Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Restoring Security of Evolving Software Models using Graph-Transformation*

Jens Bürger[13], **Jan Jürjens**[124], **Sven Wenzel**[25]

[1] TU Dortmund, Germany
[2] Fraunhofer ISST, Dortmund, Germany
[3] e-mail: jens.buerger@cs.tu-dortmund.de
[4] WWW: http://jan.jurjens.de
[5] e-mail: sven.wenzel@isst.fraunhofer.de

Received: date / Revised version: date

Abstract. Security certification of complex systems requires a high amount of effort. As a particular challenge, today's systems are increasingly long-living and subject to continuous change. After each change of some part of the system, the whole system needs to be re-certified from scratch (since security properties are not in general modular), which is usually far too much effort. When models for software get changed, this can lead to security weaknesses that are also part of the software system that is derived from those models. Hence, it is important to check the models with respect to security properties, and correct them respectively.

To address this challenge, we present an approach which not only finds security weaknesses but can also correct them in a tool-supported way. As time goes by, a diverse number of changing requirements that may be security-related and non-security-related lead to a evolving system that met its security requirements at design time but can contain vulnerabilities with respect to meanwhile updated security knowledge. Supported by patterns we can describe and detect potential flaws that may arise in models, such as inconsistencies in security requirements. Potential violations can be formalized in the patterns as well as the correction alternatives to fix these. It is based on graph transformation and can be applied to different types of models and violations. For flaw detection, these patterns are used as the left-hand sides of graph transformation rules. Using graph transformation, we can further correct the models and establish that they no longer violate the security requirements under investigation. The approach is supported by a tool which can check whether these patterns arise in models and assist the user in correcting the security vulnerabilities.

* This research is funded by the DFG project SecVolution (JU 2734/2-1) which is part of the priority program SPP 1593 "Design For Future - Managed Software Evolution".

1 Introduction

Information systems deal with critical data of companies and individuals and have thus high security requirements. Keeping an information system up-to-date with its changing environment is a challenging task, since the typical runtime of such a *long-living software system* is said to be at least 10-15 years. In such a period of time, a system is facing continuously changing requirements due to new regulations, and the changing environment leads to a constant flow of new malware, newly discovered security flaws, better cracking algorithms and so forth. This means that an information system has to be checked constantly against newly discovered security knowledge, even if the security requirements itself stay the same.

Model-based development using e.g. the Unified Modeling Language (UML) is a methodology used to improve software quality. The Business Process Model and Notation (BPMN) is used for designing business processes and to automatize them by means of tools for work flow execution. Furthermore, approaches have been proposed to extend modeling languages with security properties. For example, UMLsec extends the Unified Modeling Language (UML) with security concepts, allowing developers to specify security properties such as confidentiality as part of the UML models [25, 24, 27, 26]. Other approaches (e.g. SecureUML [36]) provide modeling concepts such as role-based access control as part of activity diagrams. Similar approaches can be found that make use of BPMN instead of UML [54, 55, 45].

A secure design is difficult to get right and many designs contain security-weaknesses which are often non-trivial to find and then to fix. And if an information system fulfilled all security requirements at the time of

initial design, it can become outdated due to a changing environment. Hence, it is necessary to constantly validate the security requirements of the design models.

The idea of the approach presented in this paper is to formalize flaws in the design of critical systems using a concept similar to anti-patterns. Anti-patterns arose from patterns in software development and project management [10]. The concept of anti-patterns helps to explicitly formalize certain risks and problems of a design. Here, we formalize vulnerabilities in the design of security-relevant systems as *violation patterns*. Beyond just identifying parts of model designs that contain security flaws, our approach also features the possibility to define correction alternatives. Alternatives are realized as the right-hand sides of graph transformations. This leads to a graph grammar for correcting modeling flaws. We use the graph transformation language Henshin [3] and apply it to the modeling languages UMLsec and BPMN.

The approach supports the detection and correction of all security vulnerabilities which can be expressed within the Henshin specification language. Furthermore, violation patterns can be interpreted as an abstraction of security knowledge. Security knowledge about detection and correction of vulnerabilities, for example, can be derived from common guidelines and it can be gathered from experience gained from previously occurred incidents [46].

The remainder of the paper is structured as follows. In the next section we introduce a running example and different types of modeling flaws that may arise. In Section 3 we present our approach to detect and correct modeling flaws. The utilization of graph transformation within our approach is described in Section 4. Section 5 reports on experiences in applying our approach. How our approach can be used to deal with model evolution with respect to security properties is covered in Section 6. Details on the tool-support are given in Section 7. We discuss related work in Section 8 before we conclude with a summary and an outlook on future work in Section 9. Further information about tool-support for security-hardening can be found on our website[1].

2 Detecting Security Vulnerabilities at the Model Level

The development of software systems in security-relevant domains can profit from the use of model-based or model-driven development of software and systems as well as business processes. There are UML extensions such as UMLsec [25] that allow modeling security requirements. Furthermore, tools exist which to some degree can *find* security weaknesses (such as [11]). However, these tools do not support automated security hardening. Supported by patterns we can describe and detect potential flaws that may arise in models, such as inconsistencies in security requirements. Moreover, we enable automatic correction of detected flaws.

2.1 Graphs

The fact that many modeling languages are graph-based enables us to interpret the meta-models as well as models of systems (called *instance models*) as graphs. Much work has been done on graph transformation systems, triple graph grammars and algebraic graph transformations [49,3,50], and tool-support has been developed for various approaches, such as [22,18]. Thus, there is diverse technology we can build on (see Section 8 for a more detailed discussion). Our approach for modeling systems, flaws and corrective steps is based on meta-modeling techniques and the graph transformation system Henshin as well as its underlying transformation language [19,3].

To detect flaws in a given instance model, we make use of the underlying graph transformation system and perform plain matching of violation patterns against the instance model in the first place. By formalizing flaws as transformation rules where the left-hand side (LHS) equals the right-hand side (RHS), we preserve the instance model and directly detect model flaws. As a result, matches of these transformation rules resemble parts of the instance model where the specific flaw exists. Regarding corrective steps, our approach uses a technique that requires only few elements to formalize. Basically, all vulnerabilities that can be expressed using the Henshin (i.e. by means of graph transformation rules) can be detected. We discuss this in detail in Section 4.

2.2 Running Example: UMLsec

Regarding defining and checking security requirements of software models, our approach currently focuses on the UML extension UMLsec [25]. UMLsec is a model-based security engineering approach based on the UML, allowing one to specify recurring security requirements (such as secrecy, integrity, authenticity and others) and security assumptions (such as encryption of connections) on the system environment. This can be done using annotations as part of the UML specifications. Thus, we encapsulate knowledge on prudent security engineering as annotations in models or code and make it available to developers who may not be security experts.

The UMLsec extension is given as a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate the security requirements and assumptions. *Constraints* give criteria that determine whether the requirements are met by the system design by referring to a precise semantics

[1] `https://www-secse.cs.tu-dortmund.de/secse/pages/research/tools/index_en.shtml`

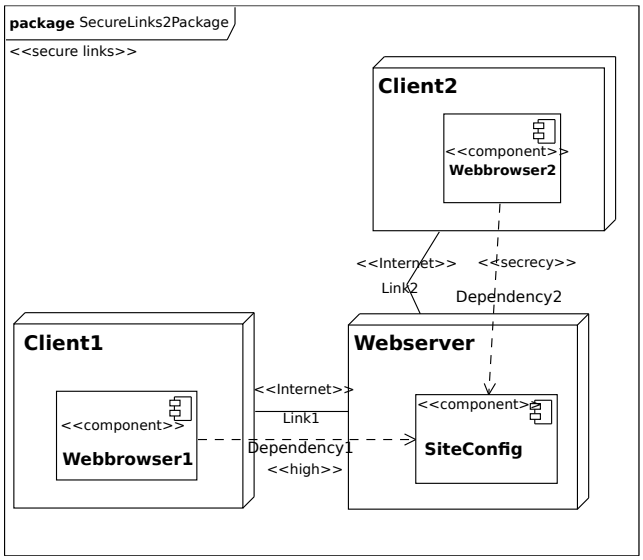

Fig. 1. UML diagram featuring UMLsec security property « secure links »

of the used fragment of UML. The security-relevant information added using stereotypes includes security assumptions on the physical level of the system, security requirements related to the secure handling and communication of data and security policies that system parts are supposed to obey.

Our approach can directly be applied to all UMLsec stereotypes within UML models for static aspects such as « secure links », « secure dependency », etc. These requirements are used mainly in class diagrams, component diagrams and deployment diagrams. They specify consistency rules between security requirements specified in the UMLsec model and security assumptions as part of the underlying infrastructure.

Moreover, our approach can also be used for detecting vulnerabilities concerning UMLsec stereotypes used within UML models for dynamic aspects (e.g. state charts, activity diagrams and sequence diagrams) which can at least be approximated safely using a static analysis (where "safely" means that the approximation may permit false positives but no false negatives in the security analysis; cf. [25] for more details).

Consider the system presented by a UML diagram shown in Figure 1 that we will use as a running example throughout the following sections. It uses the stereotype « secure links ». « secure links » deals with the security of communication: one can specify requirements such as « secrecy » to the logical relationships (dependencies) between components and annotate properties to physical relationships (communication links), e.g. « Internet » to indicate an unencrypted internet connection. UMLsec can then be used to validate the communication links against the requirements of the dependencies using attacker models. In the given example two clients are connected to a webserver through an internet link. The connections have stereotypes stating that communication has to be confidential (« secrecy ») and one case additionally requires integrity (« high »). The given configuration does not meet the security requirements, because internet links are unencrypted and can be read and manipulated. Hence, « secure links » is violated here. We will refer to this in more detail in Section 5.

2.3 Security Hardening

Our approach aims not only to detect modeling flaws, but also to correct them. This is a non-trivial challenge since in many cases it is not obvious how to correct a model which contains vulnerabilities in order to make it secure.

Flaws caused by mutually contradictory security requirements (such as non-repudiation and anonymity), can only be solved by changing the requirements in a way that they are no longer inconsistent with one another. There is no correction transformation which would enforce both of these security requirements (see [1] for more discussion on this situation).

All other flaws can be solved either by lowering the requirements or by changing the model in order to fulfill the requirements. We assume that a user always aims to fulfill the security requirements. If there exists exactly one possibility to correct the flaw, an automatic correction can easily be applied. In case of multiple possibilities for correction, the designer should get involved. As a model of a system is an abstraction, the alternative to be chosen represents a design decision. The developer has to review the correction alternatives with respect to which alternative solves the problem the desired way.

If the number of correction alternatives is not manageable for the designer, the alternatives can be ordered by their complexity. To simplify matters, we define the complexity of correction as the number of model elements affected. Other measures could also be used. Based on the ordered list of proposed alternatives, the designer can easier choose a suitable correction. In addition, it is possible to declare an alternative as default, which is then applied automatically. Nonetheless, the designer always has the option to fix a flaw by changing the model manually, or indeed to decide not to fix the vulnerability at all for some reason).

Our approach requires that all information (i.e. semantics) necessary to propose corrective transformations is part of the model. For example, recall the UMLsec stereotype « secure links » as introduced in Figure 1. If used as part of a deployment diagram, the requirement states that security requirements (such as « secrecy ») required by components implemented on distributed nodes are satisfied by the communication paths used by the respective nodes. The detection of violations of this stereotype in essence works by analyzing all point-to-point connections between nodes as well as their specified communication path type (e.g. « Internet ») and associate

them with the required security properties defined by the dependency between the respective components. In this case, all information needed to decide if there is a violation of « secure links » is available within the particular UML model. We give more detail on « secure links » in Section 5.

For example, the UML sequence diagram may not be self-contained as described above. The UML model contains the messages exchanged between objects as `OccurrenceSpecification` elements. Available tools implement the UML specification differently regarding the message order. In our case, we tested two UML modeling tools based on EMF. While TOPCASED [34] did not show a clear behavior, Papyrus [12] automacically arranges the messages to match their occurrence in the diagram. In the former case, the order of messages appearance is only captured as part of the graphical representation. It then would be necessary to treat the sequence diagram seperately, which can be interpreted as a model, too. Thus, it seems feasible to analyze the model by enriching it with relevant information about call and return sequences gathered from an analysis of the sequence diagram to be executed prior to application of our approach, if a modeling tool has been used that does not implement the ordering of elements as proposed by the UML specification.

3 Graph-Transformation-Based Correction of Security Vulnerabilities

In this section, we present the core aspects of our *4-phases-approach*. It consists of four phases that detect and correct violations of security requirements and its goal is to make sure that a given instance model *does not* undergo any security issues defined by patterns.

