot.present){
   addCorrespondenceBetween(joanaRoot.get, javaRoot)
  } else {
   val joanaRoots = javaRoot.eResource.allContents.toList.filter(
        JOANARoot).toList
   if(joanaRoots.empty){
    createJoanaRoot(javaRoot)
   } else {
    addCorrespondenceBetween(javaRoot, joanaRoots.get(0))
   }
  }

 }
}

routine createJoanaRoot(java::Package javaRoot){
 match{
  require absence of joana::JOANARoot corresponding to javaRoot
 }
 create{
  val joanaRoot = new joana::JOANARoot
  val lattice = new joana::Lattice
 }
 update{
  joanaRoot.name = "JoanaRoot"
  joanaRoot.lattice = lattice
  persistProjectRelative(javaRoot, joanaRoot, "model/" + joanaRoot
       .name + ".joanaRoot")
  addCorrespondenceBetween(javaRoot, joanaRoot)
  addCorrespondenceBetween(joanaRoot, ContainersPackage.Literals.
       PACKAGE)
 }
}
```

4.5 Example Consistency Preservation

To examine how the individual rules function, let’s revisit our running example. We have three objects: A User which can book flights through a purchase interface, and a booking interface between the *Booking Agency* and the *Airline*. Their data flow is encapsulated by facades.

We modeled the facades of the objects as PCM *Operation Signatures*, i.e., the signatures of arbitrary operations, e.g., methods. To

Listing 3: Reaction to the creation of a *SignatureInformationFlow*. If no corresponding EntryPoint is present, creates a new EntryPoint and a new FlowSpecification. Both are linked accordingly to also contain the information about the information flow on the architecture level, described by the SignatureInformationFlow.

```
1  reaction SignatureIFCreated{
2   after element c4c::SignatureInformationFlow created
3   call{
4    createEntryPoint(affectedEObject)
5   }
6  }
7
8  routine createEntryPoint(c4c::SignatureInformationFlow newFlow){
9   match{
10   require absence of joana::EntryPoint corresponding to newFlow
11   val method = retrieve java::InterfaceMethod corresponding to
          newFlow.appliedTo
12   val joanaRoot = retrieve joana::JOANARoot corresponding to
          newFlow.eContainer
13  }
14  create{
15   val point = new joana::EntryPoint
16   val spec = new joana::FlowSpecification
17  }
18  update{
19   addCorrespondenceBetween(point, newFlow)
20   addCorrespondenceBetween(ContainersPackage.Literals.PACKAGE,
          spec)
21   point.annotatedMethod = method
22   point.securitylevels.addAll(joanaRoot.securitylevel)
23   point.tag.add((method.eContainer as Interface).name + "." +
          method.name)
24   spec.entrypoint = point
25   joanaRoot.flowspecification.add(spec)
26   point.lattice = joanaRoot.lattice
27   for(anno : joanaRoot.annotation){
28    if(anno.tag.contains(point.tag.get(0))){
29     spec.annotation.add(anno)
30    }
31   }
32  }
33 }
```

model our data flow constraints, we used C4C, which is an approach for detecting confidentiality leaks in PCM. To analyze confidentiality, C4C utilizes confidentiality specifications in conjunction with the architecture model [15]. From the C4C model, we used a *SignatureInformationFlow* that is annotated to an OperationSignature. The SignatureInformationFlow includes an information element that contains a *DataSet*. This DataSet makes an assumption about the confidentiality of the data handled by the OperationSignature. The DataSet classifies the security of an OperationSignature. In our running example, we will use two DataSets: Set A for the purchase interface, between *User* and *Booking Agency*, which is allowed to contain credit card information, and Set B for the booking interface provided by the*Airline*, which is not allowed to receive credit card information.

For the code model, we use a JaMoPP *InterfaceMethod* corresponding to the OperationSignature. This is illustrated in line 11 in Listing 3, where it is retrieved. The security information added for the JOANA analysis uses *Source* and *Sink* definitions to represent a possible data flow, each annotated to the method passing on the data. This actual data flow representation is achieved by linking sources to sinks at each method to which information can flow from the source through an *EntryPoint*. Because we react to creation of a new SignatureInformationFlow, we have to create a new EntryPoint, as seen in line 15 as well as a new FlowSpecification in line 16 in Listing 3. Sink and source have an assigned security level, indicating whether a data flow is permitted. If data flows from a source with a high security level to a sink with a low security level, this data flow is not permitted. Therefore, we assign the sinks and sources of the purchase interface a high security level, as they handle credit card information. The airline on the other hand is assigned a low security level, as it handles no credit card information. This assignment of security levels is equal to the assignment of the DataSets in C4C and is illustrated in lines 21-29 in Listing 3.

