D4.3 Tool Support for Evolution-Aware Security Checks and Monitor Generation

Sven Wenzel (TUD), Daniel Warzecha (TUD), Jan Jürjens (TUD)

Document information

Document Number	D4.3
Document Title	Tool Support for Evolution-Aware Security Checks and Monitor Generation
Version	1.0
Status	Final
Work Package	WP 4
Deliverable Type	Report and Prototype
Contractual Date of Delivery	31 January 2012
Actual Date of Delivery	31 January 2012
Responsible Unit	TUD
Contributors	TUD
Keyword List	Model-based verification, Security, Evolution, Monitoring
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.01	27.09.2011	Draft	S. Wenzel (TUD)	Outline of the deliverable
0.02	04.10.2011	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	Description of planned content
0.03	31.10.2011	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	1st draft of Chapters 2,3,4,5
0.04	16.11.2011	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	Update of Chapters 2,3,4,5
0.05	09.11.2011	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	Introduction, Conclusion, Appendix
0.06	02.12.2011	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	General update, Exec. Summary
0.07	16.12.2011	Draft	S. Wenzel (TUD), D. Warzecha (TUD), J. Jürjens (TUD)	General Update Version for scientific review
0.08	22.12.2011	Draft	M. Angeli (UNITN)	1st quality check
0.09	30.12.2011	Draft	F. Bouquet (INR), E. Chiarani (UNITN), O. Delande (THA), F. Innerhofer-Oberperfler (UIB), F. Paci (UNITN), S. Paul (THA)	1st scientific review

0.10	13.01.2012	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	General update w.r.t. review results
0.11	18.01.2012	Draft	S. Wenzel (TUD), D. Warzecha (TUD), J. Jürjens (TUD)	Completion for second reviews
0.12	20.01.2012	Draft	M. Angeli (UNITN)	2nd quality check
0.13	20.01.2012	Draft	F. Bouquet (INR), E. Chiarani (UNITN), O. Delande (THA), F. Innerhofer-Oberperfler (UIB), F. Paci (UNITN), S. Paul (THA)	2nd scientific review
0.14	26.01.2012	Draft	S. Wenzel (TUD), D. Warzecha (TUD)	General update w.r.t. 2nd review results
1.0	26.01.2012	Final	S. Wenzel (TUD), D. Warzecha (TUD), J. Jürjens (TUD)	Final Version

Index

Executive summary

This deliverable describes the results of Task T4.3 "*Extend existing security analysis tools with adaptive security*" and Task T4.4 "*Develop approach for security monitor generation for adaptive security*" in Year 3 of the SecureChange project. While the description of work (DoW) declared this deliverable (D4.3) to focus only on T4.4 we decided to include T4.3 as well, because it has been continued in Year 3 and its results would be unreported otherwise.

Deliverable 4.1 [42] and Deliverable 4.2 [43] introduced a notation for describing possible model evolutions that enables automated security checks for all possible evolution paths. In Year 2, Work Package 4 has been started to implement analysis tools to perform these checks (i.e. Task T4.3, M18-M30). This task has been continued. The first prototypes presented in D4.2 have been re-developed and ported to the Eclipse platform to better integrate with tools of other work packages. Furthermore, the notation for describing evolutions has been improved. It has also been decoupled from the security notation UMLsec to enable a usage in other scenarios as well. The new notation and the new tool are discussed in the first half of this deliverable.

The second part of this deliverable focuses on Task T4.4 "*Develop approach for security monitor generation for adaptive security*" (M31-M36). We present two monitoring approaches. The first approach is an in-line monitoring approach where the monitor is generated from UML state charts and integrated into Java software by instrumenting its byte code. The second approach realizes a monitor where runtime logs of executed software systems can be compared against activity diagrams describing the expected behaviour.

Tool-Level Integration

The re-development of our tool prototype from Year 2 resulted in the new tool framework CARiSMA which smoothly fits into the tool roadmap of the SecureChange project (see Figure 1).

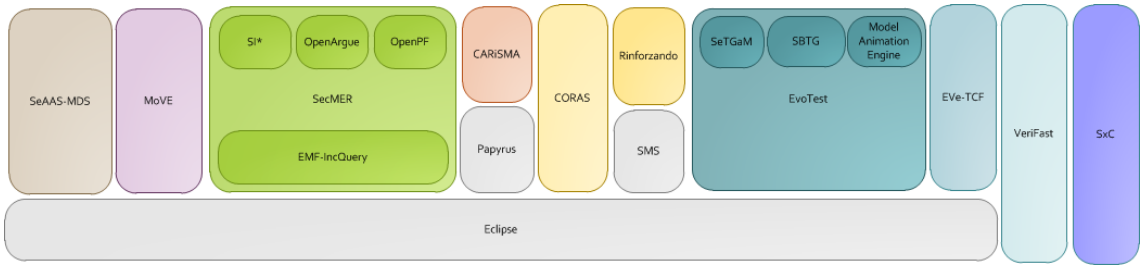

Figure 1: SecureChange tool roadmap

CARiSMA is based on Eclipse, and the Eclipse Modeling Framework (EMF), respectively. For modeling arbitrary modeling tools, e.g. Papyrus MDT, can be used. The use of Eclipse and EMF allows us to better integrate with other tools.

CARiSMA has been integrated with EvoTest/SeTGaM (WP7) for model-based testing

(see Section 4). It can benefit from our approach in two ways. On the one hand, the security requirements that are checked with CARiSMA can be exported in order to generate test cases. On the other hand, the verified evolutions of a model can be exported to analyze whether and which test cases have to be adapted. The integration has been explored within the POPS scenario. The general requirement considered is '*Specification Evolution*' and the common property is '*Life-cycle consistency*'.

In addition, the use of EMF enables the analysis of UML2 models which are used by other partners in the project, too. For example, Thales uses the Papyrus MDT modeling tool and UML2 models in the ATM case study. The compatibility between the tools enables an integration of the security analyses developed in WP4 with partners using the same modeling standards.

The monitoring approaches have not been integrated with other work packages, yet, because of their late position in project timeline. Their integration could be topic of future work.

1 Introduction

During Year 1 and Year 2, Work Package 4 worked in the security analysis of evolving models (Deliverable 4.1 [42] and Deliverable 4.2 [43]). This includes a notation for describing model evolutions, UMLseCh, and analysis tools to perform automated security checks. In the last year of the project (Year 3), this work has been continued. Task T4.3 "*Extend existing security analysis tools with adaptive security*", which started in Year 2, has been completed. The notation for describing model evolutions, UMLseCh, has been developed into the new *UMLchange* profile which fully reflects all aspects for security analysis of evolving models and beyond that supports other more general evolution-related analyses. Furthermore, the prototype tool delivered in Deliverable 4.2 has been re-developed into a new powerful analysis tool. It has been ported from the proprietary MDR library [37] onto the Eclipse platform with its Eclipse Modeling Framework (EMF), the basis for many open and commercial modeling tools. This way, the new tool can be easily integrated into existing development environments, which eased the integration with other partners of the SecureChange project. In the POPS case study, the integration between WP4 and WP7 was brought to tool level. And also in the ATM case study, Thales was able to use the new tool on Eclipse basis.

In Year 3, Work Package 4 also dealt with Task T4.4 "*Develop approach for security monitor generation for adaptive security*". The goal here is to supervise software at runtime in order to preserve security. After software has been modeled and models have been checked to be secure, the software is – following the model-driven engineering approach – generated. However, since software generation is often not 100% sufficient in industrial scenarios, it can happen, that the generated source code is manipulated subsequently. Hence, it is necessary to check, that the supplementary changes do not compromise the security properties that have earlier been verified on the models. In other words, it has to be ensured that the software conforms to the models. We have realized two alternative approaches to check the software's behavior at runtime via monitoring. One approach is based on in-line monitoring so that Java byte code is extended by routines that report each method call to a monitor. If the monitor recognizes invalid behavior, it can report an error or even stop the software execution. The other approach focuses on larger systems and checks whether the execution logs conform to the previously defined specification.

Chapters Walkthrough This deliverable can be divided into two parts. The first part deals with the security analysis of evolution. Chapter 2 introduces the new UMLchange profile that has been developed out of UMLseCh, which was described in the previous deliverables. It also discusses the grammar that can be used to describe new elements that are to be inserted into a model. The new analysis tool, CARiSMA, is described in Chapter 3. The chapter also shows details on the evolution support of the tool. The tool level integrations with model-based testing (WP7) is discussed separately in Chapter 4. The second part of this deliverable deals with the supervision of systems that might evolve. Therefore, we have developed two monitoring approaches. Chapter 5 introduces

an in-line monitoring approach that generates monitors from UML state charts. An approach for offline monitoring that compares execution logs with activity diagrams is shown in Chapter 6. Finally, we conclude our work and discuss future work in Chapter 7.

Acknowledgements We would like to thank Benjamin Berghoff, Lidiya Kaltchev, Johannes Kowald, Kubi Mensah, Yousefi Parvaneh, and Klaus Rudack, students of the TU Dortmund, for their contribution to the tool implementations of this deliverable. We also warmly thank Daniel Warzecha, former researcher at the TU Dortmund and now at Fraunhofer ISST, for his help in the tool implementation effort. Special thanks to our project partners, Michela Angeli, Fabrice Bouquet, Elisa Chiarani, Olivier Delande, Frank Innerhofer-Oberperfler, Federica Paci, and Stephane Paul for their comments on earlier versions of this document.

2 The UMLchange Notation

In Year 1 and 2 of the SecureChange project, UMLseCh has been developed as a notation for describing multiple possible evolution paths of a model [42, 43]. It was further shown how UMLseCh can be used to improve security analysis for evolving models as models do not have to be re-verified completely but the verification can be limited to the changes [26].

UMLseCh was an extension of the well-known UMLsec profile [24]. Hence, it was tightly bound to security engineering and security analysis. We have now extracted the evolution specific parts of UMLseCh and elaborated them into the UMLchange profile. The profile is thus no longer bound to security properties. Although UMLchange is still used in a security context and together with UMLsec, we decided for the separation of concerns. Security aspects and evolution aspects are now separated in two different profiles. However, the merger of both profiles will result in UMLseCh which was already presented, which can be expressed with the formula:

```
UMLseCh = UMLsec + UMLchange
```

Nonetheless, there have been various improvements of the stereotypes for describing evolution. Therefore, we use this chapter to introduce the new UMLchange profile and show how it can be used to describe different evolutions. In particular we will explain the grammar used to describe additive changes, since this has been omitted in Deliverable 4.2. Later in Section 3.1.2, we discuss a parser component that can analyze an annotated model and compute all possible evolutions out of it.

2.1 The Profile

Figure 2.1 shows the core elements of the UMLchange profile, i.e. the UMLchange stereotypes and their properties (also known as *tags*). The majority of the stereotypes (excluding *«old»* and *«keep»*) describe changes (i.e. the *change stereotypes*). The change stereotypes can be applied to any UML model element, as indicated by the extension relationships targeting meta class *Element*, the super class of all UML elements. Change stereotypes extend the abstract stereotype *Change*, which provides the basic tags {ref}, {ext} and {constraint}.

Figure 2.2 provides some examples for using UMLchange. Class *Redundant* will be deleted. Class *TooConcrete* is replaced with the Interface *IGeneral*. A new element *NewClass* is inserted into the main package. Furthermore, class *OuterClass* will be moved to package *Outside* and the class *FalseName* will be renamed to *CorrectName*.

The components of the profile are described in more detail below.

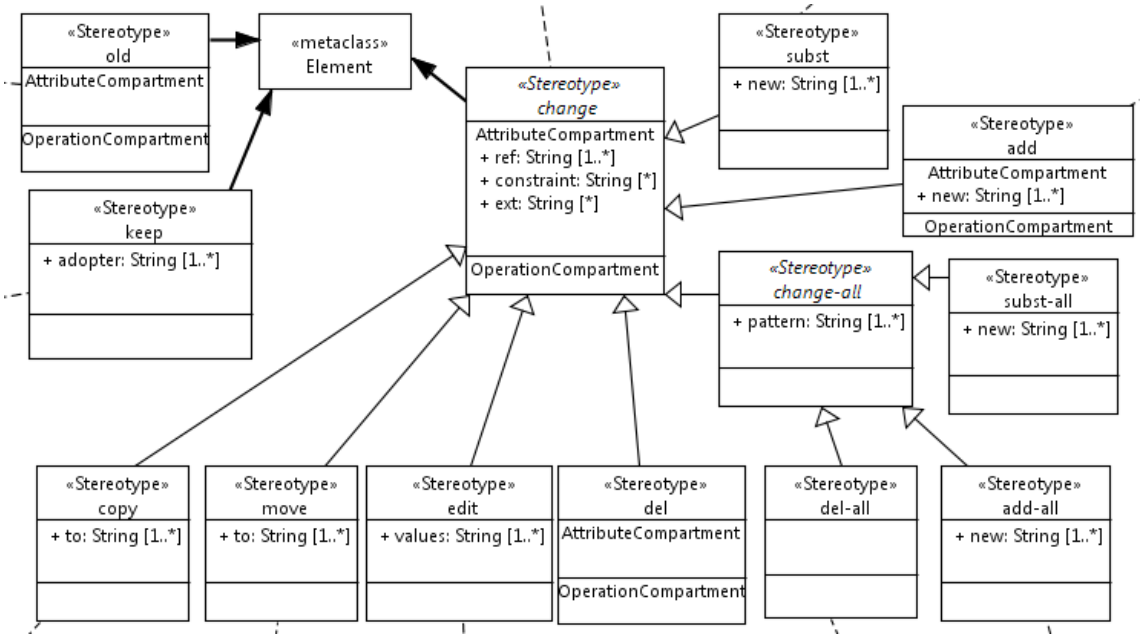

Figure 2.1: The UMLchange profile (core elements)

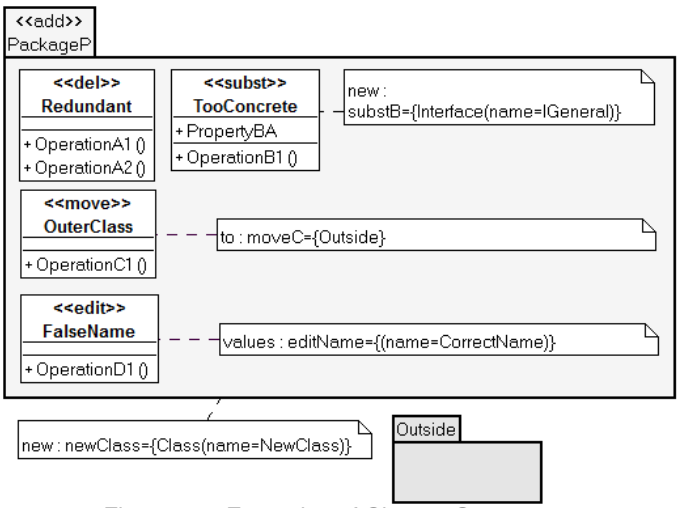

Figure 2.2: Examples of Change Stereotypes

2.1.1 Common Properties and Tags

Each UMLchange change description has the following tags: {ref},{ext} and {constraint}. To enable the description of multiple independent changes at a model element (e.g. two independent additions, each adding one operation to a class), each of these tags is multi-valued.

Every change has an ID so that it can be referenced by other changes. The tag {ref} contains the change IDs for each change at the stereotype application. Each application of a change stereotype must at least have one ID. These IDs should be unique in the model scope. The change IDs are used in constraints and in change stereotype tags to relate their entries to the corresponding change. Examples for IDs are

deleteTransition, *some_Change* and *add2Operations*.

Stereotypes cannot be applied to UML extension elements themselves. {ext} helps to describe changes of stereotype applications and their tagged values. Its format is

$$ChangeID = StereotypeName[.TagName]$$

If a change is directed at a model element, no {ext} entry is necessary. If the change target is the extension of an element, {ext} follows a convention of most UMLchange stereotype tag values. Each entry has to be prefixed with the id of the corresponding change so that entries in the value lists do not need to adhere to a certain order. If the target is a stereotype application, the name of the applied stereotype must be given. If a tagged value of a stereotype application is the target of the change, the tag name must the given in addition to that.

Every change may have constraints attached to it describing when the change may or may not take place. The corresponding tag {constraint} has the following format:

$$\begin{aligned} ChangeID = & AND(OtherChangeID)|NOT(OtherChangeID)| \\ & REQ(OtherChangeID)[,...] \end{aligned}$$

The obligatory change ID is followed by a constraint that either forces another change to be simultaneously applied (*AND(OtherChangeID)*), excludes a change from being applied simultaneously (*NOT(OtherChangeID)*) or forces a change to be applied after a certain other change (*REQ(OtherChangeID)*). A change may have more than one constraint. Each constraint can either be a separate {constraint} entry or in a comma-separated list of constraints as one entry. Contradicting constraints lead to not including any of the conflicting changes.[1]

2.1.2 «del» , «add» and «subst»

The stereotype *« del »* is used to delete the targeted model element. It recursively deletes all model elements owned by the targeted element. Any connecting model elements (e.g. associations) are also deleted to preserve the validity of the model. If the target of *« del »* is the multi-valued tagged value of a stereotype application, this stereotype deletes all values of the tag.

The stereotype *« add »* serves the purpose of describing additions to model elements. *« add »* has to be applied to the elements which will own the new elements. If the target of *« add »* is a stereotype application, multi-valued tags receive additional values. Additions to single-valued tags are treated as substituting the old tagged value with the new value.

Applying *« subst »* allows to describe the substitution of the targeted model element by one or more new model elements. The owner of the substitute element or elements is the parent of the substituted element. By substituting old elements, all of their contained elements are removed from the model, as well as all connection model elements. To

[1] The stereotype *ChangeSet* which was discussed in Deliverable D4.2 [43] is no longer contained in the profile. It was used to group changes that should be performed together which can now be enforced with the above-mentioned constraints.

prevent deleting contained elements, the stereotype *« keep »* must be applied accordingly (see 2.1.5). If tagged values are to be substituted, both single and multi-valued tags are completely substituted by the new values.

To describe the addition of new model elements or the substitutes of old elements, the stereotypes *« add »* and *« subst »* use expressions built with the UMLchange grammar. New elements are described by their metaclass names and pairs of keys and values. The new elements can be further defined by recursively describing contained elements. Changes on the grammar level are dependent on each other. Alternatives provide the ability to describe change variations. The elements described inside these alternatives are meant to be processed together.

The UMLchange grammar expressions are used in the {new} tag. Its format is

$$ChangeID = UMLchangeGrammarExpression$$

For example, to describe the addition of a new class named *someClass* to a package, *« add »* has to be applied to the package. The appropriate {new} entry is

$$someID = \{Class(name = someClass)\}$$

someID is the ID of the corresponding change. The UMLchange grammar is described in detail in 2.2.

In the example model in Figure 2.3, a new class named *ClassX* will be added to the main package. The class will have a String property named *someProperty*. The stereotype *« critical »* will be removed from *ClassA*. Finally, the class *Real* implementing the modelled interface will be substituted by a class named *Independent* containing some new void operation. As old connections are not kept, the new class will not need to implement the modelled interface.

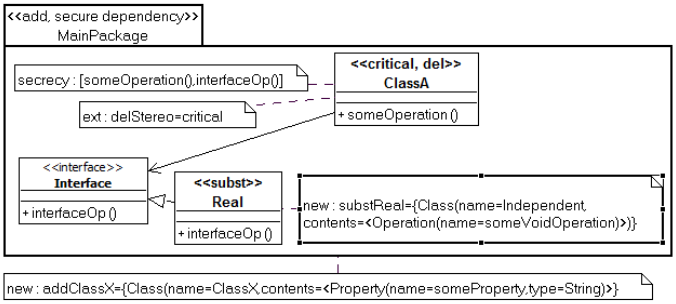

Figure 2.3: Adding, Deleting and Substituting Elements

2.1.3 «edit», «move» and «copy»

Minor changes can be expressed by applying *« edit »* to a model element. Its tag {values} has the format

$$ChangeID = \{(KeyValuePairs)\}[, ...]$$

KeyValuePairs represents the corresponding subset of the UMLchange grammar. The keys have to be valid attribute names of the targeted element. An example entry to

change the name of a class to *NewName* and its visibility to *private* would be

$$someID = \{(name = NewName, visibility = private)\}$$

As with the description of new model elements, {values} entries can describe alternative evolutions using the correct syntax. Editing stereotype applications is not possible, as changes would amount to redefining the stereotype instead of its application. Editing tagged values is analogous to substituting old with new tag values.

For structural changes, *« copy »* is used to indicate that the targeted model element is to be duplicated in one or more comma-separated namespaces given in the tag {to}. *« move »* works in the same vein, but removes the targeted model element from its original owner and only allows one target namespace. The format of {to} is

$$ChangeID = \{QualifiedNamespace[(KeyValuePairs)][, ...]\}[, ...]$$

The *QualifiedNamespace* needs to be qualified in so far that the uniqueness of the namespace in the model is guaranteed. The copied or moved model element in the target namespace can then be modified with *KeyValuePairs* using the same format as in the {values} tag of *« edit »*. Multiple destination namespaces must be comma-separated. An example for an entry in {to} is

$$\begin{aligned} copySomething = \{&mainPackage :: SubPackage(name = NewName),\\ &mainPackage :: SubPackage(name = OtherNewName)\} \end{aligned}$$

This describes two copies of the targeted model element to the same *SubPackage*, renaming each one in the process. For obvious reasons it is not allowed to copy a model element to the same namespace as the source element without changing the name of the copied element.

If a stereotype application is the target, all of its tagged values are also copied to the targeted element. If the targeted element already has the stereotype applied to it, all tagged values are replaced in the process. It is not allowed to change the name of the stereotype, as this would change the applied stereotype itself.

In the example model (see Figure 2.4), class *ClassE* has *« critical »* applied to it. *« edit »* is applied to change the value of {high} to only contain operationA. *ClassM* is moved alternatively to either package *TargetP* or *TargetP2*. Finally, *ClassC* is copied to both *TargetP* and *TargetP2*.

2.1.4 «del-all», «add-all» and «subst-all»

These three stereotypes allow to describe changes to multiple model elements. They are applied to the namespace in which the changes are to take place. Apart from the {new} tag, which works the same way as with the namesakes of the stereotypes, the {pattern} tag allows to identify the model elements in the namespace affected by the described

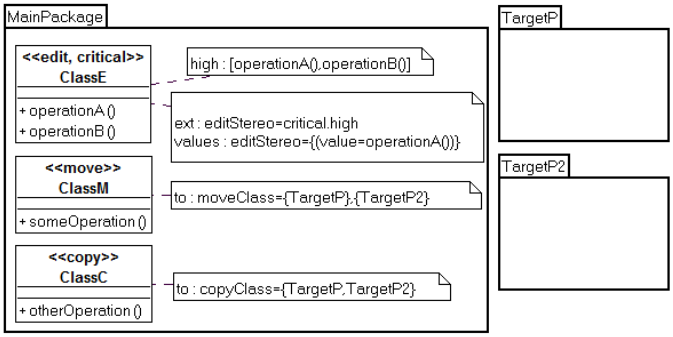

Figure 2.4: Editing, Moving and Copying Elements

change. The format of {pattern} is

$$ChangeID = TargetedElementsPattern$$

After the change ID, the *TargetedElementsPattern* uses the same syntax as the Simple Element Descriptions. First the metaclass of the targeted elements must be given. For example, if the given metaclass is *Class*, then the changes would affect all classes in the namespace marked with the *-all stereotype. Following the metaclass, the affected elements can be further filtered by giving key value pairs defining certain attributes that the affected elements must possess. For example, to affect all dependencies having a certain supplier, the entry would be *Dependency(supplier=somePackage::certainSupplier)*. Some further examples for entries in {pattern} are

- Dependency(supplier=somePackage::certainSupplier, contents=<Stereotype(name=secrecy)>)
  - all dependencies that have the supplier somePackage::certainSupplier and the stereotype application of *« secrecy »*
- Action(contents=<Stereotype>)
  - all stereotyped actions

In the example model (see Figure 2.5), all classes in package *PackageA* with the property *bitrate* will receive a new property named *length*. Furthermore, all operations named *setInput2* in *PackageB* will be removed from their respective classes.

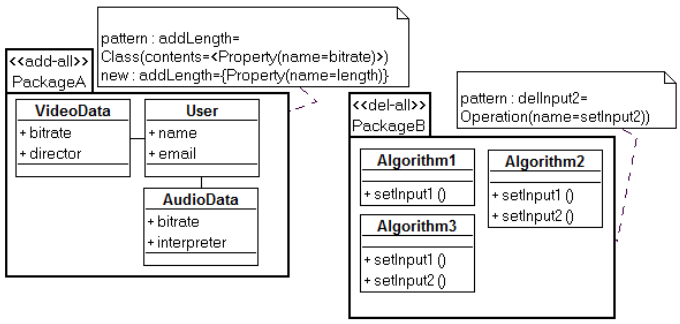

Figure 2.5: Changing Element Sets with Patterns

2.1.5 Describing Complex Changes Using «keep» and «old»

Describing complex changes with the UMLchange grammar can lead to long-winded grammar expressions. To provide a simpler method for modelling complex changes, UMLchange provides the ability to reference changes modelled in a namespace in the original model. The namespace containing the new model elements can be placed anywhere in the model.

To connect the new model elements to the correct owner in the original model, the owner relation has to be modelled in the namespace by modelling the owners of the new elements. However, it is not necessary to completely re-model the owning elements. For example, one would not need to re-model a class with all of its operations and attributes to model two new operations for it. Instead it is sufficient to just model the owning class and its name, as long as the class can be uniquely identified within the original model. To support this method, *« old »* is used to mark those incompletely modelled references to the original model.

In addition to that, *« keep »* is used to mark model elements that would otherwise be removed in the process of substituting a model element. Its tag {adopter} has the format

$$ChangeID = \{AdoptingElementDescription\}[, ...]$$

As each alternative description in {new} could describe different new elements, an entry in {adopter} must describe the receiving element for each alternative in {new}. If an alternative of {new} should not receive the element, its corresponding alternative in {adopter} is left empty. If, after a certain point, the remaining alternatives don't receive the element, then the entries can be omitted. Transferring model elements using *« keep »* is only supported when complex namespaces are used to describe the new model elements.

The *AdoptingElementDescription* uses the same syntax as the simple element descriptions (see Section 2.2). For example, let *« subst »* be applied to a class. Its {new} entry

$$substClass = \{@newElements\}, \{@otherVersion)\}$$

means that the old class is either substituted by the elements in the namespace *newElements* or alternatively by those elements in *otherVersion*. To keep some old contained element of the old class, it has to be marked with *« keep »*. If, for example, an old element is to be left out in the first alternative and should be adopted by a class NewClass when using the second alternative, the appropriate entry for {adopter} is

$$substClass = \{\}, \{Class(name = NewClass)\}$$

In the example model (see Figure 2.6), *ClassA* will be substituted by two new classes, *NewClassA* and *OtherNewClass*. The old operation *keptOperation* will be adopted by *NewClassA*. In addition to that, several new attributes and operations will be added to *ClassB*, as modelled in the namespace *NewContents*.

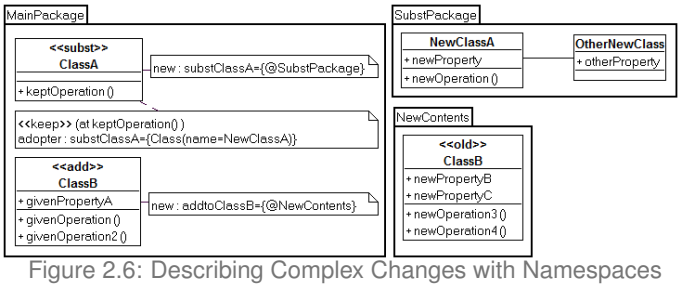

Figure 2.6: Describing Complex Changes with Namespaces

2.2 The Grammar

The UMLchange grammar can be used to describe changes adding new model elements to existing elements. Each change consists of one or more comma-separated descriptions of alternative evolutions. The format for these alternatives is:

$$\{Description\}$$

The *description* can be either a series of comma-separated simple element descriptions depicting new model elements or the single reference of a namespace wherein the additions to the model are shown.

An example for the UMLchange grammar is

$$\{Class(name = NewClass), Class(name = OtherNewClass, visibility = private)\},$$
$$\{@addClasses\}$$

This example poses two alternative evolutions. The first adds two classes named NewClass and OtherNewClass, of which the second receives a private visibility. The second alternative references a namespace *addClasses* in the model. The referenced namespace contains new model elements to be added to the original model, by either adding to old model elements using *« old »* or substituting model elements while keeping some of their contents using *« keep »*.

2.2.1 Simple Element Descriptions

Simple element descriptions (SED) succinctly describe a UML model element. The format of an SED is:

$$Metaclass(KeyValuePairs)$$

Each SED starts with the *metaclass* name of the new element. Every UML metaclass of an actual non-abstract model element can be used. Apart from that, simple comma-separated *key-value pairs* can be given to set the properties of the new model elements, ranging from common properties (e.g. name) to connection-specific ones (e.g. source and target for an association). The format of a key-value pair is:

$$key = value$$

When setting values for properties which reference other model elements in the original model, a sufficiently qualified string representation of the referenced model element has to be given. In the example model (see Figure 2.7), two different classes of the same name *WantedClass* exist in two different packages *SuperPackage* and *SubPackage*. To reference a class, the containing package namespace has to be incorporated into the attribute value. However, it is not necessary to add the model namespace to the reference, as the containing package namespace is sufficient to identify the referenced class.

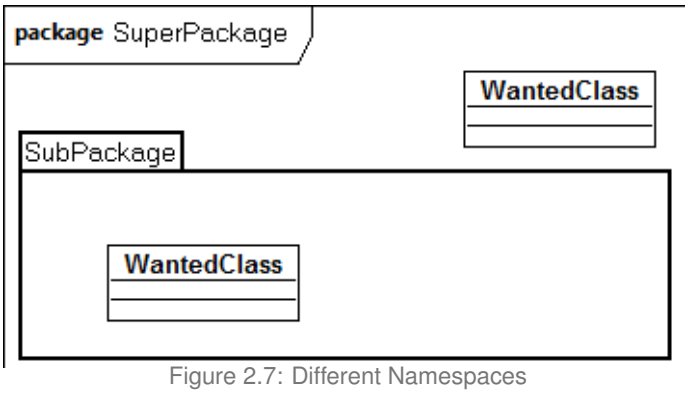

Figure 2.7: Different Namespaces

Table 2.1 shows some metaclasses, their corresponding keys, their value type and a description. The value type of a key may be a String, an element of a given enumeration, or the adequately qualified reference to a model element.

Metaclass	**Key(s)**	**Type**	**Description**
all named elements	name	String	model element name
Property (Tagged Value)	value	String, Reference	new tagged value
Class	visibility	Enumeration	public, private, protected or package
Association	sourceEndKind, targetEndKind	Enumeration	composite, shared or none
	source, target	Reference	qualified classifier
Dependency	supplier, client	Reference	qualified classifier

Table 2.1: Excerpt of Metaclasses, Keys and Values

Apart from describing the new model element itself, an additional optional key named contents with the format

$$contents =< SimpleElementDescriptions >$$

provides the means to describe further new model elements that are contained in the new element, e.g. an operation to be owned by a new class. The usage of the contents key is not restricted by a maximum depth.

2.2.2 Referencing Namespaces

To avoid long descriptions of complex additions, the UMLchange grammar allows to reference namespaces containing the new elements. The syntax for namespace referencing is

$$@NamespaceName$$

The namespaces referenced by the *namespace name* must be placed in the scope of the original model, but it is not necessary to place them in the same scope where the changes will take place. Connecting the new elements of the namespace to the original model is accomplished by modeling part of the target model element and application of the *« old »* stereotype (see Section 2.1.5).

2.2.3 Other Uses of the Grammar

Other stereotypes of the UMLchange profile use subsets of the UMLchange grammar to provide a consistent syntax (see table 2.2).

Stereotype	Tag	Subset	Example
« edit »	values	KeyValuePairs	(name=NewName, visibility=private)
« copy », *« move »*	to		
« copy », *« move »*	to	QualifiedNamespace	SomePackage:: SubPackage::TargetClass
« del-all », *« add-all »*, *« subst-all »*	pattern	SimpleElement-Description	Class(name=SomeClass, contents=<Stereotype(name=UMLsec::critical)>)
« adopter »	adopter		

Table 2.2: Other Uses of the UMLchange Grammar

{values} of stereotype *« edit »* uses the same key-value pairs to describe changes to model element attributes, as does {to} of *« copy »* and *« move »*. The target of the copy or move operation is an adequately qualified namespace equivalent to the model element references used in simple element descriptions. The descriptions of the targeted elements of the *-all stereotypes using {pattern} are the grammar's simple element descriptions, as is the target element description of {adopter}.

One of the features of the new CARiSMA tool is its ability to parse the different elements of the UMLchange grammar and create appropriate change structures. These can then be used to analyse the possible evolutions on a given model.

Due to the migration of the profile UMLseCh to UMLchange, the UMLsec analysis tool and in particular its evolution aware parts have been adapted. The new tool is discussed in the following chapter.

3 Tool Support

For all approaches towards security enhancement, it is necessary that they are applicable in practice and thus supported by tools. In Year 1 and 2 of the project, we have created formal foundations for security analyses of evolving models. Furthermore, as part of Deliverable 4.2 [43], we have presented a first prototype of an analysis tool that can perform such evolution-aware security analyses. It was implemented on the basis on the UMLsec tool, which implements already a large number of security analyses (without evolution support).

It was a problem, that the UMLsec tool was already ten years old and built on the basis of the MDR library [37], which is since 2003 no longer maintained. The extensions of the UMLsec tool were thus only realizable with enormous effort. In addition, the integration with project partners was hampered, since the tool was limited to UML 1.4 models created with ArgoUML [46], which rarely find application in industrial cases. Especially the missing support of UML 2.x was not adequate.

As a consequence, we decided to re-develop the core of the UMLsec tool into a new tool called CARiSMA. The new tool is built on the basis of Eclipse and the Eclipse Modeling Framework (see Section 3.1) and thus supports UML 2.x by using the Eclipse UML2 Plugins which is starting to become a de-facto standard for many commercial and/or open-source modeling tools such as Papyrus MDT [14], TOPCASED [47], MagicDraw [23], and IBM Rational Software Architect [22]. Furthermore, as an Eclipse plugin, the new analysis tool can be smoothly integrated into the modeling tools or other tools which are based on Eclipse. In addition, the new architecture allows users to extend the tool, as we will show in what follows.

In Section 3.2 we briefly discuss the tool validation. We conclude with a preview on future work in Section 3.3.

3.1 CARiSMA Architecture

CARiSMA [9] is implemented as an Eclipse [12] plug-in. Since version 3.0, the architecture of the integrated development environment (IDE) Eclipse is based on the Equinox kernel. This kernel was developed as a Java framework, and implements the OSGi core specification, a hardware independent and dynamic software platform enabling application management by the component model. As a result, Eclipse now exists only as a core, which reloads functions in the form of plug-ins.

CARiSMA is fully integrated into the Eclipse GUI (see Figure 3.1). It provides both an Analysis Wizard (1) for model analysis creation and an Analysis Editor GUI extension (2) to modify the settings of existing analyses. Furthermore, the results of an executed analysis are displayed in the Analysis Results view (3).

Model access for analysis is provided via the Eclipse Modeling Framework (EMF) [11],

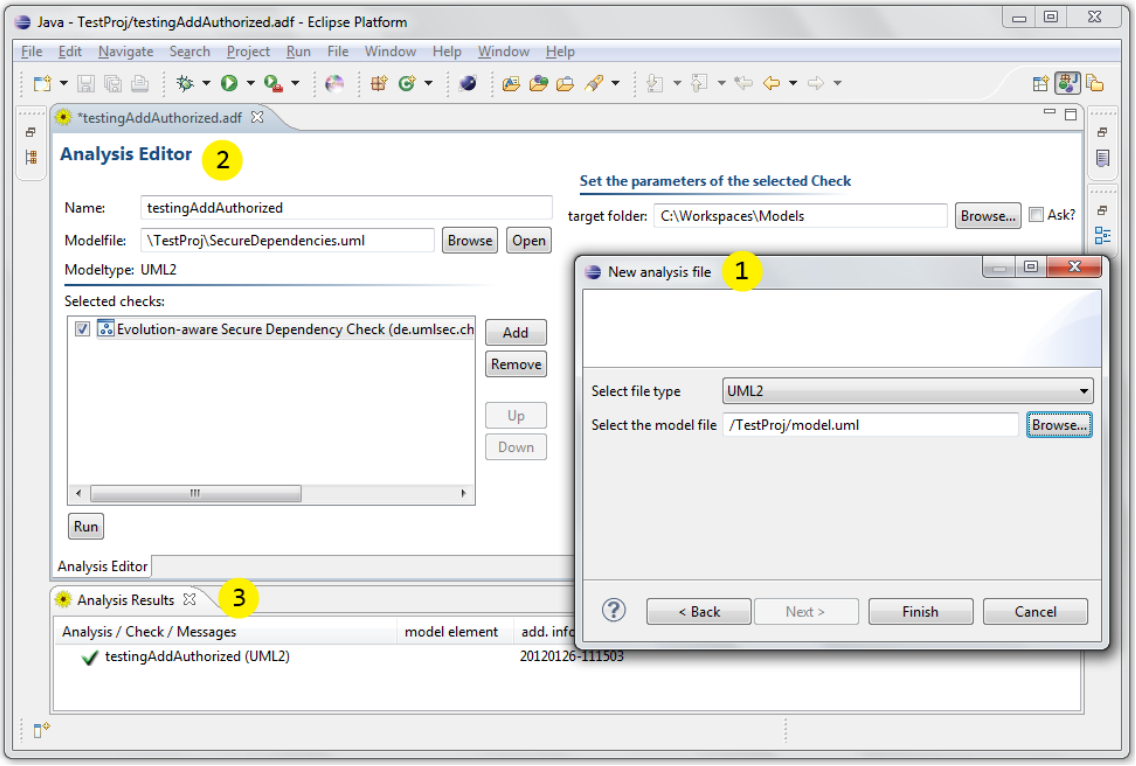

Figure 3.1: CARiSMA GUI Elements

which implements, among other tools, the OMG Meta Object Facility (MOF) specification [19]. This implementation, called Ecore, is used as a base for the UML2 metamodel implementation also provided by the Eclipse Foundation [15]. UML2 provides support for access and modification of UML 2.x models using the widely accepted `.uml` XML file format.

Like Eclipse, CARiSMA has been implemented as a plug-in based architecture. Using the modularity provided by this method, CARiSMA is distributed as several packages, of which the Core package includes the main functionality. Furthermore, CARiSMA uses extension points and extensions [10], there by facilitating the contribution of functionality of other plug-ins (see sub section 3.1.1). Part of the existing plug-ins can be seen in Figure 3.2.

Support for different modelling languages can be added by installing the corresponding modeltype packages. UMLsec and UMLchange are optional packages to incorporate support for the respective UML profiles. The checks used in a CARiSMA analysis can be installed separately to enable sleek installations.

