D4.2 FORMALLY FOUNDED AUTOMATED SECURITY ANALYSIS

Jan Jürjens (TUD), Martín Ochoa (TUD), Loïc Marchal (TUD), Marc Peschke (TUD), Arnaud Fontaine (INR), Olivier Delande (THA), Edith Felix (THA), Fabrice Bouquet (INR), Elizabeta Fourneter(INR), Frederic Dadeau (INR), Alessandra Tedeschi (DBL)

Document information

Document Number	D4.2
Document Title	Formally founded automated security analysis
Version	1.3
Status	Final
Work Package	WP 4
Deliverable Type	Report and Prototype
Contractual Date of Delivery	31 January 2011
Actual Date of Delivery	15 January 2011
Responsible Unit	TUD
Contributors	INR, THA, TUD, DBL
Keyword List	Model-based verification, Security, Evolution
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	15.06.2010	Draft	Martín Ochoa, Jan Jürjens (TUD)	Outline of the deliverable
0.2	07.09.2010	Draft	Marc Peschke (TUD)	Integration into First draft
0.3	02.10.2010	Draft	Martín Ochoa (TUD)	Added Chapter 3
0.4	12.10.2010	Draft	Jan Jürjens (TUD)	Review, comments
0.5	28.10.2010	Draft	Marc Peschke (TUD)	Chapter 7
0.6	04.11.2010	Draft	Martín Ochoa (TUD)	Update on Chapters
0.7	08.11.2010	Draft	Jan Jürjens (TUD)	Review, comments
0.8	10.11.2010	Draft	Martín Ochoa (TUD), Fabrice Bouquet (INR), Elizabeta Fourneter(INR), Frederic Dadeau (INR)	Chapter 5
0.8	14.11.2010	Draft	Martín Ochoa (TUD), Arnaud Fontaine (INR-LI)	Chapter 4
0.9	21.11.2010	Draft	Marc Peschke (TUD)	General update
1.0	22.12.2010	Draft	Edith Felix (THA) Olivier Delande (THA)	Chapter 6
1.1	01.12.2010	Draft	Martín Ochoa (TUD) Jan Jürjens (TUD) Marc Peschke (TUD)	General update
1.2	20.12.2010	Draft	Martín Ochoa (TUD) Jan Jürjens (TUD) Marc Peschke (TUD)	General update based on the internal Scientific Review
1.3	6.1.2011	Draft	Martín Ochoa (TUD) Jan Jürjens (TUD) Marc Peschke (TUD) Karmel Bekoutou (UNITN)	General update based on the quality check

Executive summary

Deliverable 4.1 [51] presented a notation that allows one to specify multiple possible evolution paths for UML Diagrams. The notation is called UMLseCh and is a further extension of the UMLsec profile [19]. This document specifies a formal foundation for this notation that aims at automatic (re)-verification of security annotated diagrams after evolution. To achieve this, we give a more precise definition of the UMLseCh semantics itself, which allows us pinning down what we mean by 'evolution' from a model M to an evolved M'. As a result of this, given an UMLseCh diagram we can extract one or more *deltas* Δ_i containing the model elements to be added, substituted or deleted from/to the original diagram.

These modifications to an original diagram M have two main consequences: they may alter the consistency of the diagram from the purely UML syntactical point of view, but more importantly *they may alter the security properties of* M. We discuss the first problem to some degree, but we focus on the latter. For this, we present sound decision procedures for different security properties that allow to establish whether a given Δ preserves them or not.

Moreover, we report on the implementation of these algorithms as plugins for the existing UMLsec Tool Suite. This allows us an automatic verification of UMLseCh annotated Diagrams drawn with the ArgoUML tool. Metrics of the efficiency gain of this implementation as opposed to trivial re-verification are presented.

As an application exercise, we model some fragments of the Global Platform (POPS case study) and verify the preservation of selected [50] security properties under evolution. Some of these fragments are used to integrate our approach with other Work Packages. We report on this integration links as summarized in the following.

D4.2 in the Project Timeline

This deliverable contains mainly results related to Task T4.2 '*Provide formal foundation for evolving security extension*' (M6-M18) and partially T4.3 '*Extend existing security analysis tools with evolving security*' (M18-M30). D4.2 corresponds to the milestone of M24 for Work Package 4 '*Providing a formal foundation for the evolving security extension including links for the code-level verification in WP6 and WP7*' and is therefore located between months 12 and 24 in the General Project Timeline 1. It contains results prominently on:

- *Models*: The application of the techniques developed in WP4 to the POPS Case Study began in D4.1. In this Derivable we model other fragments of the Global Platform (Chapter 3), partially in a joint effort with other Work Packages (WP6 and WP7, as stated in the Milestone for M24, Chapters 4,5). The ATM Case Study is also partially modelled with UMLseCh (Chapter 6).
- *Languages*: Formal foundations for the UMLseCh notation are defined in Chapter 2.
- *Tools*: D4.2 has the double nature Report/Prototype. In Chapter 7 we summarize the results of the current status of the tool implementation effort.

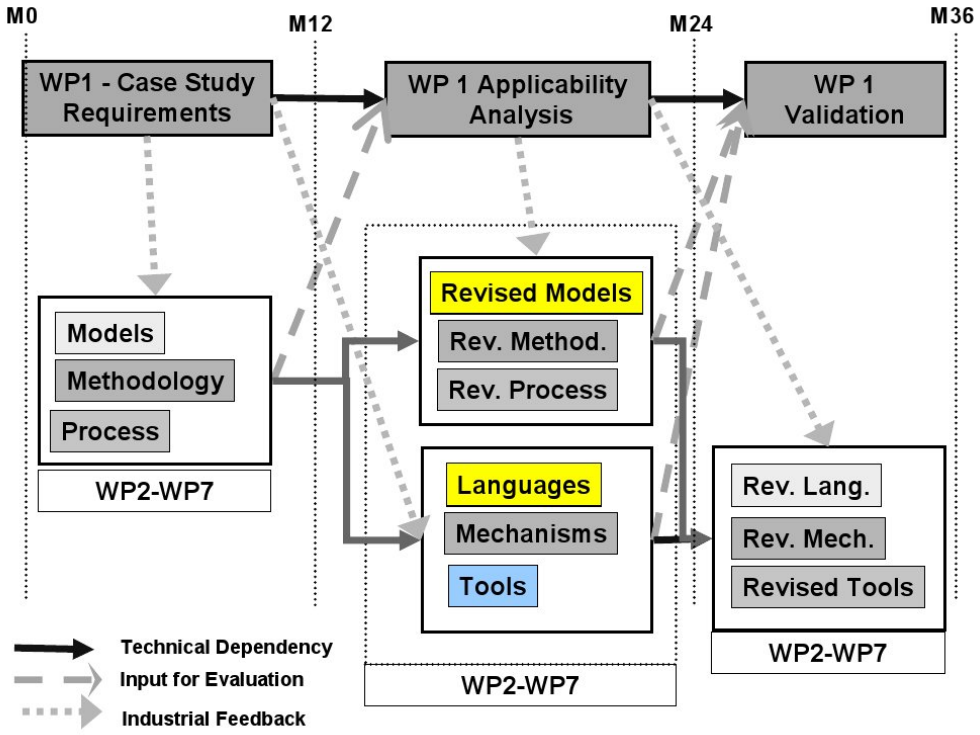


Figure 1: SecureChange project timeline

Validation

Deliverable 1.2 (due to Work Package 1) defines General Scientific and Industrial validation criteria, aimed at a general validation of the Project due to M36. The artefacts contained in this Deliverable that are subject to this validation are:

Industrial:

- The Fragment of the Global Platform modelled with UMLseCh in Chapters 3,4,5 and the reasoning techniques under evolution applied to these models. These analysis include both the '*Software update*' and the '*Specification Evolution*' requirements and specifically the properties '*Information protection*' and '*Life-cycle consistency*' as defined in [50].
- The fragment of the ATM as described in Chapter 6. The general requirement considered is '*Organizational Level Change*' and the properties considered are '*Information Access*' and '*Information Protection*'.

Scientific

- The UMLseCh notation. This falls within the category 'Modeling Languages' of the General Criteria in D1.2.
- The decision procedures for security properties in evolving scenarios as defined in Chapter 3. This corresponds to the 'Algorithms' category of the General Criteria.
- The tool support for the UMLseCh notation and the algorithms developed for the analysis of evolution.

Some preliminary discussion about the Scientific validation of these artefacts is given in Appendix A. A full list of the artefacts subject to validation will be given in M36 (D4.3).

Integration

Figure 2 summarizes the integration links among the different Work Packages within the Project. The Change requirements and Security properties in the following refer to the definition in [50].

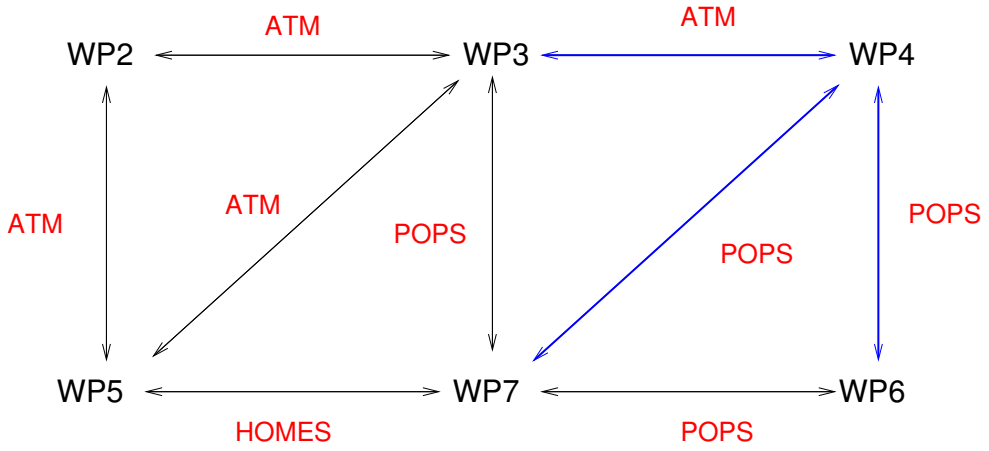


Figure 2: Integration links between work packages

This deliverable contains the following links:

WP4-WP3 Chapter 6 contains this integration link presenting a connection between the modeling and verification techniques developed by WP4 with WP3 (Requirements) based on the ATM Case Study. A risk analysis done with the Thales Security DSML gives high-level security requirements, which are reflected in the System Design and analyzed by means of the UMLseCh approach. The general requirement considered is '*Organizational Level Change*' and the properties considered are '*Information Access*' and '*Information Protection*'.

WP4-WP6 This integration link, presented in Chapter 4 describes how the result of the verification process at the model level can be used to push constraints to the verification at the code level, based on the POPS case study for a GP specific property and secure information flow. The general requirement considered is '*Software update*' and the common property is '*Information protection*'.

WP4-WP7 Based on the Global Platform life-cycle (POPS), this link (Chapter 5) shows how model-based testing for evolving systems can benefit from the techniques developed in WP4. The general requirement considered is '*Specification Evolution*' and the common property is '*Life-cycle consistency*'.