The automatic consistency preservation prevents errors by keeping the models consistent, when a change is made in another model. For instance, if the security level of a source is modified in JOANA, the DataSet of the corresponding OperationSignature is automatically adjusted. The security level of the corresponding sink is also updated as a result of the transitive change. This feature enables the prevention of errors by automatically maintaining consistency.

In order for these models to be complete so that they represent an executable program, further elements are required, but these have been omitted in this example for the sake of brevity.

5 EVALUATION

To evaluate the implementation, individual reactions were tested first, followed by the interaction of all reactions. The evaluation was conducted using two artifacts: unit tests and a test model.

Unit Test. The unit tests verify whether the modifications made to the individual CPRs align with the intended outcomes, i.e., the effects of the rules outlined in section 4. This may entail multiple unit tests for a single CPR, each examining different outcomes. A model was generated for each unit test, including the minimum elements necessary to observe all effects of the corresponding CPR.

A total of 51 unit tests were written, 17 of them for the direction from C4CBSE to JOANA, 25 for the direction from JOANA to C4CBSE, 5 for the direction from PCM to C4CBSE and 4 for the direction from JaMoPP to JOANA. The different number of unit tests stems from the different representations of the security information. The unit tests were constructed systematically to cover every Reaction written and implements the triggering action, as well as a check of the expected outcome of the consistency preservation.

The following elements were created during the tests: 255 SecurityLevels, 953 InformationFlowRelations, 12 EntryPoints, 19 Sinks and 19 Sources. The Sinks and Sources each have a SecurityLevel, an annotated method and an annotated parameter. The Sources each have one tag. The Sinks each have between 0 and 10 tags.

Out of the 51 unit tests, 96% were successful. The two failed tests were due to unexpected behavior of the Eclipse Modeling Framework [26], which we used to implement our prototype. As a result, it is not possible to determine whether the CPRs in question exhibit the desired behavior. This allowed the specific verification of almost all individual reactions.

Test Model. The test model is used to verify the feasibility of the consistency preservation from an architecture analysis annotation model to a source code analysis annotation model. The test model is an existing architecture model with security requirements in the form of an architectural security annotation model. The architecture consists of 6 components with a total of 14 interfaces, resulting in the implementation of 15 provided and 13 required interfaces. The architectural security annotations consist of 27 ParameterAndDataPairs and 17 InformationFlows. The consistency preservation has created a source code analysis annotation model as a result model from this test model. The test data includes the test model, i.e., an architecture model with corresponding security annotations, as well as a source code model with generated security annotations, which we use as a gold standard. To assess consistency, we created the source code based on the existing consistency preservation rules between PCM and JaMoPP, and the corresponding security annotations with the Reactions written conforming to this approach. We then compared the models created by our consistency preservation approach with the gold standard available. The models matched, and thus we assume, that our consistency preservation approach is correct.

6 THREATS TO VALIDITY

We discuss threats to the validity of our approach while following the guidelines from Runeson and Höst [23] and Wohlin et al. [29].

Internal Validity. We argue, that our approach can be applied to systems described by an architecture and a code model as well as annotation models for both. We implemented test cases and a case study, using prior work. The used test cases and case study conform to the preconditions we formulated. Based on the successful implementation of both, we argue, that our approach is applicable and the preconditions we proposed are necessary. Nevertheless, we only implemented a single case study and that was based on our own prior work, thus restricting the internal validity.

External Validity. The external validity of our approach is limited due to several factors. Firstly, the limited scope of the evaluation poses a risk to external validity, as the concept was only implemented for one case study with a specific combination of models and external factors. Additionally, the concept is based on the model for annotation-based static security analyses presented in the approach, which limits its applicability. While the approach is formulated generically, it is unclear how applicable it is to other annotation-based static security analyses.

7 RELATED WORK

Kebaili et al. [12] discuss the co-evolution of generated code as part of metamodel evolution. The code written by developers depends on the code generated from the metamodel. In an evolution scenario, the generated code changes and its behavior may also change. The authors envision an automatic approach to ensure correct co-evolution of the code. We focus on security annotation for architecture models and handwritten code instead. Additionally, we ensure a correct evolution by using consistency management, at the cost of partly complex consistency preservation rules.