3.1.1 Extending CARiSMA

To provide a new check for a CARiSMA analysis, a template plug-in project can be generated using the appropriate wizard. If required, the wizard generates a preference page for the check. This page can then be used to set possible global properties.

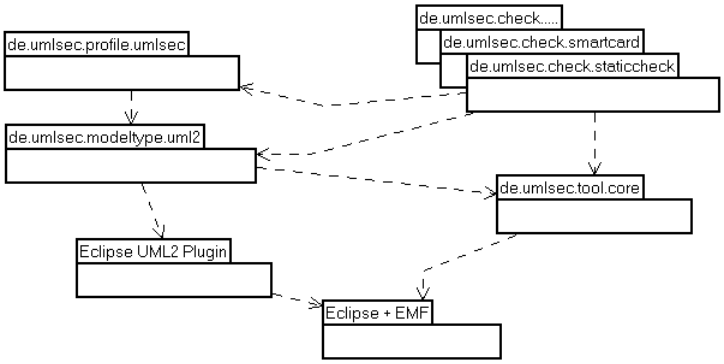

Figure 3.2: Overview of the CARiSMA architecture

A CARiSMA check extends the de.umlsec.tool.analysischeck extension point, which provides identification and description properties for the check. Additionally, parameters for the check can be defined using the extension point. A check parameter can be one of the primitive types String, Integer and Boolean, and file system references to input or output files and folders. Each parameter can also be marked optional. After defining the check and its parameters, it is added to the list of available checks and can be added to an analysis in the Analysis Editor. The check's parameters can then be either pre-set in the editor or at the appropriate time during the analysis.

The entry point for a CARiSMA check is the *perform* method, which receives the parameters for the given check and an AnalysisHost interface. This interface is used for generating entries for the Analysis Results view and access to the analysed model. The two interfaces are shown in Figure 3.3.

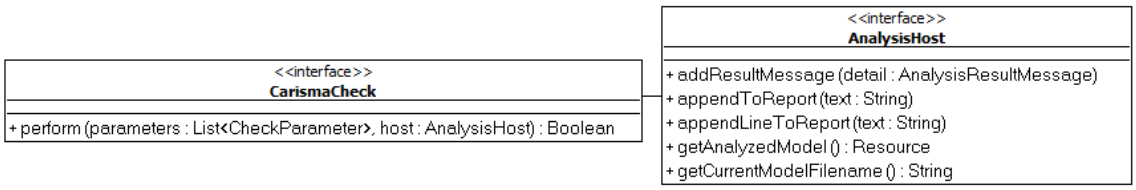

Figure 3.3: Check and AnalysisHost Interfaces

A reference to the analysed model is provided by the AnalysisHost interface. The methods to traverse and analyse the given model are supplied on a more abstract level by the Ecore metamodel implementation. In the case of UML 2.x models, the Eclipse UML2 metamodel implementation offers easier access to model elements and their properties. Furthermore, CARiSMA or rather the package de.umlsec.modeltype.uml2 provides some utility classes to help in collecting and analysing model elements. Finally, to ease working with the UMLsec and UMLchange profiles, utility classes are provided by the respective packages.

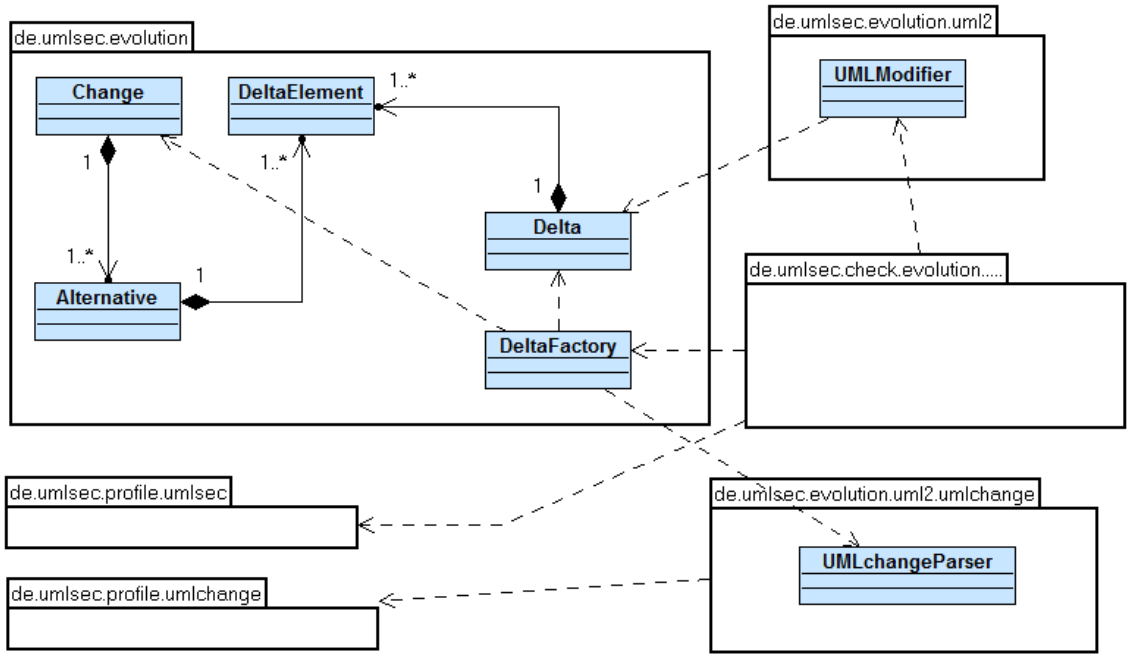

Figure 3.4: CARiSMA Evolution Architecture (simplified)

3.1.2 Evolution Support

CARiSMA is able to parse, interpret and apply evolutions described using UMLchange. The components providing the support for evolution-aware checks are shown in Figure 3.4.

After applying change descriptions to the model via the UMLchange profile, evolution-aware CARiSMA checks can be performed on the model. Internally, a parser component, the UMLchangeParser, searches the model for UMLchange applications, which are collected and then transformed to sets of equivalent delta elements.

The change structure generated by the UMLchangeParser consists of Changes which in turn contain a list of Alternatives. These Alternatives finally consist of sets of DeltaElements. In addition to multiple Alternatives, Changes themselves may have constraints attached to them restricting when the Changes may or may not take place.

The DeltaFactory processes the Change constraints and alternatives and generates all possible permutations over the alternatives while following the constraints imposed by the Changes. Each valid change permutation is saved as a Delta which can be fetched from the DeltaFactory.

A Delta can be applied to the model using the UMLModifier. The UMLModifier receives the original model, creates a copy of it and applies each DeltaElement to the model copy. The resulting model is stored and can be used in the CARiSMA evolution-aware checks.

3.2 Validation

The new CARiSMA tool, and with it the UMLchange notation, has been evaluated within the POPS case study. The validation was focused on using CARiSMA for evolution-aware security analysis in model-based development. It was organized as a half-day workshop at *Gemalto* in Paris. During the workshop the new tool and the UMLchange notation have been presented to industrial practitioners. The presentation included an introduction into architecture, user interface, and functionality of the tool, as well as teaching the UMLchange notation. Furthermore, a live demo of the tool was given. And exercises have been elaborated in order to allow the practitioners to get started with the tool. After the workshop the practitioners were able to use CARiSMA and to further evaluate the tool as "homework".

The first impression of the practitioners was that CARiSMA is a nice and powerful tool. The concept of UMLchange was rated with 4 (of maximal 5) points. The handling of the stereotype-based approach to describe evolution was rated with 3 points. Especially, the UMLchange grammar for describing new elements was seen as problematic (2 points). Typographic errors can arise, e.g. when entering qualified names of referred model elements. To tackle this problem, one could implement an auto-completion tool that allows the user to select the referenced element from a list instead of typing its name manually. This feature is evaluated for the next version of CARiSMA. Another suggestion of improvement were more expressive error descriptions in case of reported security violations. To that effect the output of CARiSMA has been extended.

The practitioners were especially interested in how the security checks have been implemented. The fact that most checks are implemented in Java lead to dissatisfaction. From industrial point of view they prefer that the checks are defined very precisely in OCL [32] or other formal notations, which would allow them to check whether the checks work as it is supposed to.

Nonetheless, the practitioners rated the tool to be applicable in daily practice, if UML is thoroughly used for modeling and if the tool is here and there a little bit improved. Furthermore, a large library of pre-defined checks should be available. However, if necessary, the checks could be implemented due to the available extension mechanism of CARiSMA. A thorough review of the SecureChange solutions and thus the basic concepts of the CARiSMA tool is presented in Deliverable 1.3 [44].

Further Applications. Due to the fact that CARiSMA is now compatible with UML2, it also allows *Thales* to integrate the tool into their processes. They use CARiSMA in the context of the ATM case study. The Thales Security DSML [43] is used to perform a risk analysis that gives high-level security requirements, which are reflected in the system design and can be analyzed by means of CARiSMA.

The feedback of Thales provides valuable input so that the tool and the contained security checks can continuously be improved. A closed validation similar to the POPS validation, however, did not happen so far.

3.3 Difference-Based Security Analysis

Recently, we have started to implement an alternative approach for evolution-aware security analysis [29]. Instead of annotating a UML model with UMLchange stereotypes, it is now possible to compare two versions of a model to gather evolution. Due to the new CARiSMA tool which is based on Eclipse and the support of the UML2-based models, we are now able to compare two versions of a model (which are EMF model instances from technical point of view) using the EMFcompare plugin of Eclipse [13]. A new CARiSMA check that we have implemented can take the difference information provided by EMFcompare and transform it into an instance of the delta model presented in Section 3.1.2. Thereby, we can run the evolution-aware security checks as if the UMLchange profile would have been used.

However, the new approach is partly limited compared to the UMLchange approach. Since only two versions of a model can be compared, it is not possible to verify all possible evolution path's at a time as they could be specified using UMLchange and the grammar. The difference computation of EMFcompare comes with a drawback. It is either based on unique identifiers used in the model or it uses a heuristic for comparison. In the first case, the engineer might be bound to certain modeling tools (i.e. those which preserve the identifiers of model elements when the model is changed). In the second case, it is not clear whether the heuristic leads to the correct result which might have an impact on the quality of the security checks. Furthermore, the difference computation is a runtime overhead, where it is not clear whether it preserves the runtime advantages of evolution-aware security checks compared to complete verification of models. We assume that it is worth in case of very large models and complex security checks.

A detailed analysis of the drawbacks and an evaluation whether the difference-based approach is a promising solution for evolution-aware security analysis is part of ongoing and future work.

4 Support for the Creation of Test Schemas

In this chapter we show how model-based testing (MBT) jointly profits from the CARiSMA tool and the UMLchange approach for security annotated models and the MBT under evolution. Our main objective is that the model used for test generation is verified for consistency with respect to the considered security properties, and this consistency should also hold after the model has evolved. If not, the model may authorize an incorrect behaviour and the produced tests will expect the same erroneous behaviour as the model from the System Under Test (SUT). The idea has already been discussed in Deliverable 4.2 (Chapter 5) [43] as well as in [17]. It is part of the integration with Work Package 7 (WP7), which focuses on MBT.

We have considered two security properties of the Global Platform that are critical for a device issuer/owner in order to have control over compromised running devices [1]: The *locked-status* (property 1) and the *authorized-status* property (property 2). The definitions are presented later in chapter 4.2.

Since we are not aiming at verifying behavioural properties, but at ensuring a structural property as a precondition to the testing process, these properties can be checked statically on UML statecharts. As presented in Deliverable 4.2, we have defined two new UMLsec stereotypes *« locked-status »* and *« authorized-status »* [43]. In the meanwhile, we have improved these stereotypes in a way, that they are no longer attached to a subsystem but directly to the affected UML state. The *state* tags in the stereotypes became thereby obsolete. Checks for each of the two properties have been implemented in CARiSMA.

Model-based testing makes use of selection criteria that indicate how to *select* the tests to be extracted from the model. The approach for testing security properties for evolving systems relies on defining additional selection criteria in the shape of *test schemas*. This is explained in Deliverable 7.4 [45].

4.1 Integrated Approach

The process is summarized in fig. 4.1. First, a validation engineer evolves a model according to the proposed changes to the previously verified test model by adding the corresponding UMLchange stereotypes (Step 1). (S)He uses CARiSMA to validate the model against the security properties (Step 2), to make sure that the model respects them. Once the model is declared correct by Step 2, a triple for each property and the delta of changes is exported. Using transformation rules, the schema is written with respect to the used property (Step 3). The created schema (Step 3) and the produced delta (Step 4), can be used to produce test cases exercising the property (Step 5).

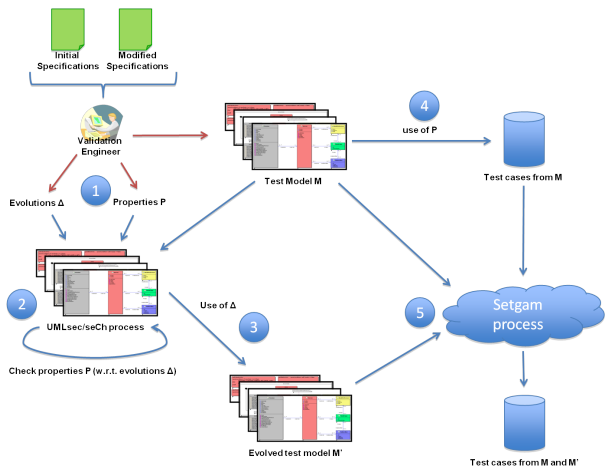

Figure 4.1: Integration of MBT, UMLsec and UMLchange

4.2 Transformation of UMLsec Stereotypes into Test Schemas

After the model has been verified with CARiSMA we want to export test schemas based on the properties specified before, encapsulating the expected behaviour of the system after executing particular instructions that could potentially violate the property and make the system not behave as expected. This generation represents thus the link from UMLsec to testing, since we can automatically generate test sequences from schemas.

In order to do so, we have implemented a check for CARiSMA that exports the security properties as hoare triples which define generic rules to create test schemas. If the expected behaviour of the system under test is modeled as a statechart where its states represent the status of the card's life-cycle, and there is a command set_status only executable by privileged applications to change the card's status from one to another and this is the event triggering all transitions in the model.

The first property, the *locked-status* property, ensures that whenever an application with enough privileges terminates the system (i.e. set to state TERMINATED), the system cannot be put back in operation. We can write property 1 as a hoare triple.

$$\{\text{state} = \text{TERMINATED}\}\ \text{all}^{+}\ \{\text{state} = \text{TERMINATED}\}$$

that is, if we reach the state *TERMINATED*, then after an arbitrary number of calls to any operation, the resulting status should be the same. In other words, there should be no *outgoing transitions* from that state to other states.

The second property, the *authorized-status* property, prohibits an application that does not have the given privilege to terminate the system. It can be written as

$$\{\text{state} \neq \text{TERMINATED}\}\ \text{all}^{+}\ \{\text{state} \neq \text{TERMINATED}\}$$

assuming that the application executing the operations has not enough privileges.

Hence, based on the hoare triples exported by CARiSMA instances of the schema language can be generated. Informally, the sequences of operations that we need for checking the security properties as specified before (that is of the form $\{T\}\ S^+\ \{Q\}$) are:

select privileged application
go to state respecting T
use an operation from S at least once
check Q

In case of the locked-status property, the concrete schema looks as follows:

```
for_each literal $X from TERMINATED
for_each literal $Y from TERMINATED
for_each $Z from any_operation
use any_operation at_least_once
to_reach state_respecting (permission = true) on_instance "instance"
then use any_operation any_number_of_times
to_reach_state_respecting (self.state=$X) on_instance "instance"
then use $Z at_least_once
to_reach_state_respecting (self.state = $Y) on_instance "instance"
```

4.3 Decreasing Model Comparison Efforts

The test generation process regarding evolutions is as follows. It takes as input the security property and two formal models, one representing the system before evolution and another representing the same system after the evolution took place. It also considers a set of test cases generated from the original model. The process, called SeTGaM, starts by a comparison of the unfolded test case specification for both models. Then a dependency impact analysis aims at identifying impacts on both models w.r.t. changes of the specification and the existing security test suite. Then test cases from the original security test suite are classified into removed, outdated, unimpacted, new tests, and so on. Details on the SeTGaM process can be found in [16].

In order to support that process, we have implemented another CARiSMA check. After a model containing UMLchange annotations has been verified, a description of the evolution is exported. If all changes pass the security verification, we can apply them to the model and the tool provides the delta to the SeTGaM process. Thereby the SeTGaM process does not have to recompute the evolution information but it can use the previously verified changes. In order to integrate the two approaches we have implemented a component that transforms the UMLchange annotations into a delta description in XML format. The XML file describes the actual changes that are applied to the model.

An example of an evolution in the GP specification that is exported to the SeTGaM process is shown in appendix A.5.

5 Statechart-Based Monitoring of Java Applications

5.1 Introduction

Implementing *safety* and *security* aspects is an integral part of software development. To ensure the safety and security of an application, requirements can be formulated using UML and the UMLsec profile or other modeling languages. Unfortunately this approach only guarantees the correctness of the model of the application and cannot verify the correctness of the implemented program. To accomplish this, static code analysis can be used, but it often proves to be inefficient for large software projects and cannot guarantee that no critical mistakes exist in an application. A solution to this problem is the *monitoring* of the application during runtime. This chapter describes an approach, which uses modifications of bytecode to inform the monitor about the state of the application (*inline-monitoring* [8]). The monitor is generated from UML state charts describing the expected behaviour.

Modern software development requires the utilization of (formal) models (e.g. UML models) during application design. In *model based security engineering*, additional information is added to the models to describe security requirements. Due to possible mistakes in the implementation of a program, validating only the model of an application is not sufficient to guarantee that an application meets all specified security requirements. The works of Bauer, Jürjens and Pironti ([2], [38]) describe how the validation of an application model can be combined with the monitoring of the program execution. In addition to that, Jürjens, Bauer and Yu [27] [3] explain how to maintain the connection between model and source code during and after the modification of the program.

Colin and Mariani [7] provide a good overview about the concept of *run time verification*. Aspects of the application are validated by an external monitor over the course of the application's execution, thus enabling to detect mistakes in the implementation of the program.

Schneider [39] describes an approach to implement a monitor by using a *finite state machine* in order to validate program execution. Each step in the program's execution is monitored and compared to the transitions of a finite state machine to determine whether the execution of the program is still valid or if it has to be terminated. We pick up this idea and describe our monitoring approach which is based on bytecode instrumentation in what follows.

5.1.1 Bytecode instrumentation

The modification of java bytecode before its execution in the Java Virtual Machine is called *bytecode instrumentation* [35]. The source code of the application is not changed and

does not have to be present to alter the bytecode. Furthermore, bytecode instrumentation enables changes to programs, even when only the `.class` files are available. One of the most used modifications of bytecode is the injection of additional operations. These operations can be used to inform a different application about the state of the program or the occurrence of certain events.

Bytecode instrumentation is very efficient in terms of performance. Since standard Java bytecode is used, the Java VM can execute the application and improve its performance by optimizing the code. Overhead is only created by the addition of operations and can be reduced by limiting the use of bytecode instrumentation to the relevant parts of the application[35].

Depending on the point in time at which the bytecode is altered three types of bytecode instrumentation can be differentiated[35]. With *static instrumentation*, all `.class` files of an application are modified and saved in a separate folder. The Java VM is launched with the altered files. A disadvantage of this approach is that all `.class` files of the application have to be processed even if they are not used during the course of the program. This problem is more serious in large software projects. *Load-time instrumentation* compensates this by altering the files only when they are loaded by the Java VM. The agent responsible for the bytecode instrumentation alters the code before it is executed. After the bytecode has been loaded, *dynamic instrumentation* enables the user to modify the bytecode while it is executed. Moreover, this adds the ability to restore the `.class` files to their original state[35].

5.2 Generation of Monitors

Our approach validates an application during runtime by using an *implemented monitor*. In order to report method calls to the monitor, before they are executed, load-time instrumentation is used to inject additional statements into the java bytecode of the application. This type of monitoring is called *inline-monitoring* [8]. The monitor checks if the desired method call is allowed at that specific point of program execution based on a *UML state diagram*. The diagram is loaded before the execution of the application and transformed into an *internal representation* (see Section 5.2.2) to increase the performance of validating method calls. In case of a forbidden method call the monitor is able to log this event and/or terminate the observed application, which may be useful if sensible data is in danger of being compromised.

There are two ways to implement the bytecode instrumentation. On the one hand, all method calls could be reported to the monitor. This would ensure the highest level of security, because each method call not represented in the state diagram will be regarded as an error in the program's execution. However, due to the increased number of messages sent to the monitor, the performance of the application would decrease. Alternatively, it is possible to monitor only those method calls represented in the diagram. This ensures that all modeled methods may only be called in the order specified by the diagram. All other methods are allowed to be executed arbitrarily between the monitored ones. The implementation presented supports both approaches.

The state diagram used for validating the allowed method calls has to meet specific re-

quirements in order to be used as an input for the monitor. These requirements are described in the following Section 5.2.1. Table 5.1 provides an overview of the most important features and how they are represented by elements of the state diagram. A more detailed description of each behavior is given in the indicated sections. Note that the monitor does not perform a security analysis. It only checks if the running program enforces the restrictions given by the state diagram. The analysis of the application's security aspects has to happen in advance, for example by model-based security engineering. Therefore the usage of the described monitor is not restricted to security analysis.

Feature	Element in diagram
allowed method call	labeled transition exists
forbidden method call	no according transition exists
allow all methods of a class or package	transition ends with `.*`
do not monitor certain methods	comment linked to a state
start monitoring	entry-point transitions
stop monitoring (monitoring may start again)	transition targeted at exit-point
end monitoring (monitor terminates)	transition targeted at final state

Table 5.1: Mapping of the desired behaviour of the monitor to elements of the state diagram

5.2.1 Notation

The following sections describe the requirements an UML state diagram has to meet in order to be used as input for the monitoring application. Furthermore, it shows how the different parts of the diagram are interpreted by the monitor.

As mentioned above, the state diagram is used to determine an allowed order of operations. When monitoring method calls a **transition** in the diagram represents an allowed method call in the application. To assign the transitions to the correct method calls, each transition is labeled with the name of the corresponding package, class and method as shown in Figure 5.1. To indicate that all methods of a class are allowed in a given state the label of the transition may end with `.*`. For example, the label `packageName.className.*` represents all methods of the class `className` in the package `packageName`.

The set of outgoing transitions of a state is therefore equivalent to the set of allowed method calls of the state. If a method with no corresponding outgoing transition at the current state is called, the application fails to meet the requirements of the state diagram and appropriate action can be taken. Otherwise, the transition to the new current state is performed.

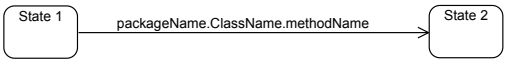

Figure 5.1: Labeling a Transition

States are of minor importance for monitoring an application, as the application is viewed as a collection of method calls. States are only used to connect transitions with each other and to model branches in the execution of the application.

Nevertheless, it is important to differentiate between *simple states*, *composite states* and

final states. While the handling of simple states is easy, composite states consist of one or more substates and are therefore more difficult to handle.

A **composite state** consists of one or more *regions* consisting of substates and transitions. If a composite state contains more than one region, the regions are called *orthogonal regions*[31, p. 566]. In this case, each region may have its own active state, thus modeling concurrency. Concurrency and therefore orthogonal regions are not supported by the presented monitor. Transitions targeted at composite states will be redirected to the substate targeted by the transition of the initial state of the composite state.

Final states indicate that the region, in which they are present, has terminated. If a final state located in an outer region of the state machine is reached, the state machine itself is considered to be terminated[31, p. 547]. In this case, monitoring of the application ends. Since orthogonal states are not processed, a final state located in a region contained by a composite state is equivalent to a state having the same outgoing transitions as the composite state.

Comments are used to define sets of method calls. While modeling security aspects, it is often useful to be able to define a set of method calls not to be monitored, for example to allow printing output to the console or to allow calls of all methods of a trusted class. A way to model this behaviour is to add transitions labeled with the allowed methods to the state and setting the target of the transition to the state itself. This leads to confusing diagrams. To prevent this, it is possible to link *comments* containing a list of all allowed methods to the state in question (the notation being the same as with transitions). If the comment is linked to a composite state, the methods are considered to be allowed for the composite state itself and all substates of it. To differentiate this comment type from other comments present in the diagram, the prefix `allowed:` is added to the list.

Figure 5.2 shows an example of this comment type: Additional to the methods specified by the transitions of the diagram, all methods of the class `java.io.PrintStream` are allowed in state 1 and state 2.

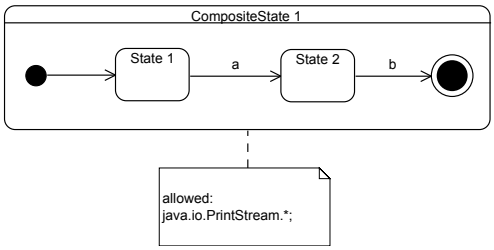

Figure 5.2: A comment allowing methods

Entry- and exit-point pseudostates are used to indicate the beginning and the end of monitoring an application. The definition of state diagrams states that each entry point has at most one outgoing transition[31, p. 557]. The outgoing transitions of all entry points form the set of methods used to begin the monitoring of the application. The monitor remains inactive until one of the methods in the set is called. Afterwards the methods called by the application are compared with the state diagram until an exit state is reached and the monitor returns to its idle state. Figure 5.3 depicts this notation. The

monitored application is allowed to call all methods until method `a` or method `b` is called. In this case, all subsequent method calls have to conform to the appropriate path in the diagram until the method call `d` stops the monitoring of the application. Note that if method `a` or method `b` are called again, the monitor will resume its function. Therefore, it is possible to specify only small parts of an application for monitoring.

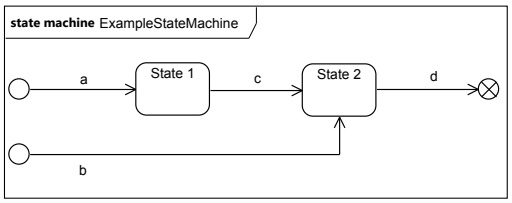

Figure 5.3: Exit- and entry-points

History pseudostates can be contained in composite states and are used to return to the last active state after the composite state is left. The state active after reaching a history state depends on whether the history state is a *deep* or *shallow* history. In the former case, the whole active state configuration of the composite state in which the history state is located is restored. In the second case only the substate becomes active which is directly contained in the composite state. History states are interpreted as specified in the OMG[31, p. 557] standard, but they present a great challenge, when the state diagram is transformed into the internal model. The transformation of history states is described in Section 5.2.2.

A **junction pseudostate** consists of a set of incoming transitions and a set of outgoing transitions, which are connected with each other. To determine which outgoing transition is used, when the junction state is reached, it is necessary to annotate the outgoing transitions with *guard conditions*. This approach only deals with the monitoring of method calls, and since the evaluation of guard conditions implies the reading of attributes, junction states with more than one outgoing transition are not supported. It is assumed that the incoming transitions of a junction state are labeled with methods while the outgoing transition has no label. Figure 5.4 shows an example of this: State 4 is reachable by each of the states 1 to 3 by calling one of the methods `a`, `b` or `c`.

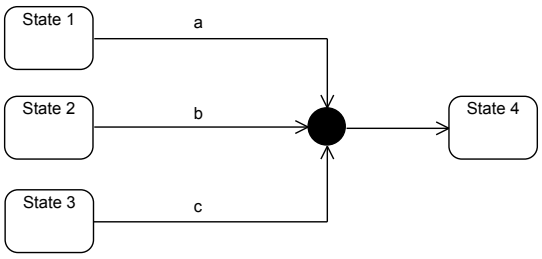

Figure 5.4: Junction pseudostate

Initial pseudostates have only one outgoing transition and indicate the first active state in a region[31, p. 556f]. They are used to redirect transitions targeted at composite states. Initial states of regions owned directly by the state machine itself are not treated in this approach. They indicate that the monitoring of the application starts with the first method call and can be easily modeled using entry-points.

A **terminate pseudostate** indicates the end of running the state machine. Upon reaching a terminal state, no methods can be called. Note that this case is different than reaching an exit point. Here the call of an outgoing transition of an entry point does not start the monitoring again.

Fork pseudostates are used to split an existing transition into several ones, which each has its target in a different region of a composite state. The opposite of a fork state is a **join state**, which combines outgoing transitions from orthogonal regions into one transition[31, p.557]. Since orthogonal regions are not supported by this approach there is no need to implement any support for fork and join states.

Choice states are used in order to implement *dynamic conditional branches*. This means that during runtime guards of transitions leaving choice states are evaluated and depending of the outcome a branch may be created[31, p. 557]. Choice states are not supported by this approach, because the evaluation of guards has not been implemented.

5.2.2 Transformation of a UML state diagram

While monitoring an application, it is important that *performance loss* of the application is as minimal as possible. Due to this, the monitor has to quickly decide if a method call is allowed at a specific point of the execution of the application.

As will be shown in Section 5.3 the monitor receives the state diagram in the form of a `.xmi` file. Working on this often complex representation is very expensive and contradicts the desired performance of the monitor. Therefore, it is necessary to transform the state diagram into an internal representation, which only contains the data needed for the validation of method calls.

The internal representation of the diagram consists of states and allowed method calls triggering the transitions. The target state of a transition can easily be determined for every model element type of state diagrams, except for history states. The target of history states is not determined by the diagram, but rather by the previous execution of the program. During transformation, the transitions targeting history states and thus affected by outgoing transitions of the composite state containing the history state are identified. Their target is set to the state active when the composite state is reached for the first time. Upon leaving the composite state, the target of all transitions leading into the according history state is changed. For this purpose a list of affected transitions is added to all transitions leaving the composite state. If a method linked to an outgoing transition of the composite state is called, the target of all affected transitions is changed to the state activated by reaching the history state. An example of this transformation is shown in Figure 5.5.

An advantage of this approach is that it reduces the complexity of the model, because the number of states in the internal representation is equal to or lower than the number

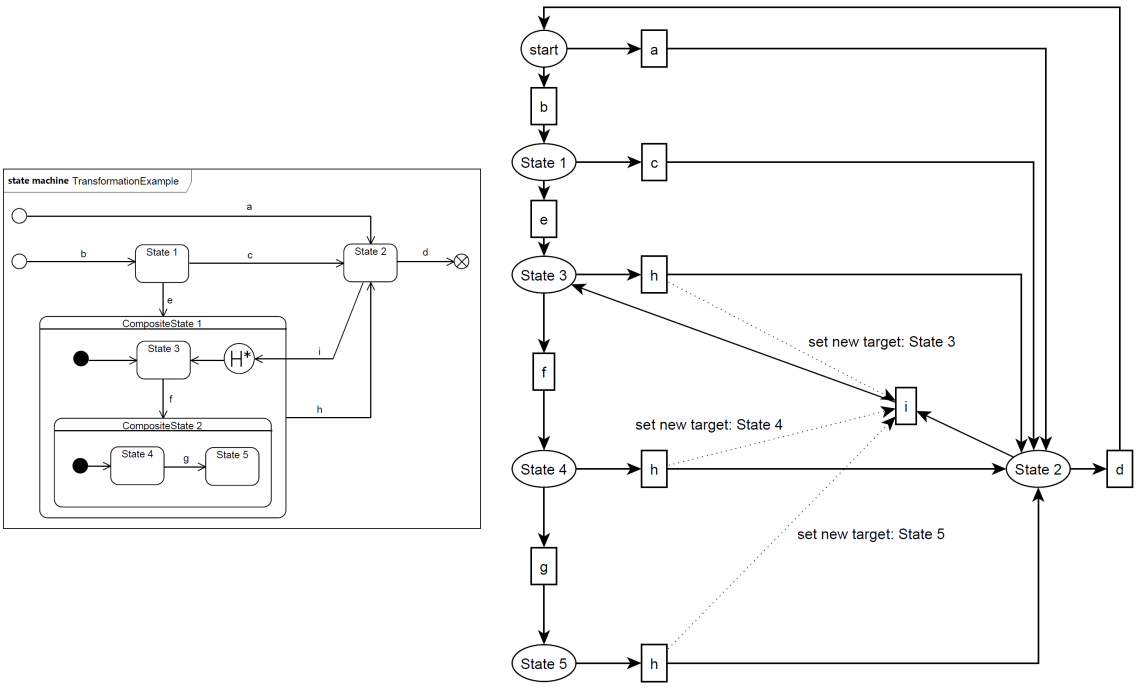

Figure 5.5: UML State Diagram And Petri Net Transformation

of states in the original model. Furthermore, the lists of affected transitions are only created once for each transition and do not have to be modified during the execution of the application. Only the targets of the affected transitions have to be changed. Therefore, the number of overall transitions in the model stays the same.

5.3 Implementation

This section describes the monitoring process. After the monitor has been initialized, the given UML state machine is transformed to the internal representation. The application is launched and bytecode instrumentation is performed. Following this, each monitored method call can be validated against the state machine.

5.3.1 Monitor Initialization

To start the monitoring process, the Java VM hosting the target application is executed with an additional parameter:

```
java monitoredApplication -javaagent:sdmonitor.jar=model.uml
```

The name of the Java class implementing the monitor is located after the `-javaagent` parameter, all statements after the equals sign are treated as parameters to the agent. In this case `model.uml` is the file containing the model used for validating the application.

Before the application is executed the Java VM calls the `premain` method of the `agent` class. Then the `agent` class loads the model specified in the argument and transforms it into the internal representation. Afterwards an instance of the `monitor` class is created and the internal model forwarded to it.

5.3.2 Internal Model Representation and Transformation

The internal model is depicted in Figure 5.6. The model class contains all the states of the model and provides a method `doCall` used to validate method calls. Additionally, `Model` has the attributes `currentState` and `idleState`. `idleState` defines the state which is initially active before the monitoring of the application starts. `currentState` defines the active state of the model and is changed to the target of a transition after each successful call of the `doCall` method.

States are represented by the class `State`. It contains two lists of allowed method calls. The list `allowedCalls` contains the names of all complete allowed method calls, while `allowedGroups` contains the names of allowed groups of methods. The method `addAllowedCall` is used to add complete method names or group of methods to their respective lists.

The class `MethodCall` represents the transitions of the UML state diagram. Since two states are linked by a method, the names of the linked states are contained in the attributes `sourceState` and `targetState`. As described in Section 5.2.2 it is possible to change the target of a method call. For this purpose, the method `setTargetState` is used. To determine which methods are affected by a called method, all affected methods are listed in `affectedCalls`.

During the transformation process, for all states without substates an instance of `State` is created and added to the internal model. If a state contains substates, these are processed recursively. Afterwards, the transitions of each entry-point are added to a new state called *start*, which is set to be the `currentState` and `idleState` of the model.

For each transition a `MethodCall` object is created and added to the list of allowed methods of the source state. If the source of a transition is a composite state, the transition is added to all substates. Determining the target of a transition is more difficult. If the target of a transition isn't a simple state the target is determined as described in Section 5.2.1. Transitions targeted at *history* states are added to the list `historyTransitions`. It is used to add methods to the list `affectedCalls`.

In the next phase the comments of the state diagram are processed. All specified methods in the comments are added to the lists of allowed methods in all states linked with the comment. If a comment is linked to a composite state, the methods are added recursively to all substates.

5.3.3 Bytecode Instrumentation

After model transformation the initialization of the monitor is finished and the application is launched. When the classes of the application are loaded, bytecode instrumentation is performed by the monitor using the package `java.lang.instrument`[36]. It identifies all

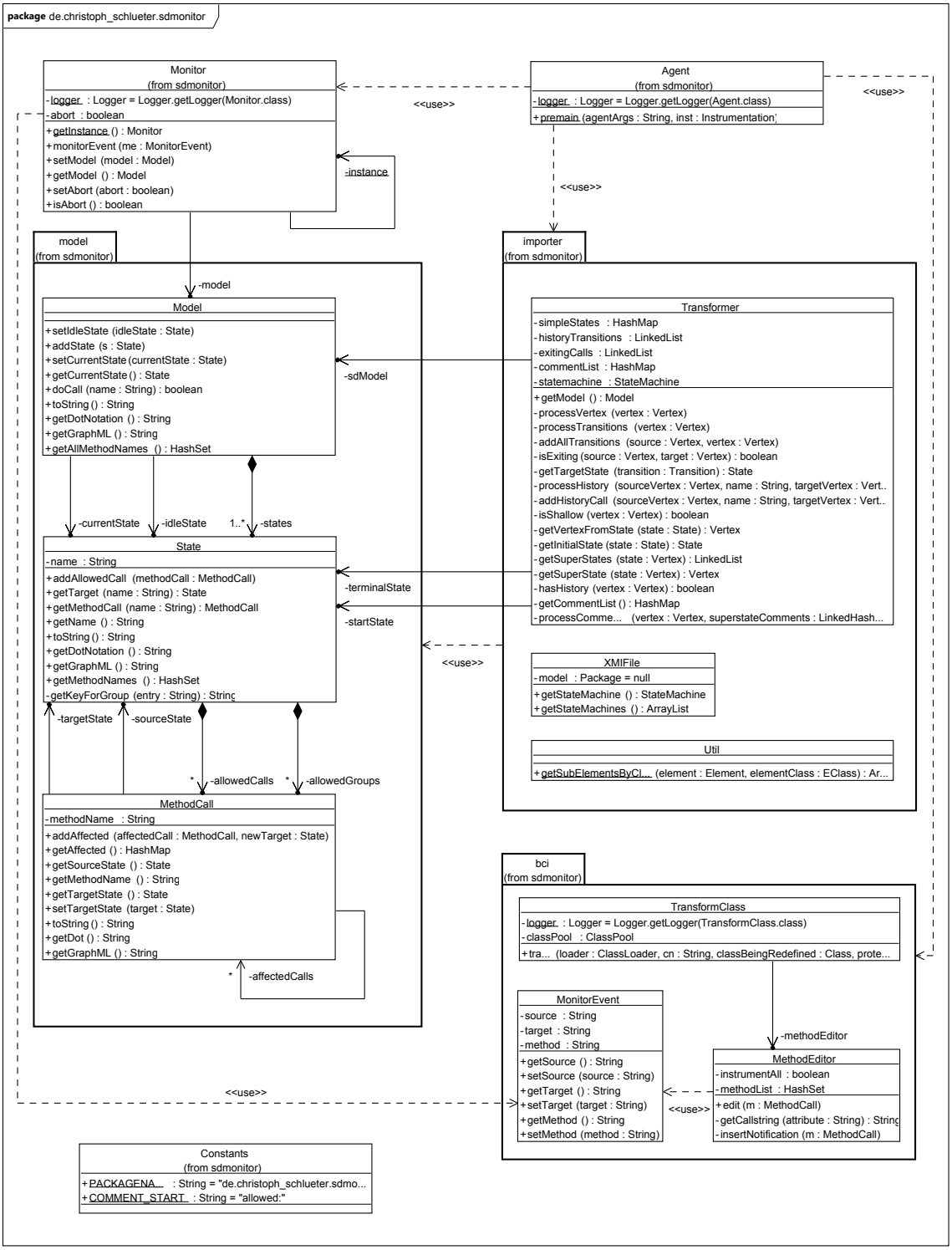

Figure 5.6: Class diagram of the internal model

method calls in the bytecode of the class which shall be monitored and replaces them with a method call to the monitor and the original method call. The argument of the monitor method call is a `MonitorEvent` object, which indicates which method of which class the application is going to execute and which class performs the method call.