Index

1 Introduction

During Year 1 (Deliverable 4.1 [51]), Work Package 4 introduced a notation that extends the UMLsec profile for describing model evolutions: UMLseCh. UMLseCh design models can be used for change exploration and decision support when considering how to integrate new or additional security functions and to explore the security implications of planned system evolution. To maintain the security properties of a system through change, the change can be explicitly expressed such that its implications can be analyzed a priori. This notation was tailored to allow an automatic analysis deciding on the preservation of security properties in the evolved models, as defined in Task T4.2 '*Provide formal foundation for evolving security extension*', which is one of the goals of this deliverable. To achieve this objective, we rely on the formal foundations (abstract syntax and behavioral semantics) of UMLsec and give a more precise formulation of the syntax and semantics of UMLseCh, already introduced in D4.1.

We then describe sound decision procedures that help to the determine the security preservation of the evolution possibilities described with the UMLseCh notation. Since the assumption is that the original UMLsec model is *secure* (it conforms to the annotated security requirements), the verification techniques proposed aim at re-using as much as possible the already existing verification information. Notice that one could trivially re-run the security analysis done to establish the security of the original model on the evolved model to decide on the preservation. This would result in general in high resource consumption for large systems. Intuitively, if the cost associated to the verification of a security property depends on a number n of model elements, but the evolution affects only a number $m \leq n$ of security relevant elements, it is more efficient to check only the modified parts if the security property is local enough. For complex diagrams with hundreds of model elements, it can actually happen that $m \ll n$.

The application of these techniques is mainly focused on the POPS case study, where several fragments are considered. In particular, this case study is used to draw integration links with Work Packages 6 and 7, where we apply the modeling and verification techniques to concrete properties of the Case Study as specified in [50]. The integration work done with WP6 gives a link with code-level verification, whereas we support the model-based testing approach of WP7 by checking the correctness of these models against security properties using the UMLseCh approach. A fragment of the ATM case study is also considered to show how the techniques developed in WP4 can be linked with WP3.

The UMLseCh modeling notation and the verification techniques should be implementable as an extension to the current UMLsec Tool Suite, as defined in task T4.3 '*Extend existing security analysis tools with evolving security*' [1]. In this document we give details about this implementation effort, including links to the binaries and screen-cast of demonstrations.

[1] An ongoing task started on Month 18 and to be completed within Month 30.

Chapters Walk-through Chapter 2 recalls the notation defined in D4.1 (with some minor modifications) and defines its semantics based on the UML Abstract syntax of UMLsec [19]. It also discusses the issue of model *correctness* after evolution. Chapter 3 presents the techniques proposed to reason soundly about security preservation under evolution. This is done for several UMLsec stereotypes, and in Section 3.2 the application of the notation and the verification techniques to a fragment of the Global Platform are presented. The link between WP4 and WP6 (Model-based and Code-based verification), based on the POPS Case Study ('Software Update') is presented in Chapter 4. Chapter 5 describes the integration work done between WP4 and WP7: verification techniques for evolving models are applied to model-based testing, also based on POPS ('Specification Evolution'). Chapter 6 presents the link between WP4 and WP3, using the Thales Security DSML approach and based on the ATM Case Study. Finally, Chapter 7 reports on the implementation of the UMLseCh verification techniques in the context of the UMLsec Tool Suite.

Acknowledgements We would like to thank Johannes Kowald, Gregor Kotainy, Yousefi Parvaneh and Daniel Warzecha, students of the TU Dortmund, for their contribution to the UMLseCh plugins and to Chapter 3 of this Deliverable. We also warmly thank Holger Schmidt, post-doc at the TU Dortmund for his help in the plug-in implementation effort. Special thanks to Federica Paci (UNITN), Frank Innerhofer-Oberperfler (UIB) and Nguyen Quang-Huy (GTO) for their comments on earlier versions of this document.

2 A Formal Foundation for UMLseCh

In D4.1 we introduced the UMLseCh notation up to some formality degree ([51]). In this chapter we give more formal details about its semantics, based on the abstract syntax of UML as defined in [19]. Some minor changes on the notation have been made with respect to the one defined in Year 1. For example, the question of complex evolutions is addressed with more detail in Section 2.1.4. We treat here only the 'concrete' notation of D4.1, since the 'abstract' notation was meant only to superficially annotate changes and had no semantics. Therefore, only the concrete notation is suitable for the verification under evolution tasks. In Section 2.5 we discuss how the changes defined in UMLseCh affect model *consistency*.

2.1 The UMLseCh Extension

2.1.1 The Profile

As it is specified in the Catalog of UML Profile Specifications [1], a UML profile does one or more of the following:

- Identifies a subset of the UML metamodel.
- Specifies "well-formedness rules" beyond those specified by the identified subset of the UML metamodel.
- Specifies "standard elements" beyond those specified by the identified subset of the UML metamodel.
- Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML metamodel.
- Specifies common model elements, expressed in terms of the profile.

This Section, together with the Sections 2.1.2, 2.1.5 and 2.5, define the UMLseCh profile, following the structure described above.

The UMLseCh profile concerns all of UML. Figure 2.1 shows the list of stereotypes, together with their tags and constraints. These stereotypes do not have parents. Figure 2.2 shows the corresponding tags. The tag ref is a DataTag and the tags substitute, add and delete are all ReferenceTags. Indeed, as it will be described in the following sections, the UMLseCh tagged values associated to these three tags are model elements, but their role is to describe possible future model elements that do not exist in the model yet. UMLseCh models **possible future changes**, thus theoretically, the substitutive or additive model elements do not exist on the model yet, but only as an attribute value

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

Figure 2.1: UMLseCh stereotypes

inside a change stereotype[1]. However, at the concrete level, i.e. in a tool, this value is either the model element itself if it can be represented with sequence of characters, or a namespace containing the model element. This could be considered as a DataTag, provided that model elements and namespaces containing model elements are considered as data. However, the name of a namespace is a reference to the namespace itself. In addition, assuming that a string-based model element notation used in the tagged values of UMLseCh represent a reference to the model element that it describes, it can then be considered as a ReferenceTag. For example, the stereotype *« Internet »* used as the value of a tag substitute represents a reference to the actual stereotype, and not the stereotype itself. UMLseCh tags are thus all considered as ReferenceTags (except the tag ref). Figure 2.1 and Figure 2.2 both follow the notation used in [19] for the UMLsec profile definition[2]. As for UMLsec, the concepts of UMLseCh can be used at both the type and the instance level. However, for simplicity reasons, the examples and description in the following will only apply to the instance level. A complete description of the UMLseCh stereotypes and their associated tags is given in the following sections. Although UMLseCh could be used alone as an evolution modeling language, it is specifically intended to model the evolution in a security oriented context (in particular, it does not aim to be an alternative for any existing general-purpose evolution specification or model transformation approaches, but in fact the results presented in this deliverable could be used in the context of those approaches). It is thus an extension of UMLsec and requires the UMLsec profile as prerequisite profile. The diagram representing the UMLseCh profile is shown in Figure 2.3.

[1]The type change represents a type of stereotype that included *« change »*,*« substitute »*, *« add »* or *« delete »*.

[2]Although the UMLsec profile was written following a previous version of UML, the UMLseCh profile follows the same notation since it still respects the current specification of UML, defined in [44].

