Specifying Model Changes with UMLchange to Support Security Verification of Potential Evolution☆

S. Wenzel[a], D. Warzecha[b], J. Jürjens[a,b], M. Ochoa[c]

[a]*Chair of Software Engineering, TU Dortmund, D-44221 Dortmund*
[b]*Fraunhofer ISST, Emil-Figge-Str. 91, D-44227 Dortmund*
[c]*Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, D-81739 Munich*

Abstract

Model-based development is a widely accepted methodology where software or parts of it is generated from models. In order to ensure quality properties such as consistency of security requirements the models are often verified prior code generation. In case of evolution, the changed models have to be reverified before re-generation. However, if several alternative evolutions of a model are possible, each alternative has to be modeled and verified in order to find the best model for further development and code generation.

In this paper we present a verification strategy to analyze whether a software evolution preserves a given security property. This is presented on the basis of the UML profile UMLchange which can be used for specifying potential evolutions of a given model simultaneously. UMLchange makes our approach independent from specific modeling tools. We also present an extensible tool that reads the annotations of EMF-based UML2 models and computes a delta model containing all possible evolution paths of the given model. The evolution paths can be verified wrt. security properties, and for each successfully verified path a new model version is generated automatically.

Keywords:
Model Evolution, Security Verification, UML Profile, Tool Support

1. Introduction

A significant portion of software development nowadays is based on models. Especially in safety or security critical domains, modeling offers advantages such as formal specification and automated code generation. The formal semantics of models allow us to specify recurring security requirements and security assumptions on the system environment. This way knowledge on prudent security engineering is encaspulated as annotations in models. The models can be verified with regard to certain properties such as consistency of security requirements. Such verification is performed by dedicated analysis tools [1, 2, 3].

However, systems are neither developed in one step nor are they carved in stone. Hence, the task of *evolving 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 software systems *and* preserving consistency of security requirements. 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 Figure 1: The starting point of our approach is a software system model M 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 evolved system models M_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

☆Partially supported by the DFG grant JU 2734/2-1 (SecVolution) within SPP 1593 "Design For Future", and the EU projects FP7 256890 (NESSoS) and ICT-FET-231101 (SecureChange).

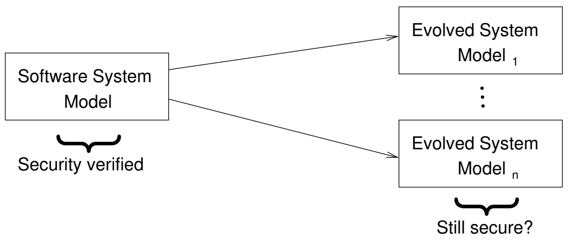


Figure 1: Model verification problem for *n* possible evolution paths

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. to determine in real-time and in parallel to the modeling activity whether the change preserves security).

We use models specified using the Unified Modeling Language (UML) and the security extension UMLsec [4]. UMLsec is given in form of a UML profile. *Stereotypes* are used together with *tags* to formulate the security requirements such as secrecy, integrity, and authenticity, and other security-relevant information. *Constraints* give criteria to determine if 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 by 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. Based on UMLsec models and the semantics defined for the different 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[1] [5].

In this paper we present a general approach for the incremental security verification of UML models regarding the consistency of 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.

To explicitly specify possible evolution paths, we have developed a further UML profile, called UMLchange, that allows a precise definition of which model elements are to be changed in a model. Constraints can be defined to coordinate and define more than one evolution path (and thus obtaining the deltas for the analysis).

Note that UMLchange is not intended as a general-purpose evolution modeling language. While existing evolution specification or model transformation approaches (such as [6, 7, 8, 9, 10]) are primarily used as a tool to change models, we needed a notation that allows us to put evolution in the focus of analysis. Thus, UMLchange is specifically intended to precisely define possible evolution paths of models. This enables the investigation of the preservation of security requirements by evolution. Thus, UMLchange does not aim to be an alternative for any of the existing approaches mentioned above. It will be interesting future work to demonstrate how the results presented in this paper can be used in the context of those approaches.

A preliminary version of UMLchange has been presented under the name *UMLseCh* at the ECMFA'11 conference [11]. Meanwhile we have improved the language. While UMLseCh was a set of additional stereotypes to UMLsec, we have now extracted the evolution specific parts and elaborated them into the UMLchange profile. This separation of concerns allowed us to deeper investigate the possible evolutions of a model, resulting in new stereotypes enabling a more precise description of evolution and supporting more complex types of evolution. Moreover, former limitations have been smoothed out. UMLchange is now fully compatible to UML version 2.4. The new grammar makes it possible to annotate multiple independent change alternatives to a single model element. New changes such as moving or copying elements can now be modeled, which better matches the perception of the user of how the model could evolve. Finally, the separation of evolution descriptions into a new profile enables its application of the evolution analysis to other quality properties. However, this is not focus of this paper. Here, we introduce the new profile and how it can be used to describe different evolutions. In particular we will explain

[1]Available online via `http://carisma.umlsec.de`

the grammar used to describe additive changes, which has been omitted in [11]. We furthermore show how the specified evolution paths can be verified wrt. preserving security properties.

The remainder of this paper is organized as follows: The change-specific extension UMLchange is defined in Section 2. Section 3 explains our general approach for evolution-aware security verification. In Section 4, we give an overview of the verification tool and evaluate our approach in Section 6. We conclude with an overview of the related work (Section 6) and a brief discussion of the results presented (Section 7).

2. The UMLchange Profile

The UMLchange profile consists of multiple stereotypes that can be used for the description of changes in a model. The majority of the stereotypes of the profile (excluding «old» and «keep») describe changes (i.e. the *change stereotypes*). To enable the description of all possible changes to a UML model, the change stereotypes can be applied to any UML model element. This is indicated by the extension relationships targeting the meta class *Element*, the super class of all UML elements.

Figure 2 provides some basic 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*.

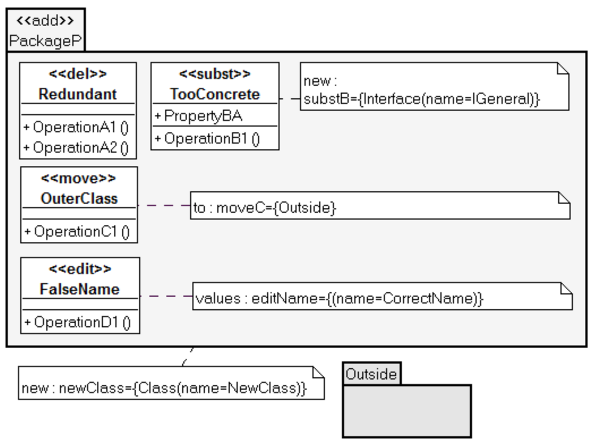


Figure 2: Examples of Change Stereotypes

The components of the profile (see Figure 3) are described in more detail below.

secure dependency. To be able to demonstrate the UMLchange profile for evolution-aware security analysis, we give a short description of the UMLsec stereotype «secure dependency». A detailed explanation of all of the tags and stereotypes defined in UMLsec can be found in [13]. «secure dependency» can be used to label subsystems containing static structure diagrams. It ensures that the «call» and «send» dependencies between objects or subsystems respect the security requirements on the data that may be communicated across 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} resp. {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} resp. {high}) in *C* then the dependency is stereotyped «secrecy» (resp. «integrity» resp. «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.