5.3.4 Method Call Validation

The instrumentation of the bytecode guarantees that the monitor is informed about each method the application is going to call. To validate a method call it is forwarded to the internal model. If the model can perform a valid transition (depending on the current value of `currentState`), `currentState` is updated and the method is considered to be allowed at this point in program execution. Otherwise no transition is performed in the model and the error is logged and/or the monitored application is terminated.

To ensure that all classes of the application are able to inform the monitor about called methods it is necessary to associate the monitor to the classes. Since it would be inconvenient to create a global instance of the monitor and a reference to it in all classes, the monitor is implemented as a *singleton*[18].

5.4 Evaluation

As mentioned in Section 5.2.2 it is important to keep the performance loss of an application while monitoring to a minimum. Therefore, it is interesting to know how much time the presented monitor needs to validate a method call. For this purpose, we create a simple benchmark, which measures the time needed to execute the method `Math.cbrt(time)`. The benchmark is executed twice, once with the application running by itself and once while the monitoring application is active. To initialize the monitor we create a simple UML state diagram consisting of one entry- and one exit-point linked to each other by a transition. Before and after the method is called the method `System.nanoTime()` is used to get (relatively) accurate timestamps of the system. The difference between the timestamps equals the time needed for the execution of the method. If we compare the computed values, we can easily calculate the loss of performance caused by the monitor.

Note that the timestamps given by `System.nanoTime()` are not completely accurate and can fluctuate in the region of microseconds depending on the operating system[21]. To reduce the effect on the benchmark, it is common to measure the time needed for thousands of method calls and to deduce the time needed for one. In this case we measure the time needed to execute 100 000 and 1 000 000 method calls on a four year old system[1]. Furthermore, executing accurate benchmarks is difficult, because the Java VM optimizes code during runtime. These difficulties and possible solutions are described by Brent Boyer in *Robust Java benchmarking*[5].

Table 5.2 shows the results of both benchmarks. We can deduce that monitoring an application needs approximately 0.62 ms per monitored method call. Although a very simple state diagram was used, this value should be relatively stable for larger diagrams,

[1] Intel Core2 Duo E6550 2,33 GHz; 6 GB RAM; Windows 7 64 Bit; JRE 6

	100 000 iterations		1 000 000 iterations	
monitor state	idle	active	idle	active
run time [ms]	182 691	244 376	1 816 806	2 432 705
run time/iteration [ms]	1,826	2,443	1,816	2,432

Table 5.2: Measurement results of the benchmark

because the validation of a method call is performed using a hashmap and occurs in constant time[34]. The only exception from this rule is a transition leaving a composite state containing a history state, since the target of all affected transitions has to be updated.

Note that the percental increase in runtime is very high (in this example 33.9%). This value depends highly on the number of monitored method calls and the time needed to perform a method without monitoring. In this example a simple method was used and all calls were monitored. Therefore, the overhead caused by the monitor can be significantly smaller in larger applications, where only a portion of time consuming methods is monitored.

5.5 Application

In this section a short application example is described for the use of the presented monitor. Based on the examples in the *Java Cryptography Architecture (JCA) Reference Guide* [33] the application is supposed to load a text from a file, encrypt it using the *DES* algorithm and save the ciphertext to a file with the same name as the input file and the additional ending `.encrypted`. The name of the file containing the plaintext and the key used for encryption are given to the application as command line parameters. The methods `readFile` and `saveFile` perform the corresponding actions, while `desCipher.doFinal` creates the ciphertext. In this example the monitor shall ensure that the file is only saved after it has been encrypted. This requirement makes more sense, if we imagine the plaintext being entered by a user (e.g. a password) and we want to prevent to store the plaintext in a file where it could be compromised by other programs or users. To initialize the monitor, the simple state diagram depicted in Figure 5.7 is used. Note that only the methods present in the diagram are monitored. Other methods can be called arbitrarily. In this test the application encrypts a `.JPG` file with a size of 4744 kB.

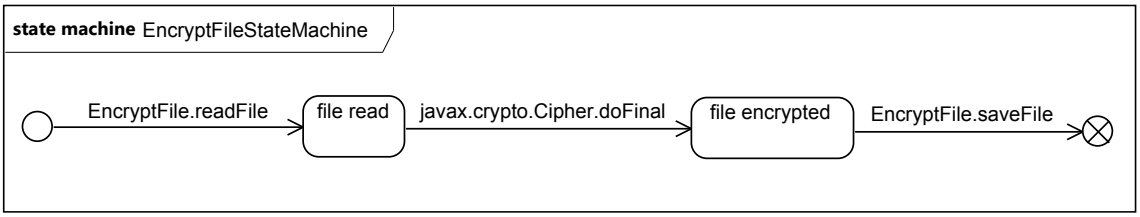

Figure 5.7: Diagram used for monitoring the encryptFile application

If we configure the logging framework to create messages for each method call the output of the monitor resembles:

```
[...] DEBUG [...] Agent: instrumentAll=false, abortOnError=false
[...] DEBUG [...] Monitor: valid call: EncryptFile.readFile
[...] DEBUG [...] Monitor: valid call: javax.crypto.Cipher.doFinal
[...] DEBUG [...] Monitor: valid call: EncryptFile.saveFile
```

It is apparent, that the application works correctly. If we comment out the call of the method `aesCipher.doFinal` in the source code we get a different output from the monitor:

```
[...] DEBUG [...] Agent: instrumentAll=false, abortOnError=false
[...] DEBUG [...] Monitor: valid call: EncryptFile.readFile
[...] ERROR [...] Monitor: invalid call: EncryptFile.saveFile
```

Note that in this case, the invalid method call is marked with the label `ERROR` and would be printed even if we would suppress the output of all `DEBUG` messages. If the flag `abortOnError` was set, application execution would stop before `saveFile` is performed. Executing the program takes 703480 ms. Activating the monitor leads to a runtime of 723165 ms, which is equal to an increase of only 2.8%. Combined with the results of Section 5.4, this example shows how the increase in runtime is highly dependent on the monitored application. Therefore it is necessary to perform a runtime analysis in the context of the actual application to be monitored.

6 Log-Based Monitoring of Processes

The in-line monitoring presented in the previous chapter is a suitable approach to enforce that a secure system behaves according to the specification that was verified with UMLsec, and the CARiSMA tool, respectively. However, some systems cannot be monitored that easily, e.g. if the system is not directly accessible as in the case of embedded systems or smart cards such as the Global Platform [1]. Here, the system is on a chip where monitoring approaches are too expensive in terms of runtime and memory usage. Here, we should be able to monitor a system from outside. This means, that events are logged into a log file, e.g. the commands sent by a host to a of a smart card or the respective responses. This log file can then be checked to be conform with the expected behaviour of the system. This approach is called *offline monitoring* [20].

Conformance checks between log files and expected behaviour are well-known for business processes [49]. Here we make use of business process management, which includes such analysis [6]. By analyzing event logs, the control flow of an executed business process can be reconstructed [52]. The challenge here is to create process models from the event logs, such that these models correspond to the dynamic process at runtime [51]. The procedure of extracting specific information from system log files at runtime may be defined as Process mining. The ProM framework, which will be discussed in detail in Section 6.1, supports plug-in based implementation of such process mining techniques. Furthermore, it supports various analysis plug-ins, one of which is the Conformance Checker plug-in. Given a Petri net model, this plug-in is used to determine the degree of conformance of a running system to the given model.

It is worth to be mentioned that this approach is general enough that it even allows us to develop secure systems in the large. In this case we have systems that are composed from several components. While each component can be monitored individually, e.g. by using the approach in Chapter 5, the overall system has to be monitored on a different level, i.e. from outside, similar to smart cards in the Global Platform scenario.

However, in case of the philosophy of model-driven development UML is the de-facto standard for modeling. That is, why UMLsec and UMLchange are suitable for security analysis of design model, even if evolution occurs. UML activity diagrams are the language of choice. They are the equivalent UML counterparts of petri nets. Thus we have realized a CARiSMA plug-in that transforms activity diagrams into petri nets. Therefore, the plug-in utilizes triple graph grammars [41]. A definition of each grammar rule is given, as well as a description of the transformation algorithm based on these individual rules. The implementation of the transformation algorithm as a CARiSMA plug-in is illustrated. Finally, an example based on the GlobalPlatform shows the application of the concept of log-based monitoring.

6.1 The ProM Framework

Various scientific and commercial tools to extract information from log events to perform process mining are being developed in the context of business activity monitoring (BAM)

and business process intelligence (BPI)[50]. However, these tools use various file formats. This makes it very difficult to use different sets of tools on the same datasets and compare the various process mining results [4]. Furthermore, some of these tools implement concepts which might be of use for other tools. But because of the lack of standardization of these tools, researchers working on new process mining techniques are forced to develop completely new mining infrastructures. To overcome these problems, the ProM framework was developed [4]. ProM accepts various file formats for both input and output files and supports the reuse of existing concepts for the implementation of new process mining techniques.

ProM features a plug-in based architecture. Plug-ins to perform imports, exports, conversions, analysis and mining can be developed and integrated into the ProM framework. The advantage of this feature is that the framework itself does not need to be modified. In order to add a new plug-in, only the name and path of the plug-in has to be added to the ini-file. Another feature of the ProM framework is the interoperability between the various plug-ins. The results generated by using the plug-in can be used as input for another plug-in.

Figure 6.1 gives an overview of the ProM framework. XML files are read by the Log Filter component which enables filtering and processing of large datasets. Input Plug-ins provide support for various models, e.g. Petri nets. Mining Plug-ins perform the actual process mining and store their results in the Result Frames. These frames may be used as visualizations as well as an input for various Analysis Plug-ins. The Conversion Plug-ins transform the mining results to and from various formats like EPCs or Petri nets.

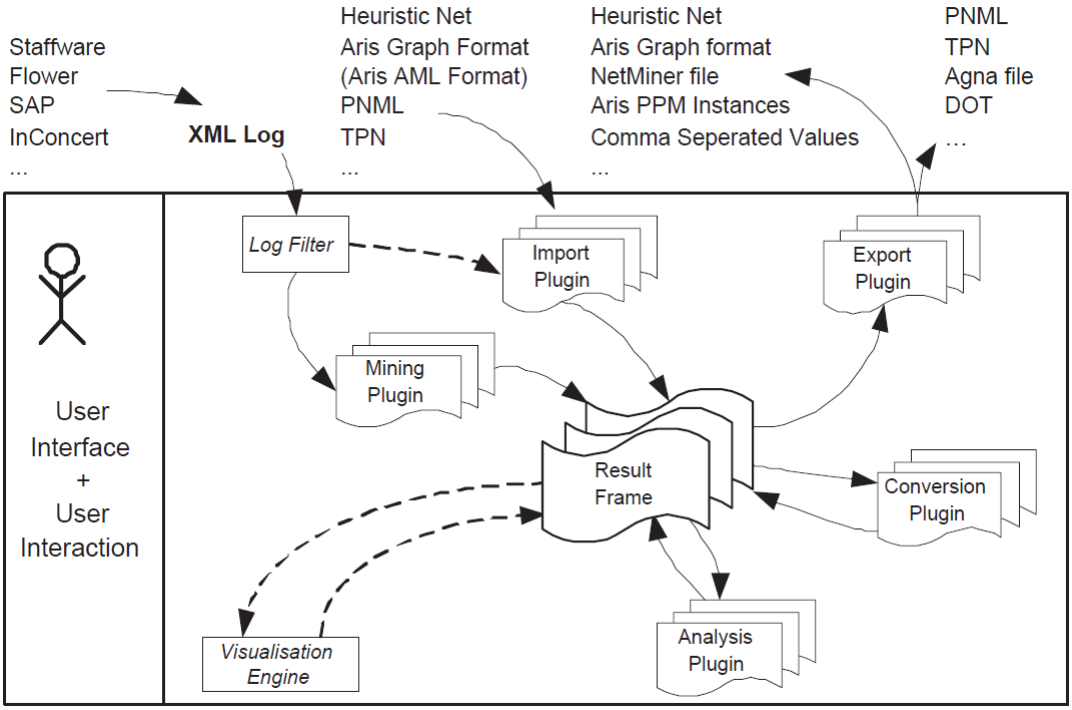

Figure 6.1: Overview of the ProM framework [4]

One of the Analysis Plug-ins of particular interest for the monitoring of evolving systems is the Conformance Checker plug-in. Conformance checking describes the ability to verify

how much a process model coincides with a process in execution. To facilitate this, the Conformance Checker makes use of various metrics to measure the degree of conformity of a given process model in form of a Petri net to a process in execution by extracting information from its log files, and comparing these log events to the process model.

6.2 Conversion of Activity Diagrams to Petri Nets

As already stated in the introduction of this chapter, the models to specify evolving systems are UML models. To make use of the Conformance Checker plug-in of the ProM framework, these models, in this case UML activity diagrams, need to be converted into Petri nets. To facilitate the conversion, we make use of a triple graph grammar (TGG) [48]. A triple graph grammar defines a relationship between two graphical models [28] by specifying a set of rules to transform one model type into the other.

A TGG rule can be divided into three components as shown in Figure 6.2. The left component represents the source model. The component in the middle maps the left component to the right component (correspondency mapping), and the right component represents the transformed model.

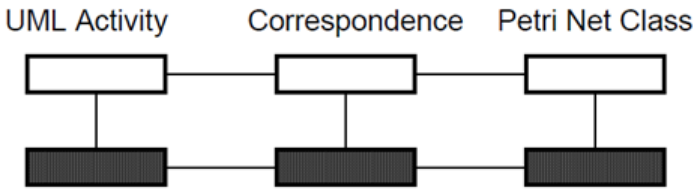

Figure 6.2: TGG Transformation of an UML activity diagram to Petri net [48]

For the conversion of an UML activity diagram into a Petri net, 6 TGG rules are defined in [48]. To alleviate the rule definition, various activity diagram elements were classified in groups. Start nodes, decision nodes, merge nodes and end nodes were classified as control nodes. Action nodes, and fork or join nodes were classified as executable nodes. Finally, edges between control nodes were classified as exception edges. The TGG rules cover all basic UML activity diagram elements.

The 6 TGG rules are defined as follows:

- Rule1: Add a new control node
- Rule2: Add a new executable node
- Rule3: Add a new edge
- Rule4: Add a new exception edge
- Rule5: Add an edge between an executable node and a control node
- Rule6: Add an edge between a control node and an executable node

These rules are illustrated in Figures 6.3 to 6.8 and described into further detail. Each rule defines which Petri net element corresponds to the specified activity diagram element.

Rule 1 states that adding a control node to an activity diagram, corresponds to adding a place to a Petri net.

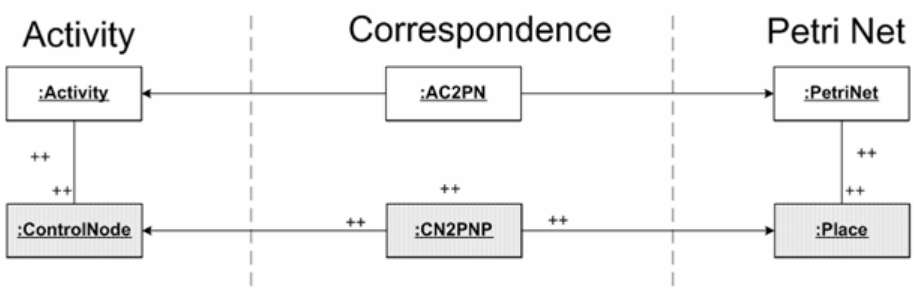

Figure 6.3: Rule1: Add a new control node [48]

Rule 2 states that adding an executable node to an activity diagram, corresponds to adding a transition to a Petri net.

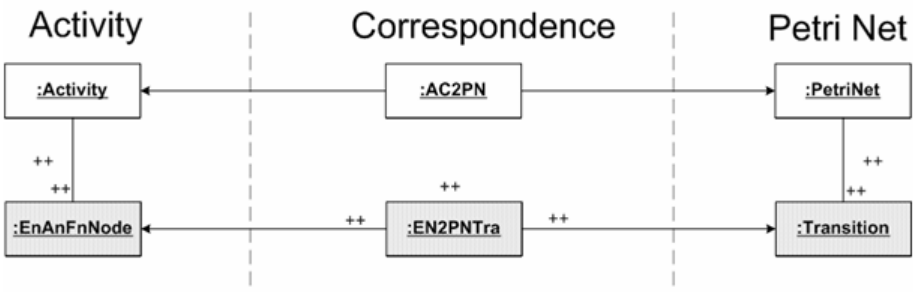

Figure 6.4: Rule2: Add a new executable node [48]

Rule 3 and 4 are a bit more complex. Rule 3 states that adding an edge between two action nodes in an activity diagram, corresponds to an edge between a transition and a place, and an edge from that place to a second transition in a Petri net. The two transitions connected by the two edges and a place between those edges, correspond to the action nodes in the activity diagram.

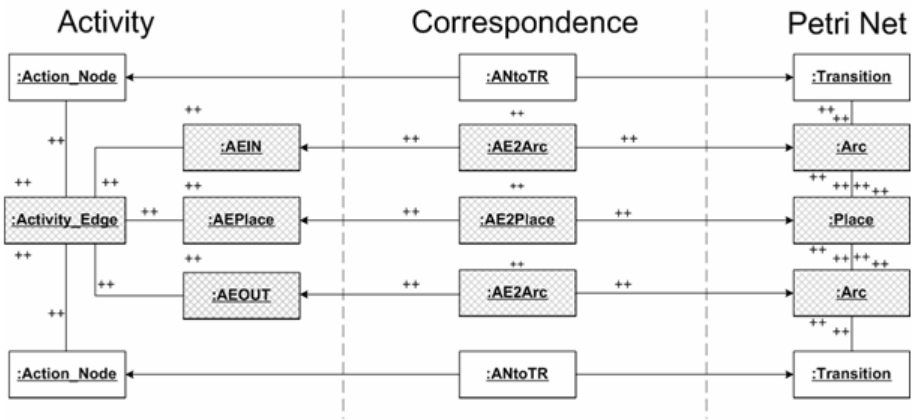

Figure 6.5: Rule3: Add a new edge [48]

Rule 4 states that an exception edge in an activity diagram (i.e. an edge between two control nodes), corresponds to an edge from a place to a transition and an edge from that transition to a second place. The two places, connected by the two edges and the transition between those edges, correspond to the control nodes in the activity diagram.

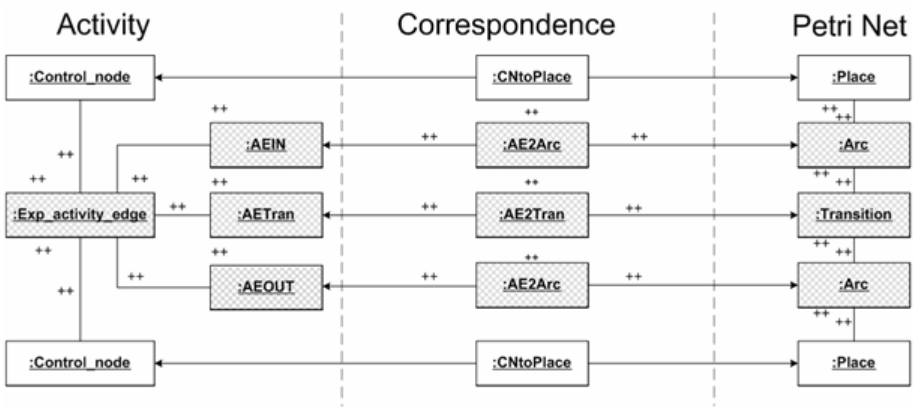

Figure 6.6: Add a new exception edge [48]

Rule 5 states that an edge from an executable node to a control node in an activity diagram corresponds to an edge from a transition to a place in a Petri net.

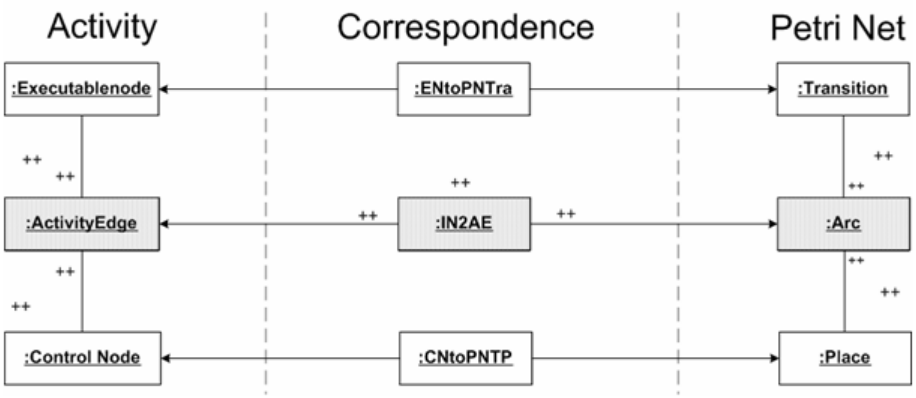

Figure 6.7: Rule5: Add an edge between an executable node and a control node [48]

Rule 6 is the counterpart of Rule 5. It states that an edge from a control node to an executable node in an activity diagram corresponds to an edge from a place to a transition in a Petri net.

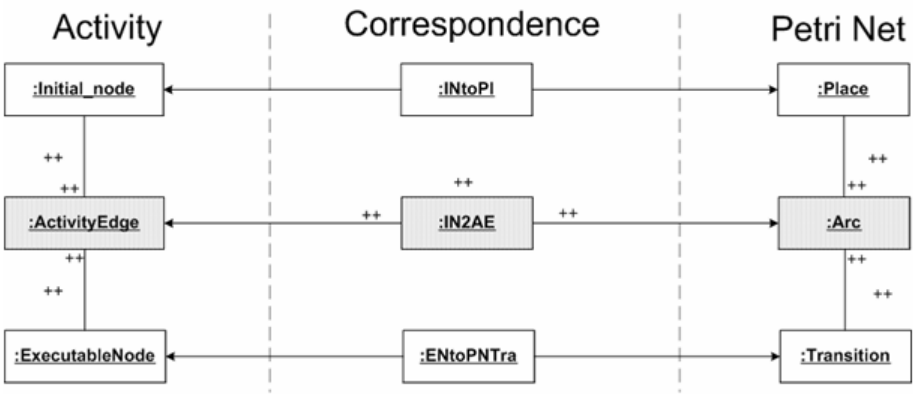

Figure 6.8: Rule6: Add an edge between a control node and an executable node [48]

These 6 TGG rules were implemented as illustrated by the algorithm convertA2P (see

Algorithm 1). ConvertA2P operates by beginning with the start node of an activity diagram, and then traversing recursively over each activity diagram element, applying each corresponding TGG rule. The applicable TGG rules are determined by the If-Statements in line 3, 6, 12, 16, 20 and 24 which check the corresponding criteria. The algorithm terminates if the end node was reached. Since activity diagrams may be cyclic, an infinite loop may occur. To prevent this, the algorithm checks if the Petri net element generated for the current activity diagram element, already exists in the Petri net (Line 1 in each rule).

Algorithm 1 convertA2P

```
1:  if endNode is reached then
2:      return
3:  else if currentElement = (JunctionNode or FinalNode or InitialNode) then
4:      apply Rule1
5:      return
6:  else if currentElement = (ActionNode or ForkNode or JoinNode) then
7:      apply Rule2
8:      return
9:  else if  currentElement is Edge  then
10:     source = source of currentElement
11:     target = target of currentElement
12:     if  (source = ActionNode AND target = (ActionNode or ForkNode JoinNode)) or
        (source = ForkNode AND target = (ActionNode or JoinNode)) or (source = JoinN-
        ode AND target = (Action Node or Fork Node))  then
13:         apply Rule3
14:         return
15:     end if
16:     if  (source = InitialNode AND target = JunctionNode) or (source = JunctionNode
        AND target = (JunctionNode or FinalNode))  then
17:         apply Rule4
18:         return
19:     end if
20:     if  (source = ActionNode AND target = (FinalNode or JunctionNode)) or (source =
        ForkNode AND target = (FinalNode or JunctionNode)) or (source = JoinNode AND
        target = (FinalNode or JunctionNode))  then
21:         apply Rule5
22:         return
23:     end if
24:     if  (source = InitialNode AND target = (ActionNode or ForkNode)) or (source =
        JunctionNode AND target = (ActionNode or ForkNode or JoinNode))  then
25:         apply Rule6
26:         return
27:     end if
28: end if
```

6.3 CARiSMA Check (Activity to Petri Net Converter)

The algorithm described in the previous section has been implemented as a CARISMA check, the activity2petrinet converter. The implementation can roughly broken down into four steps. First, the source UML model has to be loaded and accessible, which proved to be fairly easy, as CARiSMA is based on the Eclipse Modeling Framework and provides, among other modelling languages, an implementation of the UML2 metamodel.

Secondly, the implementation of a data structure to store the converted Petri net was required. Simple Petri nets, as used in the scope of this work, consist of a set of places and transitions, connected by edges. Hence a simple data structure to accommodate these requirements has been developed. Each Petri net object contains a list of place, transition and edge objects. Each object possesses a unique ID and an attribute describing the object type. In addition to the ID and attribute, edge objects store information about source and target nodes.

Finally, the converted Petri net has to be exported into the PNML format, a standard supported by most Petri net tools. PNML is an exchange format based on XML. It was developed to support the diverse requirements of the various Petri net types [40]. Though a framework implementing the international Petri net standard ISO/IEC 15909 [30] already exists, we implemented our own PNML export function to benefit from the simplicity of the used Petri net data structure. The Petri net framework would simply increase the complexity and runtime of the plug-in.

To run the CARiSMA check, a new CARiSMA analysis on a model containing an activity diagram has to be created. After adding the 'activity2petrinet' check and providing the output file parameter, i.e. the file name and path of the resulting petri net, the conversion is started by simply clicking the 'Run' button.

6.4 Application

Finally we like to conclude this chapter by demonstrating the application of the described concept of log based monitoring. For demonstrative purposes, the process of loading of an application to a GlobalPlatform card, as specified in the GlobalPlatform Card Specification Version 2.2 is used. The load and install process was modeled as an UML activity diagram illustrated in Figure 6.9.

The process is started by selecting a security domain. A security domain is the on-card entity providing support for the control, security, and communication requirements of an off-card entity, e.g. the card issuer, an application provider or a controlling authority. Then the INSTALL [for load] command followed by one or more LOAD commands processed by the security domain is executed. The process is completed by running the INSTALL [for install] command, which is also processed by the security domain and then passed to the central on-card administrator that owns the GlobalPlatform Registry for further verification and processing.

We developed a java program to simulate the back end system that hosts the GlobalPlatform system, responsible for loading an Application to a GlobalPlatform card as described

above. This simulation program logs the executed commands and writes them to a log file. These log files describe the application flow. To verify if the system at runtime corresponds to the modeled specifications, the CARiSMA activity to Petri net converter check can be used to convert the specified model (Figure 6.9) into a Petri net. The resulting Petri net after the conversion is shown in Figure 6.10.

Finally, ProM loads the Petri net model and uses its Conformance Checker plug-in to measure the degree of conformance to the specification. Figure 6.11 shows part of the result view for the Advanced Behavioral Appropriateness metric, one of several metrics which the Conformance Checker supports to measure the degree of conformance.

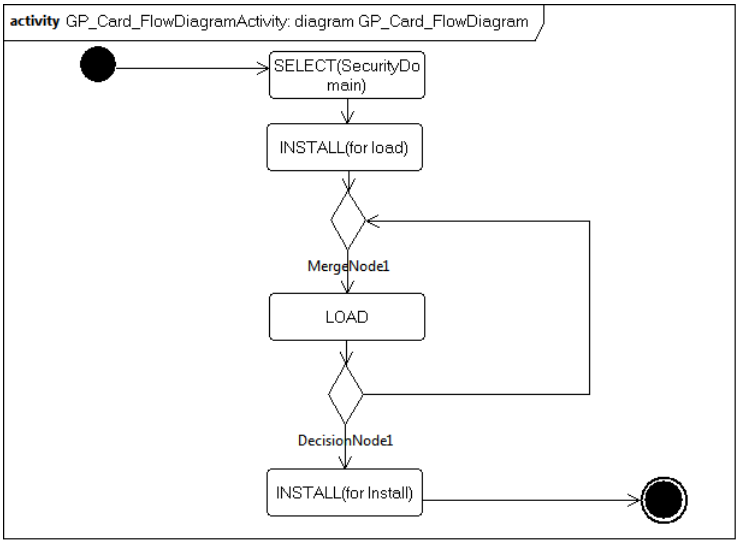

Figure 6.9: Activity Diagram specifying the load Application process of a GP card

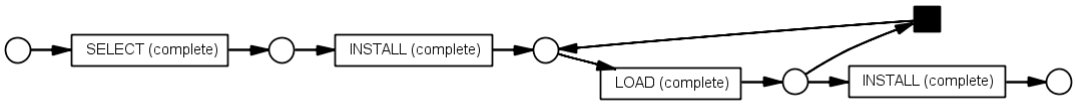

Figure 6.10: The resulting Petri net after the conversion

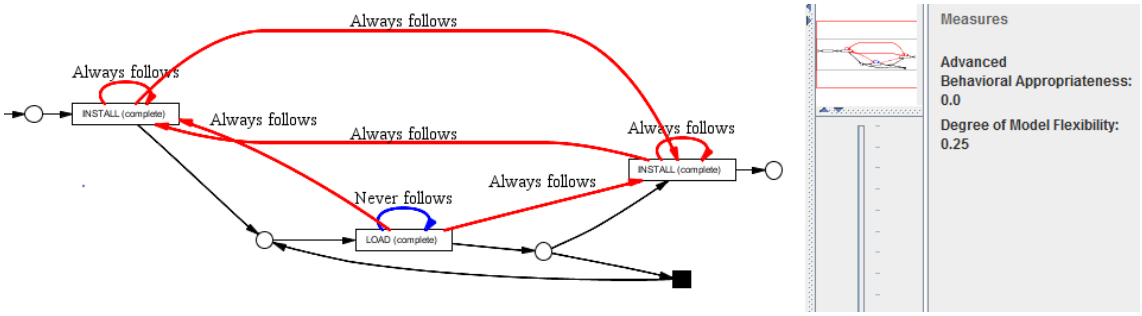

Figure 6.11: Result for the Advanced Behavioral Appropriateness metric

7 Conclusions

In this deliverable, we have reported about the results of Work Package 4 in Year 3 of SecureChange. The notation for describing possible evolutions of UML models has been improved into the UMLchange profile for UML. The notation was thereby decoupled from the security aspects defined in UMLsec in order to have separation of concerns which leads to better maintainability in the first place. Furthermore, the description of evolutions became more expressive and easier to handle. Together with the new notation, we have implemented a new analysis tool CARiSMA that allows developers to check security properties on their models and especially, if evolution is considered. It has been shown in the validation and in the feedback from project partners, that it was a good decision to re-develop the old tool and migrate it onto EMF to make it compatible with other tools and UML2. In addition, export functionality has been implemented to integrate the model-based verification approach with the model-based testing approaches of WP7.

As counterpart to the security analysis at design time, we have developed monitoring approaches to ensure secure behaviour also at runtime. Both, in-line and off-line monitoring, has been considered. Our in-line monitoring approach instruments Java byte code in order to supervise all relevant method calls and performs a conformance check against a UML state chart describing the intended behaviour. In case of violations the monitored application can be stopped. The off-line monitor was realized in a way that the intended behaviour given as UML activity diagram is transformed into a petri net that can then be compared to the process (also a petri net) that is mined from an runtime log of a system.

Continuing research is a topic of future work although it goes beyond the SecureChange project. Our first approaches towards difference-based evolution analysis should be deepened, as we think that from practical point this is an promising approach, however, the question of reliability is still to be answered. Also the monitoring approaches are planned to be brought into industrial practice. The monitors generated from UML state charts, for instance, could be improved so that they will capture field access (and thus data flow) in future as well. Both the evolution analysis and the monitoring can be extended to further notations such as BPMN for business process modeling, which makes the approaches attractive for new fields of application.

Bibliography

[1] Global platform specification. http://www.globalplatform.org/specificationscard.asp, May 2011.

[2] Andreas Bauer and Jan Jürjens. Runtime verification of cryptographic protocols. *Computers & Security*, 29(3):315–330, 2010.

[3] Andreas Bauer, Jan Jürjens, and Yijun Yu. Run-time security traceability for evolving systems. *The Computer Journal*, 54(1):58–87, 2011.

[4] H.M.W. Verbeek A.J.M.M. Weijters B.F. van Dongen, A.K.A. de Medeiros and W.M.P. van der Aalst. The prom framework: A new era in process mining tool support, applications and theory of petri nets 2005. *Lecture Notes in Computer Science, Volume 3536/2005, 1105-1116*, 2005.

[5] Brent Boyer. Robust Java benchmarking, Part 1: Issues.

[6] Patrice Briol. *The Business Process Modeling Notation Pocket Handbook*. Lulu, 2008.

[7] Séverine Colin and Leonardo Mariani. 18 run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2005.

[8] Lieven Desmet, Wouter Joosen, Fabio Massacci, Pieter Philippaerts, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe. Security-by-contract on the .net platform. *Information Security Technical Report*, 13(1):25 – 32, 2008.

[9] TU Dortmund. CARiSMA Website.
http://carisma.umlsec.de.

[10] Eclipse.org. Extension points/extensions resources.
http://www.eclipse.org/resources/?category=Extension%20points.

[11] Eclipse Foundation. Eclipse modeling framework.
http://eclipse.org/modeling/emf/.

[12] Eclipse Foundation. Eclipse website.
http://www.eclipse.org/.

[13] Eclipse Foundation. EMF compare.
http://www.eclipse.org/emf/compare/.

[14] Eclipse Foundation. Papyrus MDT Website.
http://www.eclipse.org/modeling/mdt/papyrus/.

[15] Eclipse Foundation. UML2 Metamodel EMF Implementation.
http://www.eclipse.org/modeling/mdt/?project=uml2.

[16] Elizabeta Fourneret, Fabrice Bouquet, Frédéric Dadeau, and Stéphane Debricon. Selective test generation method for evolving critical systems. In *REGRESSION'11, 1st Int. Workshop on Regression Testing - co-located with ICST'2011*, Berlin, Germany, March 2011. IEEE Computer Society Press. To appear.

[17] Elizabeta Fourneret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jürjens, and Parvaneh Yousefi. Model-based security verification and testing for smart-cards. In *ARES 2011, 6-th Int. Conf. on Availability, Reliability and Security*, Vienna, Austria, August 2011.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster*. Addison-Wesley, 6 edition, 2010.

[19] Object Management Group. MetaObject Facility.
http://www.omg.org/mof/.

[20] Klaus Havelund and Grigore Rosu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in System Design*, 24:189–215, 2004. 10.1023/B:FORM.0000017721.39909.4b.

[21] David Holmes. Inside the Hotspot VM: Clocks, Timers and Scheduling Events - Part I - Windows.

[22] IBM. IBM Rational Software Architect Website.
http://www-01.ibm.com/software/rational/products/swarchitect/.

[23] No Magic Inc. MagicDraw Website.
https://www.magicdraw.com/.

[24] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2005.

[25] J. Jürjens and M. Ochoa. Model-based security engineering for evolving systems (invited lecture). *11th School on Formal Methods (SFM 2011)*, Bertinoro (Italy) 13-18 June 2011.

[26] Jan Jürjens, Loïc Marchal, Martín Ochoa, and Holger Schmidt. Incremental Security Verification for Evolving UMLsec models. In *Proc. of the 7th European Conference on Modelling Foundations and Applications, Birmingham, UK (ECMFA'11)*, pages 52–68, 2011.

[27] Jan Jürjens, Yijun Yu, and Andreas Bauer. Tools for traceable security verification. In *BCS International Academic Conference*, pages 367–390, 2008.

[28] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *Technical Report tr-ri-07-284,Department of Computer Science, University of Paderborn*, 2007.

[29] Johannes Kowald. Differenzberechnung zur Unterstützung modellbasierter Sicherheitsaalyse von Softwareevolution. Bachelor Thesis, TU Dortmund, Germany (in German), 2011.

[30] L. Petrucci L. Hillah, F. Kordon and N. Treves. PNML Framework: an extendable reference implementation of the Petri Net Markup Language. *Petri Nets 2010, LNCS 6128, pages 318-327.*, 2010.

[31] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, May 2010.

[32] OMG. Current OMG OCL specification.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL.

[33] Oracle. Java Cryptography Architecture (JCA) Reference Guide.

[34] Oracle. Java `java.util.HashMap` API.

[35] Oracle. Java Virtual Machine Tool Interface (JVM TI) Reference.

[36] Oracle. java.lang.instrument Package.
http://docs.oracle.com/javase/6/docs/technotes/guides/instrumentation/index.html.

[37] Oracle. NetBeans MDR API.
http://netbeans.org/download/5_0/javadoc/org-netbeans-api-mdr/.

[38] Alfredo Pironti and Jan Jürjens. Formally-based black-box monitoring of security protocols. In *ESSoS*, pages 79–95, 2010.

[39] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[40] E. Schnieder. Entwurf Komplexer Automatisierungssysteme. *EKA 2006, 9. Fachtagung, Braunschweig, Germany, May 2006, pp. 35-55.*, 2006.

[41] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-59071-4_45.

[42] SecureChange. Deliverable 4.1, 2010.
http://securechange.eu/content/deliverables.

[43] SecureChange. Deliverable 4.2, 2011.
http://securechange.eu/content/deliverables.

[44] SecureChange. Deliverable 1.3, 2012.
http://securechange.eu/content/deliverables.

[45] SecureChange. Deliverable 7.4, 2012.
http://securechange.eu/content/deliverables.

[46] Open Source. ArgoUML Website.
http://argouml.tigris.org/.

[47] Open Source. Topcased Website.
http://www.topcased.org.

[48] A. Spiteri Staines. A triple graph grammar mapping of uml 2 activities into petri nets. *INTERNATIONAL JOURNAL OF COMPUTERS, Issue 1, Volume 4*, 2010.

[49] T.H.Davenport and J.E. Short. The new industrial engineering: Information technology and business process redesign. *Sloan Management Review, Vol. 31 No.4, S. 11-27.*, 1989.

[50] A. Weijters B. Vandongen A. Alvesdemedeiros W. van der Aalst, H. Reijers and H. Verbeek M. Song. Business process mining: An industrial application. *Information Systems, Vol. 32, No. 5, pp. 713-732.*, 2007.

[51] A.J.M.M. Weijters W.M.P. van der Aalst. Process mining: a research agenda. *Computers in Industry, Vol. 53, No. 3, pp. 231-244.*, 2004.

[52] C. Gunther A. Rozinat H. M. W. Verbeek W.M.P. Van Der Aalst, B.F. Van Dongen and A. J. M. M. Weijters. ProM: The Process Mining Toolkit. *BPM'09 2(Demonstration Track). CEUR-WS.org*, 2009.