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

Figure 2.2: UMLseCh tags

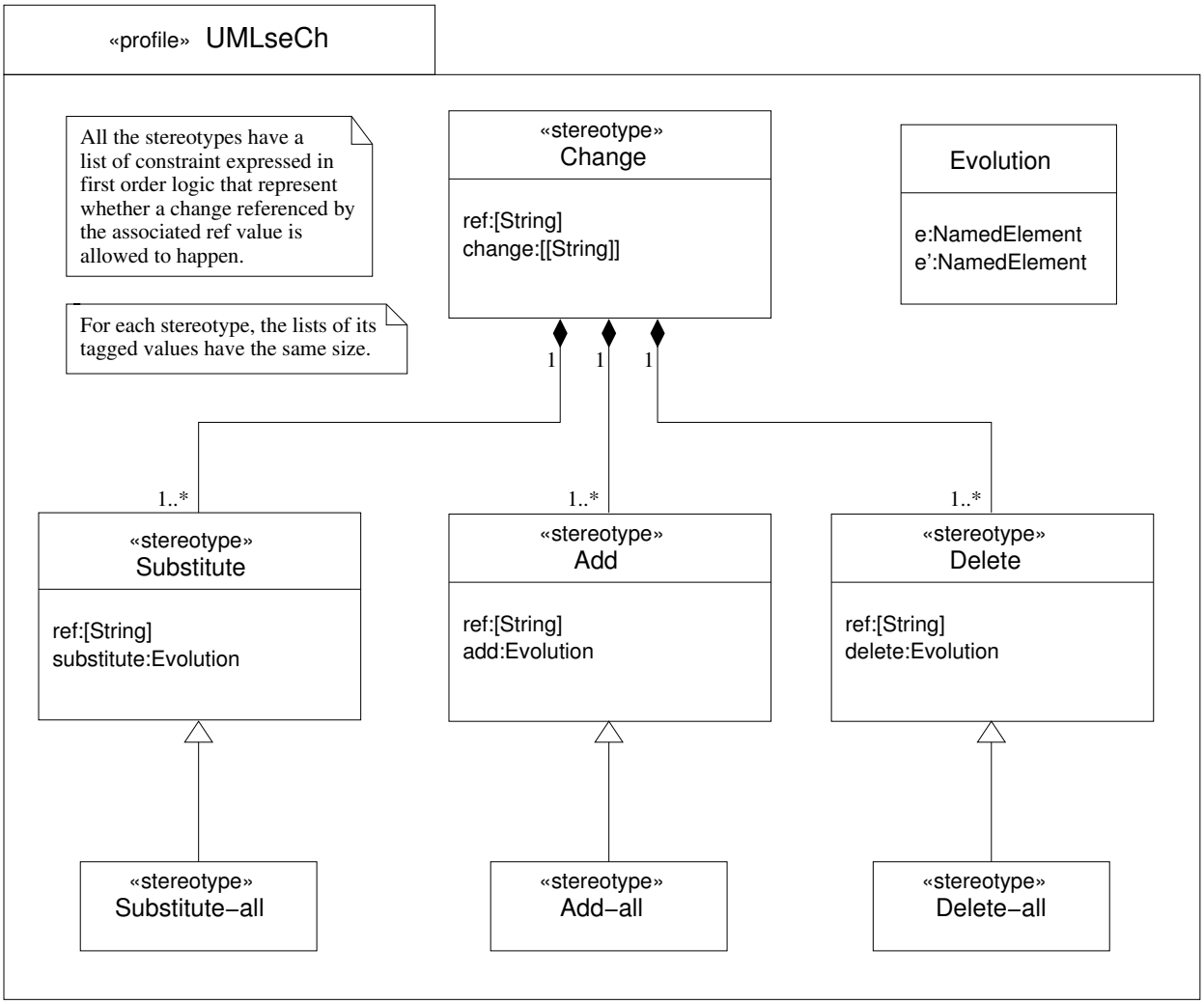


Figure 2.3: UMLseCh profile

2.1.2 Description of the Notation

2.1.2.1 *« 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
{ substitute = (ELEMENT$_1$, NEW$_1$), ... , (ELEMENT$_n$, 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. This usage of the values of tags ref will be explained further in this section. 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. More than one occurrence of the same e in the list is allowed [3]. However, two occurrences of the same pair (e, e') cannot exist in the list, since it would model the same change twice. For the notation of this list, two possibilities exist. An element of the pair is written as the model element itself if it can fit in the tag notation, i.e. if it is based on characters. It is for example the case for a stereotype, which would result to the notation *{ substitute = (« stereotype »,e') }*. On the other hand, if the model element cannot fit in the tag notation (it is the case for example with a class, a state or a component), it is placed in a namespace and the name of this namespace is the element of the pair contained in the list used as tagged value. The *namespace notation* allows UMLseCh stereotypes to graphically model more complex changes, but requires a particular behavior that will be described in Section 2.1.3. Examples will also illustrate such situations further in this chapter. The element e of a pair (e, e') representing a substitution is optional; if the model element that has to be substituted is clearly identified by the syntactic notation, i.e. if there is no possible ambiguity to state which model element is concerned by the change modeled by the stereotype *« substitute »*, the element e can be omitted. On the contrary, if that model element is not clearly identifiable, it must be used. More precisely, when 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. When the model element concerned by the substitution is a sub-element of the one to which the stereotype is attached, the element e is necessary [4]. In the case where the element e is omitted, the value of the list appears as the element e' in the tagged value, instead of the pair. Note that this is just a syntactic sugar. More precisely, in formal representations required for applying changes, the substitutions of the list of the tag substitute will always be pairs (e, e'). 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

[3] UMLseCh aims to model the **possible** changes that **could** occur, not one actual change that will happen sooner or later.

[4] The reason why the stereotype would not be attached to the sub-element itself, other than because it improves the graphical visibility and readability, is that the abstract syntax of UMLseCh, defined in Section 2.5, does not allow the sub-element to have stereotypes.

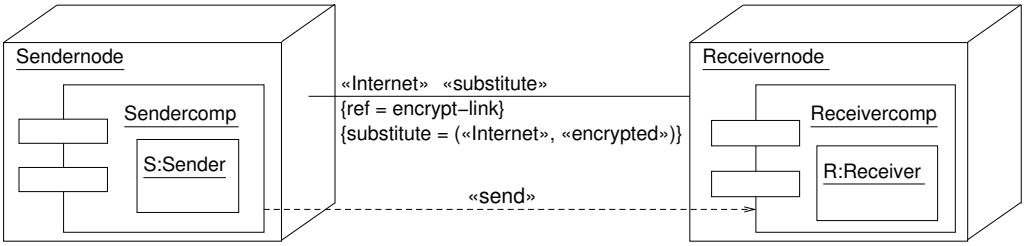


Figure 2.4: Example of stereotype *« substitute »*

of UMLseCh, defined in Section 2.5. In the case when the abstract syntax of UMLseCh is used, the expression is placed in a comment with the value of the list of the tag ref associated to the change. This comment is then attached to the concerned stereotype. If the change happens, it is also important that it leaves the resulting model in a consistent state. Therefore, to avoid any unwanted results, the values of both the elements of the pair representing the substitution must be of the same type. If the element e of the pair (e, e') is omitted, e' must be of the same type as the model element to which *« substitute »* is attached. This offers a limited protection as it only ensures that the UML models will remain correct from a syntactic point of view, but does not guarantee a consistent semantics. For example, it ensures that a method of a class will not be substituted by an attribute, leaving the diagram in an inconsistent syntactic state. However, it does not stop one from modeling the substitution of a stereotype *« critical »* attached to a class by a stereotype *« Internet »*, although this is not permitted by the UMLsec Profile definition. More rules to ensure diagrams consistency will be given in the following. To show how to use the UMLseCh notation, the following example can be considered. Assume that we want to specify the change of a link stereotyped *« Internet »* so that it is now stereotyped *« encrypted »*. For this, the following:

« substitute »

{ ref = encrypt-link }

{ substitute = (« encrypted », « Internet ») }

is attached to the link concerned by the change. Such an example is shown in Figure 2.4.

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 [(ref$_1$, CONDITION$_1$), . . . , (ref$_n$, CONDITION$_n$)], $\forall n \in \mathbb{N}$, where, $\forall i : 1 \leq i \leq n$, ref$_i$ is a value of the list of a tag ref and 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. It basically represents whether or not the change is allowed to happen (i.e. if the condition is evaluated to true, the change is allowed, otherwise the change is not allowed). 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. Formally, the predicate should be *"x occurred"* or $P(x)$, assuming that $P(x)$ is the predicate *"x occurred"* and where *x*

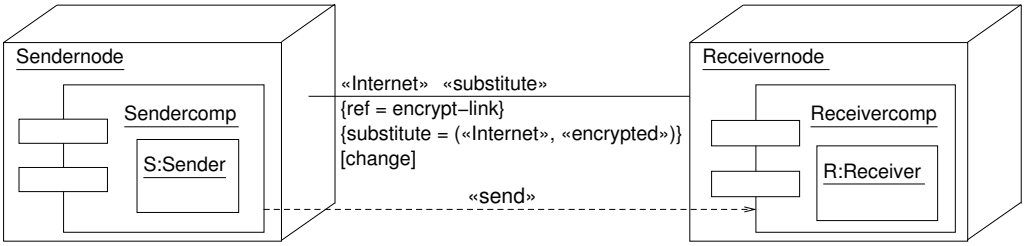


Figure 2.5: Example of a constraint of a stereotype

is one of the values of the tag ref. However, this value of the list of the tag ref, say *x*, is used as a syntactic sugar for the atomic predicate $P(x)$, where $P(x)$ is the predicate *"x occurred"*[5]. Again, if the stereotype models only one change, the condition can be shown alone on the diagram and the pair notation can be omitted. 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 following can then be attached to the link concerned by the change:

« substitute »

{ ref = encrypt-link }

{ substitute = (« encrypted », « Internet ») }

[change]

This example is shown in Figure 2.5. To express 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:

$$[\text{change}_1 \wedge \text{change}_2]$$

is added to the stereotype *« substitute »* modeling the change. If only one of both the changes is requested, but not necessarily both of them, the constraint :

$$[\text{change}_1 \vee \text{change}_2]$$

is added to the stereotype.

2.1.2.2 *« add »*

The stereotype *« add »* is similar to the stereotype *« substitute »* but, as its name indicates, denotes the addition of new model elements. It has two associated tags, namely ref and

[5] A value of the tag ref could also be considered as an atomic proposition, also called propositional variable. However, the option of an atomic predicate has been chosen because predicates can also represent sets, which can also be expressed by a function. From a high level of abstraction, a function seems easier to represent the predicate than having to keep as many propositional variables and their truth value as there are values of tags ref. It will especially be useful later in the UMLseCh abstract syntax where the function representing the predicate "ref *occurred*" will be defined.

add. The tag ref has the same meaning as in the case of the stereotype *« substitute »*. The tag add is equivalent to the tag substitute of a stereotype *« substitute »* but with the semantic of an addition. Its value is thus a list pairs of model elements, each pair representing an addition. The model elements of the pairs can either be represented as a sequence of characters and be represented directly in the tagged values or the name of a *namespace* if the additive model element is a complex model element. Again, complex additive elements will require a particular behavior that will be described in Section 2.1.4. The element e of a pair (e, e') has a slightly different meaning for a stereotype *« add »*. While with the stereotype *« substitute »*, this element represents the model element to substitute, with the stereotype *« add »* it represents the model element concerned by the addition. That can be explained easily. With a substitution, a model element is substituted by another model element of the same type. The model element to substitute hence is present on the model when the substitution takes place and can either be expressed in the pair representing the change or be the model element to which the stereotype is attached. With an addition, no element is being substituted and the stereotype *« add »* cannot be associated to a model element that does not exist yet. Instead, the model element to which the stereotype *« add »* is attached or the model element e of the pair (e, e') is the "super-element" of the element being added. For example, considering all types of UML diagrams, a class is super-element of a method or an attribute of that class, a subsystem is a super-element of a class and a stereotype can be a sub-element of a class, a link, a dependency or any other model element that can have stereotypes. Again when the super-element to which the element is added is the element to which the stereotype is attached, the element e of the pair representing additions, as for the stereotype *« substitute »*, can be omitted.

The stereotype *« add »* is a syntactic sugar of the stereotype *« substitute »*, as a stereotype *« add »* could always be represented with a stereotype *« substitute »*. Indeed, from an abstract point of view, adding a new model element consist of substituting the empty model element $\emptyset$ by the new model element. More concretely, since the stereotype *« add »* could not be attached to the empty model element (and the empty model element could not really be used in a pair of a tag add either), adding a new model element consists of replacing the set of model elements concerned by the addition by a new set containing all the elements of the previous set plus the new model element. Formally, if s is the set of model element and e the new model element, the new set is $s' = s \cup \{e\}$. An addition as a substitution would then consists of substituting s by s'. This particularity will be visible in the formal representation of UMLseCh, described in the following.

As for the stereotype *« substitute »*, the application of a change modelled by a stereotype *« add »* must leave the resulting model in a consistent state. It is thus also necessary to have values of the same type for both the elements of a pair (e, e') representing an addition or, if e is omitted, for e' and the elements of the set to which it is added. However, adding a new element might bring more difficulties and consistency problems than with a substitution, especially with dynamic structure diagrams such as activity or state diagrams. This comes from the fact that adding new elements might change the base structure of the diagram or the relations between the elements of the diagram while a substitution just change one element by another of the same type. The problems that can arise from adding a new model element as well as the possible workarounds will be

presented further in this chapter. Rules defining diagrams consistency will also be given.

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

2.1.2.3 *« delete »*

The stereotype *« delete »* is similar to the stereotypes *« substitute »* and *« add »* but, obviously, denotes the deletion of a model element. It has two associated tags, namely ref and delete, which have a similar meaning as in the case of the stereotypes *« substitute »* and *« add »*, i.e. a list of reference names and the list of model element to delete respectively. Note that here, the elements of the list used as value of the tag delete are not shown as pairs, since it just represents the model element to delete. If the list is empty, because the element to delete is the element to which the stereotype *« delete »* is attached and this stereotype models only one possible deletion, the tag delete can be omitted. On the other hand, if the stereotype *« delete »* models more than one deletion and the element to which the stereotype is attached is concerned by the change, this element must be shown in the list of the tag add. This difference from the stereotypes *« substitute »* and *« add »* ensure that the list of the tag add will always have the same size as the list of the tag ref.

As for the stereotype *« add »*, the stereotype *« delete »* is a syntactic sugar of the stereotype *« substitute »*. Indeed, it could always be represented with a stereotype *« substitute »* since deleting a model element could be expressed as the substitution of the model element by the empty model element $\emptyset$. It could also be seen as the substitution of the set containing the model element to delete by a new set that is a copy of the initial set without the element to delete. As opposed to the stereotype *« add »*, the stereotype *« delete »* could, if used as *« substitute »*, replace directly the concerned model element by $\emptyset$, since it would be attached to the model element to delete or the latter would be expressed in the pair representing the deletion. Deleting a model element might also bring similar consistency problems as in the case of an addition. As for the stereotype *« add »*, these problems and the possible workarounds will be developed further in this chapter and rules defining diagrams consistency will be given in Chapter 2.5.