For realizing this, we define a number of violation patterns and corrective steps. Currently, our focus lies on supporting detection and correction of model flaws in UMLsec instance models. We have also defined a violation pattern based on BPMN. Every violation pattern consists of a number of graph transformation rules that are used to realize different types of actions concerning analyzing and altering a given instance model. Hence, a violation pattern contains all rules that are necessary to detect a violation in its various aspects as well as to perform corrective steps to patch the violations. Figure 2 shows an overview of the 4-phases-approach as realized in our current tool support. In the following, we will discuss the formal aspects of our approach, while the specific realization using the Henshin transformation language is part of Section 4.

The input of the approach consists of three elements: First, our catalogue of violation patterns which consists of formalized security vulnerabilities and built as graph transformations. To store additional information about how the different transformation rules relate to each other, an additional file called *cliché instance file* is used. As a third input element there is the given instance model potentially undergoing security vulnerabilities. The four phases of our approach are applied once sequentially, while decisions that have to be made in phase 4 may involve the system's designer. Optionally, results of phase 3 can be used by an additional step called *preparation code*, making use of pure Java code.

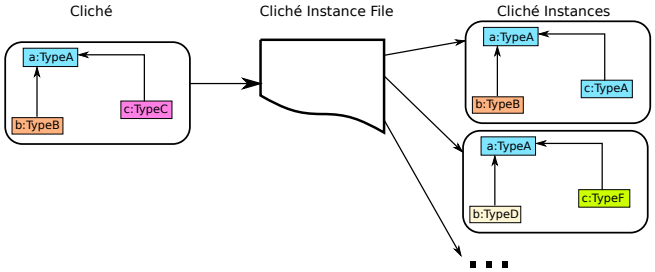

Fig. 3. Cliché used as a template to create cliché instances using mappings defined in the cliché instance file

The result of applying the approach is the checked and (where required) corrected instance model.

3.1 Phase 1: Test the environment

Prior to applying any advanced actions, it is necessary to check whether the given instance model satisfies assumptions as taken by the violation pattern. A simple precondition could be that a violation pattern designed to find and correct violations in some kind of BPMN pools may only be applied to instance models that fulfill certain preconditions. This means that a violation pattern e.g. checking BPMN collaborations contains one graph transformation rule *test* that only has a match if it is applied to a BPMN model that contains an instance of `Collaboration`. For realizing this, we make use of the matching algorithm of the underlying graph transformation system by using transformation rules where the left-hand side of the transformation (LHS) equals the right-hand side (RHS). As a consequence, phase 1 does not cause any changes to the instance model.

3.2 Phase 2: Instantiate the clichés

Before describing the process of phase 2, we introduce two terms we will use.

Cliché: When looking at an instance model, the existence or absence of one single security requirement can be shown by numerous artifacts in it. These artifacts can arise on a logical level as well as on a technical level. The former means that one security requirement can occur in different forms which lead to different characterizations in the model. The latter means that the violation of a requirement has only one form actually, but there are several possibilities to model them using the underlying

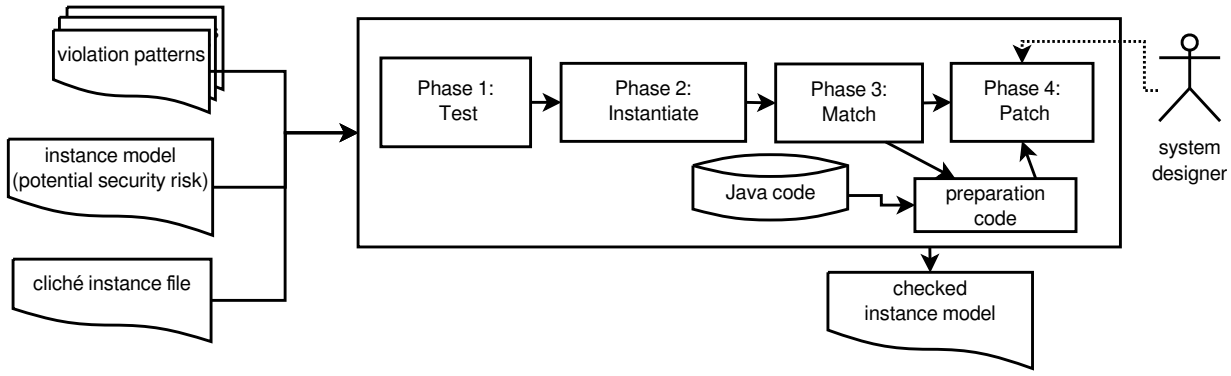

Fig. 2. Overview of the 4-phases-approach

meta model. One specific characterization, regardless of whether it is technically or logically induced, is called ***cliché***. We use this term similar to [53], where it is introduced for recurring sub-patterns of software-design-patterns.

As an example of a technical cliché, consider the following: In BPMN, pools can be annotated by text annotations. When looking at the graphical representation in the diagram, it seems that the edge that connects a pool to a text annotation is undirected. However, the BPMN meta model uses directed edges and at least it depends on the modeling tool how drawn edges are resembled by model elements. For example, the modeling tool we used for our work follows the order in which the user creates the two vertices. Given a violation pattern that incorporates a BPMN pool annotated with a text annotation, the pattern at least consists of two (technical) clichés: One cliché models a pool annotated with a text annotation with the association starting at the pool and ending at the annotation (meta model-wise). A second cliché models the same elements but with the association directed from the annotation to the pool.

We will give examples of logical and technical clichés in Section 4.

Cliché instance: In some cases, clichés may only differ in few details. More precisely, for a given violation pattern, there are transformation rules that only differ in the (model element) types of some nodes. To avoid the need for modeling a vast number of transformation rules only differing in node types, clichés can be extended by metadata making any desired number of nodes dynamic in a way that in an arbitrary combination, their types can be changed during application of our approach.

Thus, instead of having many clichés with only little differences, cliché instances support having only one cliché serving as a template plus additional metadata about which node type is supposed to be exchanged.

We call the process of taking a cliché and generating its runtime versions (cliché instances) *instantiation*. Figure 3 illustrates how the instantiation of clichés works. On the left side there is an cliché. The cliché instance file contains mappings which node types have to be exchanged how to build the respective cliché instances. This process leads to a number of cliché instances: While the general structure of the graph transformation rule is the same, the types of some nodes have changed. This is a method of easily generating *variants* of the same graph transformation rule instead of modeling one by one by hand. This also helps to keep down the number of modeled graph transformations.

We present an example of a violation pattern using cliché instances in Section 5.

To summarize, phase 2 is to instantiate clichés to cliché instances. This step is supported by metadata about which types of certain nodes have to be altered when and how.

3.3 Phase 3: Determine the matches

In phase 3, the cliché instances created during phase 2 are matched against the given instance model. More specifically, all graph transformation rules that resemble violations of a security requirement of a given violation pattern (which means cliché instances), are applied to the instance model sequentially. Whenever one of these rules does match at least once, a violation has been detected. In case we do not get a single match of the transformation rules, the given instance model does not undergo any violations modeled by the violation pattern and phase 4 will be skipped. In phase 3, we again use transformation rules where the LHS equal the RHS and thus do not cause any changes to the instance model.

3.4 Phase 4: Select and apply the patches

Application of phase 3 leads to a number of matches of clichés in the instance model, revealing some parts of the instance model that violate the security-relevant requirement modeled through the violation pattern. To perform corrective steps to re-ensure the security requirement, we use transformation rules that we call *patch rules*, realizing modifications to a specific part of an instance model.

As a security violation can occur multiple times in an instance model, the LHS of a patch rule may have multiple matches in the instance model, too. There is a general need for choosing between different alternatives for one erroneous part of a model that is identified.

For example, recall the system as introduced in Figure 1, featuring a UML deployment diagram modeling the communication between one server and two clients. Regarding the stereotypes that can be attached to communication paths for resembling the underlying communication infrastructure, think of connection types *Internet* (which means using a plaintext connection), *symmetric encryption* and *asymmetric encryption*. To make a model containing an Internet connection more secure obviously leads to two possibilities. We will discuss this kind of security violations in more detail in Section 5.

Concerning the security domain, we categorize the set of possible patches into at least two *paradigms*: first, raising the security level of the system's environment and second, reducing the security requirements on the system. Thus, we support this characterizing parameter as part of our approach at a technical level (cf. Section 4).

Hence, when more than one patch is applicable to a given match of phase 3, it has to be chosen which one should be applied. This selection can in some cases be done automatically, as discussed in Section 2. After selection, the patch rules are applied and the given instance model does not undergo the violations detected in phase 3 anymore. Obviously, phase 4 is the only phase that features transformations causing changes to the instance model.

The 4-phases-approach as introduced here realizes the application of graph transformation rules to find and correct model flaws by applying a graph transformation to erroneous parts of an instance model.

If the model is part of a set of system models, the other models will not be affected by these transformation. Hence, related models might require additional changes in order to preserve (or restore) consistency. Moreover, correction of two or more models in parallel can be handled by applying our approach iteratively to the whole UML specification of a system (containing all relevant models), where the security hardening transformations relevant to the different models are applied subsequently.

In this case, to avoid the situation that different hardening transformations undo the changes done by other transformations, it needs to be enforced that the chosen corrected model is not contained in the sequence of the models produced so far. Note that this does not guarantee the termination of this approach, since (as mentioned earlier) there are security properties which logically contradict each other, so applying the relevant hardening transformations iteratively cannot result into one specification which satisfies both contradictory specifications. Nevertheless, making use of more advanced techniques that do in-deep analyses of model's semantics is considered part of future work and discussed in Section 9.

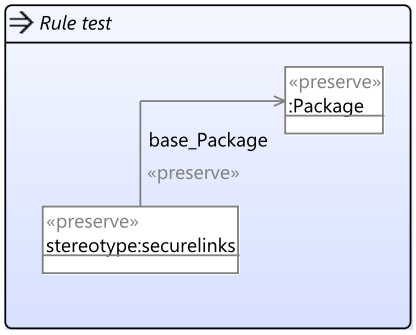

Fig. 4. Test rule for the *Insecure Links* violation pattern

4 Correction of Security Vulnerabilities: Using Henshin

In this section we show how we realized the 4-phases-approach as proposed in Section 3 using the graph transformation tool Henshin. We will give a short overview about aspects of Henshin that are important for our approach and then outline how we used Henshin to realize our approach.

4.1 Introduction into Henshin

We have carried out the approach explained in the previous sections using the graph transformation language *Henshin* and its associated toolset [3] which arose from *EMF Tiger*. The formal foundation Henshin is based on firstly was introduced as algebraic graph transformation [19]. The tool Henshin is based on the Eclipse platform and the Eclipse Modeling Framework (EMF). It allows graph-based modeling of graph transformations working on arbitrary meta models based on EMF. For example, the existing EMF implementations of the BPMN 2.0 and UML 2.x meta models can be used to perform model transformations on BPMN and UML models.

The graphical editor of Henshin uses a unified view on graph transformations. Instead of dealing with the left-hand side (LHS) and the right-hand side (RHS) of graph transformations separately, Henshin displays vertices and edges of the graph as three different types which have their own names, called *action*s. One transformation graph resembling one possible transformation is called *transformation rule*. The vertices of the graph are called *nodes*. All transformation rules including metadata are stored within a *transformation model*.

Henshin uses four different action types, annotated to the elements of a transformation rule. More precisely, all elements of a transformation rule have to be of one of the four action types that we shortly introduce: « preserve »

correlates to a plain match. Objects of this type are an element of the LHS as well as the RHS so that the respective objects are preserved. The action « delete » resembles the task of deleting objects from a given instance model such that they are part of the LHS but not of the RHS. « create » does the same for creating objects, i.e. is used for objects not part of the LHS but of the RHS. « forbid » has influence on the underlying matching algorithm, as it is a *non application clause* (NAC, see [3]). A match shall not take place if the object tagged with this action is part of a potential match.

An example of a Henshin transformation rule is shown in Figure 4. It depicts a rule for detecting a specific model type, namely a UML model containing a package that is annotated with the UMLsec stereotype « secure links ».