A Appendix

A.1 CARiSMA Plugin List

Plugin name	Description
de.umlsec.tool.core	The core Tool (GUI, etc.)
de.umlsec.tool.evolution	Evolution support
de.umlsec.tool.evolution.uml2	UML2 specific evolution support
de.umlsec.tool.evolution.uml2.umlchange	UMLchange Parser Extension
de.umlsec.tool.evolution.emfdelta	Delta generator based on EMFcompare (cf. Section 3.3)
de.umlsec.tool.ocl	Re-usable methods for the OCL support
de.umlsec.tool.ocl.library	Model to manage OCL Constraints
de.umlsec.tool.ocl.library.edit	Edit part for de.umlsec.tool.ocl.library
de.umlsec.tool.ocl.library.editor	Editor for de.umlsec.tool.ocl.library
de.umlsec.modeltype.uml2	UML2 meta model
de.umlsec.profile.umlchange	UMLchange profile
de.umlsec.profile.umlsec	UMLsec profile
de.umlsec.check.activity2petrinet	Converts an activity diagram to a petri net (cf. Chapter 6)
de.umlsec.check.activitypaths	Prints out all possible paths in an activity diagram
de.umlsec.check.oclcheck	Queries general EMF models with defined OCL constraints
de.umlsec.check.smartcard	Check for smartcard specific properties
de.umlsec.check.smartcard.evolution	Smartcard Evolution Check
de.umlsec.check.statemachinepaths	Prints out all possible paths in a state chart
de.umlsec.check.staticcheck	Checks for the static UMLsec stereotypes
de.umlsec.check.staticheck.evolution	Checks for static Smartcard Evolution content
de.umlsec.check.template	Template for creating CARiSMA plugins
de.umlsec.check.emfdelta	Plugin to test the EMF-Delta support

Table A.1: CARiSMA Plugins

A.2 UMLchange Profile Diagram

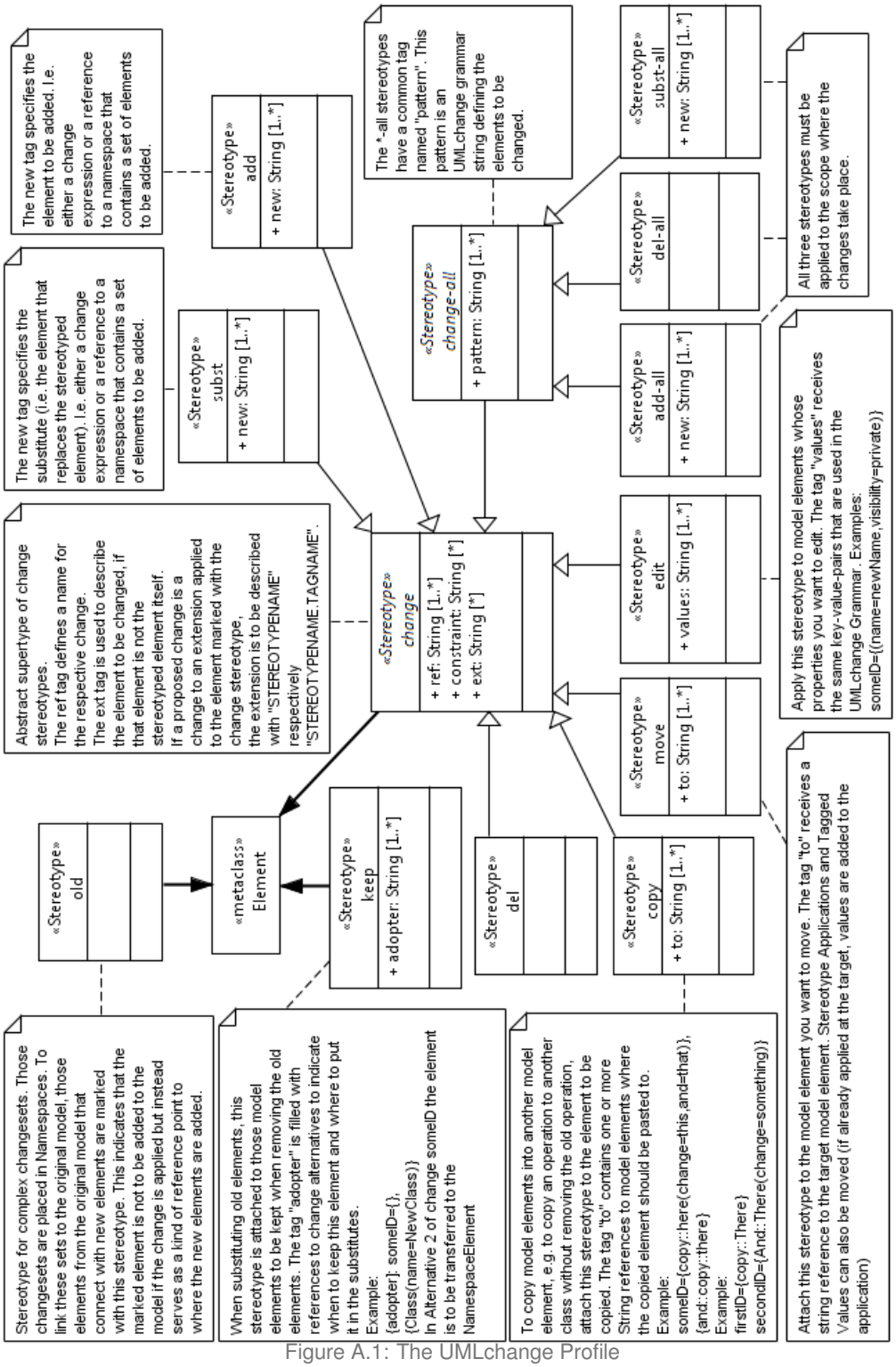

Figure A.1: The UMLchange Profile

A.3 UMLchange Grammar Keys and Values

Metaclass	Key(s)	Type	Description
all named elements	name	String	model element name
	visibility	Enumeration	public, private, protected or package
Property (Tagged Value)	value	String, Reference	new tagged value
Parameter, Property, Operation	type	String, Reference	Primitive Type or other classifier
Association	sourceEndKind, targetEndKind	Enumeration	composite, shared or none
	sourceLowerBound, targetLowerBound, sourceUpperBound, targetUpperBound	String	1,m,n or *
	sourceNavigable, targetNavigable	Boolean	true or false
	sourceEndName, targetEndName	String	end role name
	source, target	Reference	qualified classifier
Dependency, Usage	supplier, client	Reference	qualified classifier
Deployment	deployedArtifact	Reference	qualified artifact
	location	Reference	qualified node
CommunicationPath	source, target	Reference	qualified node
Transition	source, target	Reference	qualified state
Constraint	language	String	constraint language
	specification	String	constraint body

Table A.2: Common Grammar Keys and their Values

A.4 Implementation of Evolution stereotypes

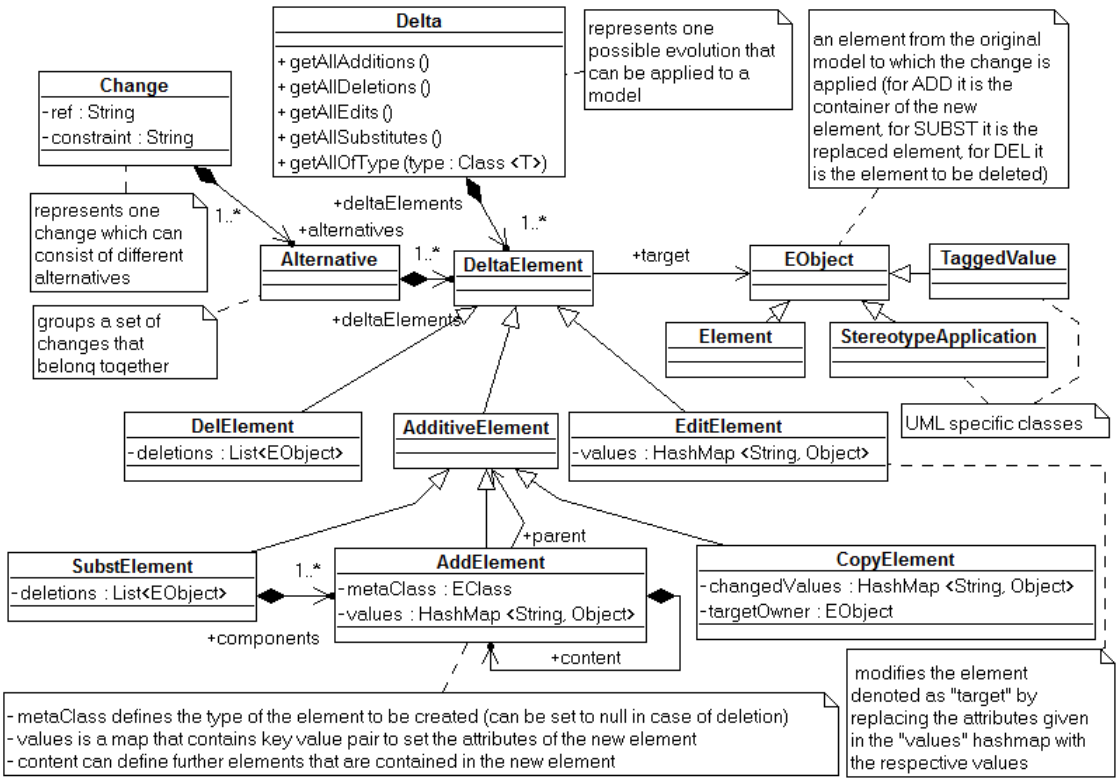

Figure A.2: The delta model

Figure A.2 shows the delta model of CARiSMA that provides the support for evolution-aware checks. Basically, all evolution descriptions given in UMLchange (see Chapter 2) are translated into elements of the delta model, so-called delta elements. While *« add »*, *« del »*, *« subst »*, *« edit »* and *« copy »* each have their own delta element types, the other stereotypes are translated to sets of delta elements equivalent to their original intent.

Using *« move »* is equivalent to changing the owner of the targeted element. Therefore, applications of the stereotype are translated to an appropriate EditElement. To manage multiple elements the stereotypes *« del-all »*, *« add-all »* and *« subst-all »* are used, which are each translated to an equivalent set of the corresponding single-change version. Their targets are identified according to the {pattern} tag entries. Elements marked with *« keep »* are processed in two possible ways. If the adopter of the marked element is described with simple element descriptions, an element description of the marked element is generated and integrated into the adequate AddElement. When the adopter is a modelled element in a change namespace, the kept element is transferred to the adopter without creating any delta elements.

New model elements modelled in complex change namespaces are translated to EditElements changing their respective owner. Only the elements directly contained in the complex change namespace or in a model element marked with *« old »* need to be edited.

A.5 Export of Evolution Information for SeTGaM

This section shows an example of how an annotated UMLchange description of an evolution is exported into XML for further processing in model-based testing, and the SeTGaM process, respectively. The integration of the tools has been discussed in Section 4.

The example is taken from the evolution of the GP specification V2.1. It is the new requirement that also non-security domain applications can terminate the card (i.e. to change card life cycle state from CARD_LOCKED to TERMINATED), if the application has the *card lock* privilege. This privilege is checked in the guard of the transition between the states *card locked* and *terminated*. Hence, in V2.2 of the GP life cycle a new transition with guard checking the privilege is added. This is expressed by the stereotype *« add »* attached to the region. The new transition is described using the UMLchange grammar in the tag *{ add }*. In this case it is:

« add »
{ add=Transition(name=setStatusCardLockedToTerminated_privilegedApp,source=CARD_LOCKED, target=TERMINATED,content=<Constraint(language=OCL, specification=...)>) }

It will insert a transition named *setStatusCardLockedToTerminated_privilegedSD* from the *card locked* state to the *terminated* state, without changing the other transitions. We furthermore add a guard (i.e. a constraint) as subselement of the new transition. In the constraint which is defined in OCL the *card lock* privilege is checked. The delta description of the above mentioned addition of the transition is as follows:

```
<Delta>
 <deltaElements>
  <AddElement>
    <target class="NamedElement">
      <type>Region</type>
      <name>CardLifeCycle</name>
      <xmiID>Hyv5YGsxEeCtltRm1xIPQ</xmiID>
    </target>
    <typename>Transition</typename>
    <values>
      <entry name="name">setStatusCardLockedToTerminated_privilegedSD</entry>
      <entry name="source">.::CARD_LOCKED</entry>
      <entry name="target">.::TERMINATED</entry>
    </values>
    <content>
      <AddElement>
        <typename>Constraint</typename>
        <values>
          <entry name="language">OCL</entry>
          <entry name="specification">...</entry>
        </values>
      </AddElement>
    </content>
  </AddElement>
 </deltaElements>
</Delta>
```

A.6 Algorithm Rules of the Activity to Petri Net Converter

Rule 1

```
1: if currentElement has not been visited then
2:     Add new Place to PetriNet
3:     for all  Element directly following currentElement  do
4:         ConvertA2P(petri, followElement)
5:     end for
6: end if
```

Rule 2

```
1: if currentElement has not been visited  then
2:     Add new Transition to PetriNet
3:     for all  Element directly following currentElement  do
4:         ConvertA2P(petri, followElement)
5:     end for
6: end if
```

Rule 3

```
1: if currentElement has not been visited then
2:     Add new Place to PetriNet
3:     Add new Arc from source to new Place
4:     Add new Arc from new Place to target
5:     ConvertA2P(petri, target)
6: end if
```

Rule 4

```
1: if currentElement has not been visited then
2:     Add new Transition to petriNet
3:     Add new Arc from source to new Transition
4:     Add new Arc from new Transition to target
5:     ConvertA2P(petri, target)
6: end if
```

Rule 5

```
1: if currentElement has not been visited then
2:     Add new Arc from source to target
3:     ConvertA2P(petri, target)
4: end if
```

Rule 6

```
1: if currentElement has not been visited then
2:     Add new Arc from source to target
3:     ConvertA2P(petri, target)
4: end if
```

A.7 ESSOS 2012: A Sound Decision Procedure for the Compositionality of Secrecy

- Martín Ochoa, Jan Jürjens, Daniel Warzecha. A Sound Decision Procedure for the Compositionality of Secrecy. To appear in *4th International Symposium on Engineering Secure Software and Systems (ESSOS 2012)*, Springer, LNCS, 2012

A Sound Decision Procedure for the Compositionality of Secrecy*

Martín Ochoa[1,3], Jan Jürjens[1,2], and Daniel Warzecha[1]

[1] Software Engineering, TU Dortmund, Germany
[2] Fraunhofer ISST, Germany
[3] Siemens AG, Germany
firstname.lastname@cs.tu-dortmund.de

Abstract. The composition of processes is in general not secrecy preserving under the Dolev-Yao attacker model. In this paper, we describe an algorithmic decision procedure which determines whether the composition of secrecy preserving processes is still secrecy preserving. As a case-study we consider a variant of the TLS protocol where, even though the client and server considered separately would be viewed as preserving the secrecy of the data to be communicated, its composition to the complete protocol does not preserve that secrecy. We also show results on tool support that allows one to validate the efficiency of our algorithm for multiple compositions.

1 Introduction

The question of compositional model-checking [5] is crucial for achieving scalable verification of systems. Moreover, compositionality of secure protocols can cause unforeseen problems (see for example problems on the SAML based single-sign-on used by Google in [3]). Although this question has been studied extensively in the literature, in this paper we propose a novel methodology to specify protocols such that given a finite set of session variables, compositionality is decidable. This is equivalent to restrict the analysis of processes to finitely many runs. Indeed vulnerabilities in authentication protocols have been shown to be limited to finitely many parallel instantiations [14]. Technically, our analysis generates finite *dependency trees* that can be stored for further deciding on future compositions. The process of merging such trees can be shown to be empirically more efficient than re-analysing the composition from scratch, and constitutes our central contribution. Moreover, this process is relatively sound and complete with respect to the First Order Logic analysis of [10].

To validate our approach we have implemented our algorithm as an extension to the UMLsec Tool Suite. This validates the usability of the approach in a

* This research was partially supported by the EU projects Security Engineering for Lifelong Evolvable Systems (Secure Change, ICT-FET-231101) and NESSoS (FP7 256890). Additionally the paper has been supported by the MoDelSec Project of the DFG Priority Programme 1496 "Reliably Secure Software Systems – RS^3".

$E ::=$	expression
d	data value ($d \in \mathcal{D}$)
N	unguessable value ($N \in$ **Secret**)
K	key ($K \in$ **Keys**)
$\mathsf{inp}(c)$	input on channel c ($c \in \mathsf{Channels}$)
x	variable ($x \in$ **Var**)
$E_1 :: E_2$	concatenation
$\{E\}_e$	encryption ($e \in$ **Enc**)
$\mathcal{D}ec_e(E)$	decryption ($e \in$ **Enc**)
$\mathcal{S}ign_e(E)$	signature creation ($e \in$ **Enc**)
$\mathcal{E}xt_e(E)$	signature extraction ($e \in$ **Enc**)

Fig. 1. Grammar for simple expressions in the Domain-Specific Language

formally sound Software Development process, and has allowed us to measure the efficiency of our approach given the derivation trees for up to 500 small components (amounting to about 1000 messages).

This paper is organized as follows: Section 2 presents some preliminaries about stream processing functions, composition and secrecy. Section 3 describes the main verification strategy, whereas Section 4 shows its application to an insecure variant of TLS. Section 5 reports on efficiency of the decision procedure compared to re-verification. Finally, Related Work is discussed on Section 6 and we conclude with Section 7.

2 Preliminaries

In [10] the underlying process model used to model component communication is based on Broy's stream-processing functions [4]. A *process* is of the form $P = (I, O, L, (p_c)_{c \in O \cup L})$ where $I \subseteq \mathsf{Channels}$ is called the set of its *input channels* and $O \subseteq \mathsf{Channels}$ the set of its *output channels* and where for each $c \in \tilde{O} \stackrel{\text{def}}{=} O \cup L$, p_c is a closed program with input channels in $\tilde{I} \stackrel{\text{def}}{=} I \cup L$ (where $L \subseteq \mathsf{Channels}$ is called the set of *local channels*). From inputs on the channels in $\tilde{I}$ at a given point in time, p_c computes the output on the channel c. Each channel defines thus a stream processing function based on its input variables allowing for a rigorous notion of sequential composition, which is denoted by $\otimes$. For cryptographic protocol analysis, the programs are specified in a domain specific language defined by the expressions as in Fig. 1 and a simple programming language with non-deterministic choice (where loops can be modelled by using local channels). To proceed with the Dolev-Yao secrecy analysis, one defines rules to translate programs to first-order logic formulas. With the predicate $\mathsf{knows}(E)$ we can express the fact that an adversary may know an expression E during the execution of the protocol, therefore it models the *man in the middle*. For example, if-constructs are translated by the following formula:

$$\begin{aligned}\phi(\textit{if } E = E' \textit{ then } p \textit{ else } p') = \forall i_1, \ldots, i_n. \big[&\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow \\ &[E(i_1, \ldots, i_n) = E'(i_1, \ldots, i_n) \Rightarrow \phi(p)] \\ &\wedge [E(i_1, \ldots, i_n) \neq E'(i_1, \ldots, i_n) \Rightarrow \phi(p')]\big]\end{aligned}$$

To verify the secrecy of data $s \in$ **Secret**, one then has to check whether the adversary can derive $\mathsf{knows}(s)$, given the formulas that arise from the evaluation ϕ of the single program constructs and the following axioms:

$$\forall E_1, E_2.$$
$$\big[\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \;\Rightarrow\; \mathsf{knows}(E_1 :: E_2) \wedge \;\mathsf{knows}(\{E_1\}_{E_2}) \wedge \;\mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\big]$$
$$\wedge \big[\mathsf{knows}(E_1 :: E_2) \;\Rightarrow\; \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big]$$
$$\wedge \big[\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \;\Rightarrow\; \mathsf{knows}(E_1)\big]$$
$$\wedge \big[\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \;\Rightarrow\; \mathsf{knows}(E_1)\big]$$

The conjunction of the formulae ϕ for all channel programs of a process is called ψ. In the following, we will discuss composition at the level of this First Order Logic translation and not at the underlying stream processing function level because the FOL translation contains implicitly all the possible actions an adversary process could perform (defined by the structural formulas). Moreover, and adversary that completely controls the communication channels between processes, might act as an adaptor creating unforeseen compositions between input and output channels. Therefore we want to approximate the knowledge an adversary can gain given all possible outputs of the processes (considering all possible well-formed inputs).

3 Decision procedure

If we assume that both P and P' preserve the secrecy of the data value s, our goal is to show a procedure so that we can decide if $\psi(P \otimes P') \nvdash \mathsf{knows}(s)$. In general this does not hold. For example consider a process P which outputs $\{s\}_K$ and a process P' which outputs K^{-1}. Independently this both processes preserve the secrecy of s, but when composed an adversary could trivially compute s. To achieve this, we will construct proof artifacts on each single process called *derivation trees*. Moreover, in order ensure that this trees are finite, we will require that the number of *keys* and *nonces* are also finite and that the conditions in the "if" constructs of the process programs admit only variables that are of type *key* or *nonce*. This will imply the decidability of our approach.

Definition 1 (Subterm). *We say that a symbol x is a subterm of the symbol T and denote it $x \hat{\in} T$ when one of the following holds:*

$x{=}T$
$T{=}\{T'\}_{\mathsf{K}}$ *and* $x \hat{\in} T'$
$T{=}\mathcal{S}ign_{\mathsf{K}}\{\mathsf{T}'\}$ *and* $x \hat{\in} T'$
$T{=}\ h{::}k$ *and* $x \hat{\in} h$ *or* $x \hat{\in} k$

Example $\mathsf{s} \hat{\in} \{\mathsf{s}\}_{\mathsf{K}}$ but is not true that $\mathsf{K} \hat{\in} \{\mathsf{s}\}_{\mathsf{K}}$. We denote this by $\mathsf{K} \hat{\notin} \{\mathsf{s}\}_{\mathsf{K}}$. This means that an adversary could potentially compute s from $\{\mathsf{s}\}_{\mathsf{K}}$ using the structural formulas with the necessary previous knowledge, but he could not compute K.

Definition 2 (Inverse). *Let $x \hat{\in} J$. We define the cryptographic inverse of a symbol J with respect to x and denote it $\mathsf{J}^{-1}(\mathsf{x})$ in the following way:*

$$\mathsf{x}^{-1}(\mathsf{x}) = \epsilon$$

If J=h::k *and* $\mathsf{x} \hat{\notin} \mathsf{h}$ *then* $\mathsf{J}^{-1}(\mathsf{x})$=$\mathsf{k}^{-1}(\mathsf{x})$
If J=h::k *and* $\mathsf{x} \hat{\notin} \mathsf{k}$ *then* $\mathsf{J}^{-1}(\mathsf{x})$=$\mathsf{h}^{-1}(\mathsf{x})$
If J=h::k *and* $\mathsf{x} \hat{\in} \mathsf{k}$, $\mathsf{x} \hat{\in} \mathsf{h}$ *then* $\mathsf{J}^{-1}(\mathsf{x}) = \mathsf{and}(\mathsf{h}^{-1}(\mathsf{x}), \mathsf{k}^{-1}(\mathsf{x}))$
If J=$\{J'\}_{\mathsf{K}}$ *or* J=$\mathcal{S}ign_{\mathsf{K}}\{J'\}$ *then* $\mathsf{J}^{-1}(\mathsf{x}) = \mathsf{or}(\mathsf{J'}^{-1}(\mathsf{x}), K^{-1})$.

Example Let $\mathsf{J} = \{\{\mathsf{s}\}_{\mathsf{K}_1}\}_{\mathsf{K}_2}$. Then $\mathsf{J}^{-1}(\mathsf{s}) = \mathsf{or}(K_1^{-1}, K_2^{-1})$ which we will interpret later as "to preserve the secrecy s we need to preserve either K_1^{-1} or K_2^{-1}".

Let $\psi(P)$ be the first order logic formula associated to P. We define $\bar{\psi}(P)$ to be the set of instantiated formulas of $\psi(P)$ with all possible values satisfying the constraints in $\psi(P)$. Since we require that all constraints only contain variables of type *key* or *nonce*, and that the respective sets are finite, then $\bar{\psi}(P)$ is also finite. It is possible to show by induction on the program constructs that $\bar{\psi}(P)$ consists of formulas F_i of the form $\mathsf{knows}(\mathsf{E}_i) \Rightarrow \mathsf{knows}(\mathsf{J}_i)$ for closed expressions E_i and J_i. Let $\mathsf{Pres(x,P)}$ be the following inductively defined predicate:

$[(\forall \mathsf{F}_i \in \bar{\psi}(P)\ \mathsf{x} \hat{\notin} J_i\ ) \Rightarrow \mathsf{Pres(x,P)})$
$\wedge\ (\ \forall \mathsf{F}_i \in \bar{\psi}(P)\ (\mathsf{x} \hat{\in} \mathsf{J}_i) \Rightarrow ((\mathsf{Pres(E}_i\mathsf{,P)} \vee \mathsf{Pres(J_i}^{-1}\mathsf{(x),P)})$
$\wedge\ (\ (\mathsf{x} = \{\mathsf{x'}\}_{\mathsf{K}} \vee \mathsf{x} = \mathcal{S}ign_{\mathsf{K}}\{\mathsf{x'}\}) \Rightarrow (\mathsf{Pres(x',P)} \vee \mathsf{Pres(K,P)})$
$\wedge\ ((\mathsf{x} = \mathsf{h::k} \Rightarrow (\mathsf{Pres(h,P)} \vee \mathsf{Pres(k,P)})$
$\wedge\ ((\mathsf{x} = \mathsf{and(h,k)} \Rightarrow (\mathsf{Pres(h,P)} \wedge \mathsf{Pres(k,P)})$
$\wedge\ ((\mathsf{x} = \mathsf{or(h,k)} \Rightarrow (\mathsf{Pres(h,P)} \vee \mathsf{Pres(k,P)})\,]$
$\Rightarrow \mathsf{Pres(x,P)}$

and $\neg\mathsf{Pres}(\epsilon\mathsf{,P})$. If we can not derive $\mathsf{Pres(x,P)}$ for some x, it follows $\neg\mathsf{Pres(x,P)}$.

Theorem 1. *If it is possible to derive* $\mathsf{Pres(x,P)}$ *(conversely* $\neg\mathsf{Pres(x,P)}$*) then* $\psi(P) \nvdash \mathsf{knows}(x)$ *(*$\psi(P) \vdash \mathsf{knows}(x)$*).*

Proof idea In case $\neg\mathsf{Pres}(\epsilon\mathsf{,P})$ since $\mathsf{knows}(\epsilon) \in \bar{\psi}(P)$ for all P. If $\forall \mathsf{F}_i \in \bar{\psi}(P)$ $\mathsf{x} \hat{\notin} J_i$ that means that there is no formula in $\bar{\psi}(P)$ containing x in a conclusive position, and therefore there is no way to derive $\mathsf{knows(x)}$ from the structural formulas. Now assume it is possible to derive $\mathsf{Pres(x,P)}$. We have already covered the base cases so we can assume that $\psi(P) \nvdash \mathsf{knows}(y)$ for all the $\mathsf{Pres(y,P)}$ $\mathsf{y} \neq \mathsf{x}$ needed in the precondition. Since in this formulas all the cases where we could apply the Structural Formulas are covered, it is impossible to derive $\mathsf{knows(x)}$. The case $\neg\mathsf{Pres(x,P)}$ is similar. □

Notice that the converse does not hold, that is $\psi(P) \nvdash \mathsf{knows}(x)$ does not mean we can derive $\mathsf{Pres(x,P)}$, because for some pathological cases we will have an infinite loop, for example for $\bar{\psi}(P) = \mathsf{knows(x)} \Rightarrow \mathsf{knows(x)}$. It is although easy to detect and avoid this loops in a machine implementation of the preservation predicate by running an initial check on the formulas. This makes the verification of the $\mathsf{Pres(x,P)}$ predicate sound and complete with respect to the First Order Logic embedding of the process programs.

As we derive $\mathsf{Pres(s,P)}$ for some symbol s and formulas P, we can build a *derivation tree* consisting of the symbols we need to consider to be able to conclude the preservation status of s. If we generate and store the derivation tree for every symbol x appearing in a process P in a relevant position (that

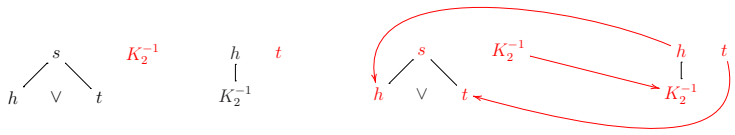

Fig. 2. Processes P and P' before and after composition

is $x \hat{\in} J_i$ for some i), then we can decide whether the composition with process P' will preserve the secrecy of any given symbol if we also have the derivation trees for P'. Consider for example $P = (\{s\}_{h::t}, K_2^{-1})$ and $P' = (\{\mathsf{h}\}_{\mathsf{K}_2}, t)$. The symbol dependency trees of both process are depicted in Fig. 2 (the symbols in red are the ones which secrecy is compromised). Clearly both processes preserve separately the secrecy of s. To see if the composition also does, we update the information on the tree of s by checking whether the truth values of h and t are altered by the composition as depicted in Fig. 2.

4 An insecure variant of the TLS protocol

As an example we apply our approach to a variant of TLS [2] (not the version of TLS in current use) that does not preserve secrecy as a composition of the client C, the server S and the authority CA . We have that the predicate for C and S after the programs are translated to F.O.L are (for details on the translation see [10]) :

$$\psi(\mathsf{C}) = \mathsf{knows}(N_C :: K_C ::\mathcal{S}ign_{\mathsf{K_C}^{-1}}\{\mathsf{C} :: \mathsf{K_C}\}) \wedge (\mathsf{knows}(s_2) \wedge \mathsf{knows}(s_3) \Rightarrow \mathsf{knows}(\{m\}_y))$$
$$\psi(\mathsf{S}) = \mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(c_3) \Rightarrow \mathsf{knows}(N_S :: \{\ \mathcal{S}ign_{\mathsf{K_S}^{-1}}\{\mathsf{kcs} :: \mathsf{c_1}\}\ \}_{c_2})$$

where $\{s_3\}_{K_{CA}} = S :: x \wedge \{Dec_{K_C^{-1}}(s_2)\}_x = y :: N_C$ and $\{c_3\}_{c_2} = C :: c_2$ where $\mathsf{key}(c_2), \mathsf{key}(x)$ and $\mathsf{key}(y)$. The set of keys is $\mathsf{Keys} = \{K_A, K_A^{-1}, k_{CS}, k_A, K_C, K_C^{-1}, K_S, K_S^{-1}, K_{CA}, K_{CA}^{-1}\}$ where k_{CS} and k_A are symmetric keys. The nonces are $\mathsf{Nonces} = \{N_C, N_S, N_A\}$. We assume that the authority CA has already distributed certificates to all parties and that the adversary is in possession of this information: $\mathsf{knows}(K_{CA}) \wedge \mathsf{knows}(\mathcal{S}ign_{\mathsf{K_{CA}}^{-1}}\{\mathsf{S} :: \mathsf{K_S}\}) \wedge \mathsf{knows}(\mathcal{S}ign_{\mathsf{K_{CA}}^{-1}}\{\mathsf{A} :: \mathsf{K_A}\})$.

We further assume that an adversary posses a key pair $\mathsf{knows}(K_A) \wedge \mathsf{knows}(K_A^{-1})$. Now we show that $\mathsf{C} \otimes \mathsf{S}$ does not preserve the secrecy of m although C and S separately do. First of all, in order to be able to apply our approach and generate the dependency tree, we have to solve the constraints for all the processes involved. So we have:

$$\bar{\psi}(\mathsf{C}) = \ \mathsf{knows}(N_C :: K_C ::\mathcal{S}ign_{\mathsf{K_C}^{-1}}\{\mathsf{C} :: \mathsf{K_C}\})$$
$$\wedge\ (\mathsf{knows}(\{\ \mathcal{S}ign_{\mathsf{x}^{-1}}\{\mathsf{y} :: \mathsf{N_C}\}\ \}_{K_C}) \wedge \mathsf{knows}(\ \mathcal{S}ign_{\mathsf{K_{CA}}^{-1}}\{\mathsf{S} :: \mathsf{x}\}\ ) \Rightarrow \mathsf{knows}(\{m\}_y))$$

where $x \in \{K_C, K_S, K_A\}$ (the public keys) and $y \in \{k_A, k_{CS}\}$ (the symmetric keys). We do not explicit the whole dependency tree for C but we note that the secrecy of m is preserved because: if $y = k_{CS}$ the adversary does not have knowledge of k_{CS}; if $y = k_A$ the adversary would need knowledge of $\mathcal{S}ign_{\mathsf{x}^{-1}}\{\mathsf{k_A} :: \mathsf{N_C}\}$

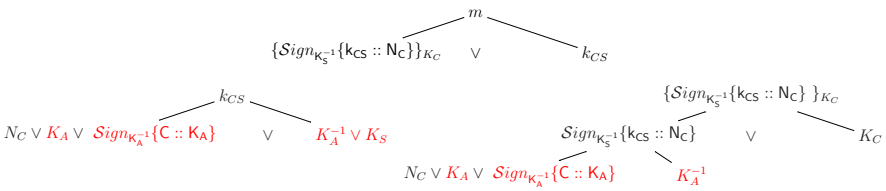

Fig. 3. Partial trees for m in C and for k_{CS} and $\{ Sign_{K_S^{-1}}\{k_{CS} :: N_C\} \}_{K_C}$ in S

and $Sign_{K_{CA}^{-1}}\{S :: x\}$ for some x. Since he only knows $Sign_{K_{CA}^{-1}}\{S :: K_S\}$ then $x = K_S$. In that case to gain knowledge of $Sign_{K_S{}^{-1}}\{k_A :: N_C\}$ he needs to posses $K_S{}^{-1}$ which he does not. In Figure 3 we depict partially this dependency tree for the case $y = k_{CS}$, $x = K_S$. Now, the instantiated formulas for S are:

$$\bar{\psi}(S) = \mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(Sign_{c_2^{-1}}\{C :: c_2\}) \Rightarrow \mathsf{knows}(N_S :: \{ Sign_{K_S^{-1}}\{k_{CS} :: c_1\} \}_{c_2})$$

with $c_1 \in \{N_S, N_C, N_A\}$, $c_2 \in \{K_C, K_S, K_A\}$. The secrecy of m is preserved in S simply because m is not a subterm of any formula in S.

To see why the composition fails to preserve secrecy, we illustrate (partially) the dependency trees of k_{CS} and $\{ Sign_{K_S^{-1}}\{k_{CS} :: c_1\} \}_{c_2}$ in case $c_1 = N_C$ and $c_2 = K_A$ in Fig. 3. In fact, since C leaks N_C and K_C, k_{CS} turns to be not secret after composition in the tree of S. This also modifies the secrecy status of $\{ Sign_{K_S^{-1}}\{k_{CS} :: N_C\} \}_{K_C}$ which results in a secrecy violation for m after updating the tree of C. We have performed a similar analysis for a fix to this protocol proposed in [10] where the composition preserves secrecy but for space reasons we do not explicit the details here.

5 Validation and Efficiency

We have implemented our approach as an extension to the UMLsec tool support [4]. That is, we can extract the protocol specification from a sequence diagram using the DSL described in Sect. 2 and translate it to First Order Logic. Since by construction each guard accepts only finitely many messages (depending on the set of keys and nonces), we can build finite dependency trees for all relevant symbols by means of a properly generated prolog program.

Reasoning about composition amounts then to join the trees from two processes. Therefore, we can at least avoid to recompute the constraint solving for the single processes. We have conducted experiments to measure the time of the composition, and compare it to the overall process of constraint-solving and prolog generation as depicted in Table 1. The first column contains the number of messages for a single session of the composition and the second column corresponds to the number of composed processes. The third column is the time

[4] http://www-jj.cs.tu-dortmund.de/jj/umlsec/

# Messages	# Compositions	Generation trees (ms)	Composition (ms)
11	5	3660	47
21	10	6214	88
31	15	9323	114
51	25	15406	198
101	50	31730	401
501	250	182771	1948
1001	500	375474	3963

Table 1. Execution times of our experiment

in ms. needed to extract the FOL formulas from the UML diagram and generate the derivation trees. The last column is the time needed for deciding the composition given the single derivation trees. In other words, if we would have a repository of 500 processes that by themselves are secrecy preserving, and we would like to check whether the composition of any 5 of them is also secrecy preserving, it would be highly desirable if we could use the existing results as opposed to re-verify from scratch every time.

6 Related Work

Overviews of applications of formal methods to security protocols can be found for example in [1, 12], some examples in [11, 13]. The question of protocol composition has been studied by different authors. More prominently, Datta, Mitchell et al. [6] have defined the PCL (Protocol Composition Logic), aimed at the verification of security protocol by re-using proofs of sub-protocols using a Hoare-like logic, focusing on authenticity. Guttmann [7] gives results about protocol composition at a lower abstraction level, considering unstructured 'blank slots' and compound keys that result from hashes of other messages. Jürjens [9] has explored the question of composability aiming at given sufficient conditions under which composition holds. Stoller [14] has computed bounds of parallel executions that could compromise the authenticity of protocols. These approaches aim at giving at a collection of theorems that if satisfied by two protocols in a composition, ensure a given property. One must show (by using a theorem prover, or by hand) that some properties are satisfied by both protocols like *disjointness* in [8]. Our approach differs from this assume/guarantee reasoning in that we efficiently check whether the composition harms secrecy given pre-computed 'proof artifacts': the dependency trees. In other words, we give accurate results about compositions (that are equivalent to re-verification), by amortizing the cost of verification at an initial phase.

7 Conclusions

The problem of compositionality is of particular importance for software development when the security of reusable components has been established, since guarantees about the composition are needed. The decision procedure should also scale efficiently to be of practical use, and most of all, sound. We have shown that our procedure is sound and complete with respect to previous work on First Order Logic protocol verification. This comes at the price of an initial

verification of the single components that considers all the possible acceptable messages. Nevertheless, this is compensated when it comes to decide compositionality with an arbitrary process for which the same process has also taken place, since this can be done very efficiently, as we have empirically tested. There are different ways in which this work could be further extended. On the one hand, one can further explore the efficiency of the approach, for example by formally deriving its complexity . On the other hand, one could extend the approach to cope with the preservation of other security properties like authenticity.

References

1. M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000. 20th International Summer School, Marktoberdorf, Germany.
2. G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Proceedings of the IEEE Infocom*, pages 717–725, 1999.
3. A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps. In V. Shmatikov, editor, *FMSE*, pages 1–10. ACM, 2008.
4. M. Broy. A logical basis for component-based systems engineering. In *Calculational System Design. IOS*. Press, 1999.
5. E. M. Clarke, D. E. Long, and K. L. Mcmillan. Compositional model checking. *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89)* IEEE Computer Society, 1989.
6. A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172(0):311 – 358, 2007. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.
7. J. D. Guttman. Cryptographic protocol composition via the authentication tests. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS '09)*, pages 303–317, Berlin, Heidelberg, 2009. Springer-Verlag.
8. J. D. Guttman, F. Javier, and F. J. T. Fábrega. Protocol independence through disjoint encryption. In *In Proceedings, 13th Computer Security Foundations Workshop. IEEE Computer*, pages 24–34. Society Press, 2000.
9. J. Jürjens. Composability of secrecy. In *Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security*, MMM-ACNS '01, pages 28–38, London, UK, 2001. Springer-Verlag.
10. J. Jürjens. A domain-specific language for cryptographic protocols based on streams. *J. Log. Algebr. Program.*, 78(2):54–73, 2009.
11. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17(3):93–102, 1996.
12. C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 237–250. IEEE Computer Society, 2000.
13. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
14. S. D. Stoller. A bound on attacks on authentication protocols. *Proc. of the 2nd IFIP International Conference on Theoretical Computer Science: Foundations of Information Technology in the Era of Network and Mobile Computing*, 2001.