The stereotype *« delete »* also has a constraint formulated in first order logic, which represents the same information as for the stereotypes *« substitute »* and *« add »*.

2.1.2.4 *« 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 and what are the possible changes. 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 *{substitute = (ELEMENT$_1$, NEW$_1$), . . . , (ELEMENT$_n$, NEW$_n$)}*. The tags ref has the exact 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

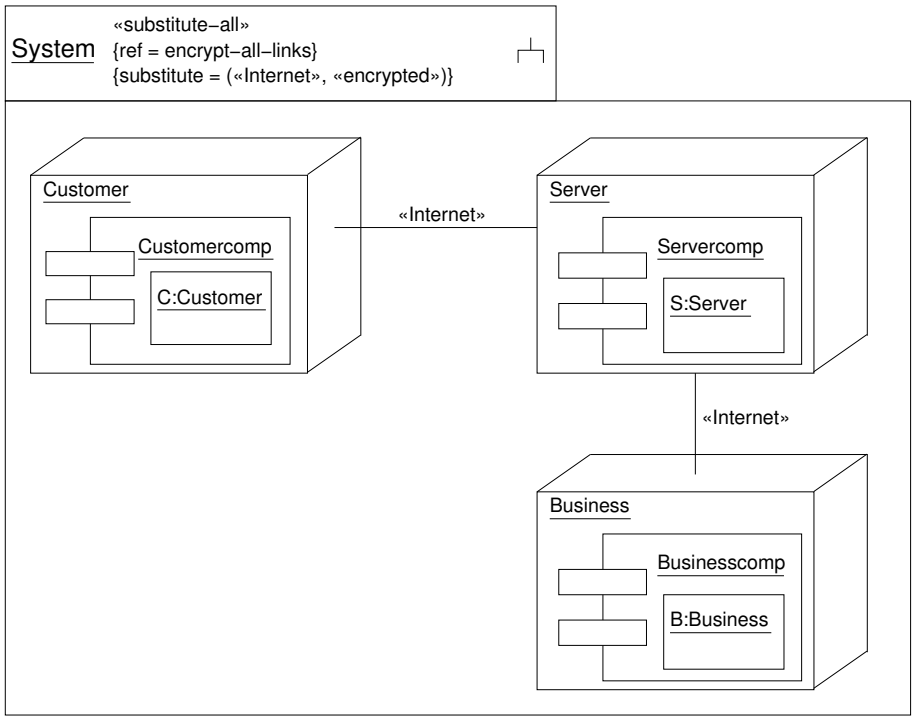


Figure 2.6: Example of stereotype *« substitute-all »*

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 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. To replace, for example, all the links stereotyped *« Internet »* of a subsystem so that they are now stereotyped *« encrypted »*, the following can be attached to the subsystem:

« substitute-all »

{ ref = encrypt-all-links }

{ substitute = (« Internet », « encrypted ») }

This example is shown in Figure 2.6.

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:

« substitute-all »

{ ref = secrecy-to-integrity }

{ substitute = ({ secrecy = X }, { integrity = X }) }

can be attached to the subsystem containing the class diagram.

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 »*.

2.1.2.5 *« add-all »*

The stereotype *« add »* also has its extension *« add-all »*, which extends the stereotype *« add »* in the same way as *« substitute-all »* extends the stereotype *« substitute »*.

2.1.2.6 *« delete-all »*

The stereotype *« delete »* also has its extension *« delete-all »*.

2.1.2.7 *« 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 exact same meaning as in the case of a stereotype *« substitute »*. The tag change, here, 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[6] 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*.

An example of the use of a stereotype *« change »* will be given in Section 2.1.4 where the use of complex additive elements will be described.

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

2.1.3 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, described in Section 2.1.5.2. 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, which will be detailed by means of examples further in this section.

To define the behavior of a complex substitution, we need to differentiate two types of model elements. The first type includes the model elements that *connect* together two other model elements. We call these elements *connectors*. For example, messages, transitions, links or dependencies are *connectors*. Concretely, a connector has the two connected model elements and additional properties, such as a name, stereotypes or boolean conditions. In our subset of UML, all *connectors* have at least a name, since the elements are all *named elements*. More precisely, they all have an attribute "*name*", but this attribute could be the empty string, which represents an unnamed element. Other properties depend on the type of *connector*. For example, a link as a set of stereotypes. A transition has an event and a guard. Changing the properties of a *connector* does not require any *namespace* since their notation is based on strings and thus they are not complex model elements. On the other hand, to model a possible substitution of a *connector* by another one connecting different model elements, *namespaces* are required and it is necessary to represent the connected model elements on the graphical representation of the substitutive element. The following notation is thus defined for a substitutive *connector*: a connected modeled element is represented by a rectangle with the name of the element inside. Again, the graphical representation that we provide is an abstract representation. At the concrete level, the usage will depend on the possibilities that the chosen tool can offer. Modeling such a change will not always be possible and will depend on the context. For example, it will not be possible if at least one of the connected model elements have no name or if it cannot be identified on the diagram. However, this situation should be rather unlikely. It will not always be possible either with state and activity diagrams, because certain tools use an abstract top state, as the one used for the UMLsec and UMLseCh abstract syntax. This state, usually used for technical reasons, contains the elements of the statemachine and those elements cannot be moved into another namespace. More details, as well as examples, will be given further in this section. The second type of model elements includes all the other elements that are not *connectors*. These elements do not request any particular semantics to model a change. Examples will also be given in the following.

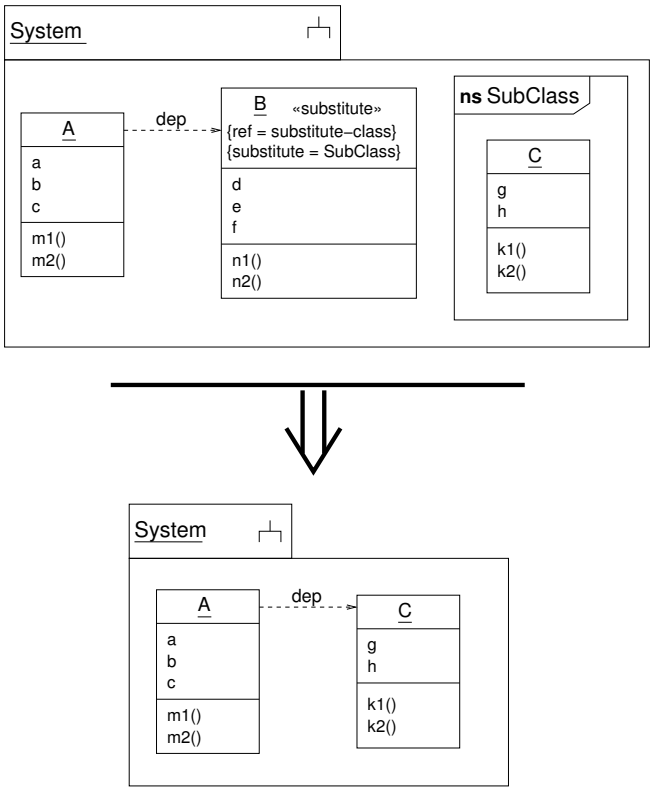


Figure 2.7: Model the possible substitution of a class

We can illustrate the use of namespaces to store complex substitutive elements with a simple case presented in the following example. Assume a class diagram with two classes, *A* and *B*, and a dependency *dep* between them. The class *A* has the attributes *a*, *b* and *c* and the methods *m1* and *m2* and the class *B* has the attributes *d*, *e* and *f* and the methods *n1* and *n2*. The change modeled for this example consists in replacing the class *B* by the class *C*, which has the attributes *g* and *h* and the methods *k1* and *k2*. The modeling of this change as well as its application is shown in Figure 2.7. Certain changes could however be simpler and thus not require complex model elements. Therefore, one should ensure that using a complex model element is imperative. Note that some parts of model elements might not be accessible to the UMLseCh notation. This is for example the case for the name of a named element. Indeed, the name of a named element is defined as an attribute of type String in [44]. However, the UMLseCh profile defines the values of the tags as pair of named elements. This means that only named elements can evolve and be used as new model elements.

The previous examples illustrated simple cases since the substitutive model elements could be easily stored in a namespace and be integrated in the model after the substitution. As mentioned above in this section, this will not be the case with the *connectors*. To illustrate the use of namespaces with *connectors*, we can consider the following example. Assume a class diagram with three classes, namely A, B and C and a dependency dep, stereotyped *«call»*, between A and B. A has the attribute a_1, a_2 and a_3 and the

methods m_1 and m_2. B has the attribute b_1 and b_2 and the methods n_1 and n_2. Finally, C has the attribute c_1, c_2 and c_3 and the method k_1. For this example, the possible change that we model is a substitution of the dependency dep by a new dependency dep', also stereotyped *« call »*, that now connects B and C. This is shown in Figure 2.8. Note that the namespace contains only the dependency, not the classes B and C. Again, the possibility to model such a change will depend on the tool that is used and the functionalities that it offers[7]. For example, as mentioned above, it will not be possible if at least one of the connected elements cannot be identified on the graphical notation, although this situation is unlikely. In particular, this applies to statemachines or activity diagrams. However, using a state in another namespace as substitutive model element will be possible since states are not *connectors* and they do not involve any *cross-reference*[8]. An example of this is shown in Figure 2.11 of Section 2.1.4, where states are placed in a package, although, for different reasons, this example models a wrong addition. Because of the requirements defined above, modeling possible changes with connectors as substitutive model elements will thus be avoided. Deleting the *connector* and adding a new one with the intended properties will be preferred to model the possible change. This can be model with a composite change, described in Section 2.1.2. Addition and deletion will be discussed in the following sections. Alternatively, one can also use an expression based on the abstract syntax of transitions, as the value of the element e' in a pair $(e,\ e')$ of a tag substitute, since it uses sequences of characters only. In this case, the readability will be slightly reduced but no complex element and thus namespace will be required. Considering the example of Figure 2.8, the value of e' could be replaced by the following expression:

$$d = (\text{"}dep'\text{"}, A, B, B, \text{«}\,\textit{call}\,\text{»}) \quad d \in \mathrm{Dep}(CD),$$

where CD is the class diagram, or this expression could be placed in a comment note attached to the concerned stereotype.