2.1. Common Properties and Tags

Each change stereotype 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:

$$ChangeID = StereotypeName[.TagName]$$

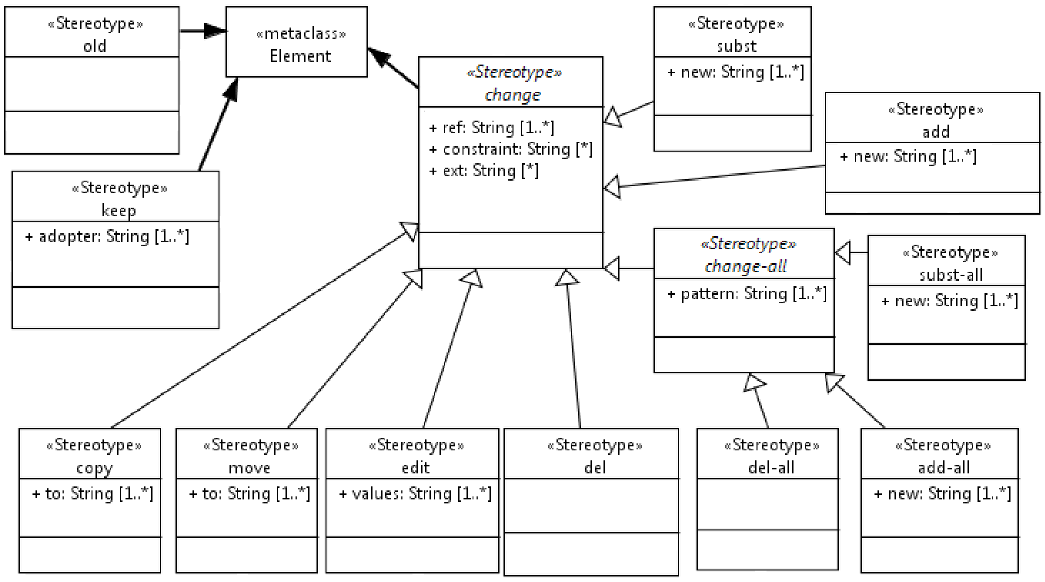


Figure 3: The UMLchange profile (core elements)

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 stereotype name must be appended with the name of the targeted tag.

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 requires another change to be applied before the change in question (*REQ(OtherChangeID)*). A change may have more than one constraint. For ease of use, each constraint can either be a separate {constraint} entry or be part of a comma-separated list of constraints. Contradicting constraints lead to not including any of the conflicting changes.

2.2. Stereotypes for Basic Change Operations

A change to a model element can be of one of three basic types:

- deletion of an element from the model
- addition of a new element to an existing model element
- substitution of an element with another element

Every possible evolution to a model can be described with one or more of these three types. The UMLchange profile offers the corresponding stereotypes to enable the description of the basic types.

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 new substitute element (or elements) is the parent of the old substituted element. By substituting old elements, all of their contained elements are removed from the model, as well as all connection model elements. To prevent deleting contained elements, the stereotype «keep» must be applied accordingly (see 2.3.3). 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 2.4.

In the example model in Figure 4, the integrity tag value for the *Server* is substituted with a high requirement. As the resulting model would be insecure, the *integrity* tag value for the *Client* is removed and a security preserving appropriate *high* value is added.

2.3. Additional Stereotypes for Convenience

Although every change to a model can be described using the three basic stereotypes, these descriptions lead to long-winded and rather cumbersome applications of these stereotypes. It would, for example, be an arduous task to create a copy of an already existing model element in another package in the model, depending on how detailed the original element has been modeled. To ease the description of certain special forms of changes to a model, UMLchange provides several stereotypes for small modifications of model elements, moving and copying elements to another namespace, changes to multiple elements, and the description of more complex changes to a model.

2.3.1. Modification and Duplication of Elements

Minor changes can be expressed by applying «edit» to a model element, which can be mapped to the substitution of old with new property values. 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. While «copy» can be mapped to the basic addition of the original element and its contents to the target namespace, «move» can simply be mapped to a substitution of the owner of the changed element.

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:

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

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.

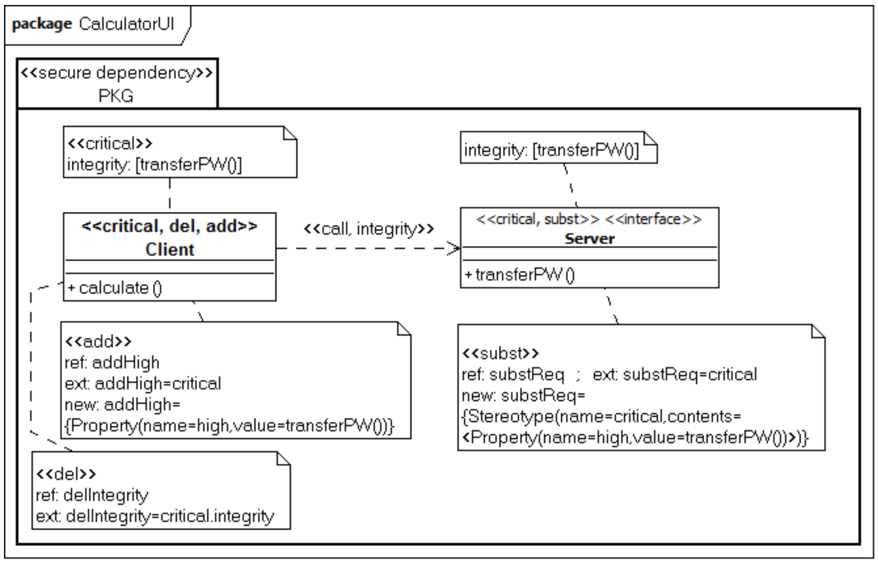

package CalculatorUI
<<secure dependency>> PKG
<<critical>> integrity: [transferPW()]
<<critical, del, add>> Client
+ calculate ()
<<call, integrity>>
integrity: [transferPW()]
<<critical, subst>> <<interface>> Server
+ transferPW ()
<<add>> ref: addHigh ext: addHigh=critical new: addHigh= {Property(name=high,value=transferPW())}
<<subst>> ref: substReq ; ext: substReq=critical new: substReq= {Stereotype(name=critical,contents= <Property(name=high,value=transferPW())>)}
<<del>> ref: delIntegrity ext: delIntegrity=critical.integrity

Figure 4: Adding, Deleting and Substituting Elements

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 5) which uses «secure-links» [13], the *Database* is to be moved to its own node *Database-Server*. As there is a «secrecy» requirement at the dependency between the *Webserver* and the *Database*, the «Internet» link has to be substituted with an «encrypted» connection. Additionally, the *adversary* attacking the system is edited to be the *custom* attacker defined in [18]. «Internet» links would be vulnerable to this attacker, but since the link in question has already been substituted with an «encrypted» connection, this change does not invalidate security of the model.

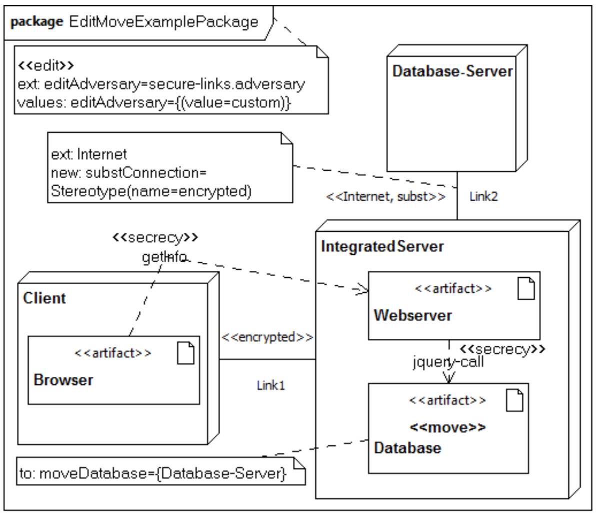

package EditMoveExamplePackage
<<edit>> ext: editAdversary=secure-links.adversary values: editAdversary={(value=custom)}
Database-Server
ext: Internet new: substConnection= Stereotype(name=encrypted)
<<Internet, subst>>
Link2
<<secrecy>> getInfo
IntegratedServer
<<artifact>> Webserver
Client
<<artifact>> Browser
<<encrypted>>
Link1
<<secrecy>> jquery-call
<<artifact>> <<move>> Database
to: moveDatabase={Database-Server}

Figure 5: Editing and Moving Elements

2.3.2. Describing Changes to Multiple Elements

Describing changes to expansive models can be hard to manage. If, for example, all applications of a certain stereotype in the model are to be removed, finding each and every application of it can be tedious task. UMLchange provides stereotypes to ease the process of describing changes to multiple elements. The changes can be focused on certain types of model elements or elements having a certain property.