A.8 AFADL 2012: Vérification et Test pour des systèmes évolutifs

- Elizabeta Fourneret, Fabrice Bouquet Martin Ochoa, Jan Jürjens, Sven Wenzel. Vérification et Test pour des systèmes évolutifs. In *Congrès Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 150–164, Grenoble, France, 2012.

Vérification et Test pour des systèmes évolutifs

Elizabeta Fourneret, Fabrice Bouquet
INRIA/Université de Franche-Comté, Besancon, France
elizabeta.fourneret,fabrice.bouquet@lifc.univ-fcomte.fr
Martín Ochoa, Sven Wenzel, Jan Jurjens
Technical University Dortmund, Germany
martin.ochoa,sven.wenzel,jan.jurjens@cs.tu-dortmund.de

Résumé

Le test à partir de modèle (MBT) est une approche utilisée pour générer des tests afin de permettre de valider que le comportement du système est bien conforme à sa spécification. Nous nous proposons d'étudier cette démarche lors d'évolution du cahier des charges (des exigences) et l'impact que cela peut avoir sur le besoin de tests associé à des propriétés de sécurité. La première étape de cette approche est de s'assurer de la correction du modèle par rapport à ces propriétés de sécurité. Pour se faire, nous avons utilisé les techniques de vérification de propriétés de sécurité pour les systèmes évolutifs proposées dans UMLseCh. Une fois le modèle vérifié, nous pouvons nous en servir avec une approche de type MBT. Sur la base des stéréotypes d'UMLseCh qui permettent de définir les propriétés, nous établissons les besoins de tests associés à ces propriétés. Pour ce faire, nous avons défini une procédure automatique pour produire des tests de sécurité à partir de modèle UMLseCh par l'intermédiaire des "schémas de test". De plus, nous montrons comment notre méthode SeTGaM permet d'améliorer les approches de type MBT en permettant de choisir de façon efficace les tests à (re)générer dans le cas d'une évolution du système. Nous illustrerons notre approche sur un fragment de l'environnement de carte à puce Global Platform [1], dans le cadre du projet Secure-Change [2].

1 Introduction

Dans le cas de systèmes critiques, il est important de pouvoir bien prendre en compte les exigences de sécurité. Pour cela, il faut d'une part que les propriétés de sécurité soient vérifiées sur les modèles d'analyse et d'architecture des systèmes. D'autre part, elles doivent être mises en œuvre

1. www.globalplatform.org
2. Ces travaux sont supportés par l'EU project Security Engineering for Lifelong Evolvable Systems (Secure Change, ICT-FET-231101)`http://www.securechange.eu/`.

par des tests (spécifiques) pour s'assurer que le comportement du système développé est correct du point de vue de la sécurité.

Typiquement, les modèles permettant de vérifier les propriétés de sécurité sont des modèles qui explicitent l'architecture du système. Différentes techniques basées sur le modèle de vérification de propriétés de sécurité sont proposés. Nous pouvons citer par exemple l'approche proposée dans [5] qui propose un profil UML pour définir les propriétés sur un modèle UML et l'outillage nécessaire à la vérification.

Pour la partie tests, nous nous intéressons à l'approche Model-Based Testing (MBT). Elle utilise les modèles pour exprimer les exigences issues d'une spécification et ensuite calculer des tests qui seront exécutés sur le système sous test (SUT - System Under Test), vu comme une *boite noire*. Le MBT permet de garantir des critères de couverture à partir du modèle et des exigences qu'il capture. La qualité du modèle vient de ce travail amont lors de la phase de validation.

Dans le cycle de vie du système, les exigences évoluent (modification, création, suppression). Afin de suivre ces évolutions, le système doit être changé. Pour cela, les modèles sont mis à jour et doivent être re-vérifiés et des nouveaux tests doivent être calculés. Ce processus de maintenance est très coûteux en terme de ressources (financières, humaines) et de temps. Cependant, pour les systèmes critiques il est indispensable. Pour cela, l'un des challenges dans l'ingénierie des systèmes est de minimiser au plus le coût de la maintenance en augmentant la qualité du logiciel [4]. Nous proposons d'utiliser la méthode SeTGaM afin de valider l'évolution du système.
La suite de cet article est organisée de la manière suivante. Dans la Section 2 nous allons définir les différentes approches sur lesquelles ce travail est basé. Le travail réalisé, les résultats et les bénéfices apportés par la technique que nous proposons sont détaillés en Section 3. Nous présentons les travaux connexes en Section 4. Enfin, nous conclurons en donnant les perspectives de ces travaux dans la Section 5.

2 Les bases d'une nouvelle approche

Aujourd'hui lors d'une évolution, le travail de re-verification et la reprise des tests pour les adapter sont vues comme deux processus distincts, comme l'illustre la Figure 1. Notre idée est d'intégrer ces deux approches. De ce fait, l'ingénieur peut profiter des avantages des deux méthodes. Dans cette section nous allons d'abord présenter le processus de vérification UMLsec, la méthode de génération de tests et ensuite la méthode de génération sélective de tests, SeTGaM.

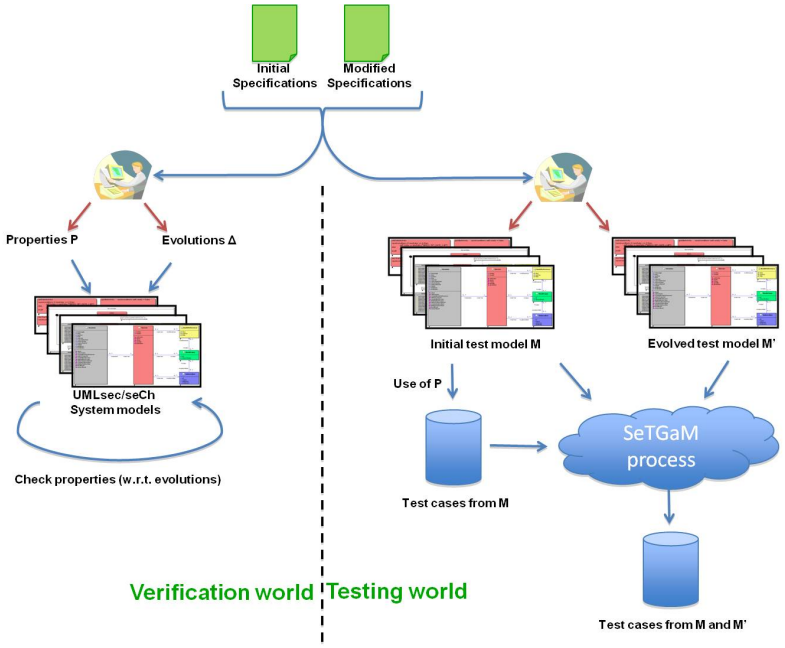

FIGURE 1 – Deux mondes : la vérification et le test

2.1 Le processus de vérification UMLsec

UMLsec se présente sous la forme d'un profil UML. Les *stéréotypes* sont utilisés en combinaison avec des *tags* afin d'exprimer les propriétés de sécurité et les assomptions. Les *contraintes* sont les critères donnés qui permettent de définir si les propriétés sont respectées par le modèle du système à l'aide d'une sémantique précise définie sur le fragment d'UML utilisé. Les informations concernant la sécurité sont ajoutées au modèle à l'aide des stéréotypes. Elles contiennent : les assomptions de sécurité portant sur la partie physique du système, les propriétés liées à la sécurité de la communication et des données échangées ou encore les politiques de sécurités auxquelles le système doit obéir. Le profil UMLsec utilise aussi les *machines à état*, pour plus d'information, le lecteur peut se reporter à la référence [5]. Sur cette base, les exigences liées à la sécurités sont définies : sécurité, intégrité, authenticité et flot de sécurité de l'information. L'outil UMLsec est capable de vérifier les contraintes associées aux stéréotypes UMLsec[12]. Il est aussi possible de générer des formules logiques de premier ordre, sur la base de la sémantique interprétative associée à UMLsec et des annotations servant à décrire les propriétés de sécurité. Un prouver automatique de théorèmes

et un *model-checker* vérifient de façon automatique si les propriétés de sécurité sont respectées. Dans un deuxième temps, il est possible d'utiliser Prolog [3] pour générer automatiquement des scénarios d'attaque qui violent les propriétés de sécurité et ainsi de déterminer et d'enlever cette faille de sécurité.

2.2 Le processus de génération de tests

Comme nous l'avons décrit précédemment, le processus de génération de tests que nous utilisons est basé sur les modèles. Les tests produits de cette façon couvrent les comportements du modèle et utilisent l'animation de ce dernier pour prédire les valeurs attendues.

Ayant pour but de créer des tests dédiés pour valider les propriétés de sécurité sur le système, nous ne pouvons pas utiliser ce modèle. Nous nous reposons sur l'utilisation de scénarios de test qui décrivent l'intention de test, illustrant la propriété de sécurité donnée (i.e préservation de secret, contrôle d'accès, authentification etc.), ou vérifiant la robustesse du système vis à vis de la sécurité. Les scénarios sont écrits avec un langage spécifique de scénarios [11].

Plus exactement, ce langage permet de créer des scénarios de test comme une séquence d'étapes, où chaque étape est composée par un ensemble d'opérations (utilisés par exemple au moins une fois) et atteint une cible donnée (un état particulier du système, activation d'une opération, etc.).

2.3 Génération sélective de tests à partir de modèles

L'approche que nous proposons dans cette sous-section est dédiée aux systèmes évolutifs critiques. Nous décrivons maintenant le processus de génération de tests que nous considérons, dédié aux tests d'évolution. Le processus prend en entrée deux modèles formels, le premier représentant le système avant l'évolution et le second après l'évolution. Il prend en compte également l'ensemble des tests calculés à partir du modèle d'origine.

Notre approche, nommée SeTGaM, représentée sur la Figure 2.3, commence par une analyse des dépendances du modèle d'origine (modèle n) et de celui après évolution (modèle $n+1$). Elle a pour but d'identifier les changements dans les dépendances de données et de contrôle (1). Ensuite, les machines à états sont comparées (2) afin d'identifier les changements entre les modèles et de calculer l'impact de l'évolution sur la suite de tests existante (3). Ce qui permet par la suite de classifier les cas de test de la suite de tests d'origine (4), en se basant sur les résultats de l'animation du modèle (5) qui permet de les définir en tant que :

- **outdated tests**, ces tests ne sont plus valides par rapport à la nouvelle version du système (ils seront utilisés dans le test de stagnation).

3. www.swi-prolog.com

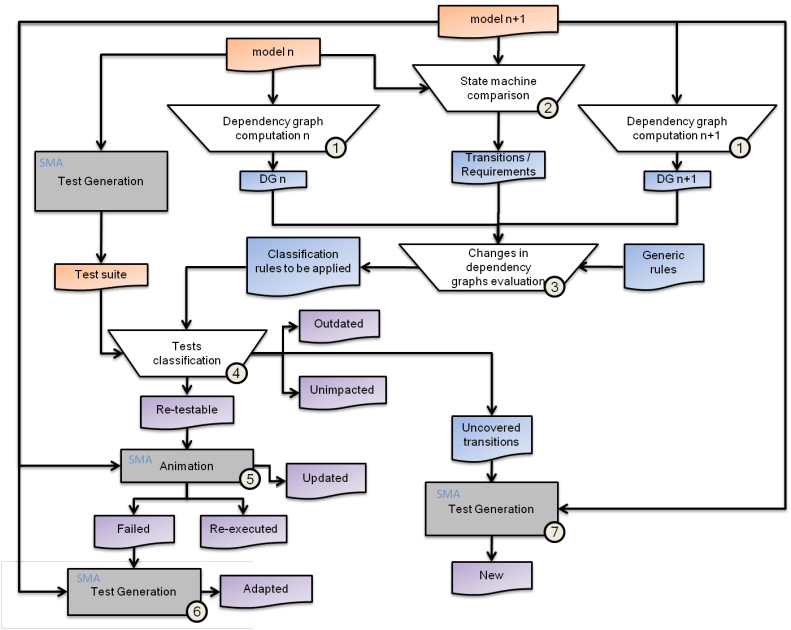

FIGURE 2 – Le processus SeTGaM

- **unimpacted tests**, ces tests ne correspondent pas à des parties/exigences impactées par l'évolution dans le système. Ils sont toujours valides par rapport à la nouvelle version (ils seront utilisés comme test de non régression)
- **updated tests**, ces tests représentent une version mise à jour de la version précédente d'un test. La mise à jour correspond à la modification de l'oracle (ils seront utilisés dans le test d'évolution).
- **adapted tests**, ces tests couvrent des parties/exigences déjà existantes, mais suite à l'évolution et du fait qu'ils couvrent des éléments évolués, le test échoue (en anglais failed). Il est nécessaire de re-calculer une nouvelle séquence de test pour maintenir la couverture (la version précédente du test sera utilisée dans le test de stagnation).
- **re-executed tests**, ces tests couvrent des parties/exigences qui ont évoluées et doivent être re-calculés. La séquence reste identique mais les paramètres et les valeurs de retours sont mises à jour (ils sont utilisés dans le test d'évolution, la version précédente du test sera utilisée dans le test de stagnation) (6)
- **new tests**, ces tests doivent être générés afin de couvrir les nouvelles

fonctionnalités du système qui n'existaient pas auparavant. (7).

3 La nouvelle technique d'intégration

Dans le cas d'une évolution, nous allons voir comment réduire l'effort nécessaire pour re-vérifier le modèle et adapter les tests. Pour cela, nous allons déterminer la différence entre l'ancien et le nouveau modèle et ainsi connaître ce sur quoi porte l'évolution.

3.1 Vérification et test pour le système évolutif

Pendant le processus de vérification, nous pouvons utiliser la différence entre les modèles pour déterminer l'impact sur les propriétés de sécurité issues de la spécification. Ainsi, on ne travaille que sur cette partie au lieu d'analyser de nouveau entièrement le modèle. Les auteurs dans [6] illustrent une telle approche pour l'extension de UML pour la sécurité nommée UMLseCh. Celle-ci permet de vérifier les propriétés de sécurité après l'évolution sur la base de conditions suffisantes et elle permet de définir explicitement une ou plusieurs évolutions.

Il est possible de classifier les tests en analysant les différences entre les deux versions de modèle et en tenant compte des dépendances entre les éléments impactés et les autres. Les auteurs dans [2] proposent la technique nommée SeTGaM, qui permet d'établir sur la base du statut des tests de la première version leur nouveau statut. Il y a actuellement 8 statuts différents : *re-executed, unimpacted, updated, adapted, outdated, failed, removed* et les nouveaux tests dont le statut est *new*. Afin d'obtenir cette classification les auteurs utilisent la différence entre les deux versions en combinaison avec l'analyse d'impact. A cela, s'ajoute aussi le statut précédent du test. Une fois cette classification effectuée, les tests sont ajoutés à 4 suites de test différentes pour permettre à l'équipe de validation d'optimiser son travail. Les suites sont : *Evolution, Regression, Stagnation et Deletion*.

Ces suites de tests sont utilisées dans le processus de maintenance afin de valider que les nouveaux éléments et ceux qui ont été modifiés dans le système, ont évolué correctement. Pour les parties inchangées, ces suites permettent de vérifier qu'elles n'ont pas été impactées par l'évolution bien que le changement a réellement eu lieu. La dernière suite de tests permet de supprimer les tests de la suite *Stagnation* par rapport aux tests issus de la version précédente. Enfin, l'utilisateur peut bénéficier de l'intégration sur plusieurs aspects :

- L'exécution en temps réel peut être réduite significativement, parce que le deuxième calcul du delta par SeTGaM sera évité (car il est déjà effectué par UMLseCh).
- L'évolution (exprimée par le delta calculé) sera formalisée avec la même sémantique pour la vérification de la propriété de sécurité et

pour l'impact sur les tests.
– L'ingénieur travaillera seulement sur *un* modèle, ce qui réduit significativement la probabilité d'introduire des erreurs humaines suite au travail en parallèle avec les deux modèles différents.

3.2 Le processus de l'approche intégrée

L'approche que nous proposons repose sur la complémentarité des deux techniques de vérification et de test. Plus particulièrement, le modèle utilisé pour la génération de tests doit être validé au préalable au regard des propriétés de sécurités considérées. Si ce n'est pas le cas, le modèle peut autoriser un comportement incorrect et voir même, ce qui est encore plus grave, les tests produits demanderont que le système sous test se comporte comme le modèle, c'est à dire d'une manière incorrecte. Cela aura comme résultat une implémentation fausse du système qui sera considérée comme correcte au regard des tests exécutés. Pour cela il est indispensable de s'assurer que le modèle de test respecte les propriétés de sécurité sur lequel les tests générés se basent.

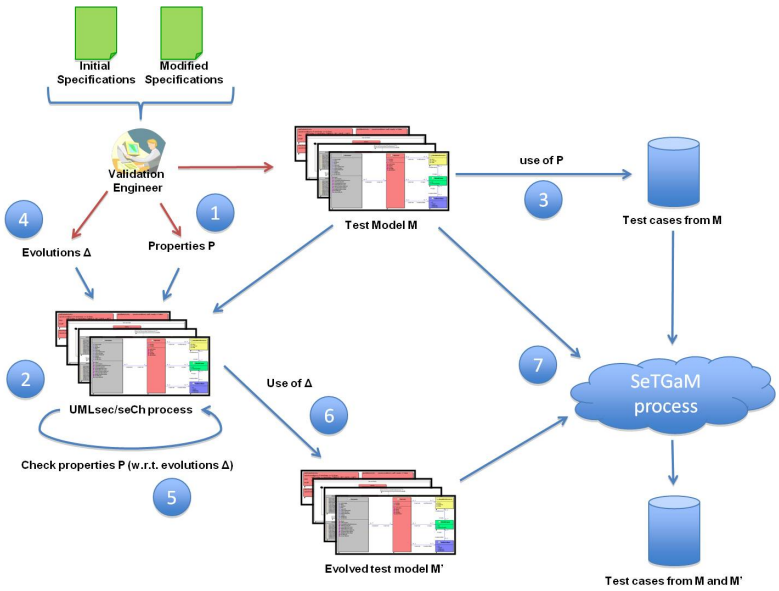

FIGURE 3 – Intégration de la vérification et du test

La collaboration entre la vérification et le test est illustrée sur la Figure 3. Tout d'abord, l'ingénieur de validation crée le modèle de test et exprime les propriétés (étape ①). Il utilise ensuite l'approche UMLsec, étape ②, afin de

valider les propriétés de sécurité et de s'assurer que le modèle respecte les propriétés considérées. Une fois que le modèle est déclaré correct, il peut être utilisé pour la génération de tests dédiés aux propriétés de sécurité (étape ③). Lors d'une évolution du système, les changements sont décrits comme un ensemble de différences entre les deux versions du modèle (étape ④). C'est alors que le processus UMLseCh est utilisé pour valider l'évolution i.e qu'il vérifie les propriétés de sécurité (étape ⑤). Une fois que les propriétés de sécurité sont validées par UMLseCh, qu'elles soient préservées par l'évolution, UMLseCh produit le fichier *delta* contenant les informations sur l'évolution. Les différences entre les deux versions, issus du *delta*, sont utilisées pour calculer le modèle de test évolué (étape ⑥). Ensuite, au niveau de l'étape ⑧ SeTGaM utilise ces trois entités : le modèle initial, le delta et le modèle évolué pour classifier les tests de la version initiale vis-à-vis l'évolution, générer des nouveaux tests si nécessaire et créer la suite de tests pour le système évolué.

Nous définissons ci-dessous les évolutions possibles pour un modèle donné :

- l'addition d'une nouvelle entité dans le modèle (class, state, etc.)
- la suppression (en anglais deletion) d'une entité existante
- la substitution d'une entité du modèle par une autre.

Ces additions, suppressions et substitutions sont spécifiées par des stéréotypes dédiés dans le modèle UML correspondant : *add, del and substitute/substitute-all*. L'exemple d'une telle description en XML est présentée sur la Figure 3.2.

Enfin, nous pouvons résumer l'approche de la manière suivante :

- la vérification du modèle basée sur le delta est appliquée sur le modèle de test et sur les propriétés vérifiées sur le modèle. De plus, les deux approches, vérification et test, sont utilisées par le même acteur (l'ingénieur de validation).
- La technique SeTGaM profite du calcul des différences déjà existant. Ceci est très utile dans deux cas : (*i*) cela rend possible le calcul automatique de la nouvelle version du modèle, et (*ii*) cela évite de recalculer les différences entre les modèles au début de SeTGaM.

3.3 Résultats et bénéfices de l'intégration

Nous avons appliqué notre approche sur un fragment de GlobalPlatform, plus exactement sur la partie de la gestion du cycle de vie de la carte à puce. Nous avons travaillé sur deux spécifications différentes 2.1.1 et 2.2 (configuration UICC) présentées par leurs diagrammes d'états/transitions sur la figure 5. Dans la version 2.1.1 de la spécification, seule une application spécifique appelée ISD peut "terminer" la carte ou rendre la carte inutilisable. Cependant dans la nouvelle version 2.2 UICC, la carte peut être "terminée" par n'importe quelle application ayant les droits. Pour des raisons de confidentialité, nous ne pouvons pas donner plus de détails sur le modèle et la spécification réalisée dans le cadre des études de cas du projet SecureChange.

```
<add>
<element type='transition' name='t1'>
<event name='e' />
<source name='S1' />
<target name='s2'/>
<guard> ocl code... </guard>
<action> ocl code... </action>
</element>
<element> ... </element>
</add>
<del>
<element type='transition' name='t2'/>
<element> ... </element>
</del>
<sub-all>
<sub type='transition' name='t1'>
<target> Ty </target>
<guard> G2 </guard>
</sub>
...
</sub-all>
```

FIGURE 4 – Exemple du fichier XML donnant les évolutions.

Pour cela, nous présentons ici une propriété de sécurité sur les droits d'accès, issue de la spécification, mais pouvant être appliquée aux cartes à puce en général. La propriété spécifie qu'une fois que la carte a été mise dans l'état *"terminated"* (par une application ayant les bons privilèges), il ne doit pas être possible de revenir à un autre état. L'intention de test associée à cette propriété pourrait être décrite de la façon suivante. Tout d'abord, une application ayant les privilèges nécessaires sera sélectionnée (i.e. celle-ci est nécessaire pour changer le statut de la carte). Puis, on utilise l'opération dédiée pour changer le statut de la carte à "terminated". Finalement, l'activation de toutes les opérations possibles sera utilisée afin de s'assurer qu'il soit impossible de changer le statut de la carte. Ce scénario, décrit sur la Figure 3.3, nous a permis de générer treize différentes séquences de tests pour la version 2.1.1.

En appliquant la méthode sur la nouvelle version, nous obtenons que la propriété précédemment décrite est correcte avec l'ajout d'une transition dans le modèle de test (exprimée par le delta). Ensuite, nous avons appliqué la méthode SeTGaM. Le calcul de dépendances sur le modèle a classé tous les tests de la première version comme impactés et ils ont été ensuite classés en "Re-executed". De plus, la méthode nous a permis de générer un nouveau

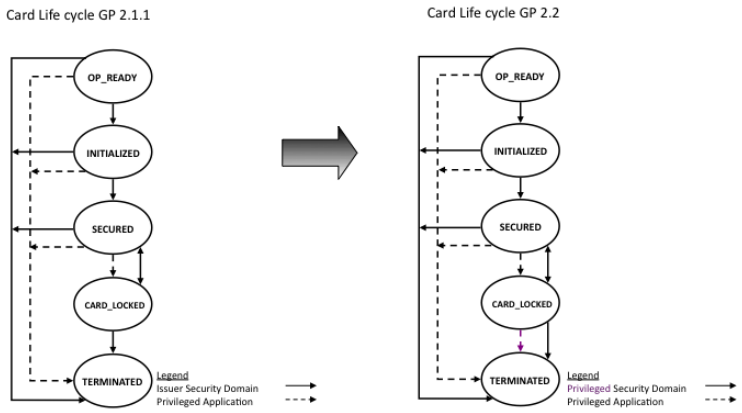

FIGURE 5 – Global Platform Card Life Cycle

```
for_each $X from any_operation
 use any_operation any_number_of_times to_reach
  state_representing selectedApp.privileges.cardTerminatePriv=TRUE
  on_instance sut
 then use setStatus at_least_once
  to_reach state_representing cardState = TERMINATED
  on_instance sut
 then use any_behavior_to_cover at_least_once to_activate $X
 then use getStatus at_least_once to_activate
   behavior_activating {@AIM : SUCCESS}
```

FIGURE 6 – Schéma de test

test, pour compléter la couverture de la nouvelle exigence correspondant à l'ajout de la transition.

Pour conclure, l'ingénieur de test peut modéliser les évolutions prévues dans le modèle de test et vérifier les changements sans forcément re-exécuter tout le processus de vérification depuis le début. En utilisant cette approche, il peut bénéficier des techniques de delta et de vérification. Plus particulièrement, il pourra exploiter les notations UMLseCh afin de calculer le delta

entre le modèle et ses évolutions. Il sera ensuite possible d'utiliser ce résultat comme entrée de la technologie de génération de tests.
D'une part, la génération de tests qui porte sur l'évolution prend en compte deux modèles et calcule leurs différences. D'autre part UMLseCh peut leur fournir un ensemble d'évolutions possibles pour un modèle donné. Il est à noter que ces deux approches considèrent des modèles qui sont créés par des outils de modélisation UML différents (e.g. UMLseCh pour la vérification est basé sur ArgoUML, cependant le logiciel de génération de tests utilise IBM Rational Software Architect). L'information pour le delta est exploitable par les différents outils puisque l'information est transmise à travers la description des changement en XML. Cette solution est la moins invasive en terme d'adaptation des outils déjà existants. Le fait d'avoir déjà le delta calculé pour le processus de génération permet de :

- créer un nouveau modèle issu de l'évolution décrite,
- éviter le calcul de la différence entre modèles, cette information sera directement utilisée depuis le delta,
- appliquer les autres traitements de l'approche en minimisant l'interférence.

4 Les travaux connexes

Lors de l'évolution d'un système, il est nécessaire de faire face à plusieurs challenges :

- définir l'évolution,
- s'assurer que le modèle est correct par rapport à la propriété de sécurité,
- s'assurer de la maintenance des artefacts.

Tout d'abord lorsque la spécification évolue, il faut que le modèle évolue en même temps que le système. Pour cela, premièrement il faut déterminer les éléments qui ont évolué. Deuxièmement, il faut s'assurer que le nouveau modèle est correct par rapport aux propriétés de sécurité issues de la spécification. Enfin, ce changement impactera la base de test. La personne en charge de cela devra s'assurer que les élément non-impactés n'ont pas changé, et vérifier que les fonctionnalités supprimées sont bien enlevées du système.

Dans le test de sécurité à partir des modèles (Model-Based Security Testing MBST), nous retrouvons souvent des approches définis autour des polices de contrôle d'accès. Par exemple, les auteurs en[9] vérifient la "soundness" de la police de sécurité et ensuite son adéquation avec les exigences afin de détecter des conflits entre des règles. Enfin, ils utilisent le test par mutation afin de s'assurer que le code est conforme au modèle de sécurité.

Les auteurs dans [3] décrivent une technique basée sur les modèles dédiés pour la validation des propriétés de sécurité des cartes à puce. Premièrement,

les propriétés de sécurité sont vérifiées sur le modèle utilisant UMLSec et ensuite le modèle est utilisé pour la génération des tests dédiés à la validation de la propriété sur la carte à puce. Ils donnent aussi des règles de transformation des stéréotypes UMLsec en schémas, utilisés pour la génération de tests de sécurité.

Le test des systèmes critiques en évolution, dont le test de non régression, malgré l'effort pour diminuer les coûts, reste toujours l'une des activités les plus chères dans la maintenance logiciels. D'une part il faut des ressources et du temps pour sélectionner les tests, d'autre part le coût dépend de l'approche automatisée ou non. Afin de réduire le coût et d'augmenter la performance, il existe plusieurs stratégies et techniques sur lesquelles les chercheurs travaillent. Cependant, elles sont orientés sur l'aspect fonctionnel. Harrold et Al. [10] expliquent clairement les stratégies de test de non-régression :

- **Tout sélectionner** (*en Anglais Retest-All*) : qui demande l'application de toute la suite de tests avant l'évolution. Dans certaines classifications nous retrouvons cette stratégie en tant qu'une technique de sélection.
- **Sélective** : qui demande de sélectionner un sous-ensemble de tests de la suite de tests avant l'évolution selon différentes techniques.

Nous retrouvons la classification suivante des techniques de sélection des tests : techniques de minimisation, techniques basées sur la couverture, techniques sûres, techniques aléatoires

Les solutions MBT pour le test de non-régression dépendent du type de modèle et de la ou des techniques choisies. Dans Korel et Al. [8] nous retrouvons une étude sur les systèmes SDL [4] et la notion d'exigence pour la génération des tests de non régression. Le but est de couvrir tout le système avec ces tests. Dans le but d'exprimer les exigences, ils utilisent des SDL représentant des fragments du système. Ensuite, ils les rassemblent en un seul système SDL. A partir de là, ils peuvent le transformer en EFSM [5] modèle, qui va être l'entrée pour la génération des tests. Le langage descriptif des *EFSM* est donc utilisé pour repérer plus facilement l'évolution du système et rattacher les exigences qu'ils utilisent. De cette manière, ils ont défini des règles pour l'addition, la suppression et la modification d'une exigence.

Dans Korel et Al. [7], nous retrouvons une méthode qui est un complément de la précédente mais plus précise car elle utilise l'analyse des dépendances sur les données et le contrôle. La sélection de tests de la suite d'origine est effectuée selon des *Patterns*. Les *Patterns* sont créés selon les différents effets : l'effet du modèle sur la modification, la modification sur le modèle et autres

4. Specification Description Language
5. Extended Finite State Machines

effets secondaires causés par la modification. Un test est inclus dans la RTS (suite de test de non régression) si au moins un des patterns d'interaction n'existe pas dans la sélection des tests.

D'autre part Chen et Al. [13], décrivent une analyse des dépendances beaucoup plus approfondie définissant les règles et les dépendances pour chaque modification élémentaire. Au total, nous retrouvons douze dépendances différentes. Ici, le mot *dépendance* est très proche de la signification du *Pattern* définie par Korel et al. La suite de test réduite contient les tests vérifiant les effets directs et indirects du modèle modifié, ayant pour but de couvrir toute les nouvelles dépendances par modification.

Les travaux de Briand et Al.[1] se sont intéressés à UML et plus particulièrement à l'étude des tests de non régression à partir de modèle utilisant les diagrammes de classe, les cas d'utilisation et les diagrammes de séquence. La méthode propose dans un premier temps de définir les différences entre les deux modèles. Ensuite, les tests sont classifiés en trois catégories : tests obsolètes, tests re-testables et tests réutilisables. Finalement, ils ont créé un outil pour des programmes en langage **C** qui automatisent cette méthode, nommé **RTSTool**. C'est ici qu'apparait la prise en compte du cycle de vie des tests. Ceci va nous servir pour le développement de notre méthode.

Nous avons vu plusieurs types de test de non-régression, cependant à notre connaissance, très peu de travaux traitent en même temps la question de la vérification basée sur le modèle, le test de sécurité et l'évolution. C'est pour cela que nous avons choisi de proposer une approche qui permet de prendre en compte en même temps ces trois aspects.

5 Conclusion et Travaux futurs

Avec ce travail nous allons au delà du test de sécurité basé sur le modèle, vu que nous travaillons sur l'aspect évolution. Quand une évolution a lieu dans la spécification, nous sommes en mesure de définir le delta en utilisant les tags crées en XML. Ensuite il nous est possible de vérifier si l'évolution est correcte en utilisant la méthodologie de UMLseCh. Finalement, si le processus de vérification n'échoue pas, le delta et les propriétés de sécurité pourront être utilisés pour la génération sélective des tests par SeTGaM.

Nous avons présenté sur une application notre approche et son implémentation. L'étape suivante est de mieux valider ces travaux. Pour cela, nous allons nous intéresser aux éléments suivants :

– Dans quelle mesure cette approche permet d'augmenter vraiment la confiance dans le modèle pour le test dans le cas d'évolution su système ?

– Est-ce que cette démarche peut être mise en œuvre dans un contexte industriel et quels sont les verrous qui peuvent en empêcher l'adoption ?

De plus, nous allons travailler sur le nombre de propriétés de sécurité et les différentes catégories qui peuvent être prises en compte dans notre approche. Pour cela, nous allons continuer à appliquer cette approche sur les études de cas du projet SecureChange. De plus, UMLsec/Ch propose de créer des scénarios d'attaque en utilisant les propriétés de sécurité. Il serait très intéressant de coupler cela dans le processus de gestion de tests afin d'obtenir d'autres types de tests dédiés à la sécurité. Nous souhaitons continuer le travail sur la transformation des stéréotypes UMLsec en schémas de tests [3] et ainsi disposer d'un processus complètement intégré permettant la génération des tests pour des systèmes évolutives en utilisant SeTGaM.

Références

[1] L.C Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. In *ELSEVIER B.V 2008*, 2008.

[2] Elizabeta Fourneret, Fabrice Bouquet, Frédéric Dadeau, and Stéphane Debricon. Selective test generation method for evolving critical systems. In *REGRESSION'11, 1st Int. Workshop on Regression Testing - co-located with ICST'2011*, Berlin, Germany, March 2011. IEEE Computer Society Press. To appear.

[3] Elizabeta Fourneret, Martn Ochoa, Fabrice Bouquet, and Jan Jürjens. Model-based security verification and testing for smart-cards. In *ARES'11*, 2011. To appear.

[4] Wolfgang Grieskamp. Multi-paradigmatic model-based testing. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, Lecture Notes in Computer Science, pages 1–19. Springer.

[5] Jan Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2005.

[6] Jan Jürjens, Loïc Marchal, Martín Ochoa, and Holger Schmidt. Incremental Security Verification for Evolving UMLsec models. In *ECMFA*, Lecture Notes in Computer Science. Springer, 2011.

[7] Bogdan Korel, Luay H.Tahat, and Boris Vaysburg. Model based regression test reduction using dependence analysis. In *IEEE ICSM '06*, 2006.

[8] Bogdan Korel, Luay H.Tahat, Boris Vaysburg, and Atef J. Bader. Requirement based automated black-box test generation. In *IEEE ICSM '01*, 2001.

[9] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In *MoDELS*, pages 537–552, 2008.

[10] Gregg Rothermel, Mary Jean Harrold, Todd L. Graves, Jung-Min Kim, and Adam Porter and. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodoligy*, 10 : 184–208, avril 2001.

[11] SecureChange. Deliverable 7.3. Available at `http://www.securechange.eu/content/deliverables`, 2011.

[12] UMLsec tool. `http://umlsec.de`, May 2011.

[13] Hasan Ural, Robert L. Probert, and Yanping Chen. Model based regression test suite generation using dependence analysis. In *Proceedings of the third internationnal workshop on Advances in model-based testing*, pages 54–62, 2007.

A.9 SFM 2011: Modelling Secure Systems Evolution

- J. Jürjens, M. Ochoa, H. Schmidt, L. Marchal, S.H. Houmb, S. Islam. Modelling Secure Systems Evolution: Abstract and Concrete Change Specifications (invited lecture). *11th School on Formal Methods (SFM 2011)*, Bertinoro (Italy) 13-18 June 2011.

Modelling Secure Systems Evolution: Abstract and Concrete Change Specifications

Jan Jürjens[1,2] , Martín Ochoa[1], Holger Schmidt[1], Loïc Marchal[3], Siv Hilde Houmb[4] and Shareeful Islam[5]

[1] Software Engineering, Dep. of Computer Science, TU Dortmund, Germany
[2] Fraunhofer ISST, Germany
[3] Hermès Engineering, Belgium
[4] Secure-NOK AS, Norway
[5] School of Computing, IT and Engineering, University of East London, UK
{jan.jurjens,martin.ochoa,holger.schmidt}@cs.tu-dortmund.de
loic.marchal@hermes-ecs.com
sivhoumb@securenok.com
shareeful@uel.ac.uk

Abstract. Developing security-critical systems is difficult, and there are many well-known examples of vulnerabilities exploited in practice. In fact, there has recently been a lot of work on methods, techniques, and tools to improve this situation already at the system specification and design. However, security-critical systems are increasingly long-living and undergo evolution throughout their lifetime. Therefore, a secure software development approach that supports maintaining the needed levels of security even through later software evolution is highly desirable. In this chapter, we recall the UMLsec approach to model-based security and discuss on tools and techniques to model and verify evolution of UMLsec models.

Keywords: Software Evolution, UMLsec, UMLseCh, Security

1 Introduction

Typically, a systematic approach focused on software quality – the degree to which a software system meets its requirements – is addressed during design time through design processes and supporting tools. Once the system is put in operation, maintenance and re-engineering operations are supposed to keep it running.

At the same time, successful software-based systems are becoming increasingly long-living [21]. This was demonstrated strikingly with the occurrence of the year 2000 bug, which occurred because software had been in use for far longer than its expected lifespan. Also, software-based systems are getting increasingly security-critical since software now pervades the whole critical infrastructures dealing with critical data of both nations and also private individuals. There is therefore a growing demand for more assurance and verifiable secure IT systems both during development and at deployment time, in particular also for

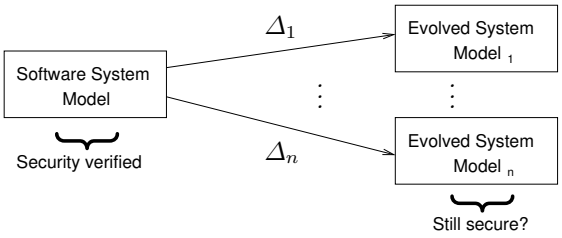

Fig. 1. Model verification problem for n possible evolution paths

long living systems. Yet a long lived system also needs to be flexible, to adapt to evolving requirements, usage, and attack models. However, using today's system engineering techniques we are forced to trade flexibility for assurance or vice versa: we know today how to provide security or flexibility taken in isolation. We can use full fledged verification for providing a high-level of assurance to fairly static requirements, or we can provide flexible software, developed in weeks using agile methodologies, but without any assurance. This raises the research challenge of whether and how we can provide some level of security assurance for something that is going to change.

Our objective is thus to develop techniques and tools that ensure "lifelong" compliance to evolving security requirements for a long-running evolving software system. This is challenging because these requirements are not necessarily preserved by system evolution [22]. In this chapter, we present results towards a security modelling notation for the evolution of security-critical designs, suitable by verification with formally founded automated security analysis tools. Most existing assessment methods used in the development of secure systems are mainly targeted at analysing a static picture of the system or infrastructure in question. For example, the system as it is at the moment, or the system as it will be when we have updated certain parts according to some specifications. In the development of secure systems for longevity, we also need descriptions and specifications of what may be foreseen as future changes, and the assessment methods must be specialized account for this kind of descriptions. Consequently, our approach allows to re-assess the impact that changes might have on the security of systems.