Jasser [9] considered a similar question, i.e., how to ensure an implementation fulfills security decisions made on the architecture level. The authors propose an approach to check the conformance of an implementation with its specified architecture by employing architectural security rules. The rules are formalized using a controlled natural language approach. Afterward, the rules are checked with dynamic analysis techniques on the system behavior. In contrast, we use architectural security specifications and keep them consistent with code security annotations, e.g., by propagating them. Furthermore, we use static analysis techniques.

Tuma et al. [27] also worked on checking security compliance between models and code. The authors point out, that a manual compliance check is labor-intensive and can be error-prone. They developed a semi-automated approach to automatically define heuristic-based mappings between design and code models. Instead, we employ consistency preservation techniques to obtain the correspondences. Using consistency preservation entails higher initial specification overhead, but enables our heuristic-free approach.

Peldszus et al. [21] propose an approach to verify that security assumptions hold at run-time. The authors use UMLsec [10] as security annotation DSL for the architecture. Their approach, named UMLsecRT, supports the annotation of system models with security properties, which are then synchronized to corresponding source code annotations. The run-time monitor then checks for violations of those source code annotations. In contrast, we focus on static analysis and the consistency preservation between architecture and code annotations. Furthermore, our generic approach is independent of the DSL used and directly embedded into consistency preservation.

Yurchenko et al. [30] also focus on the verification of security requirements on architecture in their implementation as source code. The authors focus on the advantage of a reduced specification overhead for source code annotations by keeping the annotations, explicitly specified by the architect on the software architecture, consistent with the source code annotations. Additionally, the paper presents an example specific for PCM and KEY [1], a software verification tool. We also use PCM as an architecture description language and JOANA as analysis in our example. Nonetheless, our approach is independent of the analyses used and, apart from the properties we demand from the analyses, generic. Additionally, we focus on continuous compliance through delta-based consistency preservation, instead of the generation of source code annotations as the authors do [30].

8 CONCLUSION

This paper presents a concept for preserving consistency for input models of two security analyses based on annotations. The input models are an architecture model, a corresponding source code model, and security annotation models. The concept describes consistency preservation rules that describe the required reactions to changes in the architecture and code models, as well as their annotation models. The concept establishes consistency preservation rules between the two annotation models and from the system models (i.e., architecture model and source code) to the annotation models. We assume our approach will be an addition to an existing consistency preservation, and thus require the consistency preservation between the system models. We argue our approach fulfills the two challenges outlined in the introduction, i.e., the provision of specifications and the evolution of specifications. For the provision of specification, we assume a software architecture and security requirements in the form of annotations, provided by the system architect and security experts. We then use the consistency preservation to create the corresponding source code annotations and thus provide them. The usage of a consistency preservation framework and consistency preservation rules also allows us to cope with evolving specifications, as a change in the specifications simply triggers the appropriate consistency preservation rule. Thus, we argue, that our approach can also fulfill the second challenge, i.e., the evolution of specifications.

The consistency preservation was implemented in Vitruvius and individual reactions were tested using unit tests, with 96% of the tests being successful. We also implemented a case study with concrete system and annotation models and obtained the same result as the manually defined gold standard. The success of the unit tests and the case study indicate the feasibility of our approach.

For future work, we plan to implement further case studies using different security DSLs, e.g., UMLsec. Additionally, we plan to research on the integration of intellectual property protection in consistency management for continous verification [28]. Furthermore, we plan to simplify the creation of consistency preservation rules by adding new language elements to use the annotation characteristic of the annotation models, i.e., to automatically generate deletion rules by simply annotating a model as the annotation model of another model. To improve the usability of our approach, we plan to include the automatic execution of the security analysis and the propagation of the results in the corresponding models.

ACKNOWLEDGMENTS

This work was funded by the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs and supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263 and the project FeCoMASS - 499241390.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. 2016. Deductive software verification-the key book. *Lecture notes in computer science* 10001 (2016).

[2] Mohamed Almorsy, John Grundy, and Amani S Ibrahim. 2013. Automated software architecture security risk analysis using formalized signatures. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 662–671.

[3] Martin Armbruster. 2022. Parsing and Printing Java 7-15 by Extending an Existing Metamodel. https://doi.org/10.5445/IR/1000149186

[4] Colin Atkinson, Christian Tunjic, and Torben Moller. 2015. Fundamental Realization Strategies for Multi-view Specification Environments. In *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. IEEE, Adelaide, Australia, 40–49. https://doi.org/10.1109/EDOC.2015.17

[5] Brian Chess and Jacob West. 2007. *Secure programming with static analysis*. Pearson Education.