«del-all», «add-all» and «subst-all» are applied to the namespace in which the changes are to take place and can be mapped to multiple applications of the basic stereotypes. The {new} tag has the same syntax and semantics as the {new} tag of the respective basic stereotypes, while the {pattern} tag allows to identify the model elements in the namespace affected by the described change. The format of {pattern} is :

$$ChangeID = TargetedElementsPattern$$

The tag uses the same syntax as the {new} tag of the basic stereotypes. 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. 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 6) based on a smart card life-cycle, the stereotype «locked-status» is to be added to the *TERMINATED* state. Applying this stereotype adds the constraint that the marked state may not have any outgoing transitions. To fulfill this requirement, «del-all» is applied to the region containing the state machine. Its application, named *removeOutgoing*, describes the removal of all outgoing transitions of the state *TERMINATED*, securing the model once again.

2.3.3. Describing Complex Changes

Describing complex changes with the UMLchange grammar can lead to long-winded grammar expressions. To provide a simpler method for modeling complex changes, UMLchange provides the ability to reference changes modeled 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 modeled in the namespace by modeling 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 modeled references to the original model. Complex additions using «old» can be mapped to additions to the original elements in the 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. Applications of «keep» can be mapped to the addition of the kept elements to the substitutes.

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 should not 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 tag uses the same syntax as the {new} tag of the basic stereotypes (see also Section 2.4). 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 an old contained element of substituted 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 7), the dependency between the *Algorithm interface* and the *Calculator UI* receives both «call» and «high». The *Calculator UI* would violate the *secure dependency* property, but the UI class is substituted with the *Secured Calculator UI*, which adds a host of other attributes and operations to the previously relatively small class and provides security by using the appropriate «critical» application.

2.4. The UMLchange 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:

$\{Class(name = NewClass),$

$Class(name = OtherNewClass, visibility = private)\},$

$\{@addClasses\}$

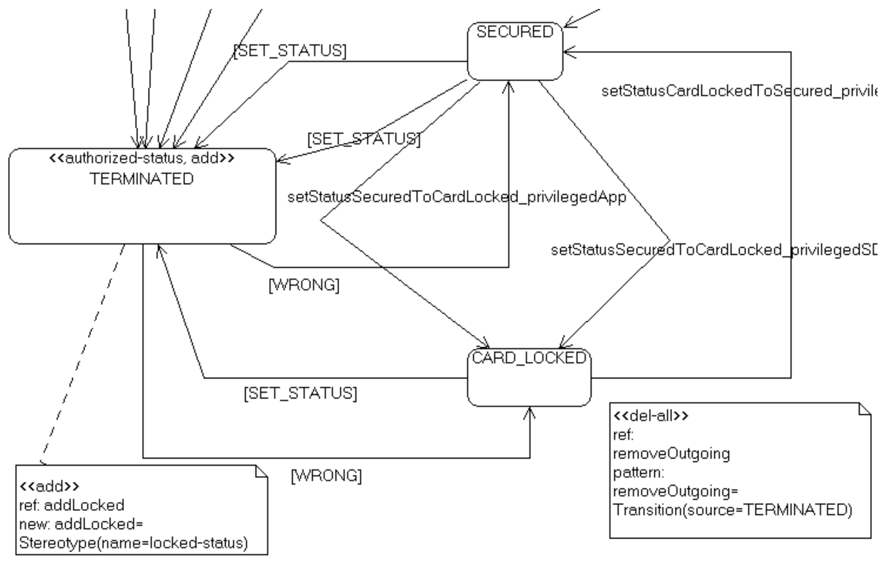

SECURED
[SET_STATUS]
setStatusCardLockedToSecured_privil
[SET_STATUS]
<<authorized-status, add>>
TERMINATED
setStatusSecuredToCardLocked_privilegedApp
setStatusSecuredToCardLocked_privilegedSD
[WRONG]
CARD_LOCKED
[SET_STATUS]
<<del-all>>
ref:
removeOutgoing
pattern:
removeOutgoing=
Transition(source=TERMINATED)
[WRONG]
<<add>>
ref: addLocked
new: addLocked=
Stereotype(name=locked-status)

Figure 6: Changing multiple elements with UMLchange

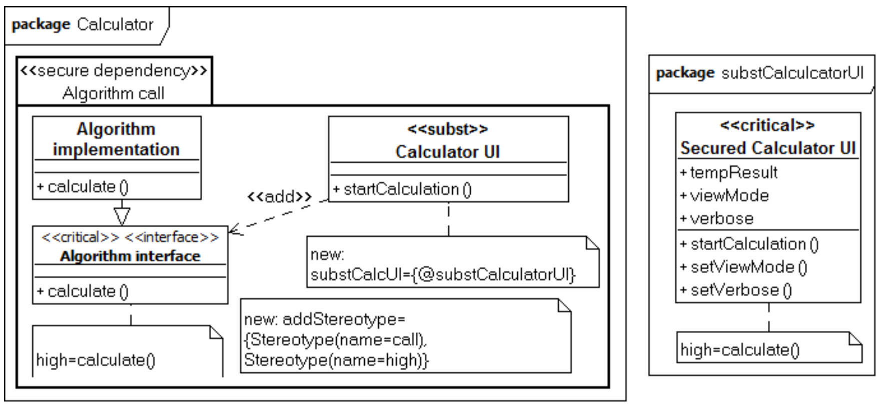


Figure 7: Describing Complex Changes with Namespaces

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».

2.4.1. Simple Element Descriptions

Simple element descriptions (SED) succinctly describe a UML model element. The format of a 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. Assuming there are two different classes of the same name *WantedClass* in two different packages *SuperPackage* and *SubPackage*, referencing a class would need the containing package namespace 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.

Table 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	Enum	public, private, protected or package
Association	sourceEndKind, targetEndKind	Enum	composite, shared or none
	source, target	Reference	qualified classifier
Dependency	supplier, client	Reference	qualified classifier

Table 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.

2.4.2. 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 2.3.3).

2.4.3. 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).

The tag {values} of stereotype «edit» uses the same key-value pairs to describe changes to model element

Stereotype	Tag	Subset	Example
«edit»	values	KeyValuePairs	(name=NewName, visibility=private)
«copy», «move»	to		
«copy», «move»	to	Qualified-Namespace	SomePackage:: SubPackage:: TargetClass
«del-all», «add-all», «subst-all»	pattern	SimpleElement-Description	Class(name=Some Class,contents= ⟨Stereotype(name= UMLsec::critical⟩))
«keep»	adopter		

Table 2: Other Uses of the UMLchange Grammar

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}.

2.5. UMLseCh vs. UMLchange

Although UMLchange originates from UMLseCh, differences between the two profiles exist. While UMLseCh allowed multiple applications of the same stereotype to the same model element, this is not allowed in UML 2.x. Hence, to make UMLchange compliant to UML, the stereotypes and tagged values have been adopted to enable the definition of multiple changes to a single model element.

With UMLseCh, it was very hard to describe more complex alternative evolutions which can now be easily modeled using the namespaces and simple grammar expressions. «keep» and «old» have been added to provide additional control when making complex changes to models.

The stereotypes «copy», «move» and «edit» also provide easier methods to make both minor changes and to duplicate model structures, while the stereotypes for changing multiple elements have been reworked to both allow a more precise pattern-matching.

While UMLseCh conceptually allowed to change UML extensions, UMLchange provides the technical realization of these concepts, which prompted the change in syntax to certain tag values, i.e. the UMLchange grammar. The grammar further provides the additional ability to describe alternative evolutions to a model, as well as describing multiple independent changes of a type to the same model element, e.g. the independent addition of two stereotypes to a class.

To provide a formal foundation for the UMLchange profile, the verification of evolutions described with the profile is discussed in the following section.

3. Verification Strategy

The evolution information annotated with UMLchange stereotypes to the model elements is parsed into an internal representation. Based on this representation we can check whether an evolution path preserves the consistency of the security requirements, which is described in what follows.