On one hand, a system needs to cope with a given change as early as possible and on the other hand, it should preserve the security properties of the overall system (Fig. 1). To achieve this, it is preferable to analyse the planned evolution before carrying it out. In this chapter we present a notation that allows to precisely determine the changes between one or more system versions, and that combined with proper analysis techniques, allows to reason about how to preserve the existing and new (if any) security properties due to the evolution. Reflecting change on the model level eases system evolution by ensuring effec-

tive control and tracking of changes. We focus in understanding and tracing the change notations for the system design model. Design models represent the early exploration of the solution space and are the intermediate between requirements and implementation. They can be used to specify, analyse, and trace changes directly.

In Sect. 2 we recall the *UMLsec* profile [10, 13], which is a UML [28] lightweight extension to develop and analyse security models, together with some applications. We present a change-modelling profile called *UMLseCh* in Sect. 3. We use UMLseCh design models for change exploration and decision support when considering how to integrate new or additional security functions and to explore the security implications of planned system evolution. To maintain the security properties of a system through change, the change can be explicitly expressed such that its implications can be analysed a priori. The UMLseCh stereotypes extend the existing UMLsec stereotypes so that the design models preserve the security properties due to change.

Although, the question of model evolution is intrinsically related with model-transformation, we do not aim to show an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [7, 1, 2, 25, 20]). However, we rely on UMLsec because it comes with sophisticated tool support[6], and our goal is to present an approach that is a) consistent with the UMLsec philosophy of extending UML b) is meant to be used on the UML fragment relevant for UMLsec.

In Sect. 4 we show some applications of UMLseCh to different diagram types and we discuss how this notation and related verification mechanisms could be supported by an extension of the UMLsec Tool Suite.

2 Background: Secure Systems Modelling with UMLsec

Generally, when using model-based development (Fig. 2a), the idea is that one first constructs a model of the system. Then, the implementation is derived from the model: either automatically using code generation, or manually, in which case one can generate test sequences from the model to establish conformance of the code regarding the model. In the model-based security engineering (MBSE) approach based on the UML [28] extension UMLsec, [11, 13], recurring security requirements (such as secrecy, integrity, authenticity, and others) and security assumptions on the system environment, can be specified either within UML specifications, or within the source code (Java or C) as annotations (Fig. 2b). This way 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 in form of 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

[6] UMLsec tool suite: `http://www.umlsec.de/`

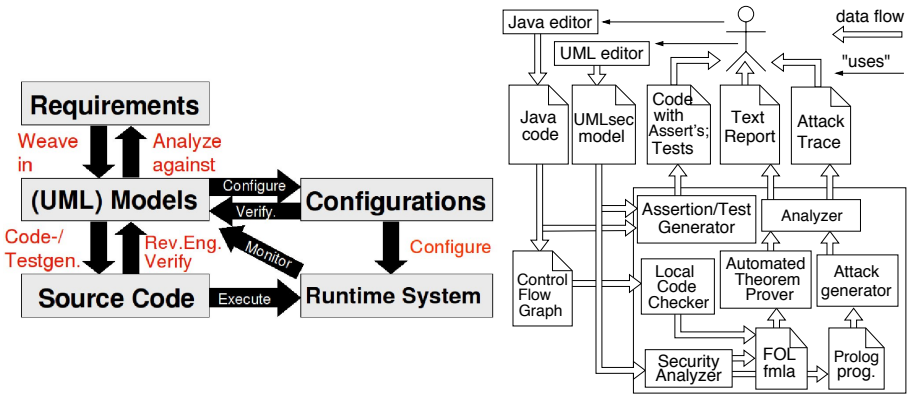

Fig. 2. a) Model-based Security Engineering; b) Model-based Security Tool Suite

determine whether the requirements are met by the system design, by referring to a precise semantics 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. The UMLsec tool-support in Fig. 2b) can be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [29, 14, 18, 8]. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes. The semantics for the fragment of UML used for UMLsec is defined in [13] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces similar to Broy's Focus model, whose behavior can be specified in a notation similar to that of Abstract State Machines (ASMs), and which is equipped with UML-type communication mechanisms. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined. To support stepwise development, we show secrecy, integrity, authenticity, and secure information flow to be *preserved* under refinement and the composition of system components. We have also developed an approach that supports the secure development of layered security services (such as layered security protocols). UMLsec can be used to specify and implement security patterns, and is supported by dedicated secure systems development processes, in particular an Aspect-Oriented Modeling approach which separates complex security mechanisms (which implement the security aspect model) from the core functionality of the system (the primary model) in order to allow a security verification of the particularly security-critical parts, and also of the composed model.

2.1 The UMLsec Profile

Because of space restrictions, we cannot recall our formal semantics here completely. Instead, we define precisely and explain the interfaces of this semantics that we need here to define the UMLsec profile. More details on the formal semantics of a simplified fragment of UML and on previous and related work in this area can be found in [9, 13]. The semantics is defined formally using so-called *UML Machines*, which is an extension of Mealy automata with UML-type communication mechanisms. It includes the following kinds of diagrams:

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called *object diagrams*.

Statechart diagrams (or *state diagrams*) give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions.

Sequence diagrams describe interaction between objects or system components via message exchange.

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statecharts and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of *packages*) integrate the information between the different kinds of diagrams and between different parts of the system specification.

There is another kind of diagrams, the use case diagrams, which describe typical interactions between a user and a computer system. They are often used in an informal way for negotiation with a customer before a system is designed. We will not use it in the following. Additionally to sequence diagrams, there are *collaboration diagrams*, which present similar information. Also, there are *component diagrams*, presenting part of the information contained in deployment diagrams.

The used fragment of UML is simplified to keep automated formal verification that is necessary for some of the more subtle security requirements feasible. Note that in our approach we identify system objects with UML objects, which is suitable for our purposes. Also, as with practically all analysis methods, also in the real-time setting [30], we are mainly concerned with instance-based models. Although, simplified, our choice of a subset of UML is reasonable for our needs, as we have demonstrated in several industrial case-studies (some of which are documented in [13]).

The formal semantics for subsystems incorporates the formal semantics of the diagrams contained in a subsystem. It

– models actions and internal activities explicitly (rather than treating them as atomic given events), in particular the operations and the parameters employed in them,

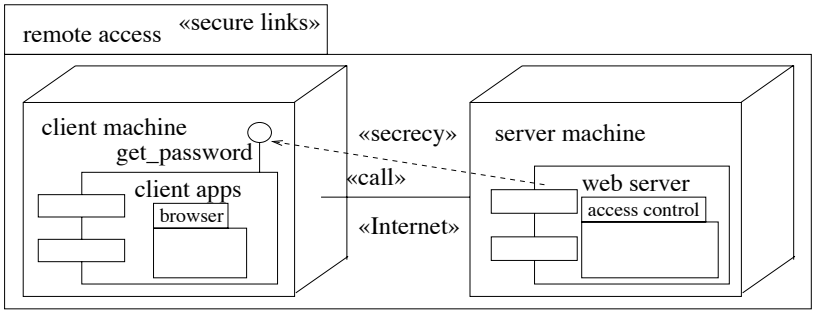

Fig. 3. Example *secure links* usage

- provides passing of messages with their parameters between objects or components specified in different diagrams, including a dispatching mechanism for events and the handling of actions, and thus
- allows in principle whole specification documents to be based on a formal foundation.

In particular, we can compose subsystems by including them into other subsystems.

For example, consider the following UMLsec Stereotype:

secure links This stereotype, which may label (instances of) subsystems, is used to ensure that security requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency d with stereotype $s \in \{\ll\texttt{secrecy}\gg, \ll\texttt{integrity}\gg, \ll\texttt{high}\gg\}$ between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype t such that

- in the case of $s = \ll\texttt{high}\gg$, we have $\mathsf{Threats}_A(t) = \emptyset$,
- in the case of $s = \ll\texttt{secrecy}\gg$, we have $\mathsf{read} \notin \mathsf{Threats}_A(t)$, and
- in the case of $s = \ll\texttt{integrity}\gg$, we have $\mathsf{insert} \notin \mathsf{Threats}_A(t)$.

Example In Fig. 3, given the *default* adversary type, the constraint for the stereotype $\ll\texttt{secure links}\gg$ is violated: The model does not provide communication secrecy against the *default* adversary, because the Internet communication link between web-server and client does not give the needed security level according to the $\mathsf{Threats}_{default}(Internet)$ scenario. Intuitively, the reason is that Internet connections do not provide secrecy against default adversaries. Technically, the constraint is violated, because the dependency carries the stereotype $\ll\texttt{secrecy}\gg$, but for the stereotype $\ll\texttt{Internet}\gg$ of corresponding link we have $\mathsf{read} \in \mathsf{Threats}_{default}(Internet)$.

Code Security Assurance [15, 16] Even if specifications exist for the implemented system, and even if these are formally analyzed, there is usually no guarantee that the implementation actually conforms to the specification. To deal with this problem, we use the following approach: After specifying the system in UMLsec and verifying the model against the given security goals as explained above, we make sure that the implementation correctly implements the specification with techniques explained below. In particular, this approach is applicable to legacy systems. In ongoing work, we are automating this approach to free one of the need to manually construct the UMLsec model.

Run-time Security Monitoring using Assertions A simple and effective alternative is to insert security checks generated from the UMLsec specification that remain in the code while in use, for example using the assertion statement that is part of the Java language. These assertions then throw security exceptions when violated at run-time. In a similar way, this can also be done for C code.

Model-based Test Generation For performance-intensive applications, it may be preferable not to leave the assertions active in the code. This can be done by making sure by extensive testing that the assertions are always satisfied. We can generate the test sequences automatically from the UMLsec specifications. More generally, this way we can ensure that the code actually conforms to the UMLsec specification. Since complete test coverage is often infeasible, our approach automatically selects those test cases that are particularly sensitive to the specified security requirements [19].

Automated Code Verification against Interface Specifications For highly nondeterministic systems such as those using cryptography, testing can only provide assurance up to a certain degree. For higher levels of trustworthiness, it may therefore be desirable to establish that the code does enforce the annotations by a formal verification of the source code against the UMLsec interface specifications. We have developed an approach that does this automatically and efficiently by proving locally that the security checks in the specification are actually enforced in the source code.

Automated Code Security Analysis We developed an approach to use automated theorem provers for first-order logic to directly formally verify crypto-based Java implementations based on control flow graphs that are automatically generated (and without first manually constructing an interface specification). It supports an abstract and modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. Currently, this approach works especially well with nicely structured code (such as created using the MBSE development process).

Secure Software-Hardware Interfaces We have tailored the code security analysis approach to software close to the hardware level. More concretely, we considered

the industrial Cryptographic Token Interface Standard PKCS 11 which defines how software on untrustworthy hardware can make use of tamper-proof hardware such as smart-cards to perform cryptographic operations on sensitive data. We developed an approach for automated security analysis with first-order logic theorem provers of crypto protocol implementations making use of this standard.

Analyzing Security Configurations We have also performed research on linking the UMLsec approach with the automated analysis of security-critical configuration data. For example, our tools automatically checks SAP R/3 user permissions for security policy rules formulated as UML specifications [13]. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software, such as firewalls or other access control configurations.

Industrial Applications of MBSE include:

Biometric Authentication For a project with an industrial partner, MBSE was chosen to support the development of a biometric authentication system at the specification level, where three significant security flaws were found [14]. We also applied it to the source-code level for a prototypical implementation constructed from the specification [12].

Common Electronic Purse Specifications MBSE was applied to a security analysis of the Common Electronic Purse Specifications (CEPS), a candidate for a globally interoperable electronic purse standard supported by organizations representing 90 % of the world's electronic purse cards (including Visa International). We found three significant security weaknesses in the purchase and load transaction protocols [13], proposed improvements to the specifications and showed that these are secure [13]. We also performed a security analysis of a prototypical Java Card implementation of CEPS.

Web-based Banking Application In a project with a German bank, MBSE was applied to a web-based banking application to be used by customers to fill out and sign digital order forms [6]. The personal data in the forms must be kept confidential, and orders securely authenticated. The system uses a proprietary client authentication protocol layered over an SSL connection supposed to provide confidentiality and server authentication. Using the MBSE approach, the system architecture and the protocol were specified and verified with regard to the relevant security requirements.

In other applications [13], MBSE was used ...

- to uncover a flaw in a variant of the Internet protocol TLS proposed at IEEE Infocom 1999, and suggest and verify a correction of the protocol.
- to perform a security verification of the Java implementation Jessie of SSL.
- to correctly employ advanced Java 2 or CORBA security concepts in a way that allows an automated security analysis of the resulting systems.

- for an analysis of the security policies of a German mobile phone operator [17].
- for a security analysis of the specifications for the German Electronic Health Card in development by the German Ministry of Health.
- for the security analysis of an electronic purse system developed for the Oktoberfest in Munich.
- for a security analysis of an electronic signature pad based contract signing architecture under consideration by a German insurance company.
- in a project with a German car manufacturer for the security analysis of an intranet-based web information system.
- with a German chip manufacturer and a German reinsurance company for security risk assessment, also regarding Return on Security Investment.
- in applications specifically targeted to service-based, health telematics, and automotive systems.

Recently, there has been some work analyzing trade-offs between security- and performance-requirements [24, 31].

3 Modelling evolution with UMLseCh

This section introduces extensions of the UMLsec profile for supporting system evolution in the context of model-based secure software development with UML.

This profile, UMLseCh, is a further extension of the UML profile UMLsec in order to support system evolution in the context of model-based secure software development with UML. It is a "light-weight" extension of the UML in the sense that it is defined based on the UML notation using the extension mechanisms stereotypes, tags, and constraints, that are provided by the UML standard. For the purposes of this section, by "UML" we mean the core of the UML 2.0 which was conservatively included from UML 1.5[7].

As such, one can define the meta-model for UMLsec and also for UMLseCh by referring to the meta-model for UML and by defining the relevant list of stereotypes and associated tags and constraints. The meta-model of the UMLsec notation was defined in this way in [13]. In this section, we define the meta-model of UMLseCh in an analogous way.

The UMLseCh notation is divided in two parts: one part intended to be used during abstract design, which tends to be more informal and less complete in its use and is thus particularly suitable for abstract documentation and discussion with customers (cf. Sect. 3.1), and one part intended to be used during detailed design, which is assumed to be more detailed and also more formal, such that it will lend itself towards automated security analysis (cf. Sect. 3.2). We discuss about possible verification strategies based on the concrete notation in Sect. 3.3.

[7] http://www.omg.org/spec/UML/1.5

3.1 Abstract Notation

We use stereotypes to model change in the UML design models. These extend the existing UMLsec stereotypes and are specific for system evolution (change). We define change stereotypes on two abstraction layers: (i) abstract stereotypes and (ii) Concrete stereotypes. This subsection given an overview of the abstract stereotypes.

The aim of the abstract change stereotypes is to document change artefacts directly on the design models to enable controlled change actions. The abstract change stereotypes are tailored for modelling a living security system, i.e., through all phases of a system's life-cycle.

We distinguish between past, current and future change. The abstract stereotypes makes up three refinement levels, where the upper level is ≪`change`≫. This stereotype can be attached to subsystems and is used across all UML diagrams. The meaning of the stereotype is that the annotated modelling element and all its sub-elements has or is ready to undergo change.

≪`change`≫ is refined into the three change schedule stereotypes:

1. ≪`past_change`≫ representing changes already made to the system (typically between two system versions).
2. ≪`current_change`≫ representing changes currently being made to a system.
3. ≪`future_change`≫ specifying the future allowed changes.

To track and ensure controlled change actions one needs to be explicit about which model elements are allowed to change and what kind of change is permitted on a particular model element. For example, it should not be allowed to introduce audit on data elements that are private or otherwise sensitive, which is annotated using the UMLsec stereotype ≪`secrecy`≫. To avoid such conflict, security analysis must be undertaken by refining the abstract notation into the concrete one.

Past and current changes are categories into addition of new elements, modification of existing elements and deletion of elements. The following stereotypes have been defined to cover these three types of change: ≪`new`≫, ≪`modified`≫ and ≪`deleted`≫.

For future change we also include the same three categories of change and the following three future change stereotypes have been defined: ≪`allowed_add`≫; ≪`allowed_modify`≫; ≪`allowed_delete`≫. These stereotypes can be attached to any model element in a subsystem. The future change stereotypes are used to specify future allowed changes for a particular model element.

Past and current change The ≪`new`≫ stereotype is attached to a new system part that is added to the system as a result of a functionality-driven or a security-driven change. For security-driven changes, we use the UMLsec stereotypes secrecy, integrity and authenticity to specify the cause of security-driven change; e.g. that a component has been added to ensure the secrecy of information being transmitted. This piece of information allows us to keep track of the

reasons behind a change. Such information is of particular importance for security analysis; e.g. to determine whether or which parts of a system (according to the associated dependencies tag) that must be analysed or added to the target of evaluation (ToE) in case of a security assurance evaluation.

Tagged values are used to assist in security analysis and holds information relevant for the associated stereotype. The tagged value: {`version=version_number`} is attached to the ≪`new`≫ stereotype to specify and trace the number of changes that has been made to the new system part. When a 'new' system part is first added to the system, the version number is set to 0. This means that if a system part has the ≪`new`≫ stereotype attached to it where the version number is > 0, the system part has been augmented with additional parts since being added to the system (e.g., addition of an new attribute to a new class). For all other changes, the ≪`modified`≫ stereotype shall be used.

The tagged value: {`dependencies=yes/no`} is used to document whether there is a dependency between other system parts and the new/modified system part. At this point in the work, we envision changes to this tag, maybe even a new stereotype to keep track of exactly which system parts that depends on each other. However, there is a need to gain more experience and to run through more examples to make a decision on this issue, as new stereotypes should only be added if necessary for the security analysis or for the security assurance evaluation. Note that the term dependencies are adopted from ISO 14508 Part 2 (Common Criteria) [5].

The ≪`modified`≫ change stereotype is attached to an already existing system part that has been modified as a result of a functional-driven or a security-driven change/change request. The tagged values is the same as for the 'new' stereotype.

The ≪`deleted`≫ change stereotype is attached to an existing system part (subsystem, package, node, class, components, etc.) for which one or more parts (component, attributes, service and similar) have been removed as a result of a functionality-driven change. This stereotype differs from the 'new' and 'modified' stereotypes in that it is only used in cases where it is essential to document the deletion. Examples of such cases are when a security component is removed as a result of a functionality-driven change, as this often affects the overall security level of a system. Information about deleted model elements are used as input to security analysis and security assurance evaluation.

Future change The allowed future change for a modelling element or system part (subsystem) is adding a new element, modifying an existing element and deleting elements (≪`allowed_add`≫, ≪`allowed_modify`≫ and ≪`allowed_delete`≫). We reuse the tagged values from the past and current change stereotypes, except for 'version_number' which is not used for future changes.

3.2 Concrete Notation

We further extend UMLsec by adding so called "concrete" stereotypes: these stereotypes allow to precisely define substitutive (sub) model elements and are

Stereotype	Base Class	Tags	Constraints	Description
substitute	all	ref, substitute, pattern	FOL formula	substitute a model element
add	all	ref, add, pattern	FOL formula	add a model element
delete	all	ref, pattern	FOL formula	delete a model element
substitute-all	subsystem	ref, substitute, pattern	FOL formula	substitute a group of elements
add-all	subsystem	ref, add, pattern	FOL formula	add a group of elements
delete-all	subsystem	ref, pattern	FOL formula	delete a group of elements

Fig. 4. UMLsecCh concrete design stereotypes

equipped with constraints that help ensuring their correct application. These differ from the abstract stereotypes basically because we define a precise semantics (similar to the one of a transformation language) that is intended to be the basis for a security-preservation analysis based on the model difference between versions.

Figure 4 shows the stereotypes defining table. The tags table is shown in Figure 5.

Tag	Stereotype	Type	Multip.	Description
ref	substitute, add, delete, substitute-all, add-all, delete-all	object name	1	Informal type of change
substitute	substitute, substitute-all	list of model elements	1	Substitutives elements
add	add, add-all	list of model elements	1	New elements
pattern	substitute, add, delete, substitute-all, add-all, delete-all	list of model elements	1	Elements to be modified

Fig. 5. UMLsecCh concrete design tags

Description of the notation We now describe informally the intended semantics of each stereotype.

substitute The stereotype substitute attached to a model element denotes the possibility for that model element to be substituted by a model element of the same type over the time. It has three associated tags, namely {`ref`}, {`substitute`} and {`pattern`}.

These tags are of the form

$$\{\,\mathsf{ref} = \mathsf{CHANGE\text{-}REFERENCE}\,\},$$
$$\{\,\mathsf{substitute} = \mathsf{MODEL\text{-}ELEMENT}\,\}$$
$$\text{and } \{\,\mathsf{pattern} = \mathsf{CONDITION}\,\}.$$

The tag {`ref`} takes a string as value, which is simply used as a reference of the change. The value of this tag can also be considered as a predicate and take a truth value to evaluate conditions on the changes, as we explain further in this section. The tag {`substitute`} has a list of model element as value, which represents the several different new model elements that can substitute the actual one if a change occurs. An element of the list contained in the tagged value is a model element itself (e.g. a stereotype, {`substitute = ≪stereotype≫`}). To textually describe UML model elements one can use an abstract syntax as in [13] or any equivalent grammar. Ideally, tool support should help the user into choosing from a graphical menu which model elements to use, without the user to learn the model-describing grammar. The last tag, {`pattern`}, is optional. If the model element to change is clearly identified by the syntactic notation, i.e. if there is no possible ambiguity to state which model element is concerned by the stereotype ≪`substitute`≫, the tag pattern can be omitted. On the other hand, if the model element concerned by the stereotype ≪`substitute`≫ is not clearly identifiable (as it will be the case for simultaneous changes where we can not attach the evolution stereotype to all targeted elements at once), the tag pattern must be used. This tag has a model element as value, which represents the model element to substitute if a change occurs. In general the value of pattern can be a function uniquely identifying one or more model elements within a diagram.

Therefore, to specify that we want to change, for example, a link stereotyped ≪`Internet`≫ with a link stereotyped ≪`encrypted`≫, using the UMLseCh notation, we attach:

$$\ll\texttt{substitute}\gg$$
$$\{\,\mathsf{ref} = \mathsf{encrypt\text{-}link}\,\}$$
$$\{\,\mathsf{substitute} = \mathsf{encrypted}\,\}$$
$$\{\,\mathsf{pattern} = \mathsf{Internet}\,\}$$

to the link concerned by the change.

The stereotype ≪`substitute`≫ also has a constraint formulated in first order logic. This constraint is of the form [CONDITION]. As mentioned earlier, the value of the tag {`ref`} of a stereotype ≪`substitute`≫ can be used as the atomic predicate for the constraint of another stereotype ≪`substitute`≫. The truth value of that atomic predicate is true if the change represented by the stereotype ≪`substitute`≫ for which the tag {`ref`} is associated occurred, false otherwise. The truth value of the condition of a stereotype ≪`substitute`≫ then

represents whether or not the change is allowed to happen (i.e. if the condition is evaluated to true, the change is allowed, otherwise the change is not allowed).

To illustrate the use of the constraint, let us refine the previous example. Assume that to allow the change with reference { ref = encrypt-link }, another change, simply named "change" for example, has to occur. We then attach the following to the link concerned by the change:

≪substitute≫
{ ref = encrypt-link }
{ substitute = encrypted }
{ pattern = Internet }
[change]

add The stereotype ≪add≫ is similar to the stereotype ≪substitute≫ but, as its name indicates, denotes the addition of a new model element. It has three associated tags, namely {ref}, {add} and {pattern}. The tag {ref} has the same meaning as in the case of the stereotype ≪substitute≫, as well as the tag {add} (i.e. a list of model elements that we wish to add). The tag{pattern} has a slightly different meaning in this case. While with stereotype ≪substitute≫, the tag {pattern} represents the model element to substitute, within the stereotype ≪add≫ it does not represent the model element to add, but the parent model element to which the new (sub)-model element is to be added.

The stereotype ≪add≫ is a syntactic sugar of the stereotype ≪substitute≫, as a stereotype ≪add≫ could always be represented with a substitute stereotype (substituting the parent element with a modified one). For example, in the case of Class Diagrams, if s is the set of methods and m a new method to be added, the new set of methods is:

$$s' = s \cup \{m\}$$

The stereotype ≪add≫ also has a constraint formulated in first order logic, which represents the same information as for the stereotype ≪substitute≫.

delete The stereotype ≪delete≫ is similar to the stereotype ≪substitute≫ but, obviously, denotes the deletion of a model element. It has two associated tags, namely {ref} and {pattern}, which have the same meaning as in the case of the stereotype ≪substitute≫, i.e. a reference name and the model element to delete respectively.

The stereotype ≪delete≫ is a syntactic sugar of the substitute stereotype, as a stereotype ≪delete≫ could always be represented with a substitution. For example, if s is the set of methods and m a method to delete, the new set of methods is:

$$s' = s \setminus m$$

As with the previous stereotypes, the stereotype ≪delete≫ also has a constraint formulated in first order logic.

substitute-all The stereotype ≪substitute-all≫ is an extension of the stereotype ≪substitute≫ that can be associated to a (sub)model element or to a whole subsystem. It denotes the possibility for **a set of (sub)model elements** to evolve over the time and what are the possible changes. The elements of the set are sub elements of the element to which this stereotype is attached (i.e. a set of methods of a class, a set of links of a Deployment diagram, etc). As the stereotype ≪substitute≫, it has the three associated tags {ref}, {substitute} and {pattern}, of the form { ref = CHANGE-REFERENCE }, { substitute = MODEL-ELEMENT } and { pattern = CONDITION }. The tags {ref} and {substitute} have the exact same meaning as in the case of the stereotype ≪substitute≫. The tag {pattern}, here, does not represent one (sub)model element but **a set of (sub)model elements** to substitute if a change occur. Again, in order to identify the list model elements precisely, we can use, if necessary, the abstract syntax of UMLsec, defined in [13].

If we want, for example, to replace all the links stereotyped ≪Internet≫ of a subsystem by links stereotyped ≪encrypted≫, we can then attach the following to the subsystem:

≪substitute-all≫
{ ref = encrypt-all-links }
{ substitute = ≪encrypted≫ }
{ pattern = ≪Internet≫ }

The tags {substitute} and {pattern} here allow a parametrisation of the tagged values MODEL-ELEMENT and CONDITION in order to keep information of the different model elements of the subsystem concerned by the substitution. For this, we allow the use of variables in the tagged value of both, the tag {substitute} and the tag {pattern}.

To illustrate the use of the parametrisation in this simultaneous substitution, consider the following example. Assume that we would like to substitute all the secrecy tags in the stereotype ≪critical≫ by the integrity tag, we can attach:

≪substitute-all≫
{ ref = secrecy-to-integrity }
{ substitute = { integrity = X } }
{ pattern = { secrecy = X } }

to the model element to which the stereotype ≪critical≫ is attached.

The stereotype ≪substitute-all≫ also has a constraint formulated in first order logic, which represents the same information as for the stereotype ≪substitute≫.

add-all The stereotype ≪`add`≫ also has its extension ≪`add-all`≫, which follows the same semantics as ≪`substitue-all`≫ but in the context of an addition.

delete-all The stereotype ≪`delete`≫ also has its extension ≪`delete-all`≫, which follows the same semantics as ≪`substitue-all`≫ but in the context of a deletion.

Example Figure 6 shows the use of ≪`add-all`≫ and ≪`substitute-all`≫ on a package containing a class diagram and a deployment diagram depicting the communication between two parties through a common proxy server. The change reflects the design choice to, in addition to protect the integrity of the message `d`, enforce the secrecy of this value as well.

Complex changes In case of complex changes, that arise for example if we want to merge two diagrams having elements in common, we can overload the aforementioned stereotypes for accepting not only lists of elements but even lists of lists of elements. This is technically not very different from what we have described so far, since the complex evolutions can be seen as syntactic sugar for multiple coordinated single-model evolutions.

3.3 Security preservation under evolution

With the use of the UMLseCh concrete stereotypes, evolving a model means that we either *add*, *delete*, or / and *substitute* elements of this model explicitly. In other words, the stereotypes induce sets **Add**, **Del**, and **Subs**, containing the model elements to be added, deleted and substituted respectively, together with information about *where* to perform these changes.

Given a diagram M and a set Δ of such modifications we denote $M[\Delta]$ the diagram resulting after the modifications have taken place. So in general let P be a diagram property. We express the fact that M enforces P by $P(M)$. *Soundness* of the security preserving rules R for a property P on diagram M can be formalized as follows:

$$P(M) \wedge R(M, \Delta) \Rightarrow P(M[\Delta]).$$

So to reason about security preservation, one has several alternatives, depending on the property P. For some static analysis, it suffices to show that simultaneous sub-changes contained in Δ preserve P. Then, incrementally, we can proceed until reaching $P(M[\Delta])$. This can be done by reasoning inductively inductively by cases given a security requirement on UML models, by considering incremental atomic changes and distinguishing them according to a) their *evolution* type (addition, deletion, substitution) and b) their *UML diagram* type.

For dealing with behavioural properties one could exploit the divide and conquer approach by means of compositionality verification results. This idea, originally described in general in [4] (and as mentioned before, used for safety

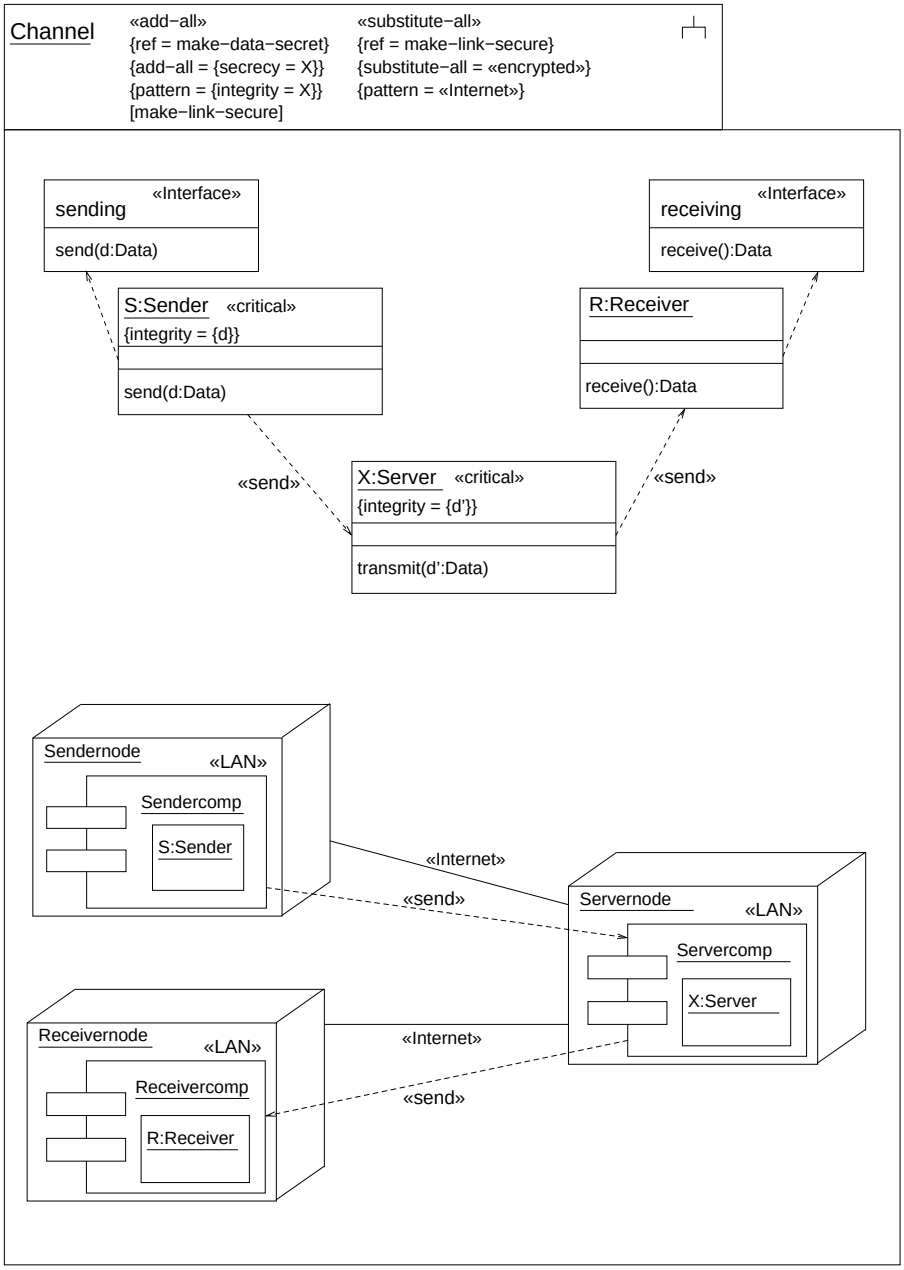

Fig. 6. A UMLseCh annotated diagram with simultaneous substitutions and additions

properties in [3]), is as follows: given components C and D, we denote with $C \otimes D$ its composition. Then, if we now that a security property P holds on both components separately and some set of rules R are satisfied then P holds in the composition, we can use this to achieve a more efficient verification under evolution, given R is easy enough to check. In practice, one often has to modify just one component in a system, and thus if one has:

$$P(C) \wedge P(D) \wedge R(C, D) \Rightarrow P(C \otimes D)$$

one can then check, for a modification Δ on one of the components:

$$P(C[\Delta]) \wedge P(D) \wedge R(C[\Delta], D) \Rightarrow P(C[\Delta] \otimes D) = P(C \otimes D)[\Delta]$$

and thus benefit from the already covered case $P(D)$ and the efficiency of R. Depending on the completeness of R, this procedure can also be relevant for an evolution of both components, since one could reapply the same decision procedure for a change in D (and therefore can be generalized to more than two components). The benefit consists in splitting the problem of verifying the composition (which is a problem with a bigger input) in two smaller subproblems. Some security properties (remarkably information flow properties like non-interference) are not safety properties, and there are interesting results for their compositionality (for example [23]).

4 Application Examples and Tool Support

4.1 Modelling change of UMLsec Diagrams

Secure Dependency This stereotype, used to label subsystems containing object diagrams or static structure diagrams, ensures that the ≪`call`≫ and ≪`send`≫ dependencies between objects or subsystems respect the security requirements on the data that may be communicated along them, as given by the tags {`secrecy`}, {`integrity`}, and {`high`} of the stereotype ≪`critical`≫. More exactly, the constraint enforced by this stereotype is that if there is a ≪`call`≫ or ≪`send`≫ dependency from an object (or subsystem) C to an interface I of an object (or subsystem) D then the following conditions are fulfilled.

- For any message name n in I, n appears in the tag {`secrecy`} (resp. {`integrity`}, {`high`}) in C if and only if it does so in D.
- If a message name in I appears in the tag {`secrecy`} (resp.{`integrity`}, {`high`}) in C then the dependency is stereotyped ≪`secrecy`≫ (resp. ≪`integrity`≫ , ≪`high`≫).

If the dependency goes directly to another object (or subsystem) without involving an interface, the same requirement applies to the trivial interface containing all messages of the server object.

This property is specially interesting to verify under evolution since it is local enough to re-use effectively previous verifications on the unmodified parts and its syntactic nature makes the incremental decision procedure relatively straightforward.

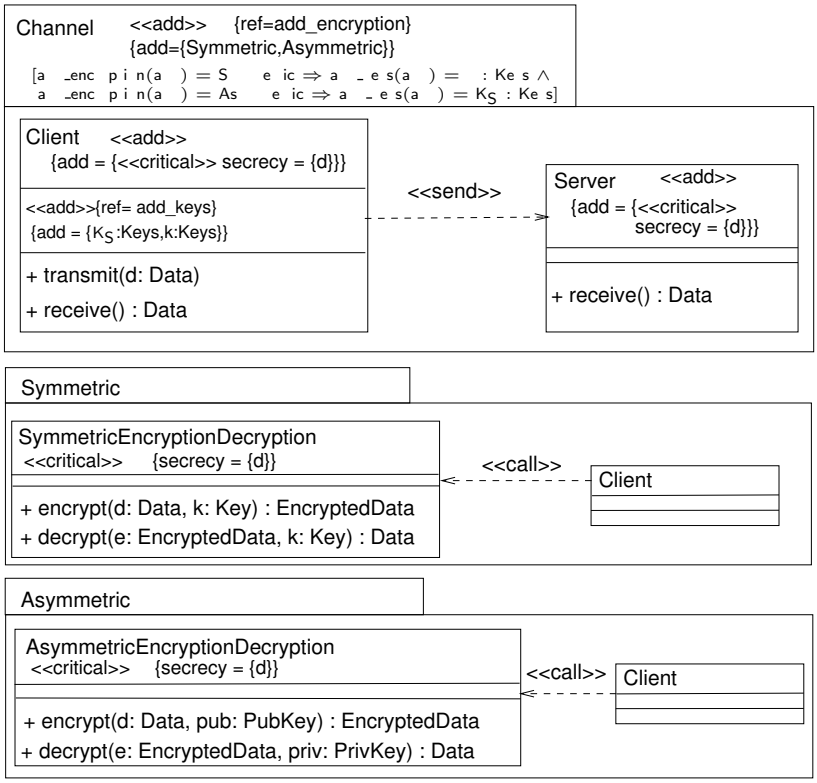

Fig. 7. An evolving class diagram with two possible evolution paths

Example The example in Fig. 7 shows the Client side of a communication channel between two parties. At first (disregarding the evolution stereotypes) the communication is unsecured. In the packages Symmetric and Asymmetric, we have classes providing cryptographic mechanisms to the Client class. Here the stereotype ≪add≫ marked with the reference tag {ref} with value add_encryption specifies two possible evolution paths: merging the classes contained in the current package (Channel) with either Symmetric or Asymmetric. There exists also a stereotype ≪add≫ associated with the Client class adding either a pre-shared private key k or a public key K_S of the server. To coordinate the intended evolution paths for these two stereotypes, we can use the following first-order logic constraint (associated with add_encryption):

$$[\mathsf{add\_encryption}(\mathsf{add}) = \mathsf{Symmetric} \Rightarrow \mathsf{add\_keys}(\mathsf{add}) = \mathsf{k} : \mathsf{Keys} \ \wedge$$
$$\mathsf{add\_encryption}(\mathsf{add}) = \mathsf{Asymmetric} \Rightarrow \mathsf{add\_keys}(\mathsf{add}) = K_S : \mathsf{Keys}]$$

The two deltas, representing two possible evolution paths induced by this notation, can be checked incrementally by case distinction. In this case, the evo-

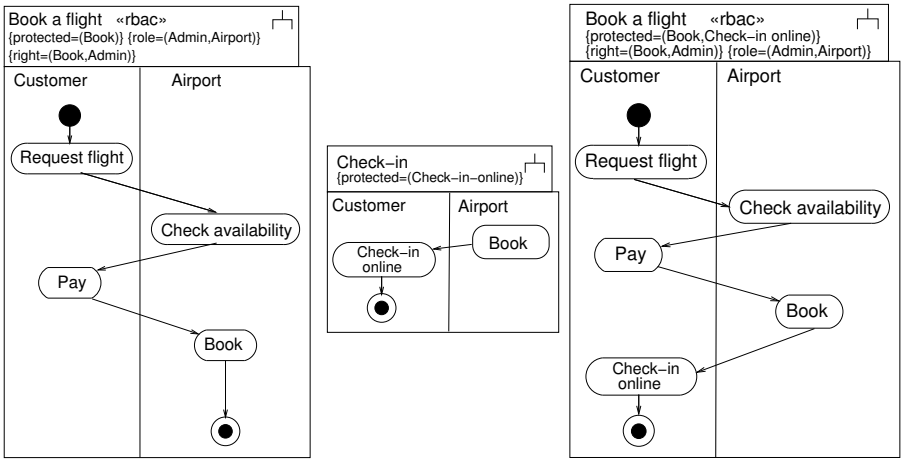

Fig. 8. Activity Diagram Annotated with ≪rbac≫ Before Evolution (left-hand side), Added Model Elements (middle), and After Evolution (right-hand side)

lution is security preserving in both cases. For more details about the verification technique see [27].