[6] Istvan David, Hans Vangheluwe, and Eugene Syriani. 2023. Model consistency as a heuristic for eventual correctness. *Journal of Computer Languages* 76 (2023), 101223.

[7] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (may 1976), 236–243. https://doi.org/10.1145/360051.360056

[8] Dennis Giffhorn and Christian Hammer. 2008. Precise analysis of java programs using joana. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 267–268.

[9] Stefanie Jasser. 2020. Enforcing Architectural Security Decisions. In *2020 IEEE International Conference on Software Architecture (ICSA)*. 35–45. https://doi.org/10.1109/ICSA47634.2020.00012

[10] Jan Jürjens. 2002. UMLsec: Extending UML for secure systems development. In *International Conference on The Unified Modeling Language*. Springer, 412–425.

[11] Kuzman Katkalov. 2013. Modeling the Travel Planner Application with IFlow. https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html. Accessed: 2024-07-13.

[12] Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, and Olivier Barais. 2023. Towards leveraging tests to identify impacts of metamodel and code co-evolution. In *International Conference on Advanced Information Systems Engineering*. Springer, 129–137.

[13] Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling consistency in view-based system development — The Vitruvius approach. *Journal of Systems and Software* 171 (Jan. 2021), 110815. https://doi.org/10.1016/j.jss.2020.110815

[14] Max Emanuel Kramer. 2017. *Specification Languages for Preserving Consistency between Models of Different Languages*. PhD Thesis. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. https://doi.org/10.5445/IR/1000069284

[15] Max E. Kramer, Martin Hecker, Simon Greiner, Kaibin Bao, and Kateryna Yurchenko. 2017. Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems. https://doi.org/10.5445/IR/1000076957

[16] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 931–936. https://doi.org/10.1109/ASE.2017.8115707

[17] S. Lipner. 2004. The trustworthy computing security development lifecycle. In *20th Annual Computer Security Applications Conference*. 2–13. https://doi.org/10.1109/CSAC.2004.41

[18] Gary McGraw. 2004. Software security. *IEEE Security & Privacy* 2, 2 (2004), 80–83.

[19] Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. 2020. Classifying approaches for constructing single underlying models. In *Model-Driven Engineering and Software Development: 7th International Conference, MODELSWARD 2019, Prague, Czech Republic, February 20–22, 2019, Revised Selected Papers 7*. Springer, 350–375.

[20] José Carlos Sancho Núñez, Andrés Caro Lindo, and Pablo García Rodríguez. 2020. A Preventive Secure Software Development Model for a Software Factory: A Case Study. *IEEE Access* 8 (2020), 77653–77665. https://doi.org/10.1109/ACCESS.2020.2989113

[21] Sven Peldszus, Jens Bürger, and Jan Jürjens. 2024. UMLsecRT: Reactive Security Monitoring of Java Applications With Round-Trip Engineering. *IEEE Transactions on Software Engineering* 50, 1 (2024), 16–47. https://doi.org/10.1109/TSE.2023.3326366

[22] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. 2011. The Palladio Component Model. https://doi.org/10.5445/IR/1000022503 ISSN: 2190-4782.

[23] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14 (2009), 131–164.

[24] Stephan Seifermann. 2022. *Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements*. Ph. D. Dissertation. Karlsruher Institut für Technologie (KIT). https://doi.org/10.5445/IR/1000148748

[25] Stephan Seifermann, Robert Heinrich, Dominik Werle, and Ralf Reussner. 2022. Detecting violations of access control and information flow policies in data flow diagrams. *Journal of Systems and Software* 184 (2022), 111138. https://doi.org/10.1016/j.jss.2021.111138

[26] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition* (2nd ed.). Addison-Wesley Professional. https://www.informit.com/store/emf-eclipse-modeling-framework-9780321331885

[27] Katja Tuma, Sven Peldszus, Daniel Strüber, Riccardo Scandariato, and Jan Jürjens. 2023. Checking security compliance between models and code. *Software and systems modeling* 22, 1 (2023), 273–296.

[28] Thomas Weber and Sebastian Weber. 2024. Model Everything but with Intellectual Property Protection — The Deltachain Approach. In *2024 ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM. https://doi.org/10.1145/3640310.3674086 in press.

[29] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[30] Kateryna Yurchenko, Moritz Behr, Heiko Klare, Max E Kramer, and Ralf H Reussner. 2017. Architecture-driven Reduction of Specification Overhead for Verifying Confidentiality in Component-based Software Systems.. In *MODELS (Satellite Events)*. 321–323.