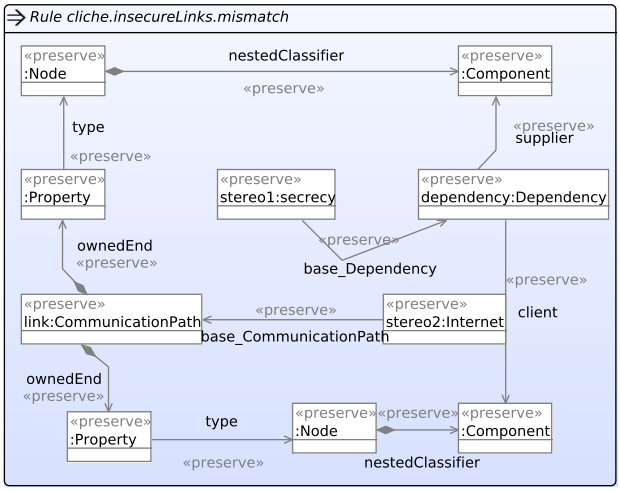

Fig. 5. Cliché matching violations of secure links

4.2 Application of Henshin

Essentially, our approach uses one transformation model per violation pattern. An advantage of structuring the different violation patterns this way is that it leads to transformation models with only a limited number of transformation rules. One transformation model consists of all transformation rules belonging to the respective violation pattern. Henshin does not have support for metadata in transformation models that would allow the separation of transformation rules by concerns, so we use a naming convention for the names used by Henshin rules. To prepare later tool support, the naming convention concerns not only the naming of the rules itself, but sometimes also the names of nodes. The naming convention uses a dot-notation and is explained in detail in Section 7.

As stated in Section 3, the transformation rules used by the phases 1-3 solely consist of « preserve »-nodes and edges, which means that we actually do not carry out any transformations but only use the techniques of Henshin for getting matches in the instance model. In the remainder of this section, we give an overview about how we use Henshin's features to realize the different phases of our approach. The realization refers to an EMF meta model as well as a given instance model.

Phase 1 is realized by using one single rule called `test`. An example of a test rule is shown in Figure 4. It just checks the existence of certain model elements to ensure that assumptions made by the rules of the violation pattern are met. As a consequence, the instance model is not altered. Thinking in terms of graph transformations, we do not invoke a transformation as such but we use the underlying matching algorithm for finding relevant spots in the instance model. We require that there is exactly one test rule per transformation model. Hence, after applying the testing rule to the given instance model successfully (e.g. resulting in a match), it is clear that the assumptions made by the violation pattern hold and arbitrary rules in terms of the remaining 3 phases can be applied.

Figure 4 shows the test rule of the running example. We want the test rule to check whether the instance model is an instance of the required meta model the violation pattern is defined for, in this case UML. This is realized by defining a node of type `Package`. Moreover, this rule ensures existence of the needed stereotype using a node named `stereotype` of type `securelinks`. The annotation with « secure links » is enforced this way.

During phase 2, clichés as part of the violation pattern together with information contained in the cliché instance file is used to build the cliché instances. For actions taking place in this phase, we restrict the usage to *preserve* in order to only cause non-altering matching. During this phase, we want to replace variable information contained in clichés by concrete values. Henshin offers a mechanism that is principally suitable for dynamically changing data of rules, called *rule parameters*. Due to technical reasons, Henshin only supports setting values of object attributes in a limited way. Nonetheless, we see the need for dynamically changing node types.

To exemplify this, see Figure 5. The node types of Henshin transformation rules are static, which means that one rule is able to detect exactly one configuration of communication path and security dependency. This means that if there are m different connection types and n different security dependency requirements, there would be a need for at least $m \cdot n$ transformation rules differing only in the type of two nodes (cf. Section 5.1 and Table 1 for further details). Regarding Figure 5, the depicted rule is only able to match the combination of « Internet » and « secrecy » in the first place. Hence, we realize the instantiation of clichés by making use of Henshin's API to keep the number of needed transformation rules low. The cliché instance file contains assignments for every node of a rule of which the type has to be

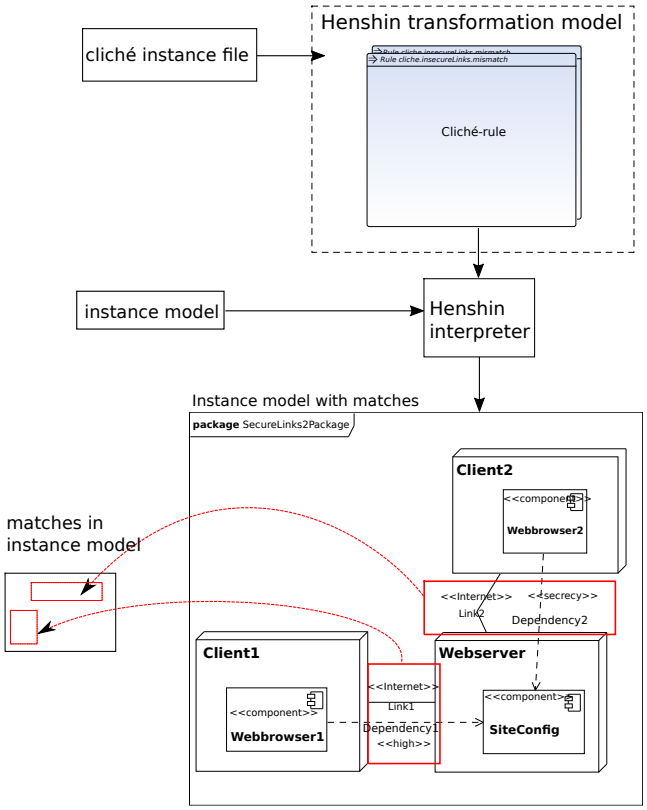

Fig. 6. Application of phase 3

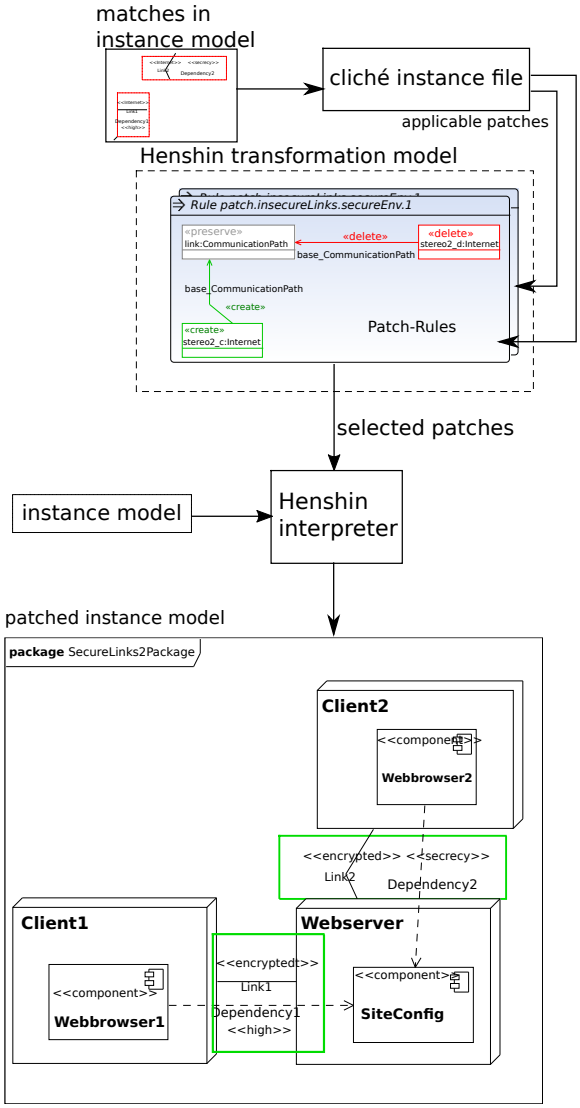

Fig. 7. Application of phase 4

replaced during runtime. We outline the contents and structure of the cliché instance file in detail in Section 7.

The purpose of phase 3 is to actually find matches of security vulnerabilities defined by the cliché rules. All clichés and cliché instances are applied to the instance model one after another. If Henshin does not find any matches, we know that the given instance model does not contain any of the vulnerabilities modeled by the used clichés. If Henshin finds matches, it returns a set of pointers into the instance model for every match found. The pointers refer to the objects in the respective transformation rule.

Phase 1 and 2 technically are rather simple steps coming down to just applying the `test` and `cliche` rules to the instance model. Phases 3 and 4 are more complex and make use of data sharing, so that we will explain the order of events taking place in phase 3 and 4 by means of Figures 6 and 7. Figure 6 shows the application of phase 3 to a UML deployment diagram. The system used in this example is the one introduced in Figure 1, undergoing two violations of the «secure links» requirement.

Regarding the 4-phases-approach (cf. Figure 2), the following operations take place: three elements are building the actual input. The Henshin transformation model handles the specific violation pattern, *Insecure Links* in this example. The respective transformation rules are stored in the Henshin transformation model. The transformation model itself gets information about dynamic type setting and cliché instances as part of a cliché instance file which is specific to the violation pattern. The third input element is the actual instance model. During phase 2, cliché instances are build, so that there is a number of cliché instances ready to be applied to the instance model by the Henshin interpreter. In our example there are two violations of «secure links», so that Henshin finds two matches of the respective cliché in the instance model. As a consequence, the result of phase 3 is an object which contains matches that reference to certain spots of the instance model which represent the vulnerability.

Finally, during phase 4 corrective steps to the instance model using patch rules are applied. These rules use actions of the types «create» and «delete» to actually apply changes to the instance model. Figure 7 shows the application of phase 4 in the running example. The result of application of phase 3 is a list of matches of cliché instances in the given instance model. Now,

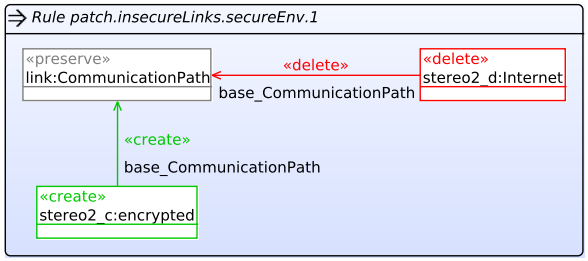

Fig. 8. Patch rule for securing the environment of a dependency

the cliché instance file contains a mapping of cliché instances to appropriate patch rules, too. Thus, supported by cliché instance file for every found match a selection of appropriate patch rules can be made. After making a selection of patches to be applied, there exists a list of parameterized patch rules. Finally, the selected patches are applied to the instance model using Henshin interpreter. As a result, we get the patched instance model. Regarding the running example, Figure 7 shows that a feasible correction of the vulnerabilities is to introduce encrypted connections.

Henshin offers the possibility to ease the design of patch rules. Because of the matches found during application of phase 3, the precise spots of the instance model that have to be patched are already known. Hence, it is not necessary for patch rules do be designed with all nodes needed for matching against the whole model again, but only a few nodes to just cause the desired changes. Using Henshin's feature *partial match*, the patch rule can be preconfigured using data gathered during phase 3 so that the patch rule is only applied to the demanded spot of the instance model. See Section 7 for more details on this.

Figure 8 shows an example of a patch rule belonging to the running example. It corrects the security vulnerability by deleting a connection path annotation and creates another one. As the name of this patch rule states, the goal of it is to secure the environment. Thus, replacing an «Internet» stereotype by «encrypted» may be a possible option. By modeling communication path type with better security, the security requirement is now fulfilled again. Note that both nodes of the patch rule (which refers communication path types) work as templates with respect to the cliché instance file. Thus, the types can be adapted during runtime.

4.3 Design of transformation rules and the effect of transformations to the model's semantics

Regarding the application of graph transformations to the model, the question arises to what extent the model's semantics is to be altered. In a traditional manner, the goal of (model) refactorings is to preserve the behavior of a system as it is. Nevertheless, our approach can detect security vulnerabilities. This means that the system's behavior represented by the given model lacks certain security-relevant properties in a sense that it does not meet given security requirements. For example, a system transfers confidential data even if a requested authentication has not taken place before. Therefore, the target of a model transformation has to be the modification of the system's behavior. Regarding the design of transformation rules, the amount of changes to the model generally should be as little as possible. Firstly, this obviously minimizes the performance overhead of the model transformation application. Secondly, it lowers the risk of side effects.

Application of corrective transformation rules to one model can be called adequate in a sense that if phase 3 does not have any matches in a model, it does not undergo the specific security vulnerability. To avoid termination problems, our tool implementation currently only does one run per violation pattern. After correction of the model, further checks should be executed to test the absence of all desired security violations, which means that the model holds the desired security requirements.