Role-based Access Control The stereotype ≪rbac≫ defines the access rights of actors to activities within an activity diagram under a role schema. For this purpose there exists tags {protected}, {role}, {right}. An activity diagram is UMLsec satisfies ≪rbac≫ if for every protected activity A in {protected}, for which an user U has access to it, there exists a pair (A,R) in {rights} and a pair (R,U) in {roles}. The verification computational cost depends therefore on the number of protected activities.

Example In Fig. 8 (left-hand side), we show an activity diagram to which we want to add a new set of activities, introducing a web-check-in functionality to a flight booking system. The new activity "Check-in online" (middle of Fig. 8) is protected, but we do not add a proper role/right association to this activity, thus resulting in a security violating diagram (right-hand side Fig. 8).

4.2 Tool Support

The UMLsec Tool Suite provides mechanical tool support for analyzing UML specifications for security requirements using model-checkers and automated theorem provers for first-order logic . The tool support is based on an XML dialect called XMI which allows interchange of UML models. For this, the developer creates a model using a UML drawing tool capable of XMI export and stores it as an XMI file. The file is imported by the UMLsec analysis tool (for example, through its web interface) which analyses the UMLsec model with respect to the security requirements that are included. The results of the analysis are

given back to the developer, together with a modified UML model, where the weaknesses that were found are highlighted.

We also have a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes. In particular, the framework includes the UMLsec tool web interface, so that new routines are also accessible over this interface. The idea behind the framework is to provide a common programming framework for the developers of different verification modules. A tool developer should be able to concentrate on the implementation of the verification logic and not be required to implement the user interface.

As mentioned in Sect. 3, tool support for UMLseCh would be beneficial in at least two ways:

- Supporting the user in modelling expected evolutions explicitly in a graphical way, without using a particular grammar or textual abstract syntax, and supporting the specification of non-elementary changes.
- Supporting the decision of including a change based on verification techniques for model consistency preservation after change.

but more importantly:

- Supporting the decision of including a change based on verification techniques for security preservation after change

First steps in this direction have been done in the context of the EU project SecureChange for statical security properties. For more details refer to [27].

5 Conclusions and Future Work

For system evolution to be reliable, it must be carried out in a controlled manner. Such control must include both functional and quality perspectives, such as security, of a system. Control can only be achieved under structured and formal identification and analysis of change implications up front, i.e. a priori. In this chapter, we presented a step-by-step controlled change process with emphasis on preservation of security properties through and throughout change, where the first is characterized as a security-driven change and the second a functionality-driven change. As there are many reasons for evolution to come about, security may drive the change process or be directly or indirectly influenced by it. Our approach covers both. In particular, the chapter introduces the change notation UMLseCh that allows for formally expressing, tracing, and analysing for security property preservation. UMLseCh can be viewed as an extension of UMLsec in the context of secure systems evolution. We showed how can one use the notation to model change for different UMLsec diagrams, and how this approach could be useful for tool-aided security verification. Consequently, this work can be extended in different directions. First of all, compositional and incremental

techniques to reason about the security properties covered by UMLsec are necessary to take advantage of the precise model difference specification offered by UMLseCh. On the other hand, comprehensive tool support for both modelling and verification is key for a successful application of UMLseCh in practical contexts.

Acknowledgements

This research was partially supported by the EU project "Security Engineering for Lifelong Evolvable Systems" (*Secure Change*, ICT-FET-231101).

References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.
2. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 440–453. Springer, 2006.
3. S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software, 2004.
4. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Annual Symposium on Logic in Computer Science (LICS)*, pages 353–362, June 1989.
5. ISO 15408:2007 Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 2: Part 2; Security Functional Components, CCMB-2007-09-002, September 2007.
6. J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and verification of layered security protocols: A bank application. In S. Anderson, M. Felici, and B. Littlewood, editors, *SAFECOMP*, volume 2788 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 2003.
7. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS) - Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.
8. S. Höhn and J. Jürjens. Rubacon: automated support for model-based compliance engineering. In Robby [26], pages 875–878.
9. J. Jürjens. Formal Semantics for Interacting UML subsystems. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 29–44. International Federation for Information Processing (IFIP), Kluwer Academic Publishers, 2002.
10. J. Jürjens. *Principles for Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, 2002.
11. J. Jürjens. Model-based security engineering with UML. In A. Aldini, R. Gorrieri, and F. Martinelli, editors, *FOSAD*, volume 3655 of *Lecture Notes in Computer Science*, pages 42–77. Springer, 2004.

12. J. Jürjens. Code security analysis of a biometric authentication system using automated theorem provers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 138–149. IEEE Computer Society, 2005.
13. J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.
14. J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 322–331. ACM Press, 2005.
15. J. Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *MEMOCODE*, pages 89–98. IEEE, 2005.
16. J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–176. IEEE Computer Society, 2006.
17. J. Jürjens, J. Schreck, and P. Bartmann. Model-based security analysis for mobile communications. In Robby [26], pages 683–692.
18. J. Jürjens and P. Shabalin. Tools for secure systems development with UML. *Intern. Journal on Software Tools for Technology Transfer*, 9(5–6):527–544, Oct. 2007. Invited submission to the special issue for FASE 2004/05.
19. J. Jürjens and G. Wimmel. Formally testing fail-safety of electronic purse protocols. In *16th International Conference on Automated Software Engineering (ASE 2001)*, pages 408–411. IEEE Computer Society, 2001.
20. D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 6(9):53–69, 2007.
21. M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, 1998.
22. H. Lipson. Evolutionary systems design: Recognizing changes in security and survivability risks. Technical Report CMU/SEI-2006-TN-027, Carnegie Mellon Software Engineering Institute, September 2006.
23. H. Mantel. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 88–101, Oakland, CA, USA, 2002. IEEE Computer Society.
24. D. C. Petriu, C. M. Woodside, D. B. Petriu, J. Xu, T. Israr, G. Georg, R. B. France, J. M. Bieman, S. H. Houmb, and J. Jürjens. Performance analysis of security aspects in UML models. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 91–102. ACM, 2007.
25. A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proceedings of the International Conference in Graph Transformation (ICGT)*, pages 226–241. Springer, 2004.
26. Robby, editor. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008.
27. Secure Change Project. Deliverable 4.2. Available as `http://www-jj.cs.tu-dortmund.de/jj/deliverable_4_2.pdf`.
28. UML Revision Task Force. *OMG Unified Modeling Language: Specification*. Object Management Group (OMG), September 2001. `http://www.omg.org/spec/UML/1.4/PDF/index.htm`.
29. UMLsec group. UMLsec Tool Suite, 2001-2011. `http://www.umlsec.de`.
30. B. Watson. Non-functional analysis for UML models. In *Real-Time and Embedded Distributed Object Computing Workshop*. Object Management Group (OMG), July 15–18, 2002.

31. C. M. Woodside, D. C. Petriu, D. B. Petriu, J. Xu, T. A. Israr, G. Georg, R. B. France, J. M. Bieman, S. H. Houmb, and J. Jürjens. Performance analysis of security aspects by weaving scenarios extracted from UML models. *Journal of Systems and Software*, 82(1):56–74, 2009.

A.10 ARES 2011: Model-based security verification and testing for smart-cards

- Elizabeta Fourneret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jürjens, and Parvaneh Yousefi. Model-based security verification and testing for smart-cards. In *ARES 2011, 6-th Int. Conf. on Availability, Reliability and Security*, Vienna, Austria, August 2011.

Model-Based Security Verification and Testing for Smart-cards

Elizabeta Fourneret*, Martín Ochoa†,
Fabrice Bouquet*, Julien Botella‡, Jan Jürjens†, Parvaneh Yousefi†

* *LIFC, Université de Franche-Comté, Besanon, France*
elizabeta.fourneret, fabrice.bouquet@lifc.univ-fcomte.fr

† *Software Engineering, Department of Computer Science, TU Dortmund, Germany*
martin.ochoa, jan.jurjens, parvaneh.yousefi@cs.tu-dortmund.de

‡*Smartesting, TEMIS Innovation, Besanon, France*
botella@smartesting.com

***Abstract*—Model-Based Testing (MBT) is widely used methodology for generating tests aiming to ensure that the system behaviour conforms to its specification. Recently, it has been successfully applied for testing certain security properties. However, it is important to consider the correctness of test models with respect to the given security property. In this paper we present an approach for smart-card specific security properties that permits to validate the system with MBT from test schemas. We combine this MBT approach with UMLsec security verification technique, by using UMLsec stereotypes to verify the model w.r.t. given security properties and gain more confidence into the model. Then, we define rule to transform the stereotype into test schema, used to generate security tests to be executed on the system. We validate this approach on a fragment of the Global Platform specification and report on available tool support.**

***Keywords*-Verification; Model-Based Testing; Model-Based Testing from schemas; UML/OCL statehchart; smart-cards;**

I. Introduction

Typically, UML models verified against security properties are explicit models of the *system* design, whereas in Model-Based Testing (MBT) we describe the expected behaviour of an application, seen thus as a *blackbox*. With the current state of the art, on one hand it is possible for a system engineer to design a conception model annotated with security properties that can be verified using automated theorem provers and model-checking, for example using the UMLsec approach [1]. On the other hand, the validation engineer designs a UML test model, and writes test scenarios that are used to produce test cases exercising security properties. This situation is depicted in Figure 1. The security properties considered for testing are typically expressed at different abstraction levels with respect to the used properties, because they will be executed on the implementation of the system (the System Under Test or SUT).

However, the test engineer has no formal guarantee that the test model under consideration is trivially violating the security property, that (s)he would like to test. In other words, little attention is paid to the fact that the *expected behaviour* expressed in the model could be contradicting the property under test[1]. Thus, our goal is to generate tests guided by the security properties and the testing model, but we want to start with a correct model in the first place. Moreover, it is desirable that the security property formalized to check the model for correctness can be further used to generate test sequences following the model based testing paradigm.

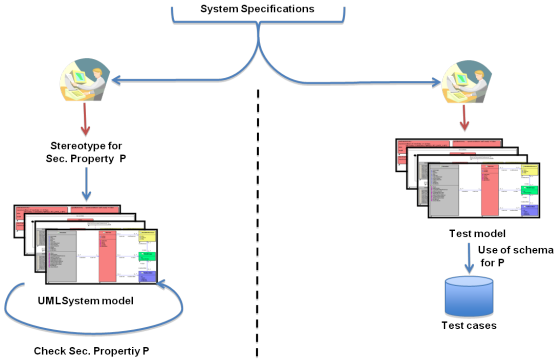

Figure 1. Our traditional approach to security verification and model-based testing

In this paper we consider two generic security properties that are relevant for smart-cards and we show how can one benefit from the UMLsec verification approach for security to ensure the correctness of the UML test model and from the MBT approach in order to generate tests for complex situations issued from the security properties. We also show

[1]This is the case in general, and not only for security properties, in this paper we focus on security.

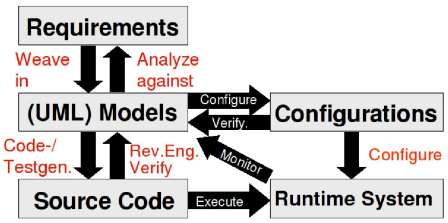

Figure 2. Model-based Security Engineering

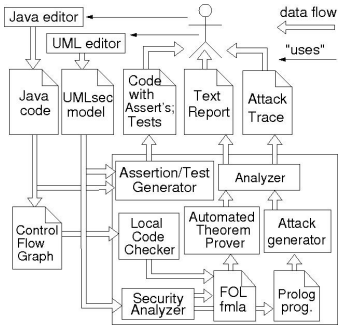

Figure 3. UMLsec Tool support

how the two approaches are linked by means of a testing Schema language, that can be used to automatically generate testing sequences.

To validate this approach, we demonstrate our results on the Global Platform Specification for smart cards [2]. We also report on available tool support for our approach.

The paper is organized as follows: Section II contains a summary of the core concepts of the UMLsec approach while Section III introduces Model-Based Testing for security, that we consider. Section IV explains how to verify consistency of the UML testing model with UMLsec with respect to the chosen security properties. Sect. V contains the validation case study and summarizes the available tool support. Relevant related work is discussed in Section VI.

II. Background: UMLsec

Generally, when using model-based development (Fig. 2), the idea is that one first constructs a model of the system. Then, the implementation is derived from the model: either automatically using code generation, or manually, in which case one can generate test sequences from the model to establish conformance of the code regarding the model. In the model-based security engineering (MBSE) approach based on the UML extension UMLsec, [1], recurring security requirements (such as secrecy, integrity, authenticity, and others) and security assumptions on the system environment, can be specified either within UML specifications, or within the source code (Java or C) as annotations (Fig. 3). This way 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 in form of 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 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. The semantics for the fragment of UML used for UMLsec is defined in [1] using so-called *UML statecharts*. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined.

The UMLsec tool support (cf. Fig.2) can be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [3]. They generate logical formulas formalizing the execution semantics and the annotated security requirements. Automated theorem provers and model checkers automatically establish whether the security requirements hold. If not, we can use Prolog to automatically generate an attack sequence violating the security requirement, which can be examined to determine and remove the weakness. Since the analysis that is performed is too sophisticated to be done manually, it is also valuable to security experts. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus, advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes.

The tags defined in UMLsec represent a set of desired properties. For instance, ”freshness” of a value means that an attacker cannot guess what its value was.

In this paper, we present an extension of UMlsec stereotypes for security relevant properties for smart-cards, which are used to drive the test generation dedicated to security. We use then the UMLsec tool to check the property on the model and extract a logical formula, that are going to be used to create test schemas, detailed in the next section.

III. Test Generation Process for Security Properties

Model-Based Testing makes use of selection criteria that indicates how to *select* the tests to be extracted from the model. These criteria usually ensure a given structural coverage of the model, such as *all the states*, or *all the transitions*, etc.

Each test is a sequence of operation calls with parameter values, which yields a distinguished execution of the model. Their results are predicted by the model. Our approach

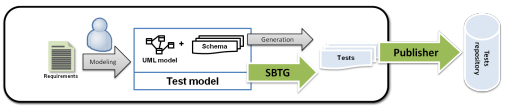

Figure 4. The Model-Based Process for testing Security Properties

(depicted on the Figure 4) for testing security properties relies on defining additional selection criteria in the shape of *test schemas*. We use the Schema Based Test Generator (SBTG) to unfold the schema and then we use the *Smartesting CertifyIt* tool to generate tests dedicated to the security property. Once the tests are generated it is possible to publish the tests into a test repository, used for test management.

A test schema is a high level expression that formalizes the test intention linked to a security property to drive the automated test generation on the behavioral model. In this approach, the security requirements that a system must fulfil are expressed as a set of *security properties*. We propose test schemas as a means to exercise the system for validating that it behaves as predicted by the model w.r.t. these security properties. Based on his know-how, an experienced security engineer will imagine possible scenarios in which he or she thinks the property might be violated by an erroneous implementation, and then on the basis of this test intention, (s)he will formalize test schemas to drive the automated test generation.

In [4] the concepts of such test schemas are defined in the shape of a language. It is based on regular expressions and allows the security testing engineer to conceive its test schemas in terms of states to be reached and operations to be called. It is based on the work done in [5] and its formal semantics has been defined in [6].

Based on this conceptual language, an operational language has been defined within the SecureChange project and implemented as a plug-in to the Smartesting suite, to describe test schemas in a "textual" way. This language is called *Smartesting schema language*. We now present it and provide a couple of illustrative examples in SectionV.

A. Presentation

A dedicated schema language editor has been implemented as a plug-in of Smartesting CertifyIt. Its aim is to provide a means to express security properties at a high level, close to a textual representation or by using usual computer programming paradigms. The expression of these properties allows the test specifications generating, called *Test Case Specification - TCS*, that are high level scenarios from which tests will be generated by CertifyIt.

The language relies on combining keywords, to produce expressions that are both powerful and easy to read by a validation engineer.
In Table I we define the language keywords. For each keyword, we give its intuitive meaning.

for_each	quantifier for an operation or a behaviour
from	to introduce a list of operations or behaviours
then	a separator for sequencing the targets to be reached
use	to introduce an operation a behaviour or a variable to use
to_reach	to introduce a state to be reached
to_activate	to introduce a behaviour to be activated
state_respecting	to introduce a constraint that characterize a set of states
on_instance	to introduce an instance on which a constraint holds
any_operation	the set of all the operations of the model
any_operation_but	the set of all the operations of the model minus the ones whose list follows
or	for a disjunction of operations or of behaviours
any_behaviour_to_cover	the set of all the behaviours of the model
any_behaviour_to_cover_but	the set of all the behaviours of the model minus the ones whose list follows
behaviour_activating	to introduce a list to be covered of behaviours tagged in the model
behaviour_not_activating	to introduce a list whose complementary must be covered of behaviours tagged in the model
at_least_once	repetition operator indicating to apply at least once the operation or behaviour previously specified
any_number_of_times	repetition operator indicating to apply any number of times the operation or behaviour previously specified
$	variable prefix
REQ	to introduce a tag that corresponds to a requirement
AIM	to introduce a tag that corresponds to an aim

Table I
KEYWORDS FOR THE SMARTESTING SCHEMA LANGUAGE

B. Language Syntax

The syntax of the language is defined by means of the grammar given in Figure 5. The language makes it possible to design test schemas as a sequence of quantifiers or blocks, each block being composed of a set of operations (possibly iterated at least once, or many times) and aiming at reaching a given target (a specific state, the activation of a given operation, etc.).

```
SCHEME ::= (QUANTIFIER_LIST ,)? SEQ
QUANTIFIER_LIST ::= QUANTIFIER (, QUANTIFIER)*
QUANTIFIER ::= for_each VAR from ( BEHAVIOR_CHOICE
                                 | OP_CHOICE )
BEHAVIOR_CHOICE ::= any_behaviour_to_cover
                  | any_behavior_to_cover_but
                                 BEHAVIOR_LIST
BEHAVIOR_LIST ::= BEHAVIOR (or BEHAVIOR)*
BEHAVIOR ::= behavior_activating TAG_LIST
           | behavior_not_activating TAG_LIST
TAG_LIST ::= { TAG (, TAG)* }
TAG ::= REQ: tag name | AIM: tag name
OP_CHOICE ::= any_operation | OP_LIST
            | any_operation_but OP_LIST
OP_LIST ::= OPERATION (or OPERATION)*
OPERATION ::= operation name
SEQ ::= BLOC (then BLOC)*
BLOC ::= use CONTROL (RESTRICTION)? (TARGET)?
CONTROL ::= OP_CHOICE | BEHAVIOR_CHOICE | VAR
VAR ::= $variable name
RESTRICTION ::= at_least_once | any_number_of_times
TARGET ::= to_reach STATE
         | to_activate BEHAVIOR
         | to_activate VAR
STATE ::= state_representing ocl constraint
          on_instance instance name
```

Figure 5. Syntax of the Smartesting Schema Language

We find, that there are several benefits from the Smartest-

ing schema language. Firstly, from a scientific point of view, the language that is described previously makes it possible for the validation engineer to express his or her test schemas by combining sequences of actions of the system to be called, along with the description by means of predicates of the states to be reached by these sequences of calls. Secondly, from a technological point of view, the language is designed to be easy to use by a validation engineer. (S)he writes test schemas in a high level language, in a textual manner, with constructions that are close to usual computer programming paradigms. That frees him from manipulating mathematical notations such as in the temporal logics. Thirdly, by its expressivity, the language is designed as a mean for a validation engineer to describe his test intentions w.r.t. a property that has to be tested. This feature strongly helps to monitor the coverage of the properties to be tested.

IV. Integrated approach

In this section, first we describe how can we improve the quality of the test models by using UMLsec. And secondly, how to obtain schemas used for test generation with respect to a given security property from a UMLsec stereotype. The model that is used for test generation has to be verified for consistency with respect to the considered security properties. If not, the model may authorize an incorrect behaviour and the tests that will be produced will expect the System Under Test to present the same behaviour as the model.

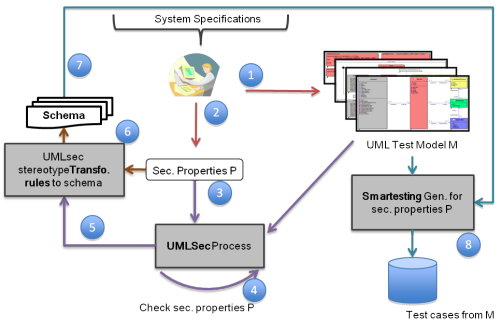

Figure 6. Integration of the two approaches

The process is summarized in Fig. 6. First, a validation engineer designs a test model (Step 1). He then extracts the security properties from the specification (Step 2). Afterwards in Step 3 (s)he writes the corresponding UMLSec stereotypes. (S)He uses the UMLsec approach to validate the model against the security properties (Step 4), to make sure that the model respects them. Once the model is declared correct, a Hoare triple (more details are given below) for each property is exported by Step 5. Then, we use transformation rule to automatize the schema writing with respect to the property (Step 6). The created schema (Step 7), can be used to produce test cases exercising the property (Step 8).

We consider the following two properties that are critical for a card issuer in order to have control over compromised running smart-cards.

Security Property 1: For any execution, whenever the card is set to the state `{status}` *by means of a operation performed by a privileged application, then it should not be possible to revert to another state.*

This property ensures that whenever an application with enough privileges terminates a card, the card cannot be put back in operation. This is an important feature to control smart-cards running malicious applications or that have been compromised in some way.

Security Property 2: It should not be possible for an application that does not have the given privilege `{permission}` *to set the card into a given state* `{status}`.

Conversely, to avoid the Denial of Service (D.O.S) attacks on the card, only applications with sufficient privileges should be able to terminate the card.

Extending UMLsec stereotypes for the given security properties: Assuming the SUT has a variable state representing the card state, we can choose to design the statechart representing the UML test model associating each UML status to a possible state in the card's life-cycle. We further assume there is a command set_status, only executable by privileged applications to change from one state to the other, and this is the event triggering all transitions in the model. To model failed attempts to change status by an unprivileged application, we allow internal transitions in a given state to represent these, for which the consequence is that a variable statusWord is affected with an error message.

Under this assumptions, a statechart in which from the `{status}` status there are not only incoming but also *outcoming* transitions [2] would be trivially violating the Security Property 1. This would contradict the very property we would like to test and could be the source of misinterpretations of the testing results. Potentially, it could also mean that the system specification is contradictory with respect to the wished security properties. To avoid this, we can extend UMLsec with a stereotype «locked-status» together with a tag `{status}` where a specific status can be defined. Semantically, a statechart annotated with this stereotype would require that there are not outgoing transitions from the state specified in `{status}`.

Similarly, we can define a stereotype «authorized» with two tags `{status}` and `{permission}`. This stereotype enforces that there exists no incoming transition to the status specified in `{status}` with a guard NOT containing

[2] With satisfiable guards, otherwise they would be superfluous.

{permission}. Under the assumption that in each transition from state to state we check for given application privileges, this stereotype would avoid having a model trivially violating Security Property 2.

These properties can be checked statically on UML statecharts since we are not aiming at verifying behavioural properties, but at ensuring a structural property as a precondition to the testing process. For example, the check performed by «authorized» on a statechart S could be summarized by the Algorithm 1, where Status is the state corresponding to the value of {status} and the auxiliary function IncomingTransitions() returns all the incoming transitions relatively to that status.

Algorithm 1: Algorithm for stereotype 'authorized'

```
Transitions := Status.IncomingTransitions();
for T in Transitions do
    if Permission ∉ T.Guard then
        return false ;
    end
end
```

In a similar way we can define an algorithm for «locked-status», where we check whether Status.OutgoingTransistions() is empty.

Rules for transformation of UMLsec stereotypes to Schemas: At the end of the model verification process we export the security property using Hoare triples [3] encapsulating the expected behaviour of the system after particular instructions are executed in states that could potentially violate the properties, should the system not behave as expected. These will be the basis for generating Testing Schemas (which will automatically generate test sequences), thus they represent the link from UMLsec to testing.

Assume S is a set of instructions performed by an application in the system and T and Q are FOL formulas quantifying over system variables. Thus, let $\{T\}\ S\ \{Q\}$ be generalized exported formula by the UMLsec tool.

When taking into account the locked-status property, the formula can be exported as:

$$\{\mathsf{state} = \texttt{\{status\}}\}\ \mathsf{set\_status}\ \{\mathsf{state} = \texttt{\{status\}}\}$$

Let A the application in the system that has a set of associated permissions $A.\mathsf{permissions}$. We assume that the set of instructions S does not include an operation that allows to select another application with different privileges. Let

$$P := \mathsf{state} \neq \texttt{\{status\}} \wedge \texttt{\{permission\}} \notin A.\mathsf{permissions}.$$

Then, the authorized-status property can be exported as:

$$\{P\}\ \mathsf{set\_status}\ \{\mathsf{statusWord} = \mathsf{Error\_not\_Privilege\_}\texttt{\{permission\}}\}$$

Intuitively we can define a generic rule to transform the exported formula by UMLsec into a Schema. We define it below: $\{T\}\ S\ \{Q\}$:

for_each $X from S,
use any_operation any_number_of_times
to_reach state_respecting T
on_instance 'chosen_instance' then
use $X at_least_once to_reach state_respecting Q
on_instance 'chosen_instance'

So for example for the locked-status property (Sec. Property 1), the Schema will generate test sequences such that: the state {status} is reached (by any application), and afterwards it will choose the set_status operations that, according to the Testing Model, will still result in preserving the Terminated state (that is, all possible internal transitions, since the model has no transition going *out* of the {status} state). A test will fail if after an execution of a set_status operation the resulting state is different from {status}, since this would contradict the expected behaviour.

We will consider two instantiations of this transformation for the given rule in the case of the Global Platform for smart-cards in the next section, where more details will be given about the test sequences generated from schemas.

V. Validation

We have applied our methodology to a real case study: the Global Platform [2] in the context of the SecureChange project [4]. The Global Platform is a non-profit organization involving over 60 industry members (including American Express, MasterCard, Visa, Nokia, Sun and Gemalto) that defines a publicly available smart card application management specification [5]. The goal of this specification is to be hardware and operating system neutral and to cover a wide range of security critical industrial applications and therefore focuses on many security aspects. For example precise protocols for the communication of the card with an application provider or central server are defined aiming at guaranteeing confidentiality, integrity and authenticity aspects of both over-the-air and terminal connections. Moreover, the Global Platform supports external software updates. Implementations of the specification with tailored applications include Financial, Mobile telecommunications, Government initiatives, Healthcare, Retail merchants and Transit domains.

The scope of our work is the management of the card life cycle, from the card's production until its destruction. We have created test models for the version on the Card Life Cycle Scope of Global Platform 2.1.1 [2] respecting the assumptions mentioned in the previous section: each

[3] A Hoare triple describes how the execution of a piece of commands changes the state of the system

[4] http://www.securechange.eu/

[5] http://www.globalplatform.org/specificationscard.asp

status of the state-chart correspond to a state of the card and each transition's guard from state to state checks for certain application privileges.

A. Obtained results for GP

As explained previously, our test generation process aims at creating security based tests. These tests are computed by animating the model, i.e. by simulating its execution through its formal description in UML/OCL.

In order to build security-based test cases, we rely on the use of dedicated test schemas that describe either nominal test cases, aiming at illustrating the considered property (i.e. the preservation of secrecy, the denial of an access to a specific security asset, etc.), or aiming at checking the robustness of the system towards security. We use them to obtain more confidence into the system using the UMLsec approach and then we use Smartesting CertifyIt tool to generate tests dedicated for security properties testing.

1) Correctness verification with UMLsec: We have verified the GP 2.1.1 life-cycle testing model using the stereotypes « locked-status » and « authorized » using the UMLsec tool [6], which we have extended for these new stereotypes. In this case, the value of `{status}` is TERMINATED (see a fragment of the statechart on Fig.7).

Figure 7. Violating fragment of the GP Life-Cycle

2) Transformation to schemas and Model-based testing for security properties: When generating test cases from security property, we first define informally the *test intention* as a scenario to test this security property. For the locked-status property we give below the test intention:

- set the status of the card to TERMINATED;
- try all operations (to see if they behave as predicted by the model, i.e. by returning a status word of error).

Using the transformation rules, we have obtained the following test schema, that completely reflects the security property w.r.t. to the test intention we have defined. We needed only to define manually the existing model instance, for which we want to generate tests.

```
for_each $X from APDU_Set_status
use any_operation any_number_of_times to_reach
state_respecting (self.state = TERMINATED)
on_instance "card" then
use $X at_least_once to_reach state_respecting
(self.state = TERMINATED) on_instance "card"
```

[6]Available on http://www-jj.cs.tu-dortmund.de/jj/umlsectool/

The test intention for the authorized-status security property that we exhibit, is defined informally as a scenario to test the nominal case of failure of this security property:

- select any application without the Card Terminate Privilege
- set the card to a state different than terminated
- try to set the status of the card to TERMINATED, which results with an error code.

Then, we can create the corresponding *test schema* from the verified stereotype. We give here one possible transformed schema to cover it. However, sometimes is impossible to express one property with only one schema and that there is only one manner to express it.

```
for_each $X from APDU_Set_status,
use any_operation any_number_of_times to_reach
state_respecting (self.lcs->exists(lc : LogicalChannel|
lc.selectedApp.privileges.cardTerminate=false))
on_instance "card" then
use any_operation any_number_of_times to_reach
state_respecting (self.state!=TERMINATED)
on_instance "card" then
use $X at_least_once to_reach state_respecting
(self.StatusWord =
APDU_SETSTATUS_ERROR_MustHaveTerminatePriv)
on_instance "card"
```

3) Discussion: Using the UMLsec approach we have verified our test model and permitted to the user to increase the confidence in it w.r.t the given properties in a realistic industrial scenario. When generating the tests we can be sure that the generated tests are consistent w.r.t. the property. The schema we have created for the *locked-status* property sets the card into the state *TERMINATED*. Then finds different manners to stay in the same state using the *APDU_Set_Status* command. The exit code of this command results with an error code. The error code corresponds to the one that the card is already *terminated*. Thus, we obtain 13 different tests for this property.

For the *authorized-status* property, using the schema we have generated 13 tests, also. Each test selects first an application without *terminated privilege* and then, the schema allows to select states different to *TERMINATED*. Afterwards, the generator tries to reach the *TERMINATE* state. It results with an error code, that the application has not the privilege to terminate the card. Then, these tests are ready to be exported and used for testing the program w.r.t the security property. But, here the created schema does not select each state different to *TERMINATED*. It selects only one among the possible list of states. To include this possibility, we need to define another variable, for example *$Y*, that will iterate the wanted states. Or we can iterate a set

of function behaviors (expressed in the model by using the keyword REQ and AIM) that we are interested in, and create tests that reflect the security property. But currently, with UMLsec we cannot identify the special tags used for testing **REQ** and **AIM**. Our goal is thus, to adapt the exported FOL formulas and add rules that will enable the transformation to benefit fully from the schema language expression power.

VI. Related work

Tests can be obtained by means of a model-checker in the shape of traces of a model that contradict the properties (see [7], [8] for example). M. Dwyer, to facilitate the use of temporal properties by validation engineers, has identified in [9] a set of design patterns that allow for expressing as temporal properties a set of temporal requirements frequently met in industrial studies.

Input/Output Labelled Transition System (IOLTS) and Input/Output Symbolic Transition System (IOSTS) have frequently been used to specify test purposes [10][11]. These formalisms specify sequencing of actions by using the same set of actions as the model, and possess two trap states named *Accept* and *Refuse*. The *Accept* states are used as end states for the test generation while the *Refuse* states allow for cutting the traces not wanted in the generated tests. These formalisms are for example used in tools such as TGV [10], STG [12], TorX [13], Agatha [14].

Some approaches are based on the definition of scenarios for the test, e.g. in [15][16], where test cases are issued from UML diagrams as a set of trees. The scenarios are extracted by a breadth-first search on the trees. A similar approach is that implemented in the tool *Telling TestStories* [17], based on defining a test model from elementary test sequences made of an initial state, a *test story* and test data.

Work close to ours is done for Tobias tool[18], which provides a combinatorial unfolding of some test schemas. The schemas are sequences of patterns made of operation calls and parameter constraints. They are unfolded independently from any model, thus the tests obtained have to be instantiated from a model. In [19], a connection between Tobias and the UCASTING tool is studied to produce instantiated tests. UCASTING [20] allows for valuating sequences of operations that are not or only partially instantiated from an UML model. Cabrera and Botella in [21], close to the previous work, use scenarios based on regular expressions, to enrich the test generation test suite produced by the Smartesting Test Designer Tool, which cannot generate tests for dynamic system properties. Thus, they propose scenario language that can be used by the validation engineer, who basing on his experience can produce interesting scenarios to generate tests involving complex situations. Their work, is an adaptation for UMl base on the work done in [4]. The conceptual language of [4] from which the Smartesting Schema Language originates, also, was designed during a project (RNTL POSE) dedicated to testing the conformance of a system to a security policy. Its conception has been guided by the experience of security practitioners, resulting in a language that well serves the aim of testing security properties. Indeed, considering both actions to perform and states to reach is the way a security engineer thinks of testing a security issue.

The originality of our Schema Language with respect to these related approaches can be summarized in three points as discussed in Section III: scientific point of view, technological point of view and its expressivity. This language allows a validation engineer to benefit plainly from its good knowledge of the model and to explicitly use all artifacts of the model (such as objects names).

Chetali in [22] has pointed out the need to have an automated approach allowing to prove security properties on a system and to write scenarios to produce functional tests as well in the smart-card context. To the extent of our knowledge there is however so far no published work on consistent model-based testing for security properties, neither for smart-cards or in general.

VII. Conclusion and Future works

This paper presents a model-based technique for test generation from schemas for UML/OCL models and its integration with the UMLsec verification approach, in order to gain more confidence in models for testing with respect to security properties and to facilitate the property expression from stereotypes into test schemas.

We enhance the verification activities to test models, and the kind of properties that are verified on the model. In addition, we propose that both approaches are used by the same actor (the validation engineer).

As underlined in the discussion part our integrated approach for now is limited only on two security properties, thus our goal is to generalize it. For instance the security property is formalized in UMLsec stereotype and then we transform it into a schema. Thus, our goal is to make a chain allowing to produce a schema from an UMLsec stereotype and vice versa, be able to produce an UMLsec stereotype from a given schema. It is in our perspective to adapt more the exported formulas and enrich the transformation rules, thus to be able to benefit as much as possible of the schema language expression power. We are limited by the schema language also, for example we are not able to use explicitly the operation parameters. We focus also on its improvement.

Furthermore, UMLsec also allows us to model the adversary that can attack the different parts of the system in a specific way. Thus, we can use UMLsec to generate security oriented tests.

Another objective is to adapt this approach to smart-card specifications under evolution and to deal with regression testing. Thus, we can manage the test life cycle and create dedicated test suites: regression, evolution, stagnation and deletion to test as given in [23] by taking into account

evolution and management of security properties w.rt. to the specification.

ACKNOWLEDGMENT

This research is supported by the EU project Security Engineering for Lifelong Evolvable Systems (Secure Change, ICT-FET-231101).

REFERENCES

[1] J. Jürjens, *Secure Systems Development with UML*. Springer-Verlag, 2005.

[2] "Global platform specification 2.1.1," March, 2003.

[3] "UMLsec tool," 2001-10, http://umlsec.de.

[4] J. Julliand, P.-A. Masson, and R. Tissot, "Generating security tests in addition to functional tests," in *AST'08, 3rd Int. workshop on Automation of Software Test*, Leipzig, Germany, May 2008, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1370042.1370051

[5] P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, and A. Haddad, "An access control model based testing approach for smart card applications: Results of the POSÉ project," *JIAS, Journal of Information Assurance and Security*, vol. 5, no. 1, pp. 335–351, 2010.

[6] J. Julliand, P.-A. Masson, R. Tissot, and P.-C. Bué, "Generating tests from B specifications and dynamic selection criteria," *FAC, Formal Aspects of Computing*, vol. 23, no. 1, pp. 3–19, 2011.

[7] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, 1999.

[8] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," *Formal Engineering Methods, International Conference on*, vol. 0, p. 46, 1998.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE'99, 21st international conference on Software engineering*, Los Angeles, California, United States, 1999, pp. 411–420.

[10] C. Jard and T. Jéron, "Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, 2005.

[11] L. Frantzen, J. Tretmans, and T. Willemse, "Test generation based on symbolic specifications," in *FATES 2004, Formal Approaches to Software Testing*, ser. LNCS, J. Grabowski and B. Nielsen, Eds., vol. 3395. Springer, 2005, pp. 1–15.

[12] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva, "STG: A symbolic test generation tool," in *TACAS'02, Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2280. Springer, 2002, pp. 151–173.

[13] G. J. Tretmans and H. Brinksma, "TorX: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, Nuremberg, Germany, Dec. 2003, pp. 31–43.

[14] C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin, "Automatic test generation with AGATHA," in *TACAS 2003, Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference*, ser. LNCS, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 591–596.

[15] A. Bertolino, E. Marchetti, and H. Muccini, "Introducing a reasonably complete and coherent approach for model-based testing," *Electron. Notes Theor. Comput. Sci.*, vol. 116, pp. 85–97, Jan. 2005.

[16] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite approach to planning and deriving test suites in UML projects," in *UML'02, 5-th int. conf. on the UML language*, ser. LNCS, vol. 2460, London, UK, 2002, pp. 383–397.

[17] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp, "Concepts for Model-based Requirements Testing of Service Oriented Systems," in *Proceedings of the IASTED International Conference*, vol. 642, 2009, p. 018.

[18] Y. Ledru, F. Dadeau, L. Du Bousquet, S. Ville, and E. Rose, "Mastering combinatorial explosion with the TOBIAS-2 test generator," in *ASE'07: Procs of the 22nd IEEE/ACM int. conf. on Automated Software Engineering*, 2007, pp. 535–536.