3.1. Incremental verification

As stated in the previous sections, all changes can be expressed with the basic change stereotypes (see Section 2.2). All other change stereotypes could be mapped to their basic counterparts. Hence, evolving a model means that we *add*, *delete*, or *substitute* elements of this model. A set of these changes is called a Delta. 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 class or deleting a dependency). In general there exist evolutions from model A to model 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*. A DeltaFactory could be used to create all possible permutations of changes that can be applied simultaneously without violating basic modeling principles of the UML.

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. 8 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.

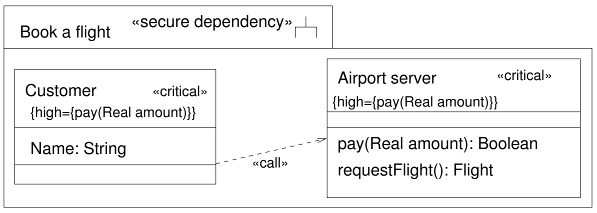


Figure 8: Class Diagram Annotated with «secure dependency»

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:

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

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

$$\nexists\, \mathsf{o}, \mathsf{o}'(\mathsf{o} \in \mathbf{Add} \lor \mathsf{o} \in \mathbf{Del}) \land ((\mathsf{o}, \mathsf{o}') \in \mathbf{Subs} \lor (\mathsf{o}', \mathsf{o}) \in \mathbf{Subs})$$

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 $\mathsf{o} \in \mathbf{Del}$, 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:

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

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 UMLchange 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.

3.2. Compositional verification

Incremental verification after change is a useful technique if only relatively small parts of the model need to be re-verified. In general, this is more challenging for security properties on behavioral models where small changes may impact the semantics in a non trivial way. To deal with this problem, a less local perspective of the system is useful: if the system model is built up on interacting components then the effects of changes to single components to the overall system security can be established if there exist a compositionality result for the security property under consideration. In other words, instead of re-verifying the complete system when modification is made to a small number of components, it suffices to re-verify the modified components only and the apply the compositionality result. In the following we summarize two such compositionality results that can be applied in the context of evolution and security.

3.2.1. Dolev-Yao secrecy

Communicating processes can be specified as sequence diagrams, and it is possible to reason about the security of the communication by means of a semantic model based on process composition and a cryptographic DSL (as it is done for example in UMLsec). In [27] a sound decision procedure is presented that given proof artifacts for Dolev-Yao secrecy on separate components can establish whether their composition will be security preserving or not. The proof artifacts considered are *dependency trees* that represent the information needed by an adversary to obtain secret messages and keys. Those trees can be calculated separately for each process, and to decide on the security of a given composition the respective trees are merged. This merging process is empirically more efficient than the complete re-verification of the composition, as we will discuss in Sect. 5.3.

3.2.2. Non-interference

Non-interference is an information flow property that relates the inputs and outputs of groups of users (typically *high* and *low*, that is *privileged* and *unprivileged*) for all possible runs of the system. This property can be specified for instance in state diagrams where the inputs and outputs are assigned to specific groups of users. For a deterministic version of this property defined in UMLsec it holds:

Theorem 1. *Let $I = I|_H \cup I|_L$ and $O = O|_H \cup O|_L$ a partition of the input and output alphabets of $A \otimes B$. If non-interference holds for an extension of the policy in I and O to the unspecified events in $I_B \cap O_A$ in A and B, then non-interference holds on $A \otimes B$.*

In other words if both components are secure (non-interferent) their composition is also secure (for a proof see [47]). It is furthermore possible to verify that a component is non-interferent in a sound way using static analysis and unwinding techniques. This technique is a sound approximation: an exact verification of non-interference would require in general to consider all possible input and output traces, which is in turn computationally unfeasible. For instance, assume there are only two possible kinds of input events to the system: a low event L and a high event H, parametric on a 64 bit long integer (that is for instance $L(10)$, $H(2^{32})$). Then to compute all possible outputs of the model one would have to consider at least 2^{128} inputs. However practical models usually contain a much smaller amount of internal transitions (usually less than 100). This number is the main input to the static approach presented in [47], allowing for practical verification, as observed empirically on models verified using a prototypical implementation of that technique.

3.3. Exemplary Application

In this section we demonstrate the incremental verification strategy by applying it to the case of the UMLsec stereotype «secure dependency», which is defined for class diagrams. The associated constraint requires that 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 the two classes (for example {secrecy={method()}}), then the other class has the same tag for this method/attribute as well (see Fig. 8 for an example). In addition to that, the communication dependencies have to have the corresponding stereotypes applied to them. 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:

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

where M.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.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 analyze the set Δ of modifications by distinguishing cases on the evolution type (deletion, addition, substitution) and the UML type.

3.3.1. 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:

$$P(M[o, D]) \Leftrightarrow$$

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

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 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.

3.3.2. 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. In addition to that, the dependency has to be marked with the stereotypes corresponding to the used «critical» tags. 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.

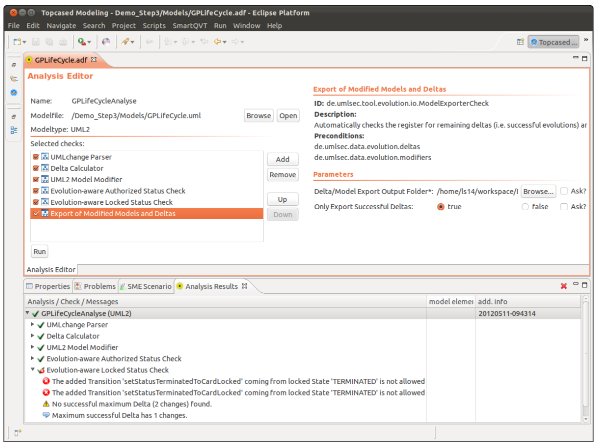
3.3.3. Substitution

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.

Tag in critical: If we substitute {requirement=method()} in class C with {requirement'=method()'}, then the same substitution must be made in every class linked to C by a dependency.

Dependency: If a «call» («send») dependency is substituted by «send» («call») then P is preserved if the substitute dependency has the same subset of the stereotypes «secrecy»,«integrity» and «high» applied to it. If this is not the case, we check for simultaneous additions that apply the required stereotypes to the new dependency.

Example. The example in Fig. 9 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.


Topcased Modeling - Demo_Step3/Models/GPLifeCycle.adf - Eclipse Platform
File Edit Navigate Search Project Scripts SmartQVT Run Window Help
Topcased ...
GPLifeCycle.adf
Add
Remove
Up
Down
Run
Browse
Open
Browse...
Properties
Problems
SME Scenario
Analysis Results

Analysis Editor

Name: GPLifeCycleAnalyse
Modelfile: /Demo_Step3/Models/GPLifeCycle.uml
Modeltype: UML2

Selected checks:
- ✓ UMLchange Parser
- ✓ Delta Calculator
- ✓ UML2 Model Modifier
- ✓ Evolution-aware Authorized Status Check
- ✓ Evolution-aware Locked Status Check
- ✓ Export of Modified Models and Deltas

Export of Modified Models and Deltas

ID: de.umlsec.tool.evolution.io.ModelExporterCheck
Description:
Automatically checks the register for remaining deltas (i.e. successful evolutions) ar
Preconditions:
de.umlsec.data.evolution.deltas
de.umlsec.data.evolution.modifiers

Parameters

Delta/Model Export Output Folder*: /home/ls14/workspace/I ☐ Ask?
Only Export Successful Deltas: [✓ true] false ☐ Ask?

Analysis Editor

Analysis / Check / Messages	model elemer	add. info
✓ GPLifeCycleAnalyse (UML2)		20120511-094314
✓ UMLchange Parser		
✓ Delta Calculator		
✓ UML2 Model Modifier		
✓ Evolution-aware Authorized Status Check		
✗ Evolution-aware Locked Status Check		
✗ The added Transition 'setStatusTerminatedToCardLocked' coming from locked State 'TERMINATED' is not allowed		
✗ The added Transition 'setStatusTerminatedToCardLocked' coming from locked State 'TERMINATED' is not allowed		
⚠ No successful maximum Delta (2 changes) found.		
Maximum successful Delta has 1 changes.		