Nevertheless, the approach as a whole cannot be free of side effects. It is possible that correction of a security vulnerability to hold a security requirement again introduces a new vulnerability. One reason is that there are security properties that logically contradict each other. This kind of problems rather needs the assistance of the system designer, as a design decision has to be made to resolve this conflicting requirements. Apart from this, it seems reasonable that applying a patch rule for one cliché reveals another model flaw. How to take these *side effects* into account is not in the focus for this publication and considered part of future work. We will refer to this in Section 9.

5 Security Hardening of UMLsec & BPMN Models: Case-Study

We have validated our approach in a number of applications. We considered the adherence to security properties in both UMLsec [25] and BPMN models, examining two kinds of model flaws dealing with UMLsec models and one example of compliance issues concerning BPMN models. To explain the approach, we present one example of each of the applications here. Please note that presenting an entire catalogue of all UMLsec stereotypes is beyond the scope of this publication. Thus, we chose a selection of patterns to show the application and realization of our approach in terms of a case study.

5.1 (In)Secure Links in UMLsec models

The running example as introduced in Figure 1 and used as part of the previous section relates to a security vulnerability that can occur in UMLsec models. We sum

up the essentials of this violation pattern and add some details to complete the description of the respective violation pattern.

Insecure Links concerns security vulnerabilities and corrections regarding the requirement «secure links». The running example considers a scenario which features a deployment diagram annotated with the UMLsec stereotype «secure links». It deals with the following scenario: Two nodes are connected via a communication path. There are at least two components, each deployed to one node,

and the two components are themselves connected by a dependency, which represents the logical communication carried out over the physical communication link that is specified between the two nodes. The type of the link and the link itself is defined by a communication path annotated by a UMLsec stereotype (here, an «Internet» connection).

For a given attacker (e.g. the *default* attacker), the UMLsec approach defines in which ways this kind of attacker can manipulate the given communication link in terms of the threats *read*, *delete* and *insert*, named after the corresponding security-relevant actions that can be performed regarding data transmitted over the given communication link. For example, the *default* attacker can read, delete and insert data communicated over a plain, unencrypted Internet link. Now, there may be security requirements concerning the communication of the respective software components, also given as a UMLsec stereotype attached to the communication dependency (here, the stereotype «high» means that highly security-critical data is to be communicated over the respective communication path).

The constraint of the UMLsec property «secure links» is fulfilled if the attack scenario derived from the type of communication link does not violate the security requirements demanded by the corresponding dependency. To demand that one or multiple security requirements (secrecy, integrity, *high*) shall be satisfied and certain threats shall be prevented, the relevant dependency has to be annotated with the respective requirements (cf. [25]). More formally, let n and m be two nodes connected via a communication link of type

$$l \in \{\text{«Internet»}, \text{«encrypted»}, \text{«LAN»}, \text{«wire»}\}.$$

Let there be one component per node with a communication dependency between them, marked with a security requirement $s \in \{\text{«secrecy»}, \text{«integrity»}, \text{«high»}\}$. If, for a given attacker type, the threat scenario derived from the type l of communication link violates this security requirement s, the UMLsec property «secure links» is violated. For example, given the communication link type $l = \text{«Internet»}$, the security property $s = \text{«secrecy»}$ and the *default* attacker, «secure links» is violated as explained above. We can derive a set of *critical configurations* (consisting of a communication link l, an attacker type A, and a dependency s) which lead to a violation of «secure links» (cf. Table 1).

Table 1. Critical configurations of A, d and l

Adversary A	Stereotype s	Stereotype l
default	high	Internet
”	”	encrypted
”	secrecy	Internet
”	integrity	Internet
insider	high	Internet
”	”	encrypted
”	”	LAN
”	”	wire
”	secrecy	Internet
”	”	encrypted
”	”	LAN
”	”	wire
”	integrity	Internet
”	”	encrypted
”	”	LAN
”	”	wire

We use the 4-phases-approach to enforce the requirements of «secure links». See [25] for the possible values for dependencies and applicable communication path types.

We formalize violation and correction of the security vulnerability *Insecure Links* by defining the following transformation rules.

First, this violation pattern features a test rule that checks the existence of a package annotated with «secure links» as shown in Figure 4.

For phase 2, we define one cliché as introduced in Figure 5. This cliché has a match whenever a critical configuration of two nodes and two deployed components with corresponding communication links and communication dependencies regarding Table 1 is found. Regarding the running example in Figure 1, the communication path type «Internet» is used in a critical configuration with the security requirements «secrecy» and «high».

Following the shown cliché, the dependency `dependency` and the communication path `link` are modeled. Both of these elements are annotated with a respective UMLsec stereotype. The `link` connects two nodes of type `Property`, which are again connected to a `Node`. Finally, both *Nodes* are connected to a `Component`, and both `Component` nodes are connected through a `Dependency`. Moreover, we make use of the cliché instance file for this violation pattern in a sense that the critical configurations shown in Table 1 are part of the cliché instance file and lead to 16 cliché instances during runtime. The node types of the nodes `stereo1` and `stereo2`, used as placeholders in the cliché, are then changed to other types as needed and defined by the cliché instance file.

Regarding the correction of this security vulnerability, we suggest three transformations. Firstly, following the paradigm of securing the environment, the corrective step consists of choosing a type for the communication

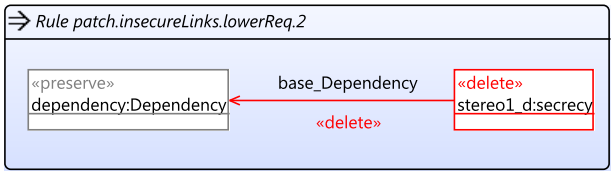

Fig. 9. Patch rule for correction of *insecure links* by deleting the security requirements of the dependency stereotype

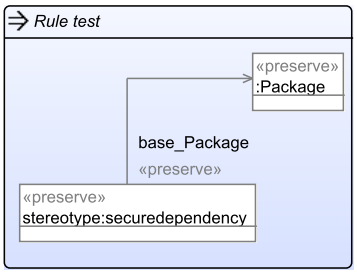

Fig. 10. Test rule for insecure dependency pattern

path that satisfies the requirements demanded by the respective dependency. The patch rule `patch.insecureLinks.secureEnv.1` intended for this is shown in Figure 8. The communication path is modeled as node of type `CommunicationPath`. An existing stereotype is modeled by the node `stereo2_d` and is deleted, while a new stereotype, represented by `stereo2_c`, is created, which tags the communication path then.

Regarding the running example, the corrected diagram is shown at the bottom part of Figure 7.

The goal of the given patch rule is to match against a given CommunicationPath `link` annotated with a stereotype `stereo2_d`. The stereotype `stereo2_d` has to be deleted and a new stereotype instance, `stereo2_c` will be created.

Normally one would need one patch rule for every given stereotype. For example, [25] initially defines four different link types. Thus, given a number of n stereotypes, the number of needed patch rules of this kind would be $n^2 - n$. Hence, in this case there would be a need for 12 patch rules differing only in the types of two nodes but not in their structure. To keep the number of needed elements in the transformation model low here, we support the dynamic adaption of node types during runtime. The configuration of which elements are to be adopted under which circumstances is defined through the cliché instance file.

As an alternative step, we define two rules for correcting the security vulnerability following the second paradigm (reducing the security requirements). The rule `patch.insecureLinks.lowerReq.1` reduces the security requirements by exchanging the dependency's stereotype by a stereotype with fewer requirements on the communication path. The rule is modeled in the same manner as `patch.insecureLinks.secureEnv.1`. For example, consider the default adversary, a communication path stereotyped «encrypted» and a dependency stereotype «high». After changing the dependency's stereotype to «secrecy», the requirements of «secure links» are clearly satisfied.

As a third (although drastic) correction alternative we define the rule `patch.insecureLinks.lowerReq.2` as shown in Figure 9. This rule simply removes the security requirement by deleting the stereotype of a given dependency. Although this rule looks trivial, there may be models in which this rule is the only possibility of correcting the model to fulfill «secure links» again. This especially applies when facing the adversary type *insider*. Since an insider has access to all types of communication paths (cf. [25]), it is not possible to satisfy any of the three security requirements of {«secrecy», «integrity», «high»}.

5.2 (In)Secure Dependency in UMLsec models

This violation pattern concerns the UMLsec property «secure dependency». This stereotype can be applied to class diagrams. The security requirement that is ensured by this stereotype is as follows: Security properties forced by a class C implementing an interface I are respected by a class D having a «call» or «send» dependency to I (see [25] for further details). To sum it up, the security requirement is violated if one of the following conditions applies: the tagged values of the classes C and D mismatch or one of the tags $t \in \{\text{secrecy}, \text{integrity}, \text{high}\}$ is used for messages as part of the «critical» tag of C, but the dependency is not tagged respectively.

Firstly, we define the test rule for this violation pattern. As shown in Figure 10, it consists of two nodes. This rule ensures that the instance model is a UML model containing a package that is annotated with the «secure dependency» stereotype.

To detect the actual model flaw, we formalized it using four clichés. The first one is `cliche.insecureDependency.criticalMismatch` and detects the mismatch of the tagged values of the «critical» tags. As shown in Figure 11, the cliché models the existence of the interface `i`, its implementation `c` and a class `d`. Due to technical limitations of Henshin concerning the analysis of data types of attributes, we assume the tagged values to be stored as part of a comment which is annotated to the class.

By making use of the Henshin API and Henshin's feature to use boolean expressions as part of transformation models that influence the matching (called *nested conditions*), we can detect a mismatch of the tagged values of `c` and `d` that resembles the violation of «secure dependency».

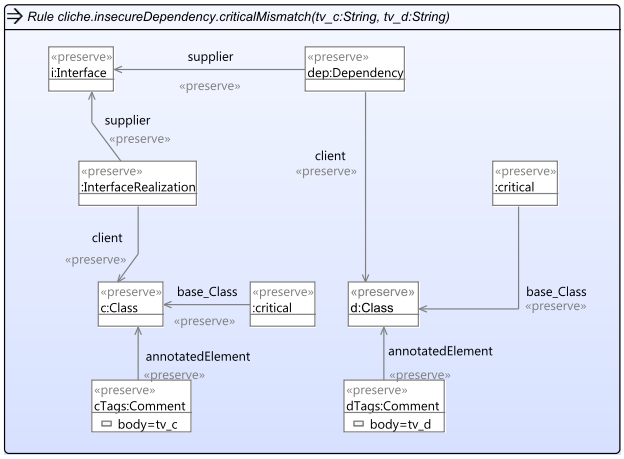

Fig. 11. Cliché for detecting a mismatch of tagged values of the two classes C and D

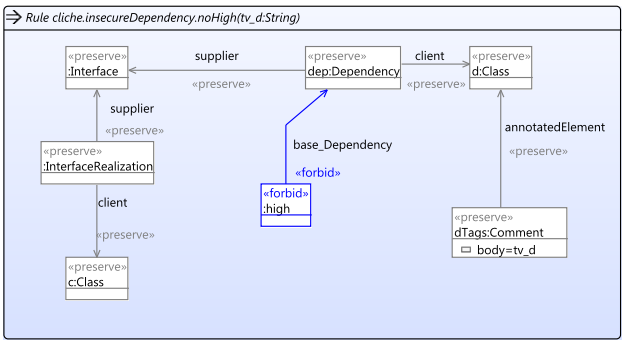

Fig. 12. Cliché detecting a missing *high* stereotype

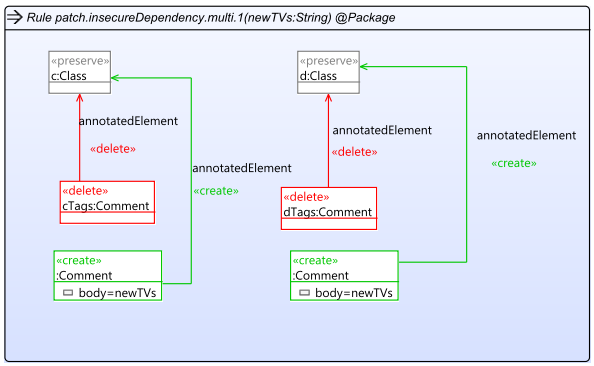

Fig. 13. Patch rule supporting multiple paradigms

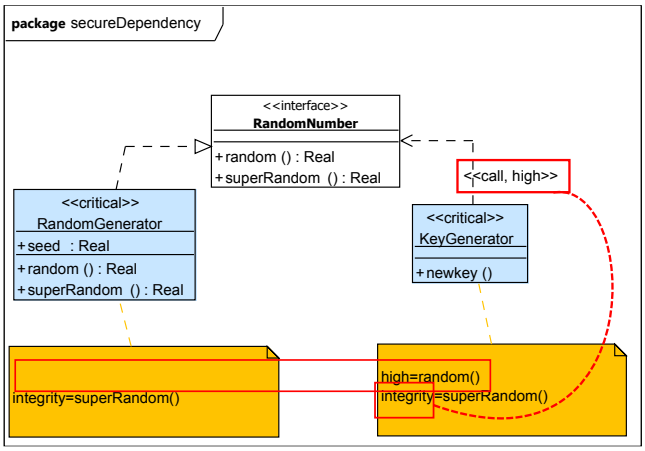

Fig. 14. Example of a flaw consisting of two clichés

The second possibility of violating the requirements of this stereotype concerns a missing tag $t \in \{\mathsf{secrecy}, \mathsf{integrity}, \mathsf{high}\}$. Figure 12 shows the cliché for detecting a missing *high* stereotype. This transformation rule depicts the classes `c` and `d` as well as the tagged values of `d` shown as comment element here. This rule internally requires the *high* tagged value to be set. To fulfill the requirements of «secure dependency», the dependency `dep` has to be tagged with «high» also. By using «forbid» for the stereotype *high*, we can detect the violation of this security requirement caused by the missing stereotype. For detecting the lack of a tag *secrecy* or *integrity*, two further clichés are defined analogously.