2.1.4 Complex Additive Elements

Complex additive elements also require a specific semantics. As for the substitutions, we will differentiate two types of model elements, the *connectors* and the rest of the model elements. In addition, our subset of diagrams is partitioned into two groups: the static structure diagrams, which include the Class diagram, the Object diagram and the Deployment diagram, and the dynamic behavior diagrams, which includes Statechart diagrams, Activity diagrams and Sequence diagrams. For model elements that belong to static structure diagrams and that are not *connectors*, no particular semantics is necessary to model a possible addition. On the other hand, additions will be slightly more complicated

[7] With ArgoUML, this example can be modeled in the following way: assuming that the class diagram has been modelled, one creates a dependency between the classes B and C and then, creates a package. The namespace of the dependency can then be change to the package. Once it is done, the new dependency can be deleted from the diagram and the name of the package (i.e. the namespace) containing that dependency can be used as a reference in the value of the tag substitute. This solution however will not work with statemachines or sequence diagrams

[8] A *cross-reference* here means that the elements used as *connected* model elements refer to elements that belong to another namespace. Replacing a *connector* by another one having the same *connected* elements would thus not involve any *cross-reference* either, but this will be covered by the string-based modification of the *connector* properties.

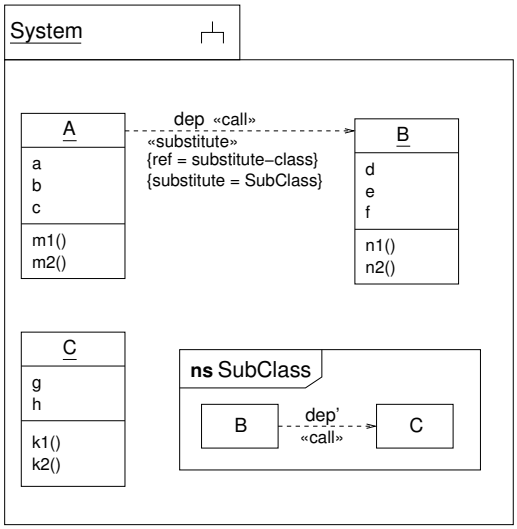


Figure 2.8: Model the possible substitution of a dependency

for *connectors* and dynamic behavior diagrams. As mentioned in Section 2.1.2, certain model elements also require to be integrated to the model after being added. This means that they need to be connected to the rest of the diagram, otherwise it would leave the model in an inconsistent state. This category of model elements includes *connectors* of all types of diagrams and any model element from dynamic behavior diagrams, with one exception described below. In certain cases, we can ensure the integration of the new added model elements with a special behavior that we call *merge*. This operation will be described further in this section. Other situations will require more complex additions, which will be detailed in Section 2.1.5.

For static structure diagrams, as mentioned above, adding a model element that is not a *connector* is easy. Those model elements are, for example, Nodes, Classes or Components. In this case, modeling the change thus simply consists in placing the model element into a namespace and use the name of this namespace as a reference in the relevant pair of the tag add. Such changes are trivial.

The previous examples illustrated a simple case since the additive model elements belong to a static structure diagram and are not *connectors*. It could thus be easily stored in a namespace and be integrated in the model after the addition. Some situations will be different and will require the *merge* behavior . The *merge*, as its name indicates, adds the new model elements and *merges* the parts of the additive model elements that are already present on the model. This behavior will be used automatically if elements from the existing model are included into the additive part. To illustrate the use of a *merge*, we can consider the following example. Assume a Statechart diagram with three states, namely A, B and C, and two transitions, one from A to B and one from B to C. The initial and final states are also present on the diagram. Assume now that one wants to model the possible addition of a transition from C to B. This can be modeled by placing two states, B and C, and a transition from C to B in a namespace[9]. The name of this

[9] As explained, this namespace will be a state machine at the concrete level.

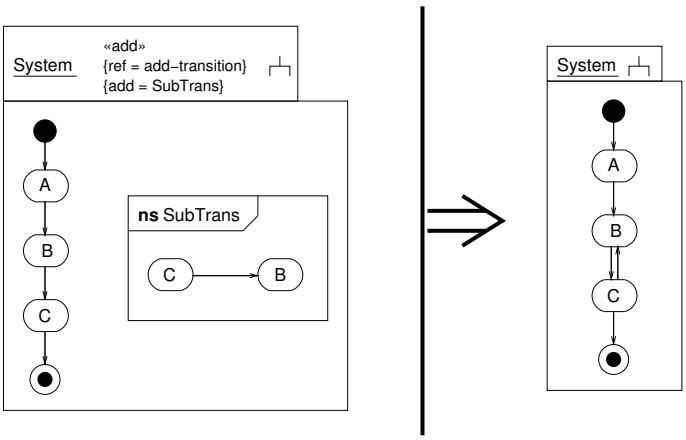


Figure 2.9: Possible addition of a new transition using the *merge*

namespace is then used as the element e' of the pair (e, e') representing the relevant addition in the tag add[10]. This model as well as the result of applying the change is shown in Figure 2.9. Note that if final states are placed in the namespace containing the additive elements, they will not be merged with the existing final states, since a statemachine can have more than one final state. This property also stands for activity diagrams.

There exists one exception to the general principles mentioned in the previous paragraphs. This exception concerns *lifelines* of Sequence diagrams. Indeed, although Sequence diagrams are dynamic behavior diagrams, *lifelines* can be added alone since they do not need to be directly connected to the rest of the diagram. They can thus be added in a similar way as the *non-connectors* elements of static structure diagrams. To model a possible addition of a *lifeline* to a Sequence diagram, we will thus place this *lifeline* into an *namespace* and use the name of this *namespace* as a reference in the pair of the tag add representing the change. Applying the change will then simply consist in adding the *lifeline* contained in the *namespace* to the model. An example of such a change is shown in Figure 2.10, where a *lifeline*, named *C*, could possibly be added, if the change occurs, to a Sequence diagram containing two *lifelines*, *A* and *B*.

2.1.5 Problems with Stereotypes *« add »* and *« delete »*

As mentioned in Section 2.1.2, adding or deleting a model element might generate problems or difficulties that do not exist with a substitution. This is mainly due to the fact that a substitution simply means to change a model element by another one of the same type. On the other hand, an addition of a model element means, in addition to adding the element, to adapt the new model in order to integrate the new element. Deleting also requests to adapt the model resulting from the deletion. This section illustrates such situations by means of examples. More generally, both the addition and the deletion will have to respect constraints to ensure the diagrams consistency. These rules will be detailed in

[10]This principle could also be used with substitutions. However, even if it could be useful in certain cases, it is not indispensable. For simplicity reasons, it will not be defined here.

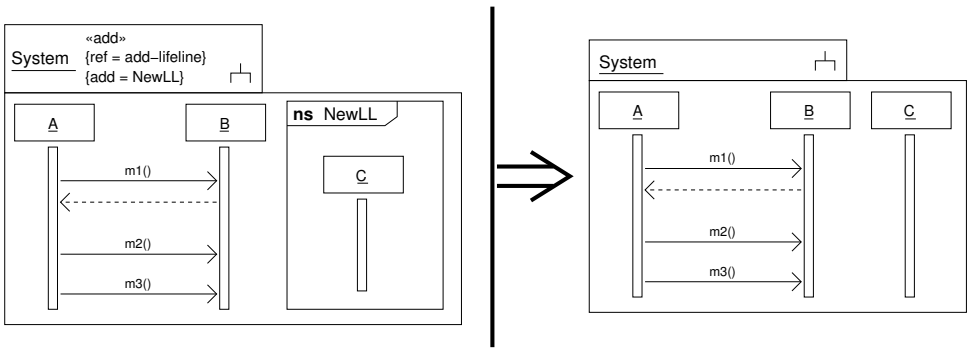


Figure 2.10: Example of an addition of a *lifeline* in a sequence diagram

the formal foundations of the notation in the following.

2.1.5.1 The case of *« add »*

To illustrate a case of inconsistency created by an addition, we can consider the following example. Assume a State Diagram with four states: the initial and final states, and the states *A* and *B*. In addition, the possibility of adding a new state, called *C*, is considered. This new state would directly follow the state *B* and precede the final state. This can be seen as adding a state on the transition from B to the final state. A stereotype *« add »*, together with the related information in the tagged values, could thus be attached to the subsystem. However, applying the change modeled by this stereotype would lead to the situation shown in Figure 2.11, which presents an inconsistent diagram (the state *C* is pending on the diagram and is not connected to any part of it) and represent a change obviously different from the one initially intended. This problem actually comes from the fact that, although intuitively the change can be seen as the addition of a state on the last transition, the concrete addition will just add a state disconnected from the rest of the diagram. This follows the correct semantics of the stereotype. Indeed, the stereotype *« add »* only models the addition of the state *C*, and nothing else. But the change that was intended was the addition of the state *C*, the modification of the transition between the state *B* and the final state so that it is now connecting the state *B* to the state *C*, and finally the addition of a transition between the state *C* and the final state. It is hence a wrong modeling of the intended change more than a wrong semantics of the stereotype *« add »*. To model the change correctly, several possibilities exist.

A first solution would be to substitute the state B by a sequential composite state containing and arranging the additional elements in the intended way. To model such a change, a stereotype *« substitute »* with the related information can be attached to the state B of the statechart diagram. This situation is shown in Figure 2.12. Following the semantics of the stereotype *« substitute »* in the case of complex substitutive elements, presented in Section 2.1.3, the application of this change would generate a result rather close to the intended one. Concretely, the last part of the statechart diagram would be contained in the composite state, which would hence represent a sub-diagram. The result obtained after the substitution and the result initially intended are equivalent, since when the composite

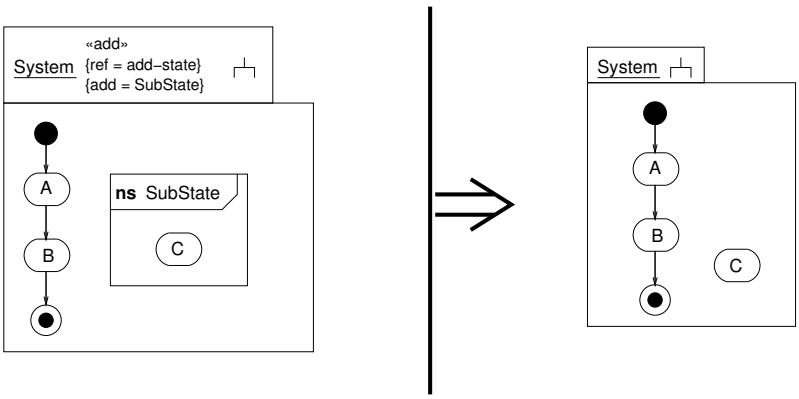


Figure 2.11: Inconsistent diagram resulting from an inappropriate addition

state is entered, the flow will visit the state B then he state C, then exit the composite state and leave the diagram through the final state. However, although both the diagrams are equivalent, the result remains slightly different from the one expected[11]. Other possibilities exist.

Another solution would be to use the *merge* operation. However, as explained in Section 2.1.4, the final states will not be merged. In this case, a final state is necessary in the additive namespace in order to model the transition from the state C to the final state. Therefore, after the *merge*, the remaining final state and final transition from the state B have to be deleted. These changes should also happen together, since they represent one global change, and thus should be modeled by a stereotype *« change »* representing composite changes.

Finally, another solution would be to model the change with three stereotypes, each of them modeling one of the three changes necessary to add the state C. These three changes, as mentioned above, are the addition of the state *C*, the modification of the transition between the state *B* and the final state so that it is now connecting the state *C* to the final state and finally, the addition of a transition between the state *B* and the state *C*. However, this solution will work only if the three changes **happen simultaneously**. This can be modeled by using a composite change, as shown in Figure 2.13. For simplicity reasons, we name the final state f. Note that this solution is not simpler than the solution using the *merge*.

