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)

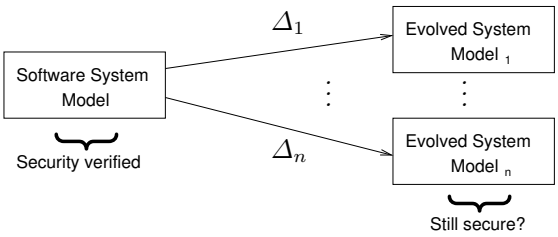

Fig. 1. Model verification problem for n possible evolution paths

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.

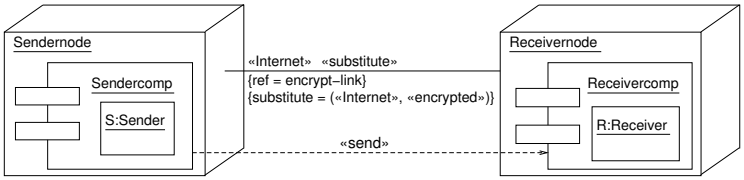

Fig. 4. Example of stereotype substitute

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

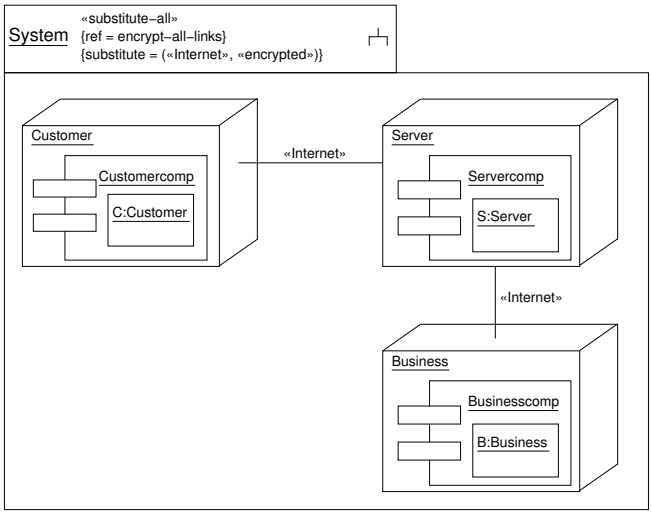

Fig. 5. Example of stereotype substitute-all

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 ».

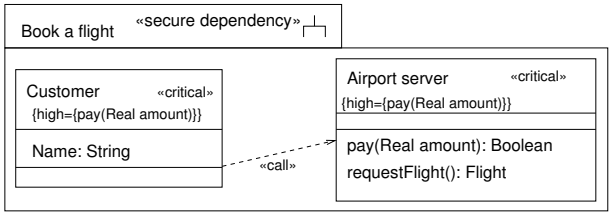

Fig. 6. Class Diagram Annotated with « secure dependency »

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:

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

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

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

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:

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

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.

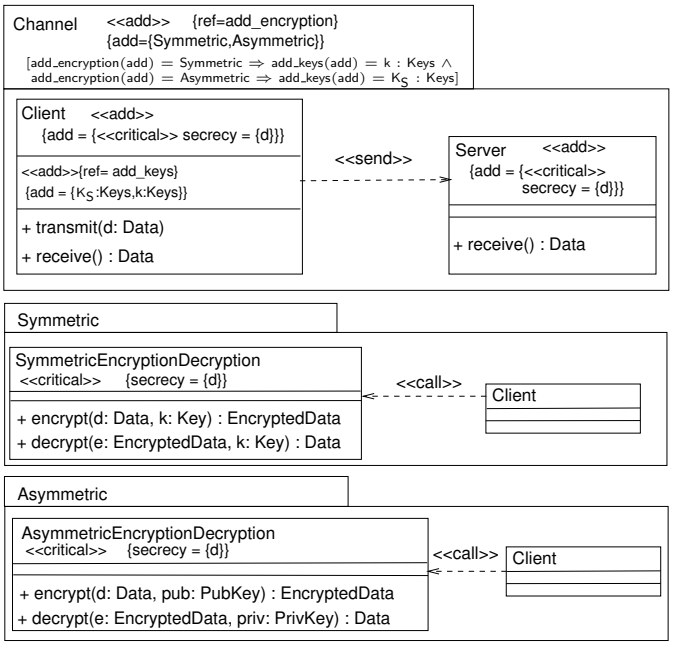

Fig. 7. An evolving class diagram with two possible evolution paths

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

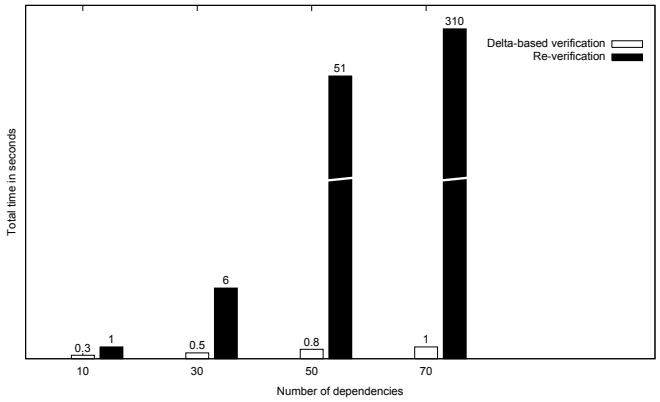

Fig. 8. Running time comparison of the verification

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.