Regarding correction of the latter cliché, a missing tag $t \in \{\mathsf{secrecy}, \mathsf{integrity}, \mathsf{high}\}$ is compensated by adding the respective stereotype to the dependency. This can be done by a transformation rule similar to the one in Figure 8.

With respect to the mismatch of message usage in the «critical» tags of the classes C and D, more complex operations have to take place. Method names of the classes have to be parsed and a corrective set of tagged values following one of the two paradigms has to be compiled. These operations cannot easily be modeled by graph transformations. Hence, and to support future complex operations, we have implemented a mechanism called *preparation code* to run pure Java code prior to the application of phase 4 (cf. Figure 2). The technical details on this are presented in Section 7.

Figure 13 shows the rule `patch.insecureDependency.multi.1` we defined for these corrective actions. This rule matches against the two modeled classes `c` and `d`. The deleting and creating actions cause the deletion of the tagged values of classes `c` and `d` and the creation of new tagged values. The content of the tagged values to be created is given by the input parameter `newTVs` of this rule. In fact, Henshin is used for doing the model transformation in terms of matching and deleting as well as creating nodes here, but the work on parsing, comparing and correcting the methods used as part of the tagged values is done by running external Java code provided as part of the violation pattern.

An application of *Insecure Dependency* is depicted in Figure 14: it shows a fraction of a UML class diagram using the stereotype «secure dependency». Due to the fact that most UML modeling tools do not display tagged values of stereotypes in the diagram view, we show the tagged values of the stereotype «critical» as notes con-

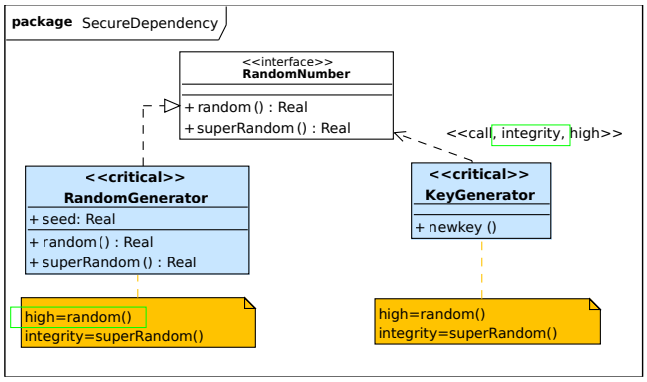

Fig. 15. Diagram with corrected model flaw

nected to the owning class for better readability. The model specifies a key generator which makes uses of a random number generator by calling an interface. The interface provides two different methods for generating random numbers: `random()` and `superRandom()`. There is one class implementing the interface that features a `seed` additionally. Following the requirements of «secure dependency», both classes are tagged with «critical». As the key generator calls methods of the interface, the dependency is tagged with a stereotype «call».

For example, the diagram in Figure 14 does not meet the requirements of «secure dependency»: Firstly, the occurrences of method names of the «critical»-tags of the classes `RandomGenerator` and `KeyGenerator` do not match. Secondly, the method `superRandom()` is part of the integrity tagged value, but the dependency between `KeyGenerator` and `RandomNumber` is missing «integrity».

To correct the model flaw, we apply our approach which leads to two model transformations. First, using `patch.insecureDependency.secureEnv.1` the missing stereotype «integrity» of the dependency between the interface and the class `KeyGenerator` is added. The second corrective action is caused by the use of `patch.insecureDependency.multi.1`. The external Java code which is executed prior to applying the actual model transformation accepts a parameter which encodes the corrective paradigm to use. In this case, one can think of securing the environment which leads to adding the `high` tagged value with the respective method to the «critical»-tag of `RandomGenerator` as well.

After applying these two corrective actions, the violation of the requirements of «secure dependency» is fixed and the resulting diagram is shown in Figure 15.

Note that the diagram shown in Figure 14 is an example of a logical cliché, because the definition of «secure dependency»'s security requirements consists of two independent requirements that both have to be fulfilled (cf. [25, chapter 4.1]). Thus, the security issue concerning the violation of «secure dependency» consists of at least two clichés which need to be treated independently. For that reason, we need to define two (actually four due to technical reasons) transformation rules as introduced above.

5.3 Information Leak in BPMN models

With this violation pattern, we look at a scenario concerning business processes and show that our approach can also be applied to further modeling languages.

This violation pattern represents an aspect of compliance requirements, in this case information flow between different entities. We assume that different entities, represented by different pools, own confidential information to work with during execution of the business process. The information is supposed not to leave a group of trustworthy entities. The compliance requirement is that there must not be an information flow from a trustworthy entity to an untrustworthy one. Since we wanted this pattern to be modeled using standard BPMN editors, we decided to realize this violation pattern avoiding modifications to the BPMN meta model. Hence, the property of trustworthiness is represented by a text annotation *confidential* here.

The model depicted in Figure 16 represents an example of a forbidden information flow: There are three subcontractors participating a common business process. `subcontractor A` and `subcontractor B` are trustworthy, so that the respective pools are annotated with *confidential*.

Regarding the application of the 4-phases-approach for this violation pattern, the test rule just has to ensure that a given instance model represents a BPMN model containing a *collaboration*. Hence, the respective rule just contains one node of type `Collaboration` and is designed analogously to the ones shown in Figure 4 and 10.

Regarding the formalization of the model flaw itself, our goal is to detect an information flow that begins at a pool annotated with *confidential* and targeting a pool that lacks this annotation. This corresponds to modeling one cliché at first hand. Due to the structure of the BPMN meta model (cf. [42]), there are many ways to model message flow between two pools that all have nontrivial differences on a structural level. Thus, we cannot model these differences by one cliché and supported by further definitions in the cliché instance file, but we rather have to model separate clichés. Note that this is an example of technical clichés we introduced in Section 3.

If edges between objects such as the connecting edge between a pool and a text annotation are undirected, they in fact are directed on the model level. We restricted us to a number of possible message flow connections between objects to concentrate on modeling the fundamental aspects of this violation pattern. Based on [42, Tab. 7.4], we used a number of fundamental combinations between begin and end of a message flow which led to Table 2.

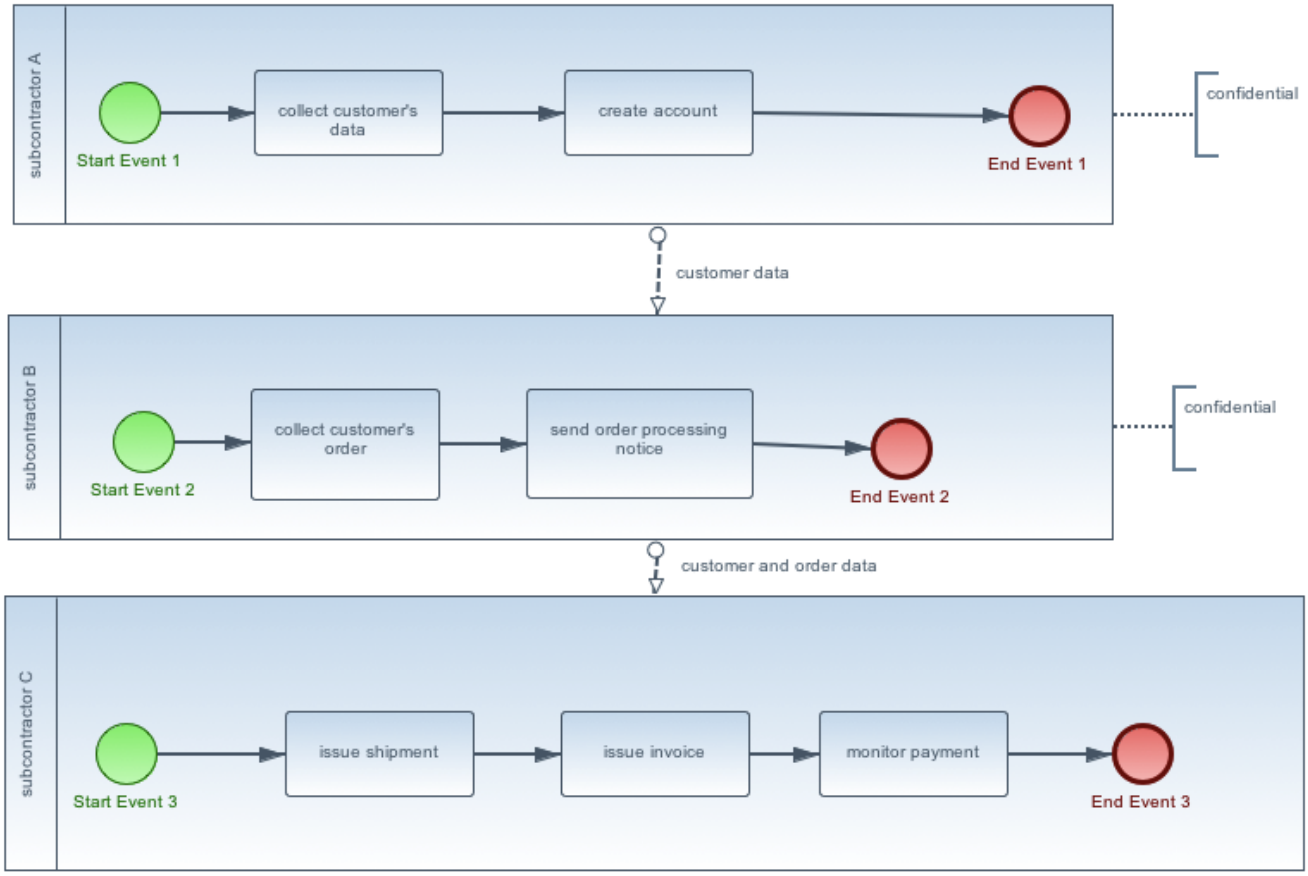

Fig. 16. Example BPMN model containing *information leak*

No.	Source	Target
1	Message End Event	Message Start Event
2	”	Pool
3	”	Task
4	Task	Message Start Event
5	”	Pool
6	”	Task
7	Pool	Message Start Event
8	”	Pool
9	”	Task

Table 2. Clichés modeled to detect *Information Leak*

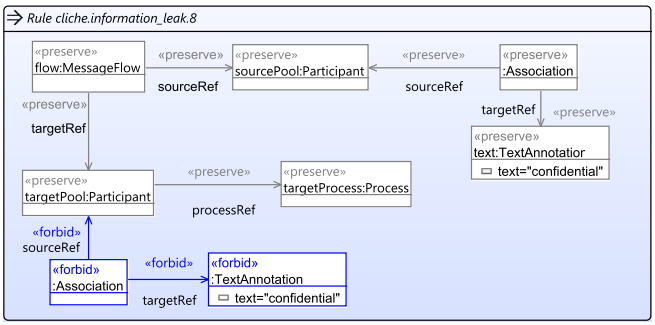

Fig. 17. Cliché for recognition an illegal message flow representing a compliance issue

For clarity, we focus on showing only one cliché here, as the further clichés only differ in the structure but content-wise represent the same facts.

Figure 17 shows `cliche.information_leak.8` that is used to detect a message flow beginning at a pool `sourcePool` annotated with `confidential` and ending at a pool `targetPool` lacking this annotation. The two pools are connected through a message flow `flow`. The violation of the compliance requirement is formalized by ensuring the existence of the text annotation at the source pool by using a « preserve » action. Furthermore, we force the absence of the annotation by modeling it using « forbid ». As the BPMN meta model allows the edge between a text annotation and a pool to be directed on the modeling level even if it is always undirected on the graphical level, we assume it to be directed from the pool to the text annotation.