As mentioned above, the changes modeled by stereotypes *« add »* will have to respect additional constraints to ensure the consistency of the diagram, otherwise the model will not be allowed. This will be described in Section 2.5.

[11] Provided that the notation accepts the possibility for the two elements of a pair of a tag substitute to be of different types, the namespace having the substitutive elements could contain the state B, the state C and the transition between them. This solution is however not allowed by the current version of UMLseCh.

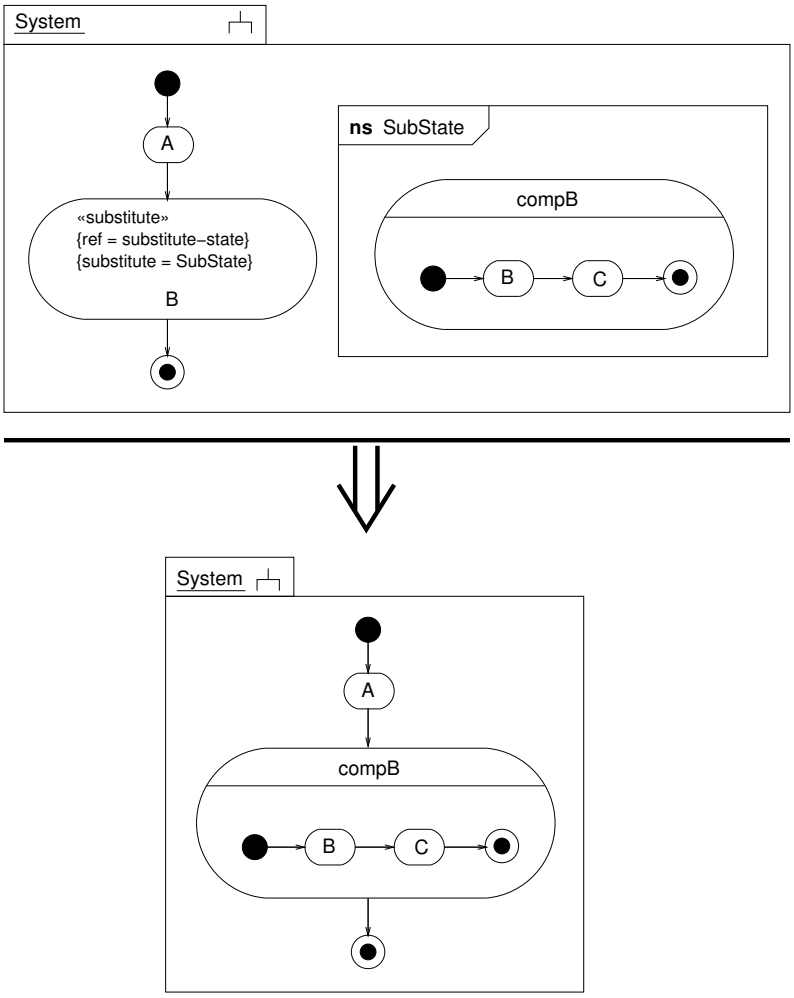


Figure 2.12: A solution to the problem of Figure 2.11

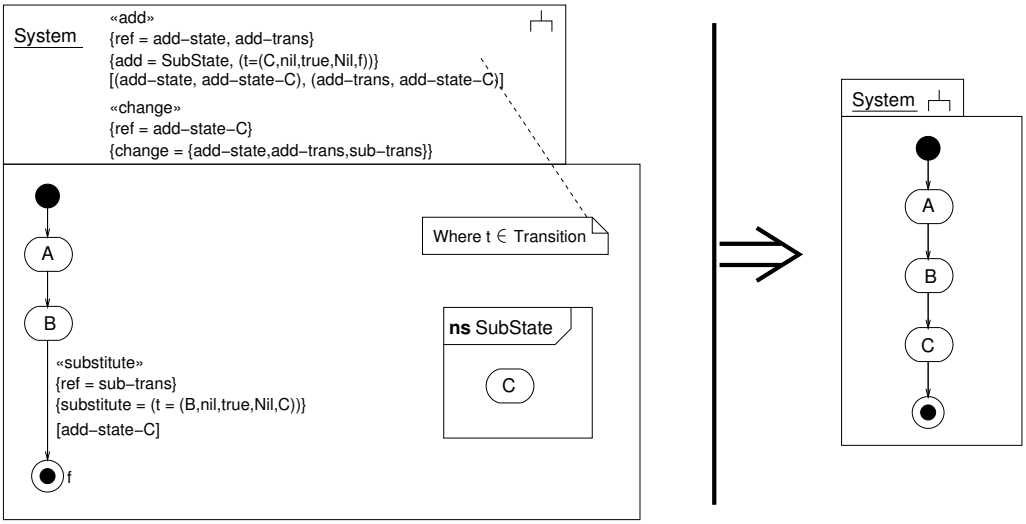


Figure 2.13: Another solution to the problem of Figure 2.11

2.1.5.2 The case of *« delete »*

Applying a change modeled by a stereotype *« delete »* could also leave the model in an inconsistent state. For example, deleting a *lifeline* of a sequence diagram connected to another *lifeline* by messages would result to the situation shown in Figure 2.14. No extra semantics have been defined for the behavior of the application of a change modeled by a stereotype *« delete »*. Applying a deletion modeled by a stereotype *« delete »* on a model element will thus not be allowed if it does not fulfill the constraint defined in Section 2.5, which is the case of the stereotype *« delete »* of Figure 2.14.

2.2 General Concepts

Before defining the formal representation of the UMLseCh diagrams and the semantics of the application of a change, we need to define general principles. The UMLsec and UMLseCh abstract syntax are both based on n-tuples and sets and thus, all of the concepts will be described using set theory.

The most general concept that we define is the concept of UML named element, which is an element that "may have a name" [44]. We define the set **Elements** as the set of the instances of the named model elements defined on a model instance. In the abstract syntax, the model elements will all be represented with their own representation using n-tuples. At this point, we define the general representation of a *UML named element*.

Definition 1. *A UML named element is a tuple* $e \in \mathbf{Elements}$ *such that* $e = (e_1, \ldots, e_n)$*, with* $n \in \mathbb{N}$*, where the k first elements, with* $k \in \mathbb{N}$*,* $1 \leq k \leq n$*, are names and the* $n - k$ *other elements are named elements or sets of named elements. If* $k = 0$*, the named element has no name. If* $k \geq 1$*, the named element has k names and is defined by* $e = (e_1, \ldots, e_k, \ldots, e_n)$ *where* $e_1, \ldots, e_k$ *are names.*

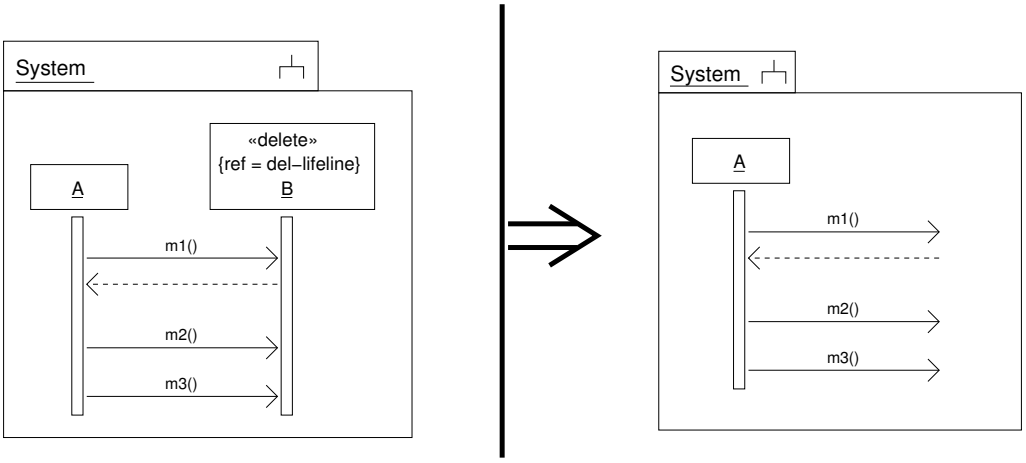


Figure 2.14: Incorrect diagram resulting from an unallowed *« delete »*

In addition to *named elements*, we can define *namespaces*. A *namespace* is defined in [44] as an abstract container of *named elements*, the *namespace* being itself a *named element*. We assume a set $\mathbf{Namespaces}$ that contains all the namespaces of the instance of a model and give the following definition.

Definition 2. *A UML namespace is a pair* $(n, elts)$*, with* $n \in \mathbf{String}$ *the name of the namespace and* $elts \subseteq \mathbf{Elements}$ *a set of model elements contained in the namespace.*

Since *namespaces* are themselves *named elements*, they can be contained in *namespaces*. We thus define a *top namespace* as a *namespace* that is not contained in any other *namespace*. In addition, any *named element* is contained, possibly in a nested way, in a *top namespace*.

Definition 3. *A top namespace is a namespace that is not contained in any namespace. All of the UML named elements are contained directly or indirectly in a top namespace.*

The set **StereoNm** is the set of *stereotype* names. This set includes the UMLseCh stereotypes, such as *« change »*, *« substitute »* or *« delete-all »*, as well as the UMLsec stereotypes, such as *« critical »*, *« fair exchange »* or *« secure dependency »*. It also includes some of the UML stereotypes, such as *« call »* and *« send »*, and the user defined stereotypes. We differentiate the stereotype definitions from the stereotype instances. A stereotype definition is a stereotype name $s \in \mathbf{StereoNm}$. We assume a function that maps a stereotype definition to its associated tag definitions. A stereotype instance is a stereotype applied on a model element in the instance of a system. It has its owned associated tagged values for each tag definition. The set **Stereotypes** represents the set of all instances of stereotypes. A stereotype instance must be an instance of a stereotype definition, as defined in the following.

Definition 4. *An instance of a stereotype is defined as* $s^\tau \in \mathbf{Stereotypes}$*, where* $s \in \mathbf{StereoNm}$ *is the stereotype definition of* s^τ*.*

This simply ensures that any stereotype instance is an instance of a stereotype defined in the set $\mathbf{StereoNm}$. Note that by *stereotype* in the following, we always mean *stereotype instance*. We will explicitly precise when an element refers to a *stereotype definition*. We also define a function τ that returns the stereotype definition of a stereotype instance.

Definition 5. *Let s be a stereotype definition, such that $s \in \mathbf{StereoNm}$ and s^τ be a stereotype instance, such that $s^\tau \in \mathbf{Stereotypes}$ and such that s^τ is an instance of the stereotype definition s, as defined in the definition 4, τ is defined as:*

$\tau : \mathbf{Stereotypes} \rightarrow \mathbf{StereoNm}$
$\tau(s^\tau) = s$

The semantics of the *stereotype definitions* can be refined by defining particular types. For example, the stereotypes of UMLsec could be considered as being of the type security, which represents stereotypes modeling security requirements. In the following, we will define the type change for the stereotype definitions of UMLseCh. More precisely, all of the stereotypes defined in Figure 2.1 are of the type change. Formally, the set **ChangeNm** represents the set of stereotype definitions of type change. It is defined as follow.