Figure 10: The user interface of CARiSMA

Applied to the Channel is «add» describing two changes with the reference ID's add_symenc and add_asymenc specifying two possible evolution paths: merging the classes contained in the current package (Channel) with either Symmetric or Asymmetric.

Another application of «add», this time to the Client, describes the addition of either a pre-shared private key K_{sym} (ak1) or a public key K_{server} of the server (ak2). To limit the allowed evolution paths for the model, the changes have constraints associated with them indicating that the addition of the symmetric and asymmetric mechanism have to be applied simultaneously with the addition of their respective keys (AND(ak1),AND(ak2)). The simultaneous addition of both keys is also forbidden (NOT(ak2)).

The two deltas, representing two possible evolution paths induced by this notation, can be then given as input to the decision procedure described for checking «secure dependency». Both evolution paths respect sufficient conditions for the consistency of the security requirement to be satisfied.

4. Tool Support

This section briefly describes the tool environment we have implemented to verify security properties of evolving models.

4.1. The User Interface

The approach of evolution-based security analysis with the UMLchange profile has been implemented in the model analysis tool CARiSMA[2]. The tool is built on

[2]http://carisma.umlsec.de

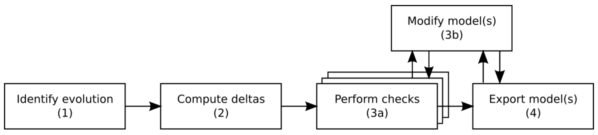


Figure 11: The pipeline for evolution analysis

the basis of Eclipse and fully integrates into the Eclipse GUI by providing different views for defining analyses and presenting results to the user.

In order to analyze a model, the user has to define the analysis to be performed. The *Analysis Editor* provides a graphical user interface for the definition of analyses (see Figure 10). The user can select the model he wants to analyze and choose one or more checks from a list of available checks. The different checks might require additional parameters. Therefore, the analysis editor displays a list of input fields for the currently selected check. The supported parameter types for a check are the primitives string, integer, float, and boolean, as well as folders and files. The analysis configuration is stored in a file which allows the user to repeat the analysis with the same configuration of parameters.

During an analysis, CARiSMA loads the model and triggers all checks listed in the analysis configuration. The results are displayed in the *Analysis Result View*. Each performed analysis is shown as a root entry while the checks are second level entries. The checks return either success or failure as the result of their execution, which is indicated by different icons in the result view. Each check entry may contain further subentries offering details about the execution of a check. Subentries may be simple status information for the user, warnings in case of precautionary findings (i.e. noteworthy, but the check can still be successful), and errors that express the reasons why the check fails.

In addition to the output presented in the analysis result view, CARiSMA generates a textual report. Each check can verbosely contribute to the report, e.g. by describing counterexamples for violated security properties or proposing improvements.

Analysis of Evolution. Apart from allowing the static analysis of models, CARiSMA provides mechanisms for analyzing evolutions to models. This is supported by certain checks that can be selected when defining an analysis.

Basically, an evolution analysis follows the pipeline illustrated in Figure 11. The steps shown have to be followed in the given order, using checks provided by CARiSMA. After identifying the changes to a model (Step 1), a second component computes the sets of

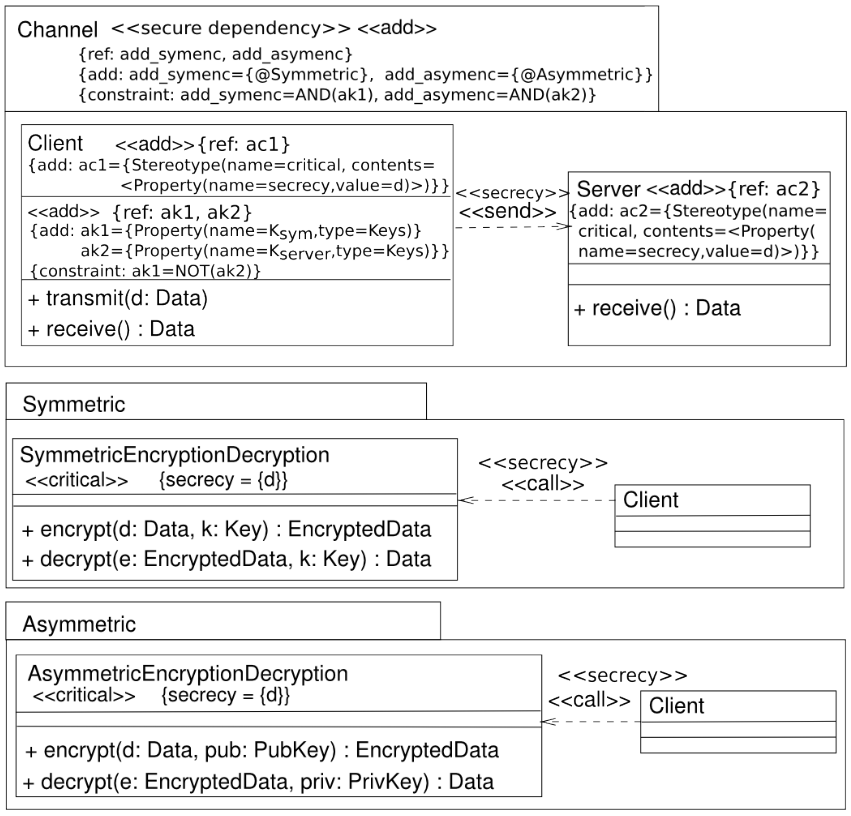

Channel <<secure dependency>> <<add>>
{ref: add_symenc, add_asymenc}
{add: add_symenc={@Symmetric}, add_asymenc={@Asymmetric}}
{constraint: add_symenc=AND(ak1), add_asymenc=AND(ak2)}
Client <<add>>{ref: ac1}
{add: ac1={Stereotype(name=critical, contents=<Property(name=secrecy,value=d)>)}}
<<add>> {ref: ak1, ak2}
{add: ak1={Property(name=Ksym,type=Keys)}
ak2={Property(name=Kserver,type=Keys)}}
{constraint: ak1=NOT(ak2)}
+ transmit(d: Data)
+ receive() : Data
<<secrecy>>
<<send>>
Server <<add>>{ref: ac2}
{add: ac2={Stereotype(name=critical, contents=<Property(name=secrecy,value=d)>)}}
+ receive() : Data
Symmetric
SymmetricEncryptionDecryption
<<critical>> {secrecy = {d}}
+ encrypt(d: Data, k: Key) : EncryptedData
+ decrypt(e: EncryptedData, k: Key) : Data
<<secrecy>>
<<call>>
Client
Asymmetric
AsymmetricEncryptionDecryption
<<critical>> {secrecy = {d}}
+ encrypt(d: Data, pub: PubKey) : EncryptedData
+ decrypt(e: EncryptedData, priv: PrivKey) : Data
<<secrecy>>
<<call>>
Client

Figure 9: An evolving class diagram with two possible evolution paths

deltas from the changes (Step 2). When analyzing model evolution with respect to security or compliance, it is possible that a subset of the changes are valid evolutions while others are not. The delta computation component computes these subsets for analysis. The actual security checks are then performed on either the deltas (Step 3a) or the modified models themselves (Step 3b). Valid deltas and the corresponding models can be persistently stored (Step 4). The process is explained in more detail in Section 4.3. CARiSMA allows the user to convert a normal analysis to an evolution analysis by providing a context menu entry for automatic conversion.

4.2. *Tool Architecture*

Like Eclipse, CARiSMA has been implemented as a plugin based architecture. Using the modularity provided by this method, CARiSMA is distributed as plugins, of which the core plugin includes the main functionality. Furthermore, CARiSMA offers extension points facilitating the contribution of functionality of other plugins to provide different checks. CARiSMA can be started standalone as an RCP application or within an existing Eclipse. It is also possible to smoothly integrate it into existing modeling tools such as TOPCASED[3], IBM Rational Software Architect[4] or other tools based on Eclipse.