Regarding the correction of this model flaw, we defined and formalized two patch rules. `patch.information_leak.secureEnv.1` (as shown in Figure 18) follows the paradigm of securing the environment. It assumes that the `confidential` annotations were set on

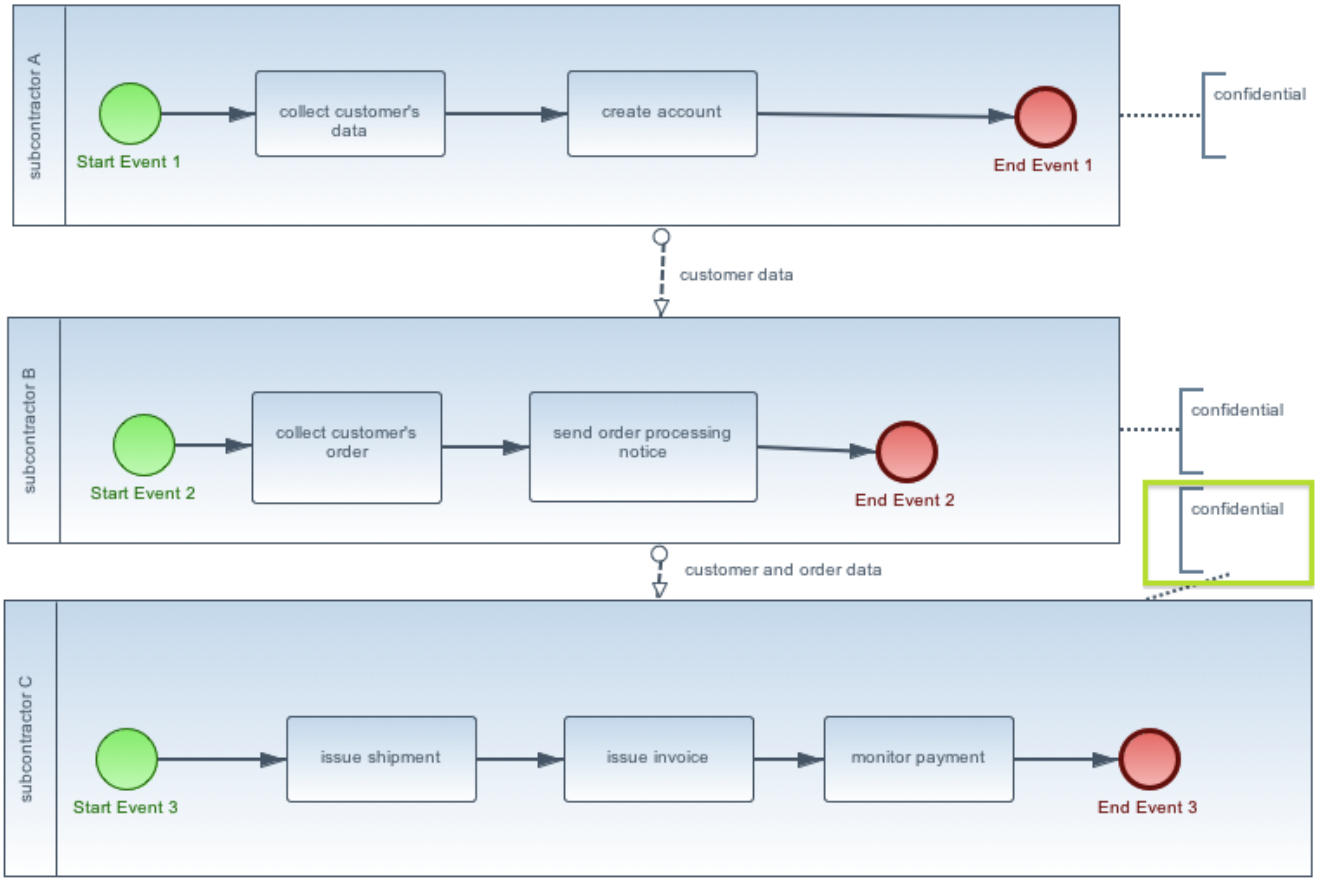

Fig. 19. Example BPMN model corrected using the 4-phases-approach

Fig. 18. Rule for correcting *Information Leak* by deleting a harmful message flow

purpose and thus considers the existing message flow between the two pools as the model flaw. Hence, this patch rule realizes the deletion of it. The design of this rule shows the advantage of applying the four phases in sequence and an information flow between the phases. Our approach makes use of the results of phase 3, where matching the cliché rules lead to the elements that form the model flaw in the actual instance model. Thus, we can model a small corrective rule just containing the most significant model elements that are needed to express the corrective action and use, in this case, the information, which specific message flow `flow` that connects the pools `sourcePool` and `targetPool` is to be deleted.

As a further possibility, following the paradigm of lowering the security requirements, an additional patch rule `patch.information_leak.lowerReq.1` assumes that it is important not to break the communication between the involved entities and just creates an additional `confidential` text annotation and adds it to the respective pool.

Figure 19 shows the resulting BPMN model. Assuming that subcontractors need to share confidential information among each other, application of `patch.information_leak.lowerReq.1` leads to adding a *confidential* annotation to `subcontractor C` here.

6 Application to Evolution

In the previous sections, we introduced our approach and presented how it can be generally applied to check if a given system undergoes security vulnerabilities with respect to given security requirements and, if so, recover certain security properties. Moreover, our approach can be used to support model evolution. Information about the evolution of a software model can assist the security analysis and support recovering a system's security properties. To make use of the fact that the model under consideration is the result of a system evolution, we

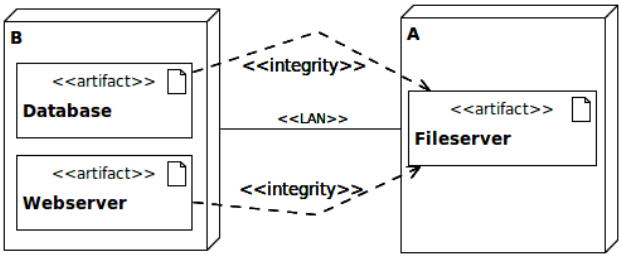

Fig. 20. Example system prior evolution

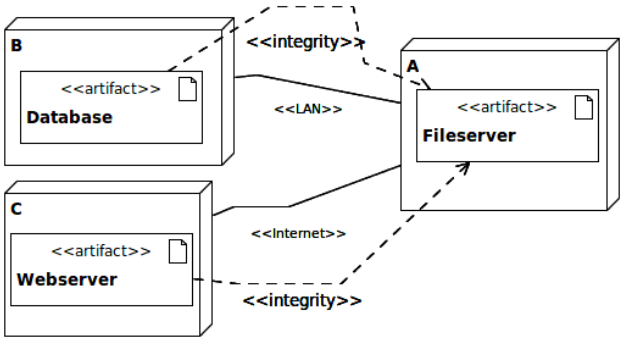

Fig. 21. Example of evolved system

should explicitly take the changes to the model under account. Therefore, we need to consider the difference between the model in its evolved state and in its original state.

To achieve this, we can build on a number of approaches available that realize the calculation of model differences, like EMF Compare [17] and SiDiff [52]. The difficulty with model differences as they are generated by common approaches is that they describe the model difference on a technical, fine grained level, tightly related to the underlying meta model. This is insufficient for deriving potential evolutions on the semantic level, because one single edit step like the addition of a communication path can be the result of a multitude of possible *atomic* evolutions [46]. Moreover, the structure of the meta model (e.g. in the case of UML) can lead to complex sequences of edit operations, as even the addition of one single communication path maps to the addition of at least three model elements from the meta model view (the communication path itself and two properties for referencing the association ends).

We illustrate this using an example. Figure 20 shows a UML deployment diagram featuring an information system that consists of three artifacts that are deployed on two nodes. The model is extended by UMLsec annotations of the security property secure links. In this example, the two nodes `A` and `B` are located in the same data-center so that they are connected via LAN. The security requirement is that the integrity of the data transferred between `A` and `B` is to be preserved. Assuming the default adversary, the requirements of «secure links» are satisfied.

Due to performance issues, a new server is introduced so that one of the artifacts deployed on node `B` can be handled by a separate physical system. From the model perspective, this change corresponds to one evolution: node `B` is *split* into two nodes. Figure 21 shows the system after the model evolution has taken place. Let us assume that the new node `C` is located in another data-center that has only an internet connection to external premises. Now, the requirements of «secure links» are violated (cf. Table 1). The evolution we show here is simple regarding its complexity, but is the result of a number of edit steps at the model level. Hence, we need to aggregate and generalize from the multitude of atomic edit operations (or *semantically lift* them) to evolution steps that are relevant for and can be processed by our approach (*semantic changes*).

Kehrer et al. propose an approach called SiLift [28] that features recognition of semantic model differences out of atomic edit operations. To calculate the initial atomic edit operations, SiLift can make use of arbitrary difference matching engines using an Java interface but can especially use EMF Compare and SiDiff (which is mainly developed at the same research group, cf. [28]). The semantic lifting is realized by the use of Henshin transformation rules. They can partly be generated automatically. Kehrer et al. define an EMF based meta model to describe model differences and extend it to support semantic lifting of these differences, so that SiLift uses the same base techniques as our approach does ([29]). Based on the semantic lifting approach, two extensions to our approach are conceivable.

Firstly, evolution of a model can be supported by *co-evolution*. This means that the evolution of the model itself triggers a second evolution to retain the system's security. Using the SiLift's meta model describing semantic change sets [29], it is possible to use the semantic changes between two revisions of a model as an input to the 4-phases-approach. SiLift then is applied prior to our approach. It calculates the semantic changes out of the two model revisions using definitions of semantic changes. The 4-phases-approach is then run with a Henshin transformation model that concerns flaw-related model differences (*diff*s) instead of model flaws. The cliché instance file of this pattern is e.g. used for hardening the model. This is realized by directly referencing appropriate patch rules of respective violation patterns. Based on the semantic changes that have been detected, a preselection of possible patches can be made or, under certain circumstances, applied automatically.

As an example, consider Figure 20 and assume an evolution such that a third artifact like `XML-Server`, also with a dependency from `Fileserver`, is deployed

on node B. It is unlikely that integrity should not be required for this new dependency.

A co-evolution that can be triggered in this case is the automatic addition of « integrity ». The semantic difference that can be obtained from the evolution in this case is an additional artifact getting deployed while a communication path and an artifact annotated with security requirements are already given. Hence, we can support that a model that met all security requirements prior to evolution does not get in an insecure state when evolving. To fully support this extension (e.g. referencing clichés of different violation patterns using one cliché instance file), our prototypical tool may have to be adapted slightly.

As a second extension, the user can be assisted when analyzing an evolved software model. In Section 4, we argued that there are at least two paradigms to make a system secure again. As we showed during the case study in Section 5, especially actions of the paradigm which *lowers the security* can lead to the removal of a security requirement annotation. This may be a justified step in some scenarios (e.g. when contradictory security properties are demanded), but it does not make sense if the action that initiated the model evolution actually was the addition of the respective requirement. In this case, evolution information can be used within our tool by marking inappropriate corrective steps or even disable them in the UI.

7 Model-based Security Hardening: Tool Support

The approach as introduced in the previous sections has been implemented as a prototypical tool. It is built upon the Eclipse platform and making use of EMF [15,16]. The implementation features a core application designed independently from modeling languages. It lists the detected flaws and lets the user select from correction alternatives and apply them.

The tool architecture is designed to provide flexible support for the modeling languages as well as additional violation patterns. No recompilation is needed when adding additional violation patterns. Furthermore, using a generic interface `PatchEngine` that is extended by modeling language specific implementations eases support of further modeling languages. Currently, we support BPMN and UML by making use of their EMF implementations.

Henshin does not feature powerful possibilities for passing arguments to transformation rules by now. Only the use of node's attributes values as parameters is supported. The user interface of Henshin does not support modeling to pass node types to be set to certain nodes via rule parameters. Hence, our tool realizes setting the types of transformation nodes' rules via Henshin API. The types of nodes are set by directly manipulating the transformation graph referring to the respective rule. The information of which node types are related to which cliché instances is provided by the cliché instance file.

The cliché instance file is realized as a XML file containing all metadata for mapping between clichés, cliché instances and patch rules. There is one cliché instance file per violation pattern and by externalizing this data from the program code, additional violation patterns can easily be added without need for recompiling the tool. Thus, the tool can be used as a stand-alone application which can be used and extended by its users.