Definition 6. *The set $\mathbf{ChangeNm} \subset \mathbf{StereoNm}$ is the set of definitions of stereotypes of type* change*, such that:*

$$\mathbf{ChangeNm} \equiv \{\text{« change »}, \text{« substitute »}, \ldots, \text{« delete-all »}\}^{12}$$

The stereotypes of type change thus also belong to a particular set. The set **Change** is the set of instances of stereotypes of type change, such that
$\mathbf{Change} \subset \mathbf{Stereotypes}$. Formally, a change stereotype can hence be precisely specified by the following definition.

Definition 7. *A stereotype of type* change *is a stereotype instance $s^\tau \in \mathbf{Change}$ such that $s \in \mathbf{ChangeNm}$.*

For tag definitions, we focus on the instance level because the elements that will be used for the application of changes are tagged values. These values, associated to tags on the instance of a model, are sufficient to apply the concepts presented in this chapter. However, to completely define tags, assume the set $\mathbf{TagNm}$ of tag definitions. It includes all the tag definitions of the stereotype definitions. Formally, we have $\mathbf{TagNm} \equiv \cup_s tagnm(s)$, $\forall s \in \mathbf{StereoNm}$, where we suppose a function $tagnm$ that return the tag definitions of a stereotype definition. On the other hand, the set **Tag** is the set of the tags of stereotype instances. Any tag associated to an instance of a stereotype applied on a model element belongs to this set. As for stereotypes, the semantics of tags can be refined by giving them a type. In particular, the tags associated to UMLseCh stereotypes will be considered as tag of type change. The set **TagChange** is the set of the tags of such type. As described in Chapter 2.1, the tags that can be attached to stereotype of type change are the tags ref, change, substitute, add and delete. Therefore, the set **Tag** can be disjointly partitioned into the sets **TagRef** of instances of tag ref and **TagTr** of

[12] The complete list of stereotypes can be found in Figure 2.1 where the UMLseCh profile is defined

instances of tag change, substitute, add and delete. We also define the set $\mathbf{Values}$ of values of the tags. This set contains all the values that are given to tags on an instance of a model. Formally, the type of tags change can be defined as follow.

Definition 8. *Let $tags(s)$ be a function that returns the tags of a stereotype s. A tag $t : t \in tags(s)$ is of type* change *if $s \in Change$.*

The function $tags$ will be defined in the following. We can also refine the UMLseCh stereotypes since they only have associated tags of type change.

Definition 9. *Let $tags(s)$ be a function that returns the tags of a stereotype s. If $s \in \mathbf{Change}$, we have $tags(s) \subseteq \mathbf{TagChange}$.*

The values associated to a tag ref of a change stereotype represent a list of labels such that each element of this list can be used as a reference of the change modelled by the stereotype and its associated tagged value, which is the value of the list in the tag of type change at the same position as the label in the list of the tag ref. Such a label provides a means to identify a change and thus must be unique among the values of all of the tags ref associated to stereotypes of a model instance. This constraint is verified by the following definition.

Definition 10 (Unicity of the ref values)**.** *Let $\mathbf{TagRef}$ be defined as $\mathbf{TagRef} \equiv \{t_1, t_2, \ldots, t_k, \ldots\}$ and $v(n)$ be a function that returns the value of a tag n. Assume $T \equiv (v(t_1) \uplus v(t_2) \uplus \cdots \uplus v(t_k) \uplus \cdots)$ the multiset containing the values of all the tags* ref *associated to stereotypes of a model instance and represented as $T \equiv \{x_1, x_2, \ldots, x_k, \ldots\}$. Then $\forall i, j \in \mathbb{N}$ such that $i \neq j$, we have $x_i \neq x_j$.*

The function v that returns the value of a tag, as well as the function σ that returns the name of a tag, will be defined in the following. In the next section, we will also define the formal representation of stereotypes and tagged values. These representations will extend the abstract syntax of UMLsec. We also need the set of boolean values and the set of strings for signatures of functions defined below. We thus define the set **Boolean** of boolean values as $\mathbf{Boolean} \equiv \{true, false\}$. We also assume the set of boolean expressions $\mathbf{BoolExp}$. The set **String** is the common set of sequences of symbols and digits.

2.3 New Elements for the UMLseCh Abstract Syntax

To provide formal semantics of the changes modeled by the UMLseCh stereotypes, some additional concepts need to be added to the abstract syntax. Stereotypes are mentioned in the abstract syntax of UMLsec, but no precise representation is given. More precisely, only a set of *stereotype* names is defined. For UMLseCh, the situation is different, because the semantics of the changes need to directly use the stereotypes and their associated tagged values. We thus define the abstract representation of a stereotype as follow. A stereotype $s \in \mathbf{Stereotypes}$ is a tuple given by $s = (name, tag, constraint)$ where:

- $name \in \mathbf{StereoNm}$ is the name of the stereotype;
- the set $tag \subseteq \mathbf{Tag}$ is the set of associated tags; and
- $constraint \in \mathbf{BoolExp}$ is the constraint of the stereotype.

Since we define stereotypes as tuples, we can consider the inductive definition of a tuple from the Kuratowski's definition of ordered pairs and thus define the k^{th} element of the tuple n as $\pi_k(n)$, $\forall k \in \mathbb{N}$. With such a representation, it is easy to define the function $tags(s)$ that return the set of tags of a stereotype s, mentioned in the previous section.

Definition 11. *Let s be a stereotype such that $s \in \mathbf{Stereotypes}$, its set of tags is given by the function:*

$tags : \mathbf{Stereotypes} \to \mathcal{P}(\mathbf{Tag})$
$n \mapsto \pi_2(n)$

Note that $\mathcal{P}(X)$ represents the power set of X. Following the same principle, the constraint associated to a stereotype can be obtained as easily as for the set of tags. Concretely, we have the following function.

Definition 12. *Let s be a stereotype such that $s \in \mathbf{Stereotypes}$, its constraint is given by the function:*

$cons : \mathbf{Stereotypes} \to \mathbf{BoolExp}$
$n \mapsto \pi_3(n)$

The representation of a stereotype defined above also allows us to refine the definition of a stereotype of type change. More precisely, as defined in Figure 2.1 in Chapter 2.1, a stereotype of type change must have two associated tags and an associated constraint. Formally, we have the following.

Definition 13. $\forall s \in \mathbf{Change}$ *such that* $s = (name, tag, constraint)$*, we have:*

$$tag \equiv \{ref, change\}$$

where $ref \in \mathbf{TagRef}$ *and* $change \in \mathbf{TagTr}$*.*

This definition restricts the use of a stereotype of type change and ensure that any of such a stereotype has exactly two tags, one of type ref and one of type change. We also define two functions ref and $change$, which return respectively the tag of type ref and the tag of type change of a stereotype s, such that $s \in \mathbf{Change}$.

Definition 14. $\forall s \in \mathbf{Change}$*, its tag of type* ref *is given by the function:* $ref : \mathbf{Change} \to \mathbf{TagRef}$ $n \mapsto e$ *where* $e \in tags(n) \cap \mathbf{TagRef}$.

Definition 15. $\forall s \in \mathbf{Change}$*, its tag of type* change *is given by the function:* $change : \mathbf{Change} \to \mathbf{TagTr}$ $n \mapsto e$ *where* $e \in tags(n) \cap \mathbf{TagTr}$.

Note that we can ensure the results of the functions defined above, since a stereotype of type change has exactly two tags, one in $\mathbf{TagRef}$ and one $\mathbf{TagTr}$, and $\mathbf{TagRef} \cap \mathbf{TagTr} = \emptyset$.

In [44], stereotypes are described as elements that can extend the semantics of any other model element. In the UMLsec abstract syntax, however, only certain model elements contain stereotypes. This restriction, justified in the context of UMLsec, has to be overcome in the abstract syntax of UMLseCh, since the stereotypes semantics represent possible evolution, which can be attached to any model elements. Precisely, the abstract representation of elements that do not contain stereotypes in UMLsec, such as states or transition, will be extended so that they can contain stereotypes. Note, however, that this extension does not concern all model elements. In particular, certain model elements, such as attributes or operations of a class, will not be extended and thus will not have any stereotype. The abstract of UMLseCh will be presented further in this Chapter.

Tagged values also need to be given a formal abstract representation. In UMLsec, a function mapping a stereotype to its associated tagged values and constraint is assumed, but again, no precise representation is defined. In UMLseCh, tags of stereotypes of type change represent an important information since they contain the information describing the change to apply. We define a tag as an ordered pair $t \in \mathbf{Tag}$ given by $t = (tag, value)$, where:

- $tag \in \mathbf{TagNm}$, is the name of the tag; and
- $value \subseteq \mathbf{Value}$, is the set of values associated to the tag.

Our representation of tags is then equivalent to the definition of tagged values in the UML specification. We also define two functions, σ and υ, mentioned above, that we can apply on a tag. Assume a tag n, such that $n \in \mathbf{Tag}$, the function $\sigma(n)$ returns the *name* of the tag n and the function $\upsilon(n)$, mentioned in the previous section, returns the *value* of the tag n. As for stereotypes, since tagged values are represented as ordered pairs, it is easy to define those functions following the Kuratowski's definition of ordered pairs.

Definition 16. *Let t be a tag such that $t \in \mathbf{Tag}$, its name is given by the function:*

$\sigma : \mathbf{Tag} \to \mathbf{TagNm}$
$n \mapsto \pi_1(n)$

Definition 17. *Let t be a tag such that $t \in \mathbf{Tag}$, its set of values is given by the function:*

$\upsilon : \mathbf{Tag} \to \mathcal{P}(\mathbf{Value})$
$n \mapsto \pi_2(n)$

As defined in the UML specification [44], a tagged value can only be represented as an attribute defined on a stereotype. Therefore, we follow the specifications by attaching a set of tags only to stereotypes in our abstract syntax. If the stereotype does not have any associated tag, the set *tag* is simply the empty set $\emptyset$. We also define the empty tag written $\emptyset$, such that $\emptyset \in \mathbf{Tag}$ and all its relevant subsets.

We can refine the value of the tags substitute, add and delete. These tags have a list of pairs of the form (e, e') as value, where e is the model element concerned by the change and e' is the new model element. Note that although certain cases of the instance level, described in Section 2.1.2, will allow the elements of the list to be single elements or the list itself to be omitted, this is just syntactic sugar. For the formal semantics, we will

assume that the elements can all be identified from the list of values given by the tags. The value of a tag t of type ref is an ordered set of strings such that $\upsilon(t) \subset \mathbf{String}$. The value of a tag t of type change is an ordered set of sets of strings, such that $\forall l \in \upsilon(t)$, we have $l \subseteq T$, where T is the set defined in definition 10. Tagged values of stereotypes of type change have another particularity. They are in the form of lists where the order of the elements matter, since elements of same subscript in each list are related and represent one change modeled by the stereotype. Before we can define this precisely, we need to refine the definition of the empty tag $\emptyset$.