[19] O. Maury, Y. Ledru, and L. du Bousquet, "Intégration de TOBIAS et UCASTING pour la génération des tests," in *ICSSEA'03, 16th Int. Conf. on Software and Systems Engineering and their Applications*, Paris, France, 2003.

[20] L. Van Aertryck and T. Jensen, "UML-CASTING: Test synthesis from UML models using constraint resolution," in *AFADL'03*, 2003.

[21] K. Cabrera Castillos and J. Botella, "Scenario based test generation using test designer," in *SCENARIOS'11, 1st Int. Workshop on Scenario Based Testing – co-located with ICST'2011*. Berlin, Germany: IEEE Computer Society Press, Mar. 2011, pp. ***–***, to appear.

[22] B. Chetali, "Security testing and formal methods for high levels certification of smart cards," in *Proceedings of the 3rd International Conference on Tests and Proofs*, ser. TAP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02949-3_1

[23] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon, "Selective test generation method for evolving critical systems," in *REGRESSION'11, 1st Int. Workshop on Regression Testing - co-located with ICST'2011*. Berlin, Germany: IEEE Computer Society Press, Mar. 2011, pp. ***–***, to appear.

A.11 ECMFA 2012: Incremental Security Verification for Evolving UMLsec Models

- Jan Jürjens, Loïc Marchal, Martín Ochoa, and Holger Schmidt. Incremental Security Verification for Evolving UMLsec models. In *Proc. of the 7th European Conference on Modelling Foundations and Applications, Birmingham, UK (ECMFA'11)*, pages 52–68, 2011.

Incremental Security Verification for Evolving UMLsec models*

Jan Jürjens[1,2], Loïc Marchal[3], Martín Ochoa[1] and Holger Schmidt[1]

[1] Software Engineering, Department of Computer Science, TU Dortmund, Germany
[2] Fraunhofer ISST, Germany
[3] Hermès Engineering, Belgium
{jan.jurjens,martin.ochoa,holger.schmidt}@cs.tu-dortmund.de
loic.marchal@hermes-ecs.com

Abstract. There exists a substantial amount of work on methods, techniques and tools for developing security-critical systems. However, these approaches focus on ensuring that the security properties are enforced during the initial system development and they usually have a significant cost associated with their use (in time and resources). In order to enforce that the systems remain secure despite their later evolution, it would be infeasible to re-apply the whole secure software development methodology from scratch. This work presents results towards addressing this challenge in the context of the UML security extension UMLsec. We investigate the security analysis of UMLsec models by means of a change-specific notation allowing multiple evolution paths and sound algorithms supporting the incremental verification process of evolving models. The approach is validated by a tool implementation of these verification techniques that extends the existing UMLsec tool support.

1 Introduction

The task of *evolving secure software systems* such that the desired security requirements are preserved through a system's lifetime is of great importance in practice. We propose a *model-based approach to support the evolution of secure software systems*. Our approach allows the verification of *potential future evolutions* using an automatic analysis tool. An explicit model evolution implies the transformation of the model and defines a difference Δ between the original model and the transformed one. The proposed approach supports the definition of multiple evolution paths, and provides tool support to verify evolved models based on the delta of changes. This idea is visualized in Fig. 1: The starting point of our approach is a Software System Model which was already verified against certain security properties. Then, this model can evolve within a range of possible evolutions (the evolution space). We consider the different possible evolutions as *evolution paths* each of which defines a *delta* Δ_i. The result is a number of

* This research was partially supported by the EU project Security Engineering for Lifelong Evolvable Systems (Secure Change, ICT-FET-231101)

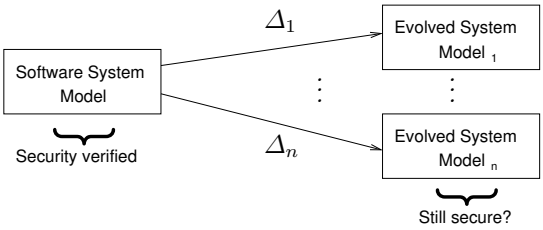

Fig. 1. Model verification problem for n possible evolution paths

evolved Evolved System Model$_i$. The main research question is "Which of the evolution paths leads to a target model that still fulfills the security properties of the source model?".

Theoretically, one could simply re-run the security analysis done to establish the security of the original model on the evolved model to decide whether these properties are preserved after evolution. This would, however, result in general in a high resource consumption for models of realistic size, in particular since the goal in general is to investigate the complete potential evolution space (rather than just one particular evolution) in order to determine which of the possible evolutions preserve security. Also, verification efficiency is very critical if a continuous verification is desired (i.e. it should be determined in real-time and in parallel to the modelling activity whether the modelled change preserves security).

We use models specified using the Unified Modeling Language (UML) [1] and the security extension UMLsec [6]. The UMLsec profile offers new UML language elements (i.e., *stereotypes*, *tags*, and *constraints*) to specify typical security requirements such as secrecy, integrity, and authenticity, and other security-relevant information. Based on UMLsec models and the semantics defined for the different UMLsec language elements, possible security vulnerabilities can be identified at an early stage of software development. One can thus verify that the desired security requirements, if fulfilled, enforce a given security policy. This verification is supported by a tool suite [2] [8].

In this paper we present a general approach for the incremental security verification of UML models against security requirements inserted as UMLsec stereotypes. We discuss the possible atomic (i.e. single model element) evolutions annotated with certain security requirements according to UMLsec. Moreover, we present *sufficient conditions* for a set of model evolutions, which, if satisfied, ensure that the desired security properties of the original model are preserved under evolution. We demonstrate our general approach by applying it to a representative UMLsec stereotype, «secure dependency». As one result of our work, we demonstrate that the security checks defined for UMLsec allow significant efficiency gains by considering this incremental verification technique.

[1] The Unified Modeling Language http://www.uml.org/
[2] Available online via `http://www-jj.cs.tu-dortmund.de/jj/umlsectool`

To explicitly specify possible evolution paths, we have developed a further extension of the UMLsec profile (called UMLseCh) that allows a precise definition of which model elements are to be *added*, *deleted*, and *substituted* in a model. Constraints in first-order predicate logic allow to coordinate and define more than one evolution path (and thus obtaining the deltas for the analysis).

Note that UMLseCh is not intended as a general-purpose evolution modeling language: it is specifically intended to model the evolution in a security-oriented context in order to investigate the research questions wrt. security preservation by evolution (in particular, it is an extension of UMLsec and requires the UMLsec profile as prerequisite profile). Thus, UMLseCh does not aim to be an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [4, 1, 2, 14, 9]). It will be interesting future work to demonstrate how the results presented in this paper can be used in the context of those approaches.

This paper is organized as follows: The change-specific extension UMLseCh is defined in Sect. 2. Sect. 3 explains our general approach for evolution-specific security verification. Using class diagrams as an example application, this approach is instantiated in Sect. 4. In Sect.5, we give an overview of the UMLsec verification tool and how this tool has been extended to support our reasoning for evolving systems based on UMLseCh. We conclude with an overview of the related work (Sect. 6) and a brief discussion of the results presented (Sect. 7).

2 UMLseCh: Supporting Evolution of UMLsec Models

In this section we present a further extension of the UML security profile UMLsec to deal with potential model evolutions, called UMLseCh (that is, an extension to UML which itself includes the UMLsec profile). Figure 2 shows the list of UMLseCh stereotypes, together with their tags and constraints, while Fig. 3 describes the tags.

The UMLseCh tagged values associated to the tags {`add`} and {`substitute`} are strings, their role is to describe possible future model evolutions. UMLseCh describes **possible future changes**, thus conceptually, the substitutive or additive model elements are not actually part of the current system design model, but only an attribute value inside a change stereotype[3]. At the concrete level, i.e. in a tool, this value is either the model element itself if it can be represented with a sequence of characters (for example an attribute or an operation within a class), or a namespace containing the model element.

Note that the UMLseCh notation is complete in the sense that any kind of evolution between two UMLsec models can be captured by adding a suitable number of UMLseCh annotations to the initial UMLsec model. This can be seen by considering that for any two UML models M and N there exists a sequence of deletions, additions, and substitutions through which the model M can be transformed to the model N. In fact, this is true even when only

[3] The type change represents a type of stereotype that includes «change»,«substitute», «add» or «delete».

Stereotype	Base Class	Tags	Constraints	Description
change	all	ref, change	FOL formula	execute sub-changes in parallel
substitute	all	ref, substitute,	FOL formula	substitute a model element
add	all	ref, add,	FOL formula	add a model element
delete	all	ref, delete	FOL formula	delete a model element
substitute-all	all	ref, substitute,	FOL formula	substitute a group of elements
add-all	all	ref, add,	FOL formula	add a group of elements
delete-all	all	ref, delete	FOL formula	delete a group of elements

Fig. 2. UMLseCh stereotypes

Tag	Stereotype	Type	Multip.	Description
ref	change, substitute, add, delete, substitute-all, add-all, delete-all	list of strings	1	List of labels identifying a change
substitute	substitute, substitute-all	list of pairs of model elements	1	List of substitutions
add	add, add-all	list of pairs of model elements	1	List of additions
delete	delete, delete-all	list of pairs of model elements	1	List of deletions
change	change	list of references	1	List of simultaneous changes

Fig. 3. UMLseCh tags

considering deletions and additions: the trivial solution would be to sequentially remove all model elements from M by subsequent atomic deletions, and then to add all model elements needed in N by subsequent additions. Of course, this is only a thereotical argument supporting the theoretical expressiveness of the UMLseCh notation, and this approach would neither be useful from a modelling perspective, nor would it result in a meaningful incremental verification strategy. This is the reason that the substitution of model elements has also been added to the UMLseCh notation, and the incremental verification strategy explained later in this paper will crucially rely on this.

2.1 Description of the Notation

In the following we give an informal description of the notation and its semantics.

substitute The stereotype «substitute» attached to a model element denotes the possibility for that model element to evolve over time and defines what the possible changes are. It has two associated tags, namely ref and substitute. These tags are of the form {ref = CHANGE-REFERENCE} and

$$\{\,\mathsf{substitute} = (\mathsf{ELEMENT}_1,\, \mathsf{NEW}_1),\, \ldots\, ,\, (\mathsf{ELEMENT}_n,\, \mathsf{NEW}_n)\,\}$$

with $n \in \mathbb{N}$. The tag ref takes a list of sequences of characters as value, each element of this list being simply used as a reference of one of the changes modeled by the stereotype «substitute». In other words, the values contained in this tag can be seen as labels identifying the changes. The values of this tag can also be considered as predicates which take a truth value that can be used to evaluate conditions on other changes (as we will explain in the following). The tag substitute has a list of pairs of model element as value, which represent the substitutions that will happen if the related change occurs. The pairs are of the form (e, e'), where e is the element to substitute and e' is the substitutive model element [4]. For the notation of this list, two possibilities exist: The elements of the pair are written textually using the abstract syntax of a fragment of UML defined in [6] or alternatively the name of a namespace containing an element is used instead. The *namespace notation* allows UMLseCh stereotypes to graphically model more complex changes (cf. Sect. 2.2).

If the model element to substitute is the one to which the stereotype «substitute» is attached, the element e of the pair (e, e') is not necessary. In this case the list consists only of the second elements e' in the tagged value, instead of the pairs (this notational variation is just syntactic sugar). If a change is specified, it is important that it leaves the resulting model in a syntactically consistent state. In this paper however we focus only on the preservation of security.

Example We illustrate the UMLseCh notation with the following example. Assume that we want to specify the change of a link stereotyped «Internet» so that it will instead be stereotyped «encrypted». For this, the following three annotations are attached to the link concerned by the change (cf. Figure 4):

$$\text{«substitute»},\ \{\,\mathsf{ref} = \mathsf{encrypt\text{-}link}\,\},\ \{\,\mathsf{substitute} = (\text{«encrypted»},\ \text{«Internet»})\,\}$$

The stereotype «substitute» also has a list of optional constraints formulated in first order logic. This list of constraints is written between square brackets and is of the form $[(\mathsf{ref}_1,\, \mathsf{CONDITION}_1),\, \ldots\, ,\, (\mathsf{ref}_n,\, \mathsf{CONDITION}_n)]$, $n \in \mathbb{N}$, where, $\forall i : 1 \leq i \leq n$, ref_i is a value of the list of a tag ref and $\mathsf{CONDITION}_n$ can be any type of first order logic expression, such as $A \wedge B$, $A \vee B$, $A \wedge (B \vee \neg C)$, $(A \wedge B) \Rightarrow C$, $\forall x \in N.P(x)$, etc. Its intended use is to define under which conditions the change is allowed to happen (i.e. if the condition is evaluated to

[4] More than one occurrence of the same e in the list is allowed. However, two occurrences of the same pair (e, e') cannot exist in the list, since it would model the same change twice.

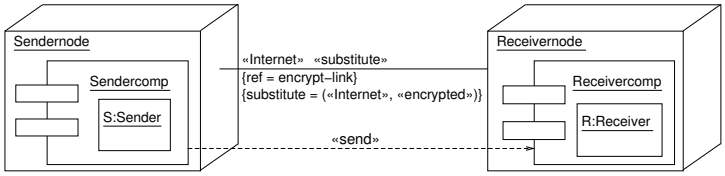

Fig. 4. Example of stereotype substitute

true, the change is allowed, otherwise the change is not allowed). As mentioned earlier, an element of the list used as the value of the tag ref of a stereotype « substitute » can be used as an atomic predicate for the constraint of another stereotype « substitute ». The truth value of that predicate is true if the change represented by the stereotype « substitute » to which the tag ref is associated occurred, false otherwise.

To illustrate the use of the constraint, the previous example can be refined. Assume that to allow the change with reference encrypt-link, another change, simply referenced as change for the example, has to occur. The constraint [change] can then be attached to the link concerned by the change. To express for example that two changes, referenced respectively by change_1 and change_2, have to occur first in order to allow the change referenced encrypt-link to happen, the constraint $[\mathsf{change}_1 \wedge \mathsf{change}_2]$ is added to the stereotype « substitute » modeling the change.

add and delete Both « add » and « delete » can be seen as syntactic sugar for « substitute ». The stereotype « add » attached to a parent model element describes a list of possible sub-model elements to be added as children to the parent model element. It thus substitutes a collection of sub-model elements with a new, extended collection.

The stereotype « delete » attached to a (sub)-model element marks this element for deletion. Deleting a model element could be expressed as the substitution of the model element by the empty model element $\emptyset$. Both stereotypes « add » and « delete » may also have associated constraints in first order logic.

substitute-all The stereotype « substitute-all » is an extension of the stereotype « substitute ». It denotes the possibility for **a set of model elements of same type and sharing common characteristics** to evolve over time. In this case, « substitute-all » will always be attached to the super-element to which the sub-elements concerned by the substitution belong. As the stereotype « substitute », it has the two associated tags ref and substitute, of the form { ref = CHANGE-REFERENCE } and

$$\{\,\mathsf{substitute} = (\mathsf{ELEMENT}_1,\, \mathsf{NEW}_1),\, \ldots\, ,\, (\mathsf{ELEMENT}_n,\, \mathsf{NEW}_n)\,\}.$$

The tags ref has the same meaning as in the case of the stereotype « substitute ». For the tag substitute the element e of a pair representing a substitution does not represent one model element but **a set of model elements** to substitute if a change occurs. This set can be, for example, a set of classes, a set of methods of a

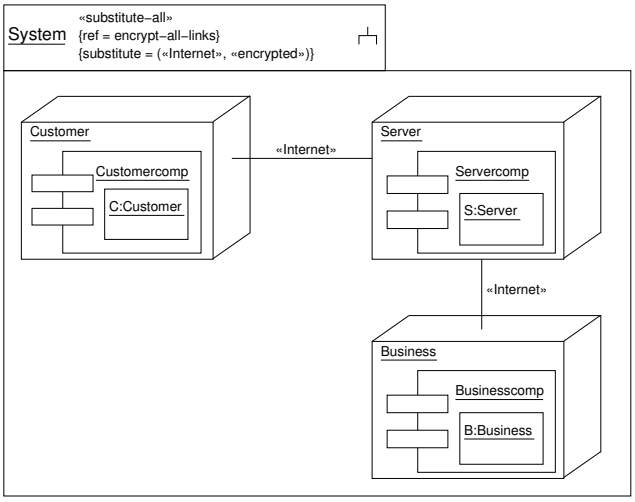

Fig. 5. Example of stereotype substitute-all

class, a set of links, a set of states, etc. All the elements of the set share common characteristics. For instance, the elements to substitute are the methods having the integer argument *"count"*, the links being stereotyped « Internet » or the classes having the stereotype « critical » with the associated tag secrecy. Again, in order to identify the model element precisely, we can use, if necessary, either the UML namespaces notation or, if this notation is insufficient, the abstract syntax of UMLseCh.

Example To replace all the links stereotyped « Internet » of a subsystem so that they are now stereotyped « encrypted », the following three annotations can be attached to the subsystem: « substitute-all », { ref = encrypt-all-links }, and { substitute = (« Internet », « encrypted ») }. This is shown in Figure 5.

A pair (e, e') of the list of values of a tag substitute here allows us a parameterization of the values e and e' in order to keep information of the different model elements of the subsystem concerned by the substitution. To allow this, variables can be used in the value of both the elements of a pair. The following example illustrates the use of the parameterization in the stereotype « substitute-all ». To substitute all the tags secrecy of stereotypes « critical » by tags integrity, but in a way that it keeps the values given to the tags secrecy (e.g. { secrecy = d }), the following three annotations can be attached to the subsystem containing the class diagram: « substitute-all », { ref = secrecy-to-integrity }, and { substitute = ({ secrecy = X }, { integrity = X }) }.

The stereotype « substitute-all » also has a list of constraints formulated in first order logic, which represents the same information as for the stereotype « substitute ».

change The stereotype « change » is a particular stereotype that represents a *composite change*. It has two associated tags, namely ref and change. These tags are of the form { ref = CHANGE-REFERENCES } and { change = CHANGE-REFERENCES$_1$, ..., CHANGE-REFERENCES$_n$ }, with $n \in \mathbb{N}$. The tag ref has the same meaning as in the case of a stereotype « substitute ». The tag change takes a list of lists of strings as value. Each element of a list is a value of a tag ref from another stereotype of type change.[5] Each list thus represents the list of *sub-changes* of a *composite change* modeled by the stereotype « change ». Applying a change modeled by « change » hence consists in applying all of the concerned *sub-changes* **in parallel**.

Any change being a *sub-change* of a change modeled by « change » **must** have the value of the tag ref of that change in its condition. Therefore, any change modeled by a *sub-change* can only happen if the change modeled by the *super-stereotype* takes place. However, if this change happens, the *sub-changes* will be applied and the *sub-changes* will thus be removed from the model. This ensures that *sub-changes* cannot be applied by themselves, independently from their *super-stereotype* « change » modeling the *composite change*.

2.2 Complex Substitutive Elements

As mentioned above, using a complex model element as substitutive element requires a syntactic notation as well as an adapted semantics. An element is complex if it is not represented by a sequence of characters (i.e. it is represented by a graphical icon, such as a class, an activity or a transition). Such complex model elements cannot be represented in a tagged value since tag definitions have a string-based notation. To allow such complex model elements to be used as substitutive elements, they will be placed in a UML namespace. The name of this namespace being a sequence of characters, it can thus be used in a pair of a tag substitute where it will then represent a reference to the complex model element. Of course, this is just a notational mechanism that allows the UMLseCh stereotypes to graphically model more complex changes. From a semantic point of view, when an element in a pair representing a substitution is the name of a namespace, the model element concerned by the change will be substituted by the content of the namespace, and not the namespace itself. This type of change will request a special semantics, depending on the type of element. For details about this complex substitutions we refer to [15].

3 Verification Strategy

As stated in the previous section, evolving a model means that we either *add*, *delete*, or / and *substitute* elements of this model. To distinguish between big-step and small-step evolutions, we will call "atomic" the modifications involving only one model element (or sub-element, e.g. adding a method to an existing

[5] By type change, we mean the type that includes « substitute », « add », « delete » and « change ».

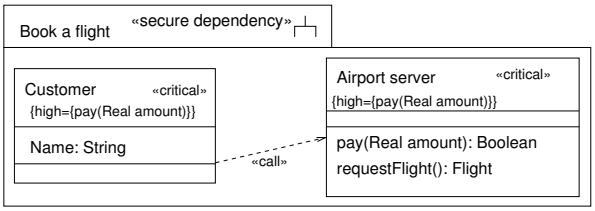

Fig. 6. Class Diagram Annotated with « secure dependency »

class or deleting a dependency). In general there exist evolutions from diagram A to diagram B such that there is no sequence of atomic modifications for which security is preserved when applying them one after another, but such that both A and B are secure. Therefore the goal of our verification is to allow some modifications to happen *simultaneously*.

Since the evolution is defined by additions, deletion and substitutions of model elements, we introduce the sets **Add**, **Del**, and **Subs**, where **Add** and **Del** contain objects representing model elements together with methods id, type, path, parent returning respectively an identifier for the model element, its type, its path within the diagram, and its parent model element. These objects also contain all the relevant information of the model element according to its type (for example, if it represents a class, we can query for its associated stereotypes, methods, and attributes). For example, the class "Customer" in Fig. 6 can be seen as an object with the subsystem "Book a flight" as its parent. It has associated a list of methods (empty in this case), a list of attributes ("Name" of type String, which is in turn an model element object), a list of stereotypes (« critical ») and a list of dependencies (« call » dependency with "Airport Server") attached to it. By recursively comparing all the attributes of two objects, we can establish whether they are equal.

The set **Subs** contains pairs of objects as above, where the type, path (and therefore parent) methods of both objects must coincide. We assume that there are no conflicts between the three sets, more specifically, the following condition guarantees that one does not delete and add the same model element:

$$\nexists\, o, o'(o \in \mathbf{Add} \wedge o' \in \mathbf{Del} \wedge o = o')$$

Additionally, the following condition prevents adding/deleting a model element present in a substitution (as target or as substitutive element):

$$\nexists\, o, o'(o \in \mathbf{Add} \vee o \in \mathbf{Del}) \ \wedge ((o, o') \in \mathbf{Subs} \vee (o', o) \in \mathbf{Subs})$$

As explained above, in general, an "atomic" modification (that is the action represented by a single model element in any of the sets above) could by itself harm the security of the model. So, one has to take into account other modifications in order to establish the security status of the resulting model. We proceed algorithmically as follows: we iterate over the modification sets starting with an object $o \in \mathbf{Del}$, and if the relevant simultaneous changes that preserve security are found in the delta, then we perform the operation on the original model

(delete o and necessary simultaneous changes) and remove the processed objects until **Del** is empty. We then continue similarly with **Add** and finally with **Subs**. If at any point we establish the security is not preserved by the evolution we conclude the analysis. Given a diagram M and a set Δ of atomic modifications we denote $M[\Delta]$ the diagram resulting after the modifications have taken place. So in general let P be a diagram property. We express the fact that M enforces P by $P(M)$. *Soundness* of the security preserving rules R for a property P on diagram M can be formalized as follows:

$$P(M) \wedge R(M, \Delta) \Rightarrow P(M[\Delta]).$$

To prove that the algorithm described above is sound with respect to a given property P, we show that every set of simultaneous changes accepted by the algorithm preserves P. Then, transitively, if all steps were sound until the delta is empty, we reach the desired $P(M[\Delta])$.

One can obtain these deltas by interpreting the UMLseCh annotations presented in the previous section. Alternatively, one could compute the difference between an original diagram M and the modified M'. This is nevertheless not central to this analysis, which focuses on the verification of evolving systems rather than on model transformation itself.

To define the set of rules R, one can reason inductively by cases given a security requirement on UML models, by considering incremental atomic changes and distinguishing them according to a) their *evolution* type (addition, deletion, substitution) and b) their *UML diagram* type. In the following section we will spell-out a set of possible sufficient rules for the sound and secure evolution of class diagrams annotated with the « secure dependency » stereotype.

4 Application to <<secure dependency>>

In this section we demonstrate the verification strategy explained in the previous section by applying it to the case of the UMLsec stereotype « secure dependency » applied to class diagrams. The associated constraint requires for every communication dependency (i.e. a dependency annotated « send » or « call ») between two classes in a class diagram the following condition holds: if a method or attribute is annotated with a security requirement in one of both classes (for example { secrecy = {method()} }), then the other class has the same tag for this method/attribute as well (see Fig. 6 for an example). It follows that the computational cost associated with verifying this property depends on the number of *dependencies*. We analyze the possible changes involving classes, dependencies and security requirements as specified by tags and their consequences to the security properties of the class diagram.

Formally, we can express this property as follows:

$$P(M): \ \forall C, C' \in M.\mathsf{Classes}\ (\exists d \in M.\mathsf{dependencies}(C, C') \Rightarrow C.\mathsf{critical} = C'.\mathsf{critical})$$

where $M.\mathsf{Classes}$ is the set of classes of diagram M, $M.\mathsf{dependencies}(C, C')$ returns the set of dependencies between classes C and C' and $C.\mathsf{critical}$ returns

the set of pairs (m, s) where m is a method or an object shared in the dependency and $s \in \{\mathsf{high}, \mathsf{secrecy}, \mathsf{integrity}\}$ as specified in the «critical» stereotype for that class.

We now analyse the set Δ of modifications by distinguishing cases on the evolution type (deletion, addition, substitution) and the UML type.

Deletion

Class : We assume that if a class $\bar{C}$ is deleted then also the dependencies coming in and out of the class are deleted, say by deletions $D = \{o_1, ..., o_n\}$, and therefore, after the execution of o and D in the model M (expressed $M[o, D]$) property P holds since:

$$P(M[o, D]) :$$

$$\forall C, C' \in M.\mathsf{Classes} \setminus \bar{C}\ (\exists d \in M[o, D].\mathsf{dependencies}(C, C') \Rightarrow C.\mathsf{critical} = C'.\mathsf{critical})$$

and this predicate holds given $P(M)$, because the new set of dependencies of $M[o, D]$ does not contain any pair of the type $(x, \bar{C})$, $(\bar{C}, x)$.

Tag in critical : If a security requirement (m, s) associated to in class $\bar{C}$ is deleted then it must also be removed from other methods having dependencies with C (and so on recursively for all classes $C_{\bar{C}}$ associated through dependencies to $\bar{C}$) in order to preserve the secure dependencies requirement. We assume $P(M)$ holds, and since clearly $M.\mathsf{Classes} = (M.\mathsf{Classes} \setminus C_{\bar{C}}) \cup C_{\bar{C}}$ it follows $P(M[o, D])$ because the only modified objects in the diagram are the classes in $C_{\bar{C}}$ and for that set we deleted symmetrically (m, s), thus respecting P.

Dependency : The deletion of a dependency does not alter the property P since by assumption we had a statement quantifying over all dependencies (C, C'), that trivially also holds for a subset.

Addition

Class : The addition of a class, without any dependency, clearly preserves the security of P since this property depends only on the classes with dependencies associated to them.

Tag in critical : To preserve the security of the system, every time a method is tagged within the «critical» stereotype in a class C, the same tag referring to the same method should be added to every class with dependencies to and from C (and recursively to all dependent classes). The execution of these simultaneous additions preserves P since the symmetry of the critical tags is respected through all dependency-connected classes.

Dependency : Whenever a dependency is added between classes C and C', for every security tagged method in C (C') the same method must be tagged (with the same security requirement) in C' (C) to preserve P. So if in the original model this is not the case, we check for simultaneous additions that preserve this symmetry for C and C' and transitively on all their dependent classes.

Substitution

Class : If class C is substituted with class C' and class C' has the same security tagged methods as C then the security of the diagram is preserved.

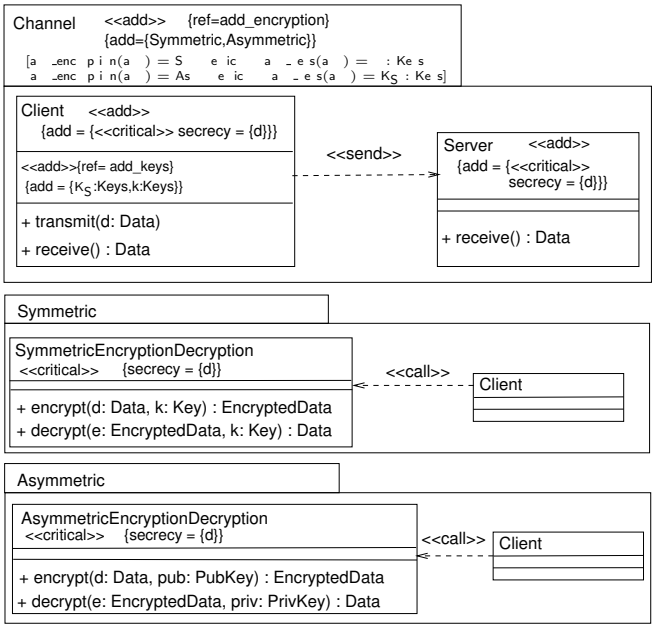

Fig. 7. An evolving class diagram with two possible evolution paths

Tag in critical : If we substitute $\{\,\mathsf{requirement} = \mathsf{method}()\,\}$ by $\{\,\mathsf{requirement'} = \mathsf{method}()'\,\}$ in class C, then the same substitution must be made in every class linked to C by a dependency.
Dependency : If a $\ll\mathsf{call}\gg$ ($\ll\mathsf{send}\gg$) dependency is substituted by $\ll\mathsf{send}\gg$ ($\ll\mathsf{call}\gg$) then P is clearly preserved.

Example The example in Fig. 7 shows the Client side of a communication channel between two parties. At first (disregarding the evolution stereotypes) the communication is unsecured. In the packages Symmetric and Asymmetric, we have classes providing cryptographic mechanisms to the Client class. Here the stereotype $\ll\mathsf{add}\gg$ marked with the reference tag {ref} with value add_encryption specifies two possible evolution paths: merging the classes contained in the current package (Channel) with either Symmetric or Asymmetric. There exists also a stereotype $\ll\mathsf{add}\gg$ associated with the Client class adding either a pre-shared private key k or a public key $\mathsf{K_S}$ of the server. To coordinate the intended evolution paths for these two stereotypes, we can use the following first-order logic constraint (associated with add_encryption):

$$[\mathsf{add\_encryption}(\mathsf{add}) = \mathsf{Symmetric} \Rightarrow \mathsf{add\_keys}(\mathsf{add}) = \mathsf{k} : \mathsf{Keys} \ \wedge$$
$$\mathsf{add\_encryption}(\mathsf{add}) = \mathsf{Asymmetric} \Rightarrow \mathsf{add\_keys}(\mathsf{add}) = \mathsf{K_S} : \mathsf{Keys}]$$

The two deltas, representing two possible evolution paths induced by this notation, can be then given as input to the decision procedure described for

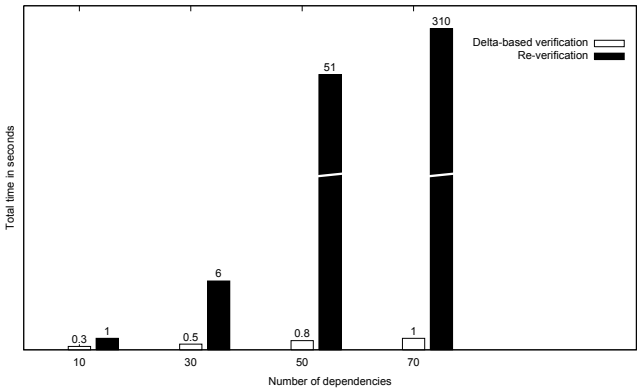

Fig. 8. Running time comparison of the verification

checking « secure dependency ». Both evolution paths respect sufficient conditions for this security requirement to be satisfied.

5 Tool support

The UMLsec extension [6] together with its formal semantics offers the possibility to verify models against security requirements. Currently, there exists tool support to verify a wide range of diagrams and requirements. Such requirements can be specified in the UML model using the UMLsec extension (created with the ArgoUML editor) or within the source-code (Java or C) as annotations. As explained in this paper, the UMLsec extension has been further extended to include evolution stereotypes that precisely define which model elements are to be added, deleted, or substituted in a model (see also the UMLseCh profile in [15]). To support the UMLseCh notation, the UMLsec Tool Suite has been extended to process UML models including annotations for possible future evolutions.[6]

Given the sufficient conditions presented in the previous sections, if the transformation does not violate them then the resulting model preserves security. Nevertheless, security preserving evolutions may fail to pass the tests discussed, and be however valid: With respect to the security preservation analysis procedures, there is a trade-off between their efficiency and their completeness. Essentially, if one would require a security preservation analysis which is complete in the sense that every specified evolution which preserves security is actually shown to preserve security, the computational difficulty of this analysis could be comparable to a simple re-verification of the evolved model using the UMLsec tools. Therefore if a specified evolution could not be established to preserve security, there is still the option to re-verify the evolved model.

It is of interest that the duration of the check for « secure dependency » implemented in the UMLsec tool behaves in a more than linear way depending on the

[6] Available online at `http://www-jj.cs.tu-dortmund.de/jj/umlsectool/manuals_new/UMLseCh_Static_Check_SecureDependency/index.htm`

number of dependencies. In Fig. 8 we present a comparison between the running time of the verification[7] on a class diagram where only 10% of the model elements were modified. One should note that the inefficiency of a simple re-verification would prevent analyzing evolution spaces of significant size, or to support online verification (i.e. verifying security evolution in parallel to the modelling activity), which provides the motivation to profit from the gains provided by the delta-verification presented in this paper. Similar gains can be achieved for other UMLsec checks such as «rbac», «secure links» and other domain-specific security properties for smart-cards, for which sound decision procedures under evolution have been worked out (see [15]).

6 Related Work

There are different approaches to deal with evolution that are related to our work. Within *Software Evolution Approaches*, [10] derives several *laws of software evolution* such as "Continuing Change" and "Declining Quality". [12] argue that it is necessary to treat and support evolution throughout all development phases. They extend the UML metamodel by *evolution contracts* to automatically detect conflicts that may arise when evolving the same UML model in parallel. [16] proposes an approach for transforming non-secure applications into secure applications through requirements and software architecture models using UML. However, the further evolution of the secure applications is not considered, nor verification of the UML models. [5] discussed consistency of models for incremental changes of models. This work is not security-specific and it considers one evolution path only.

Also related is the large body of work on software verification based on *Assume-Guarantee reasoning*. A difference is that our approach can reason incrementally without the need for the user to explicitly formulate assume-guarantee conditions.

In the context of *Requirements Engineering for Secure Evolution* there exists some recent work on requirements engineering for secure systems evolution such as [17]. However, this does not target the security verification of evolving design models. A research topic related to software evolution is *software product lines*, where different versions of a software are considered. For example, Mellado et al. [11] consider product lines and security requirements engineering. However, their approach does not target the verification of UML models for security properties. *Evolving Architectures* is a similar context with a different level of abstraction. [3] discusses different evolution styles for high-level architectural views of the system. It also discusses the possibility of having more than one evolution path and describes tool support for choosing the "correct" paths with respect to properties described in temporal logic (similar to our constraints in FOL). However, this approach is not security specific. On a similar fashion, but more focused on critical properties, [13] also discusses the evolution of Architectures.

The UMLseCh notation is informally introduced in [7], however no details about verification are given. Both the notation and the verification aspects are

[7] On a 2.26 GhZ dual core processor

treated in more detail in the (unpublished) technical report [15] of the SecureChange Project. Note that UMLseCh does not aim to be an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [4, 1, 2, 14, 9]) or model transformation languages such as QVT [8] or ATL [9]. It will be interesting future work to demonstrate how the results presented in this paper can be used in the context of those approaches.

To summarize, to the extent of our knowledge there is so far no published work that considers evolution in the context of a model-based development approach for security-critical software involving more than one evolution path and automated model verification.

7 Conclusion

This paper concerns the preservation of security properties of models in different evolution scenarios. We considered selected classes of model evolutions such as addition, deletion, and substitution of model elements based on UMLsec diagrams. Assuming that the starting UMLsec diagrams are secure, which one can verify using the UMLsec tool framework, our goal is to re-use these existing verification results to minimize the effort for the security verification of the evolved UMLsec diagrams. This is critical since simple re-verification would in general result in a high resource consumption for models of realistic size, specially if a continuous verification is desired (i.e. it should be determined in real-time and in parallel to the modelling activity whether the modelled change preserves security).

We achieved this goal by providing a general approach for the specification and analysis of a number of sufficient conditions for the preservation of different security properties of the starting models in the evolved models. We demonstrated this approach at the hand of the UMLsec stereotype « secure dependency ». This work has been used as a basis to extend the existing UMLsec tool framework by the ability to support secure model evolution. This extended tool supports the development of evolving systems by pointing out possible security-violating modifications of secure models. We also show that the implementation of the techniques described in this paper leads to a significant efficiency gain compared to the simple re-verification of the entire model.

Our work can be extended in different directions. For example, we plan to increase the completeness of the approach by analyzing additional interesting model evolution classes. Also, it would be interesting to generalize our approach to handle other kinds of properties beyond security properties.

References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[8] Query/View/Transformation Specification http://www.omg.org/spec/QVT/
[9] The ATLAS Transformation Language http://www.eclipse.org/atl/

2. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 440–453. Springer, 2006.
3. D. Garlan, J. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *WICSA/ECSA 2009*, pages 131 –140, sept. 2009.
4. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Proceedings of international conference on Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.
5. S. Johann and A. Egyed. Instant and incremental transformation of models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 362–365, Washington, DC, USA, 2004. IEEE Computer Society.
6. J. Jürjens. *Principles for Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, 2002.
7. J. Jürjens, M. Ochoa, H. Schmidt, L. Marchal, S. Houmb, and S. Islam. Modelling secure systems evolution: Abstract and concrete change specifications (invited lecture). In I. Bernardo, editor, *11th School on Formal Methods (SFM 2011), Bertinoro (Italy) 13-18 June 2011*, LNCS. Springer, 2011.
8. J. Jürjens and P. Shabalin. Tools for secure systems development with UML. *Intern. Journal on Software Tools for Technology Transfer*, 9(5–6):527–544, Oct. 2007. Invited submission to the special issue for FASE 2004/05.
9. D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 6(9):53–69, 2007.
10. M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution – The Nineties View. In *METRICS'97*, pages 20–32, Washington, DC, USA, 1997. IEEE Computer Society.
11. D. Mellado, J. Rodriguez, E. Fernandez-Medina, and M. Piattini. Automated Support for Security Requirements Engineering in Software Product Line Domain Engineering. In *AReS'09*, pages 224–231, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
12. T. Mens and T. D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering Journal*, 7(1):39–59, February 2000.
13. T. Mens, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *Computer*, 43(5):42 –48, May 2010.
14. A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proceedings of the International Conference in Graph Transformation (ICGT)*, pages 226–241. Springer, 2004.
15. Secure Change Project. Deliverable 4.2. Available as http://www-jj.cs.tu-dortmund.de/jj/deliverable_4_2.pdf.
16. M. E. Shin and H. Gomaa. Software requirements and architecture modeling for evolving non-secure applications into secure applications. *Science of Computer Programming*, 66(1):60–70, 2007.
17. T. T. Tun, Y. Yu, C. B. Haley, and B. Nuseibeh. Model-based argument analysis for evolving security requirements. In *SSIRI'10*, pages 88–97. IEEE Computer Society, 2010.