As stated in Section 4, modeling of patch rules is possible using just a few nodes. Consider the transformation rule of Figure 8: There are only three nodes needed to delete a stereotype from a dependency and create a new one. To avoid the need for modeling a bigger part of the dependency's environment to clearly identify which actual objects in the instance model are to be altered, we make use of Henshin's feature *partial match* and our results of application of phase 3 shown in Figure 6. The output of phase 3 is a set of pointers into the instance model, showing all critical model parts. Particularly, we get a link to the concrete dependency of the instance model in this case. We can use this information to restrict the matching in phase 4: Instead of getting matches for all dependency objects of the instance model, we get only one match, because we force the matching to be only applied to the one respective dependency. Our tool uses the mechanism of partial match this way to ease the modeling of transformation rules.

As Henshin does not support storing metadata of rules or structuring them, we support this by defining a naming scheme for rule names and some node names as far as necessary for the approach. Rules to be used in phase 2 are named using the scheme `cliche.FLAW.NAME`. Note that name components written in capital letters are meant as placeholders to be filled with the corresponding values in the violation pattern. As we stated in Section 3, a security vulnerability can consist of numerous clichés, so that we use `FLAW` to identify the given vulnerability and `NAME` to identify the cliché. The names of the nodes contained in `cliche` rules can be chosen freely, but one has to keep in mind that if there is an intended use of matched nodes of a cliché rule inside a patch rule (for data sharing between phase 3 and 4), the names used for the corresponding nodes in the corresponding rules have to be the same. Considering naming conventions of node names of phase 3, the same regulations apply as to the cliché rules consequentially.

Note that for a given match, several patches can theoretically be applicable. For a given security vulnerability, there can be different strategies (paradigms) for treating the issue and *patch* the erroneous part of the instance model. The first paradigm supported by our approach considers lowering the security requirements. When a rule realizes this paradigm, it is resembled by `lowerReq` as part of its name. The second paradigm we

look at considers raising the security level of the environment, called `secureEnv` (secure the environment). As a third case discussed in this paper, we consider that a single patch rule supports different paradigms controlled by parameters. This is resembled by the string `multi` in the rule's name. This especially can be used when making use of the possibility to integrate program code into the approach using the feature *preparation code* implemented by us. We will discuss this in detail at the end of this section.

The naming scheme for patch rules is `patch.FLAW.PARADIGM.#`. As stated for cliché rules before, the part `FLAW` of the rule's name is used for stating the security vulnerability this rule refers to. As patch rules are not necessarily linked to a specific vulnerability and do not need an explicit name, patch rules of the same kind are just numbered consecutively using an integer in place of `#`. The part `PARADIGM` resembles the correction paradigm as stated above. Patch rules are the only rules to correct vulnerabilities in models.

Our tool implementation currently does not support automatic application of patch rules or all violation patterns in sequence. As we discussed in Section 2, automatic correction seems reasonable in some cases. Adding an automatic application of patch rules would be an interesting task for future work.

It is conceivable to sort the possible patch rules such that the ones securing the environment and affecting fewest number of model elements are listed first. The assumption is that the user added the security requirements with reason, so that it is more likely that the model should be altered so that it fulfills the requirements. Furthermore, we can assume that the changes applied to the model should be limited. As all work and transformation is directly carried out at the model level and we do not make use of intermediate representations or (textual) representations of the system under consideration, the user is able to see what changes are (proposed to be) made at any time with direct impact on the model elements. He is able to accept proposed patches and also can further adjust the model afterwards (or prior to applying the approach certainly).

At present, implementations `UMLPatchEngine` and `BPMNPatchEngine` exist. Thus, we support these two modeling languages at the moment, but adding support for additional modeling languages just requires the existence of a respective EMF based meta model and the implementation of the `PatchEngine` interface.

The tool developed by us mainly makes use of the API of Henshin. All features that make use of Henshin are realized with calls to the official API.

In some cases the plain application of transformation rules is not sufficient and more complex changes have to be applied to the instance model prior to apply phase 4. Firstly, Henshin provides aggregation of transformation rules such that rules can be executed in a loop, multiple transformation rules can be executed sequentially or even conditionally (cf. [3]).

Using this technique, it is possible to raise the complexity of problems that can be addressed within the approach. This aggregation can be used by our tool in principle but is not used for our current violation patterns.

Also, as an alternative to the current realization of the manipulation of cliché instances, there would be the possibility of using a declarative approach, which would tightly relate to the power of the Henshin language itself.

Moreover, we implemented an additional step called *preparation code* that takes place after application of phase 3. It represents a possibility of executing arbitrary Java code by calling a standardized interface (cf. Figure 2). Preparation code especially has access to the transformation rule and match of phase 3 and also can set parameters prior to the execution of the proposed patch rule of phase 4. Thus, it is possible to integrate the execution of application logic when the application of simple graph transformation rules seems insufficient. The tool then performs the following steps prior to application of a patch rule: The class referenced in the cliché instance file is instantiated and loaded using the Java reflection mechanism. This gives the possibility of extending the project with further violation patterns including preparation code without code changes or recompilation. Thus, complex operations that cannot be modeled with graph transformation rules can be performed this way.

7.1 Performance Measurement

We did a performance test to measure the time needed to perform the analysis of exemplary instance models. For each of the three violation patterns we presented in Section 5 we did a run of 10 analyses of an exemplary instance model having model flaws. We chose an arbitrary set of corrective actions to be applied to the instance model. Additionally, we also did a run of 10 analyses of exemplary instance models not containing any flaws. The times measured are for loading the instance model and transformation model, and applying the phases 1-4 of our approach. The test platform is a Intel Core i5 M540 system running at 2,53 GHz and equipped with 4 GB RAM, running Linux. The test results are shown in Table 3 and Table 4.

The time needed to execute the first run is significantly higher than in the other ones. As external data such as the needed modeling language meta model only has to be loaded once prior the execution of the first run and is resident then for the remaining ones, all runs following the first one profit from this initial effort.

This sort of *caching effect* especially comes into account when our approach has to be applied multiple times to the same instance model, as stated in Section 2.

violation pattern/run	1	2	3	4	5	6	7	8	9	10	∅
insecure links	10311	3793	2153	1969	1445	1321	1370	1401	1487	1682	**2693**
insecure dependency	7050	1610	1290	1355	1284	2710	1622	1453	1393	1311	**2108**
information leak	1055	1017	307	169	228	266	157	182	161	142	**368**

Table 3. Performance measurement of security hardening tool using models containing vulnerabilities (time values in ms)

violation pattern/run	1	2	3	4	5	6	7	8	9	10	∅
insecure links	6139	2105	1115	1424	1588	999	1002	1049	1039	1003	**1746**
insecure dependency	4895	1962	1481	1769	1386	1381	1374	1852	1393	1355	**1885**
information leak	1542	365	256	184	201	186	185	178	155	192	**344**

Table 4. Performance measurement of security hardening tool using models without vulnerabilities (time values in ms)

8 Related Work

Several approaches to model and analyze security requirements for software systems or business processes have been proposed recently, such as UMLsec [25]. Menzel et. al. present an approach based on an external data structure to extend BPMN for security requirements [40], [45] presents a BPMN security extension similar to UMLsec, while Rodriguez et. al. suggest a solution using the native possibility of BPMN to enhance artifacts. The latter approach leads to an extension with additional symbols and a consistent view diagrams and hence is similar to UMLsec [45]. Approaches for tool supported analysis of models have also been proposed. Examples of dedicated tools for analyzing UML models with respect to the fulfillment of requirements can be found in [33,37,31,13,47,5]. Awad [4] presented a similar approach for process models given in BPMN.

Tun et al. [51] did research on evolving security requirements. From the requirements engineering view, they propose a meta model to capture changes to security requirements of a system. By making use of formal methods, an event based, formal description of the system can be generated. Regarding analysis approaches based on model transformations, [9] uses graph grammars to formalize and validate OCL constraints. [48] incorporates transformation for model checking and can thus perform model checking for visual languages. [20] presents an approach for checking consistency rules based on graph grammars. [23] presents a family of languages to cover the life cycle of transformation development enabling the engineering of transformations. [2] presents an approach for studying the formal verification of model transformations. An overview on the state of the art in model smells and model refactoring can be found in [38]. All these approaches can be adapted to validate models concerning security requirements. However, to our knowledge the automatic correction of security violations has not been addressed yet.

Reder and Egyed [44] propose an approach for repairing model inconsistencies. The approach is build upon formalizing general requirements to the model as boolean expressions. Whenever these expressions do not evaluate to true regarding the model, this is encountered as an inconsistency and trees are calculated that resemble every feasible repair operation as sequences and alternatives. The approach does not take into account how inconsistencies may relate to possible repair operations and it does not consider the actual choice and application of repair operations.

Mens et al. [39] did research on refactoring models of software and they propose a tool supported approach to apply refactorings of software models using graph transformations of typed, attributed graphs. Their approach concerns the detection of, but, so far no solution to, conflicts of different atomic-like refactoring steps concerning mutual exclusion and sequential dependencies. In contrast to Mens et al., our approach tightly relates to modeling a flaw and its various aspects. Additionally, the corrective steps (patch rules) we propose always reference to a state of the model previously detected, i.e. matched. Nonetheless, the results in [39] concerning detection of dependencies and conflicts between different refactorings may be relevant for future work when we tend to implement more complex violation patterns and analyses.

Montrieux presented some thoughts on automated security hardening in [41]. He discusses problems occurring in modeling with UMLsec only exemplary and proposes for instance how to correct a misused «secure dependency» annotation. His work builds a basis for the formalized flaws and their correction alternatives. The work of Montrieux misses a systematic approach for automating corrections, because technical details on how to formalize model flaws and their corrections are not given. Neither is a proposal on how to implement the corrections.

Bergmann et al. propose a semiautomatic methodology that targets maintaining security requirements [7]. The approach is called *SecMER* and supported by a tool. The focus lies on managing security requirements under evolution aspects. To achieve this, requirements are captured in two different models with different expressive power. The approach mainly covers the elicitation

of requirements, an argument analysis and the evolution of requirements. The first two steps are supported by a graph based model (*SI**) and an argument analysis realized using a textual input and a graphical visualization. Evolution of requirements is handled by a declarative approach used to define patterns which are in turn used to assess the evolution's impact on the requirements. The SeCMER approach mainly addresses to model social dependencies as can be modeled by SI* (i.e. using assets, actors and actions) and an argument model (i.e. using claim, argument, fact, warrant). Hence, models of concrete systems are not considered. As a consequence, how to realize security properties in a software system or how to deal with evolution of security requirements of a given system is not addressed so far. In contrast to this, our approach is based on modeling languages to build (critical) software systems. This can be for example done with BPMN using workflow execution engines or using UMLsec with code generation techniques. Security requirements can be directly annotated to a system model, so that there is no need to treat requirements and the model itself in two different models. However, Bergmann et al. make use of a declarative model query language as part of their EMF-IncQuery approach [6] to express violations and security properties using patterns. This can be relevant for future work to extend our cliché mechanism by to support generation of clichés.

The idea behind our violation patterns is similar to the one of anti-patterns. Anti-patterns, which arose from patterns in software development and project management [10], are a concept which helps to explicitly formalize certain risks and problems of a design. An approach to detect and correct anti-patterns has been proposed by Llano and Pooley [35]. They specify anti-patterns in object-oriented development using a notation similar to UML. Based on this, they establish transformation rules for their correction. Although this approach enables the automation of detection and correction of object-oriented software defects, it is not directly applicable to security-related modeling flaws, since aspects of the domain-specific extensions of the modeling languages have to be considered.

While there exist repositories of anti-patterns concerning mistakes in software-development from an architectural viewpoint [14], anti-patterns for security and compliance risks have, to the best of our knowledge, not been discussed so far.

We realize our approach using Henshin, an EMF based model transformation tool built upon the Henshin transformation language [3]. The base of this language is the graph transformation system which has been used for the former EMF Tiger tool [8]. Due to its handling of EMF based models, the flaw detection and model correction can be applied easily to different types of models, since EMF based meta models for many modeling languages exist.

Kindler and Wagner propose a constructive algorithm to perform graph transformations using triple graph grammars [30]. There is tool support for triple graph grammars in the TGG interpreter tool [22]. It uses triple graph grammars and is also based on EMF. However, with TGG interpreter we were not able to define rules that remove model elements. Thus this tool is not suitable for implementing our approach. An overview on other transformation approaches can be found in [21].