CARiSMA is kept as model-type independent as possible. The framework and the core components are model-type independent, while the checks are mostly model type specific. Checking security properties specified with UMLsec stereotypes is of course bound to UML. However, it is also possible to realize model-type independent checks. For instance, we offer an OCL check plugin that can read arbitrary constraints from a catalog file and evaluate them on the structural properties of a model regardless of its type.

Model access is provided via the Eclipse Modeling Framework (EMF) [19], which implements, among other tools, the OMG Meta Object Facility (MOF) specification [20]. It provides the basis for the metamodel implementation and allows us to work with models of arbitrary types.

[3]`http://www.topcased.org/`

[4]`http://www.ibm.com/software/awdtools/swarchitect/`

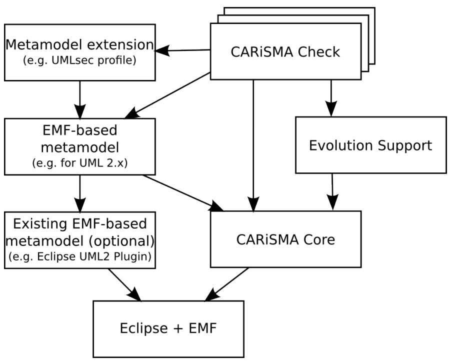


Figure 12: Overview of the CARiSMA architecture

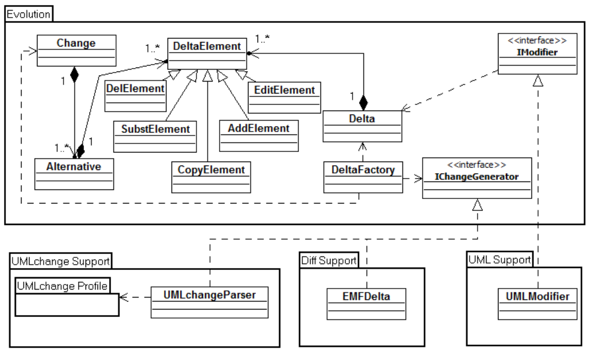


Figure 13: Components for evolution-based analysis

Figure 12 visualizes the overall architecture of CARiSMA. The CARiSMA core contains the graphical user interface and the routines for performing analyses, i.e. managing preferences, executing checks, etc. In order to enable the analysis of models of certain types, a metamodel wrapper plugin has to provide the EMF-based metamodel and register it at the CARiSMA core. The metamodel wrapper can further provide convenience functions for working with models of that type, e.g. for getting all elements of a certain type. The metamodel wrapper can be based on an existing metamodel implementation such as the Eclipse UML2 Plugin in case of UML models. Currently, we offer metamodel wrappers for UML2 and BPMN2. On top of the core and the metamodel wrappers, different plugins can be realized that contain the checks performed during model analysis. Some plugins, e.g. those checking UMLsec properties, might require further extensions of the metamodels such as the UMLsec profile offering security-related stereotypes. The evolution support will be explained separately in Section 4.3.

The CARiSMA core provides a blackboard architecture that allows checks to exchange data. Required data can be specified as a pre-condition in the metadata of a check. Data provided by a check can be specified as a post-condition.

Future Contributions to CARiSMA. Users can easily contribute additional checks to CARiSMA. An extension point provides ability to define meta information of the check such as name, parameters, or conditions. The implementation of a checks is basically a simple Java class realizing the interface `CarismaCheck` provided by CARiSMA. The interface specifies the `perform` operations which will contain the actual model analysis of a check. The parameter values entered by the user are passed to that operation at runtime. The check also receives a reference to the framework to access the analyzed model and the blackboard as well as to generate output. For the latter, CARiSMA offers operations to append text to the analysis report and to create result entries. The report itself and the graphical components cannot be accessed by a check directly. The perform operation has to return a boolean expressing whether the check was successful.

Support for additional modeling languages can be added by creating a corresponding metamodel wrapper plugin. It has to provide an EMF-based metamodel implementation or import it from some existing plugin and register it at the CARiSMA core.

4.3. Evolution Support

The components that provide the support for analyzing evolving models are shown in Figure 13.

Evolutions to a model are reflected by collections of change descriptions. A *change* consists of one or more exclusive alternative evolutions to a part of the model. These alternatives can then be broken down to one or more *delta elements*, which describe atomic changes to model elements. More specific, we support the description of the addition, deletion, substitution, editing and copying of model elements. The content of an alternative has to be applied in a single evolutionary step.

Changes can be constrained using the three constraint types AND, NOT and REQ. While AND forces another change to be applied simultaneously, meaning during the same evolutionary step of the model, NOT excludes the referenced change from being applied to the model in the same evolution. Finally, REQ ensures that the referenced change is applied to the model before the constrained change takes place. This is to make sure that dependent changes are applied in the correct order.

CARiSMA provides an interface to generate change descriptions (see Figure 11, Step 1). Two implementations of this interface are incorporated into CARiSMA: the UMLchangeParser, which creates the change descriptions from UMLchange applications on the model [21], and the EMFDelta, which generates the change descriptions by computing the difference between the original and the evolved model using EMF Compare [22].

A *delta* is a set of the above mentioned delta elements. The elements in the set are the atomic changes that amount to a possible evolution to the model. The *delta factory* (Step 2) receives a collection of change descriptions and processes the change constraints and alternatives, thereby generating all possible permutations over the alternatives, including the empty set, while following the constraints imposed by the changes. The elements of each valid change permutation are stored as a delta.

A delta can be applied to the model using implementations of the IModifier interface (Step 3b). An existing implementation is the UMLModifier. The UMLModifier receives the original model, creates a copy of it and applies each atomic change described by a delta element to the model copy. The resulting model is stored for use in CARiSMA evolution-aware checks (Step 3a). To avoid unnecessary computation of model modifications, the models are lazily initialized only when a check needs the modified model for validation. Modifications are mainly executed via generic EMF methods, while the selection of UML-specific aspects are handled using a custom logic. It would therefore be easy to build other model-specific modifier implementations.

The existing CARiSMA evolution checks each receive the list of computed deltas and iterate over each delta while checking the validity of the delta's modifications. If any change leads a model violating the checked security property, the corresponding delta is removed from the list to prevent subsequent checks from unnecessarily checking this delta again.

The evolved models which passed all checks can then be persistently stored using the ModelExporter (Step 4), which is also able to store their corresponding deltas in an XML format alongside the models.

5. Evaluation

The UMLchange profile and the approach of delta-based security analysis have been successfully evaluated in different case studies.

5.1. Evolution Analysis of a Smartcard Specification

Beside adopting our approach to different security properties of UMLsec, we have also considered domain-specific properties. In context of the EU project SecureChange[5] we have developed new stereotypes for behavioral models, namely statecharts. In the *Global Platform* smart card specification [23], we can identify two properties: 1.) "For any execution, whenever the card is put in the TERMINATED state by means of a set status issued by a privileged application, then it should not be possible to revert to another state", and 2.) "It should not be possible for an application that doesn't have the Card Terminate privilege to switch the card life cycle state to Terminated, whether via a SET STATUS command (if the application is a SD) or the invocation to the GPSystem.terminateCard() method". These properties have been developed into the new stereotypes «locked status» and «authorized status». Furthermore, we have implemented the delta-based verification of these properties and successfully applied that to the evolution of the changes of the smartcard specification from version 2.1.1 to version 2.2 [24].

5.2. Usability & Applicability in Industrial Practice

In order to evaluate the usability and applicability of UMLchange, we have organized a half-day workshop at an industrial partner which is a leading company for smart card solutions. The participants of the workshop have been technical experts with longtime experiences in UML and security modeling. 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. 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 the tool and to further evaluate the tool as "homework".

The first impression of the practitioners was that UMLchange 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. The major problem is, that many modeling tools have only limited support for stereotypes and tagged values from the usability perspective. Especially, the UMLchange grammar for describing new elements was seen as problematic, as typographic errors can arise, e.g. when entering qualified names of referred model elements. To tackle this problem, one could implement an auto-completion

[5]`https://www.securechange.eu`

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.