To enhance our analysis of evolved models, we need to extract semantic evolution information out of the fine grained and technical edit operation description of common model difference approaches. To achieve this, we can make use of recent work of Kelter et al. [28]. Their tool supported approach *SiLift* realizes building the difference of two models as instances of an EMF meta model (i.e. a model in its original and evolved state) and calculating a basic model difference. Based on this, information about differences on a semantic level is extracted.

9 Conclusion & Outlook

When models for software get changed, this can lead to security weaknesses that are also part of the software system derived from those models. Hence, it is important to check the models with respect to security properties, and correct them respectively. In this paper we presented an approach which not only finds security weaknesses but can also correct them in a tool-supported way. The approach is based on patterns that can describe potential flaws, such as inconsistencies in security requirements. The tool can check whether these patterns arise in models and assist the user in correcting the security vulnerabilities.

Potential violations can be formalized in the patterns as well as the correction alternatives to fix these. It is based on graph transformation and can be applied to different types of models and violations. Although it is limited to violations that have decidable number of correction alternatives, it is well applicable to several modeling flaws that can occur, if the given modeling language has clear semantics. Especially in case of evolving models (e.g. during maintenance of long living software systems) vulnerabilities can arise and tool-assisted correction becomes necessary as it has been shown in the given examples. The approach has been implemented in a tool making use of the graph transformation language and framework Henshin. Hence, it supports the detection and correction of all security vulnerabilities which can be expressed within the Henshin specification language (directly or by approximation). It can also deal with specifications containing several UML models.

With respect to the question whether the application of a patch using our approach actually leads to the desired security property at the model level, we use the idea

of "translation validation" as proposed in [43], which means that instead of verifying the transformation as such, each resulting model is verified: After the patches have been applied, the user checks the resulting model with the automated tools available for checking UMLsec models for the included security properties. If this shows that the model is now secure, it means:

- the patches indeed did match (or the security weaknesses by fixed in passing by some other patch, but the important thing is that the model is now secure)
- the patches did resolve the vulnerabilities (otherwise the model would not be secure now)
- the patches did not introduce another security flaw concerning the security requirements contained in the UMLsec annotations (for the same reason as above).

Note that this does not prove that the patterns indeed only match actual vulnerabilities, or more generally, it could be that the model was changed more than strictly necessary to restore security. It would be very interesting future work to prove this as well.

Furthermore, when modeling a violation pattern, effort needs to be put into showing that a given pattern exactly matches the actual vulnerabilities. For now, this can be done by providing test cases that test all possible aspects of the respective pattern. As this is an effortful method, further research can investigate how advanced methods could be used to prove security properties of violation patterns, for example by incorporating formal methods (for restricted cases).

Support for evolving models is given, due to computation of differences. The difference information can be used in co-evolution manner, i.e. changes leading to security inconsistencies can be corrected automatically. Furthermore, interpretation of changes can lead to better tool-assistance when proposing correction alternatives to the user. For the security properties that can be expressed directly, the detection and correction approach is fully precisely; for properties that are expressed by approximation, this depends on the precision of this approximation. The feasibility of our approach was demonstrated in a case study where vulnerabilities in UMLsec and BPMN models were successfully detected and corrected.

Our ongoing work focuses on how to make it easier to express security vulnerabilities by making use of supporting rules to simplify the definition and application of patches, and how to generalize them, e.g. to treat directed edges equal to undirected ones if direction not considered. Particularly, the current realization of the manipulation of cliché instances could be replaced by a declarative approach, which would tightly relate to the power of the Henshin language itself.

As we stated at the end of Section 4, it seems conceivable that there are violation patterns that, when being applied to a instance model to correct it, may lead to *side effects* in a sense that a new flaw belonging to another violation pattern or even the same violation pattern is introduced to the instance model. One possible solution to address this problem could be to enhance the usage of clichés by extending the possibilities of cliché application by control their application using boolean formulas, for example. Firstly, Henshin offers different types of so-called transformation units that, e.g., can be used to execute one single transformation rule in a loop, multiple transformation rules sequentially, or conditionally. Secondly, the approach could be enhanced in a sense that the mapping contained in the cliché instance file holds these information. Moreover, it could be useful to alter the sequence of the approaches' steps and introduce a conditional loop. In this case, the termination problem has to be addressed first.

Beside the technical formalization of violation patterns and corrections, the preliminary tool support should be extended to a more user-friendly interface. The correction alternatives which are currently presented as textual descriptions could for instance be replaced by a graphical representation of the result if it was applied to the violated model.

Since we earlier drew the comparison with anti-patterns, it would be interesting to investigate whether actual anti-patterns for software/system models regarding security or compliance exist. This would require an empirical study analyzing models from different sources. However, since these models are typically very critical, obtaining the models to perform such a study may be challenging. Nonetheless, we are planning to extend our library of violation patterns and correction transformations for certain domains, e.g. compliance rules in business processes of insurances.

In [32] an extension to Henshin is presented that features another way of doing dynamic node typing. In contrast to our solution which adapts the model containing the transformation rules via the Henshin API at runtime, [32] proposes a wrapper model. To use dynamic typing, the transformation rules are modeled at a meta level (which however leads to an increase in the number of elements needed in a transformation rule). Integrating this model to lift the concept of cliché instances more to the user-side is nevertheless interesting future work.

Another point for future work could be to integrate the 4-phases-approach with our platform for risk and compliance checks, CARiSMA [11]. CARiSMA could be executed as a pre-step to the 4-phases-approach. This would make it possible to analyze the execution semantics of a model and create annotations at the model. Our tool could then make use of the annotations and detect and correct flaws that cannot be detected using static analysis.

References

1. Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD 2009*, pages 39–50. ACM, 2009.
2. Moussa Amrani, Levi Lucio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A tridimensional approach for studying the formal verification of model transformations. In *ICST 2012*, pages 921–928. IEEE, 2012.
3. Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MoDELS 2010*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.
4. Ahmed Awad. BPMN-Q: A Language to Query Business Processes. In *EMISA*, pages 115–128, 2007.
5. Federico Banti, Rosario Pugliese, and Francesco Tiezzi. An accessible verification environment for UML models of services. *J. Symb. Comput.*, 46(2):119–149, 2011.
6. Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over emf models. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2010.
7. Gábor Bergmann, Fabio Massacci, Federica Paci, TheinThan Tun, Dániel Varró, and Yijun Yu. A tool for managing evolving security requirements. In Selmin Nurcan, editor, *IS Olympics: Information Systems in a Diverse World*, volume 107 of *Lecture Notes in Business Information Processing*, pages 110–125. Springer Berlin Heidelberg, 2012.
8. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Lifting parallel graph transformation concepts to model transformation based on the Eclipse Modeling Framework. *Electronic Communications of the EASST*, 26, 2010.
9. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In *UML*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.
10. William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.
11. Carisma tool homepage, 2013. http://carisma.umlsec.de/.
12. CEA. Papyrus UML. http://www.papyrusuml.org.
13. María Victoria Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing. A heterogeneous approach to UML semantics. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 383–402. Springer, 2008.
14. Ward Cunningham et al. Portland pattern repository. http://c2.com/cgi/wiki?AntiPatternsCatalog.
15. Eclipse Foundation. Eclipse. http://www.eclipse.org/.
16. Eclipse Foundation. Eclipse modeling framework project (EMF). http://eclipse.org/modeling/emf/.
17. Eclipse Foundation. EMF Compare. http://www.eclipse.org/emf/compare/.
18. Eclipse Foundation. Henshin Project. http://www.eclipse.org/projects/project.php?id=modeling.emft.henshin.
19. H. Ehrig and H.J. Kreowski. *Automata, Languages, Development*, chapter Parallel graph grammars, pages 425–447. North Holland, 1976.
20. Gregor Engels, Reiko Heckel, and Jochen Malte Küster. The Consistency Workbench: A tool for consistency management in UML-based development. In *UML 2003*, volume 2863 of *LNCS*, pages 356–359. Springer, 2003.
21. Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.
22. Greenyer et al. TGG-Interpreter. http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/forschung/projekte/tgg-interpreter.html.
23. Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. Engineering model transformations with transML. *Software and System Modeling*, 12(3):555–577, 2013.
24. Jan Jürjens. Modelling audit security for smart-cart payment schemes with UML-SEC. In Michel Dupuy and Pierre Paradinas, editors, *Trusted Information: The New Decade Challenge, IFIP TC11 Sixteenth Annual Working Conference on Information Security (IFIP/Sec'01), June 11-13, 2001, Paris, France*, volume 193 of *IFIP Conference Proceedings*, pages 93–108. Kluwer, 2001.
25. Jan Jürjens. *Secure Systems Development with UML*. Springer Verlag, 2005.
26. Jan Jürjens and Guido Wimmel. Formally testing fail-safety of electronic purse protocols. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pages 408–411. IEEE Computer Society, 2001.
27. Jan Jürjens and Guido Wimmel. Security modelling for electronic commerce: The common electronic purse specifications. In Beat Schmid, Katarina Stanoevska-Slabeva, and Volker Tschammer, editors, *Towards The E-Society: E-Commerce, E-Business, and E-Government, The First IFIP Conference on E-Commerce, E-Business, E-Government (I3E 2001), October 3-5, Zürich, Switzerland*, volume 202 of *IFIP Conference Proceedings*, pages 489–505. Kluwer, 2001.
28. Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638 –641, sept. 2012.
29. Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *ASE*, pages 163–172, 2011.
30. E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, tr-ri-07-284, University of Paderborn, 2007.
31. Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In *FTRTFT*, pages 395–416, 2002.

32. Christian Krause, Johannes Dyck, and Holger Giese. Metamodel-specific coupled evolution based on dynamically typed graph transformations. In *ICMT 2013*, volume 7909 of *LNCS*, pages 76–91. Springer, 2013.
33. Diego Latella, István Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.
34. LBC. Topcased, the open-source toolkit for critical systems. http://www.topcased.org/.
35. Maria Teresa Llano and Rob Pooley. UML Specification and Correction of Object-Oriented Anti-patterns. In *ICSEA '09*, pages 39–44. IEEE Computer Society, 2009.
36. Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A UML-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441, 2002.
37. Fabio Massacci and Katsiaryna Naliuka. Towards practical security monitors of UML policies for mobile applications. In *POLICY*, page 278, 2007.
38. Tom Mens, Gabriele Taentzer, and Dirk Müller. Challenges in model refactoring. In *Proc. 1st Workshop on Refactoring Tools*. University of Berlin, 2007.
39. Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.
40. Michael Menzel, Ivonne Thomas, and Christoph Meinel. Security requirements specification in service-oriented business process management. In *ARES*, pages 41–48, 2009.
41. L. Montrieux. Implementation of access control using aspect-oriented programming. Master's thesis, Facults Universitaires Notre-Dame de la Paix, Namur, 2009.
42. Object Management Group. Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0/PDF.
43. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.
44. Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 220–229, New York, NY, USA, 2012. ACM.
45. Alfonso Rodríguez, Eduardo Fernández-Medina, and Mario Piattini. A BPMN extension for the modeling of security requirements in business processes. *IEICE Transactions*, 90-D(4):745–752, 2007.
46. T. Ruhroth, S. Gärtner, J. Bürger, J. Jürjens, and K. Schneider. Versioning and evolution requirements for model-based system development. In *Int. Workshop on Comparison and Versioning of Software Models (CVSM)*, 2014.
47. Rick Salay, Marsha Chechik, Steve M. Easterbrook, Zinovy Diskin, Pete McCormick, Shiva Nejati, Mehrdad Sabetzadeh, and Petcharat Viriyakattiyaporn. An Eclipse-based tool framework for software model management. In *OOPSLA workshop on Eclipse Technology eXchange (ETX 2007)*, pages 55–59. ACM, 2007.
48. Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In *UML 2003*, volume 2863 of *LNCS*, pages 92–95. Springer, 2003.
49. Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163, 1995.
50. Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996.
51. Thein Than Tun, Yijun Yu, Charles B. Haley, and Bashar Nuseibeh. Model-based argument analysis for evolving security requirements. In *SSIRI*, pages 88–97, 2010.
52. University of Siegen. SiDiff. http://www.sidiff.org.
53. L. Wendehals. Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets. Master's thesis, in German, Universität Paderborn, 2001.
54. Christian Wolter, Michael Menzel, and Christoph Meinel. Modelling security goals in business processes. In *Modellierung*, 2008.
55. Christian Wolter and Andreas Schaad. Modeling of task-based authorization constraints in BPMN. In *5th BPM*, pages 64–79, 2007.