Nonetheless, the practitioners rated the tool to be applicable in daily practice, if UML is thoroughly used for modeling and if the usability of the tool implementation was improved to meet industrial standards. The ability to evaluate potential evolution paths and alternative solutions at the same time was liked. The fact that the modeling of arbitrary many models to just find the valuable evolution path became dispensable, was seen as major argument for our approach, although the direct editing of the models would be easier. A thorough review of the basic concepts of UMLchange and the CARiSMA tool is contained in [25].

5.3. Runtime Efficiency

Another validation activity focused on the run-time behavior of a sound delta-based verification (as discussed in Sect. 3) compared to a complete re-verification of the changed model. The duration of the check for «secure dependency» implemented in the UMLsec tool behaves in a more than linear way depending on the number of dependencies. Table 3 shows a comparison between the running time of the verification[6] 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 modeling 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 [26]). For instance, the evaluation for evolving models marked with the «secure links» stereotype [18] showed that analyzing a model using the delta of changes is significantly faster than re-analyzing the modified model (see Table 4).

As discussed in Sect. 3, the incremental analysis is appropriated when small parts of the model need to be re-verified. In general, this is more challenging for security properties on behavioral models where small changes of the model may impact the semantics in a non trivial way. Nevertheless, this problem can be tackled by taking a less local perspective of the system: if the system model is built up on interacting components then the effects of changes to single components to the overall system security can be established if there exist a compositionality result for the security property under consideration. For instance, we have conducted experiments to measure the time of the composition compared to the overall re-verification run-time as depicted in Table 5 for the Dolev-Yao secrecy result we discussed in Sect. 3. Again, a significant time gain results from reusing information of previously verified components of the model. Since the compositionality result guarantees the security of the composition, a complete re-verification is not required.

# dependencies	Run time Δ-verification (in s)	Run time re-verification (in s)
10	0.3	1
30	0.5	6
50	0.8	51
70	1	310

Table 3: Running time comparison for verification of *secure dependency*

# changes	Run time Δ-verification (in ms)	Run time re-verification (in ms)
10	0,41	1,93
20	0,43	0,93
30	0,67	2,71
50	1,12	5,18
100	3,30	20,02
500	63,8	448,93
1000	269,64	2054,15

Table 4: Running time with UMLchange vs. re-verification for *secure links*

# 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 5: Composition vs. full verification of secrecy

6. Related Work

Model-based development is nowadays a well accepted paradigm. The formal semantics of models allow us to verify them with regard to certain properties such as security requirements. Hence, there has so far been work on tools for analyzing security properties on models. Most approaches are dedicated to specific problems. Basin et al. have presented a tool for

[6]On a 2.26 GHz dual core processor

the analysis of access control properties in UML models [2]. Siveroni et al. developed a tool which performs static verification on models composed of UML class and state machine diagrams [3]. The UMLsec tool [13], which can be seen as a predecessor of CARiSMA, combines the analysis of various security properties in a single tool. However, these tools did not target the situation of model evolution as in the case of CARiSMA and the UMLchange approach. In addition, the Eclipse-based architecture of CARiSMA enables a smooth integration into the secure development processes with existing modeling tools.

There are different approaches to deal with evolution that are related to our work. Within *Software Evolution Approaches*, [28] derives several *laws of software evolution* such as “Continuing Change” and “Declining Quality”. [29] 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. Similarly [30] discusses the verification of consistency through refinement, and [31] discusses consistency of models for incremental changes of models. These works can be integrated with the approach presented in this paper to enhance the detection of inconsistencies and conflicts. Our approach, however, focuses on the evolution of software systems while preserving security properties. [32] 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.

In the context of *Requirements Engineering for Secure Evolution* there exists some recent work on requirements engineering for secure systems evolution such as [33]. 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. [34] 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. [35] 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). Again, this approach is not security specific. On a similar fashion, but more focused on critical properties, [36] also discusses the evolution of Architectures.

The UMLchange notation was informally introduced in [37], however, there it was called *UMLseCh* and tightly bound to UMLsec. Note that UMLchange does not aim to be an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [6, 7, 8, 9, 10]) or model transformation languages such as QVT [38] or ATL [39]. The latter offers the possibility to express model transformations rules for horizontal and vertical transformations from any source metamodel to any target metamodel. Since ATL was initially developed to answer the QVT RFC issued by the OMG (Object Management Group), it shares common requirements [40] with QVT. Some efforts to have graphical notations for transformation languages include UMLX [41], a graphical model transformation language that was also developed to answer the QVP RFP [42] issued by the OMG. MOLA (MOdeling transformation LAnguage) [43] is another graphical language for model transformations. The Epsilon framework offers both model analysis and transformation capabilities for EMF-based models [44]. However, security analysis is not in the focus of these frameworks. It will be interesting future work to demonstrate how the security results presented in this paper can be used in the context of those approaches, that is, whether the mentioned transformation approaches can be used to extract the relevant delta information which can then be plugged-in in the security preservation decision procedures. In particular, the Epsilon approach sounds promising since it is based on EMF and should be compatible to our tool implementation.

Tools for difference computation and analysis are discussed in [45]. It is obvious that one could integrate model differencing with security analysis. In this case, one would have two models as input and computes the delta in order to verify it. Currently, we are evaluating this idea. First results can be found in [21]. Very interesting in that area is also the approach presented in [46], which defines a profile for describing changes. However, that paper focuses on configuration management aspects and model merging. Security aspects or quality properties in general are not considered.

7. Conclusion

This paper focuses on the preservation of security properties of models given different evolution scenarios. We considered selected classes of model evolutions such as addition, deletion, and substitution of model elements based on UMLsec models. Assuming that the starting UMLsec diagrams are secure, which one can verify using UMLsec checks (e.g. implemented in CARiSMA), our goal is to re-use these existing verification results to minimize the effort for the security verification of the evolved UMLsec models. 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 modeling activity whether the modeled 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. The approach contains a novel UML profile, called *UMLchange*, which allows modelers to specify possible evolution paths of a model and to verify them with respect to quality properties such as consistency of security requirements. We demonstrated this approach at the hand of the UMLsec stereotype «secure dependency». Applications for the consistency of other UMLsec security requirements such as «secure links» can be found in [26]. The evolution of behavioral security properties of UMLsec is challenging to verify incrementally. Changes can be also specified with UMLchange for those properties, but the verification technique is different: compositionality leads to more suitable results (see [27] for Dolev-Yao, or [47] for non-interference).

The approach has been implemented in the analysis tool CARiSMA. It can parse the UMLchange stereotypes applied to model elements, compute all possible evolution paths, and verify whether they preserve the consistency of the security requirements. The industrial validation has shown that the approach enables secure model evolution. Evolving systems can be developed by pointing out possible security-violating modifications of previously secure models. We also showed 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. So far we have researched the different classes of evolution and change types and successfully applied our approach to security properties. Beside supporting further security properties, it would be interesting to prove in more detail whether our approach is sufficient to handle other kinds of quality properties beyond security properties. Another direction of research is alternative source of evolution information. Instead of annotating models with UMLchange stereotypes, one could just compare two versions of a model to gain evolution information. We have recently started to implement such an approach [22] based on the model comparison framework EMFcompare [48]. However, such a solution is not as powerful as the approach presented in this paper. Model comparison is limited to two models and it has a higher resource consumption, which reduces the runtime advantages of evolution-aware security checks compared to re-verification of entire models. A detailed analysis of the drawbacks and an evaluation whether the difference-based approach is sufficient is part of ongoing and future work.

References

[1] J. Jürjens, Y. Yu, Tools for model-based security engineering: Models vs. code, in: 22nd International Conference on Automated Software Engineering (ASE 2007), 2007.

[2] D. A. Basin, M. Clavel, J. Doser, M. Egea, Automated analysis of security-design models, Information & Software Technology 51 (5) (2009) 815–831.

[3] I. Siveroni, A. Zisman, G. Spanoudakis, A UML-based static verification framework for security, Requir. Eng. 15 (1) (2010) 95–118.

[4] J. Jürjens, Principles for secure systems design, Ph.D. thesis, Oxford University Computing Laboratory (2002).

[5] J. Jürjens, P. Shabalin, Tools for secure systems development with UML, Intern. Journal on Software Tools for Technology Transfer 9 (5–6) (2007) 527–544, invited submission to the special issue for FASE 2004/05.

[6] R. Heckel, Compositional verification of reactive systems specified by graph transformation, in: E. Astesiano (Ed.), Proceedings of international conference on Fundamental Approaches to Software Engineering (FASE), Vol. 1382 of LNCS, Springer, 1998, pp. 138–153.

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

[8] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow, Model transformations? transformation models!, in: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), Springer, 2006, pp. 440–453.

[9] A. Rensink, Á. Schmidt, D. Varró, Model checking graph transformations: A comparison of two approaches, in: Proceedings of the International Conference in Graph Transformation (ICGT), Springer, 2004, pp. 226–241.

[10] D. S. Kolovos, R. F. Paige, F. Polack, L. M. Rose, Update transformations in the small with the epsilon wizard language, Journal of Object Technology 6 (9) (2007) 53–69.

[11] J. Jürjens, L. Marchal, M. Ochoa, H. Schmidt, Incremental Security Verification for Evolving UMLsec models, in: Proc. of the 7th European Conference on Modelling Foundations and Applications, Birmingham, UK (ECMFA'11), 2011, pp. 52–68.

[12] J. Jürjens, Model-based security engineering with UML, in: A. Aldini, R. Gorrieri, F. Martinelli (Eds.), FOSAD, Vol. 3655 of Lecture Notes in Computer Science, Springer, 2004, pp. 42–77.

[13] J. Jürjens, Secure Systems Development with UML, 1st Edition, Springer, 2005.
URL `http://amazon.com/o/ASIN/3540007016/`

[14] UMLsec group, UMLsec Tool Suite, http://www.umlsec.de (2001-2012).

[15] J. Jürjens, Sound methods and effective tools for model-based security engineering with UML, in: G.-C. Roman, W. G. Griswold, B. Nuseibeh (Eds.), Proceedings of the International Conference on Software Engineering (ICSE), ACM Press, 2005, pp. 322–331.

[16] S. Höhn, J. Jürjens, Rubacon: automated support for model-based compliance engineering, in: Robby (Ed.), 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, ACM, 2008, pp. 875–878.

[17] M. Broy, K. Stølen, Specification and Development of Interactive Systems, 2001.

[18] D. Warzecha, Entwurf und Realisierung einer Komponente zur modellbasierten Sicherheitsanalyse von Softwareevolution für Netzwerkarchitekturen, Bachelor Thesis, TU Dortmund, Germany (in German) (2011).

[19] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd Edition, Addison-Wesley, 2009.

[20] Object Management Group, Meta Object Facility Core Specification (MOF) 2.0 (formal 06-01-01), http://www.omg.org/docs/formal/02-04-03.pdf (January 2006).

[21] SecureChange, Deliverable 4.3, tool support for evolution-aware security checks and monitor generation, http://securechange.eu/content/deliverables (2012).

[22] J. Kowald, Differenzberechnung zur Unterstützung modellbasierter Sicherheitsanalyse von Softwareevolution, Bachelor Thesis, TU Dortmund, Germany (in German) (2011).

[23] Globalplatform card specification version 2.2, http://www.globalplatform.org/specificationscard.asp (March 2006).

[24] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jürjens, P. Yousefi, Model-based security verification and testing for smart-cards, in: ARES 2011, 6-th Int. Conf. on Availability, Reliability and Security, Vienna, Austria, 2011.

[25] SecureChange, Deliverable 1.3, http://securechange.eu/content/deliverables (2012).

[26] Secure Change Project, Deliverable 4.2, available as http://www-jj.cs.tu-dortmund.de/jj/deliverable_4_2.pdf.

[27] M. Ochoa, J. Jürjens, D. Warzecha, A sound decision procedure for the compositionality of secrecy, in: 4th International Symposium on Engineering Secure Software and Systems (ESSOS 2012), Lecture Notes in Computer Science, Springer, 2012.

[28] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and Laws of Software Evolution – The Nineties View, in: METRICS'97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 20–32.

[29] T. Mens, T. D'Hondt, Automating support for software evolution in UML, Automated Software Engineering Journal 7 (1) (2000) 39–59.

[30] A. Egyed, Consistent adaptation and evolution of class diagrams during refinement, in: FASE'04, Vol. 2984 of LNCS, Springer, 2004, pp. 37–53.

[31] S. Johann, A. Egyed, Instant and incremental transformation of models, in: Proceedings of the International Conference on Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, 2004, pp. 362–365.

[32] M. E. Shin, H. Gomaa, Software requirements and architecture modeling for evolving non-secure applications into secure applications, Science of Computer Programming 66 (1) (2007) 60–70. doi:http://dx.doi.org/10.1016/j.scico.2006.10.009.

[33] T. T. Tun, Y. Yu, C. B. Haley, B. Nuseibeh, Model-based argument analysis for evolving security requirements, in: SSIRI'10, IEEE Computer Society, 2010, pp. 88–97.

[34] D. Mellado, J. Rodriguez, E. Fernandez-Medina, M. Piattini, Automated Support for Security Requirements Engineering in Software Product Line Domain Engineering, in: AReS'09, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 224–231.

[35] D. Garlan, J. Barnes, B. Schmerl, O. Celiku, Evolution styles: Foundations and tool support for software architecture evolution, in: WICSA/ECSA 2009, 2009, pp. 131 –140. doi:10.1109/WICSA.2009.5290799.

[36] T. Mens, J. Magee, B. Rumpe, Evolving Software Architecture Descriptions of Critical Systems, Computer 43 (5) (2010) 42 – 48. doi:10.1109/MC.2010.136.

[37] J. Jürjens, M. Ochoa, H. Schmidt, L. Marchal, S. Houmb, S. Islam, Modelling secure systems evolution: Abstract and concrete change specifications (invited lecture), in: I. Bernardo (Ed.), 11th School on Formal Methods (SFM 2011), Bertinoro (Italy) 13-18 June 2011, LNCS, Springer, 2011.

[38] Object Management Group, QVT Query/View/Transformation Specification, http://www.omg.org/spec/QVT/1.0/PDF/ (2005).

[39] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Sci. Comput. Program. 72 (1-2) (2008) 31–39.

[40] F. Jouault, I. Kurtev, On the architectural alignment of ATL and QVT, in: Proceedings of the Symposium on Applied Computing (SAC), ACM, New York, NY, USA, 2006, pp. 1188–1195.

[41] E. D. Willink, On challenges for a graphical transformation notation and the UMLX approach, Electronic Notes in Theoretical Computer Science 211 (2008) 171–179.

[42] OMG, MOF 2.0 query/views/transformations RFP (2002).

[43] A. Kalnins, J. Barzdins, E. Celms, Model transformation language MOLA, in: Proceedings of the International Conference on Model-Driven Architecture: Foundations and Applications (MDAFA), Vol. 3599 of LNCS, Springer, 2004, pp. 62–76.

[44] D. S. Kolovos, R. F. Paige, F. A. Polack, Epsilon development tools for eclipse, in: Eclipse Modeling Symposium, Eclipse Summit Europe, 2006.

[45] D. S. Kolovos, D. D. Ruscio, R. F. Paige, A. Pierantonio, Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing, in: Proc. 2nd Workshop on Comparison and Versioning of Software Models (CVSM'09), IEEE Computer Society, 2009.

[46] P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, Conflicts as first-class entities: A uml profile for model versioning, in: Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers, Lecture Notes in Computer Science Volume 6627, Springer, 2011, pp. 184–193.

[47] M. Ochoa, J. Jürjens, J. Cuellar, Non-interference on uml statecharts, in: 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe 2012), Lecture Notes in Computer Science, Springer, 2012.

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