Definition 18. *The set of values of the empty tag $\emptyset$ is such that $\forall t \in \mathbf{Tag}$, $|\upsilon(\emptyset)| = |\upsilon(t)|$*

We can now define the set of values of a tag of type change as an ordered set and give an additional condition to stereotypes of type change so that the tagged values can be associated together. This condition is that the ordered sets have the same arity.

Definition 19. *Let s be a stereotype such that $s \in \mathbf{Change}$ and $tags(s) = \{t_1, t_2\}$. The value of the tags t_1 and t_2 are ordered sets such that $|\upsilon(t_1)| = |\upsilon(t_2)|$. We also assume a function g that returns the k^{th} element of the set of values of a tag of type* change *t, $\forall k$ $1 \leq k \leq |\upsilon(t)|$ and a function f that returns the position of an element in the set of values of a tag of type* change*. These functions will be defined in the next section.*

Finally, *namespaces* need to be included in the UMLseCh abstract syntax. As mentioned in Section 2.1.5.2, a *namespace* does not exist by itself. It is modeled by an abstract metaclass and only takes the form of a model element, such as a class, a statemachine or a package, at the concrete level. However, as for the graphical notation, we need to be able to represent *namespaces* in the abstract syntax in a general form and at a high level of abstraction. Since all of the UMLseCh model elements are *named elements* that belong to the set **Elements** and following the definition 2, a *namespace* can simply be define as an ordered pair $n = (sname, elts) \in \mathbf{Namespaces}$ given by:

- a *namespace* name $sname \in \mathbf{String}$; and
- a set $elts \subset \mathbf{Elements}$ of *named elements*.

Again, a function that returns the set of named elements of a namespace can easily be defined. Note that all of the named elements defined here have a name based on string representation, even if they belong to a different set, such as $\mathbf{StereoNm}$. It is thus correct to consider the name $sname$ of a namespace as an element of $\mathbf{String}$.

Definition 20. *Let ns be a namespace such that $ns \in \mathbf{Namespaces}$, its set of named elements is given by the function:*

$elts : \mathbf{Namespaces} \rightarrow \mathcal{P}(\mathbf{Elements})$
$n \mapsto \pi_2(n)$

Theoretically, any UML *named element* that can contain other UML *named elements* is a *namespace*. This concerns a large part of the model elements of our simplified UML since many of them are *namespaces*. However, although these model elements are

namespaces, they will not be represented in the form described above because this form is too abstract for some of the concepts that will be required in the following. All of the elements will thus be defined with their own representation, extending the UMLsec abstract syntax in order to include the UMLseCh concepts. As mentioned above, these model element representations are defined with n-tuples. The general form for *namespaces*, defined above, will nevertheless be useful for certain cases that will describe the application of a change at the highest level of abstraction. It is therefore useful to have a function mapping a namespace in its model element representation to the general representation of *namespaces*. This is defined in the following.

Definition 21. *Let e be a namespace in its model element representation, such that $e = (e_1, \ldots, e_n)$, as defined in the definition 1, with $n \in \mathbb{N}$, $e \in \mathbf{Elements}$ and $e_k \in \mathbf{Elements} \cup \mathcal{P}(\mathbf{Elements})$, $\forall k : 1 \leq k \leq n$. The function ns, which gives the general representation of e, is given by:*

$$ns(e) = (sname, elts)$$

where $sname \in \{e_1, \ldots, e_k\}$, with $k \geq 1$ the number of names in e, as defined in the definition 1, or $sname$ is the empty name $\emptyset$ if $k = 0$, is the name of e and $elts \equiv s_1 \cup \cdots \cup s_{n-k}$ given, $\forall i\ 1 \leq i \leq n - k$, by:

- $s_i = e_{i+k}$, *if e_{i+k} is a set;*
- $s_i = \{e_{i+k}\}$, *if e_{i+k} is a model element; and*
- $s_i = z_1 \cup \cdots \cup z_l$, *if e_{i+k} is a diagram of the form $D = (d_1, \ldots, d_l) \quad \forall l \in \mathbb{N}$ and where, $\forall j\ 1 \leq j \leq l$, $z_j = d_j$ if d_j is a set and $z_j = \{d_j\}$ if d_j is not a set*[13].

2.4 General Application of a Change

At the highest level of abstraction, it is possible to simply represent a change using the concepts defined in Section 2.2 and 2.3. Assume a function $space$ that returns the namespace $n \in \mathbf{Namespace}$ of a model element e, with $e \in \mathbf{Elements}$, or the model element e itself if e is not contained in any namespace[14]. Each change, namely a substitution, an addition or a deletion, can easily be defined as follow.

Definition 22 (Local substitution)**.** *Let e be a UML model element and e' the substitutive model element of e, such that $e, e' \in \mathbf{Elements}$, and let $N \in \mathbf{Namespaces}$, such that $N = (n, S)$, with n a name of e and $S \subset \mathbf{Elements}$, be the namespace in which e is contained, such that $N = space(e)$. A substitution of e by e' in N is defined as $(S \setminus \{e\}) \cup \{e'\}$.*

Definition 23 (Local addition)**.** *Let e be a UML model element and e' the set of additive model elements (which may contain only one element if only one element is added) to add*

[13] This condition is necessary to apply ns on subsystems. See Section 2.5 for the abstract syntax of subsystems

[14] In our subset of UML, this second possibility concerns subsystems.

in e, such that $e \in \mathbf{Elements}$ and $e' \subset \mathbf{Elements}$, and assume $ns(e)$ the namespace general representation of e, such that $ns(e) = (n, S)$, with n a name of e and $S \subset \mathbf{Elements}$. An addition of e' in e is defined as $S \cup e'$.

This definition of an addition provides an automatic support of the *merge* behavior described in Section 2.1.4. Indeed, assume that e' is a set of instances of model elements, such that $e' = \{e'_1, e'_2, \ldots, e'_{k-1}, e'_k, e'_{k+1}, \ldots, e'_n\}$, with $e'_i \in \mathbf{Elements}$, $\forall i \in \mathbb{N}$, $1 \leq i \leq n$, and such that e'_{k-1}, e'_k and e'_{k+1} are instances already present in the namespace e. The model elements e'_{k-1}, e'_k and e'_{k+1} will automatically be merged in e, by definition of the union operator $\cup$, which removes the duplicate elements of the set.

Definition 24 (Local deletion)**.** *Let e be a UML model element to delete from the model, such that $e \in \mathbf{Elements}$, and N be the model element in which e is contained, such that $N = (n, S)$ is the namespace general representation of e, with n a name of e and $S \subset \mathbf{Elements}$, and $space(e) = N$. A deletion of e from N is defined as $S \setminus \{e\}$.*

Note that at the concrete level, following the UML nested representation of namespaces, it is easy to define the namespace of a model element, and thus a result of the function $space$. Indeed, using the composite name $n_1 :: n_2 :: \ldots :: n_k :: e$ of a model element e, with $k \in \mathbb{N}$, the namespace of e is n_k. The elements e and e' can also easily be found at the concrete level since they will be specified in a stereotype of the instance of a model. More precisely, the element e' will be found in the values of the tags substitute or add, as the second element of a pair. The element e will either be the first element of a pair in the values of the tags substitute, add or delete, or be the element to which the stereotype is attached. Note however that with certain complex changes, the element concerned by the change will not be directly expressible in the pair representing the change or be the element to which the stereotype is attached. In such a case, the abstract syntax will be used as a language to precisely represent this element. Concretely, modifying this element will require formal rules that interpret the expression given to identify the element so that it can be passed to the function. Such a means to interpret the rules is beyond the scope of this deliverable and concerns tool support. It is thus not considered here and in particular, we assume that the model element concerned by the change is given as an argument of the function. The definitions of the application of a substitution, an addition or a deletion given above are specific to the concerned model elements, but do not represent a complete change in an instance model. In particular, a change should also integrate the results of the definitions 22, 23 and 24 to the model, remove the tagged values associated to that change and update the function that evaluate the predicate *"occurred"*. To update the tagged values of a change stereotype, such that the information relative to the change is removed after the change occurs, the following functions will be necessary.

Definition 25. *Let $t = (tag, values)$, with $t \in \mathbf{TagChange}$, be a tag of type* change *such that the ordered set $values \equiv \{v_1, \ldots, v_k, \ldots, v_n\}$, $n \in \mathbb{N}$ and $\forall k \in \mathbb{N}$, $1 \leq k \leq n$. The function f is the function that returns the position of an element of the set values, such that:*

$$f((v_1, \ldots, v_k, \ldots, v_n), v_k) = k$$

$\forall n \in \mathbb{N}$ and $\forall k \in \mathbb{N}$, $1 \leq k \leq n$.

Definition 26. *Let $t = (tag, values)$, with $t \in \mathbf{TagChange}$, be a tag of type* change *such that the ordered set $values \equiv \{v_1, \ldots, v_k, \ldots, v_n\}$, $n \in \mathbb{N}$ and $\forall k \in \mathbb{N}, 1 \leq k \leq n$. The function g is the function that returns the element of a given position in a set values, such that:*

$$g((v_1, \ldots, v_k, \ldots, v_n), k) = v_k$$

$\forall n \in \mathbb{N}$ *and* $\forall k \in \mathbb{N}, 1 \leq k \leq n$.

We also assume a function that returns the right condition of a change, based on the value of the tag ref. To update the predicate *"occurred"*, we need to define a function mapping a boolean value to a value of a tag ref. The set $\mathbf{TagRef}$ is the set of the instances of tags of type ref. When a change occurs, its label is removed from the list of the associated tag ref, this tag being an instance that belongs to the set $\mathbf{TagRef}$. It is hence possible to define a function ψ representing the predicate "occurred", which is true if the change labeled by the tag ref given as argument of the predicate occurred.

Definition 27. *The function ψ representing the predicate "occurred" is defined as:*

$$\psi : \mathbf{String} \rightarrow \mathbf{Boolean}$$
$$n \mapsto \begin{cases} true & \text{if } \nexists r : r \in \mathbf{TagRef}, n \in v(r) \\ false & \text{if } \exists r : r \in \mathbf{TagRef}, n \in v(r) \end{cases}$$

To integrate the result of a substitution, an addition or a deletion to the rest of the model, the definitions of the application of a change given above can be refined so that the local change is reflected to the whole model. Intuitively, substituting a model element simply consists in replacing that model element by another one on the model. This is described in the definition 22. However, formally, when the model element is substituted by the other one, the set containing the new element is a new set that needs itself to substitute the set containing the former model element on the model. A substitution, represented by the function $substitute$, can thus be defined recursively.

Definition 28 (Substitution)**.** *Let e be a UML model element and e' the substitutive model element of e, such that $e, e' \in \mathbf{Elements}$, and let $N \in \mathbf{Namespaces}$, such that $N = (n, S)$, with n a name of e and $S \subset \mathbf{Elements}$, be the namespace in which e is contained, or the element e itself in its namespace representation if e is a top namespace, such that $N = space(e)$. A substitution of e by e' is defined by a function $substitute(e, e')$ such that:*

$$substitute : (\mathbf{Elements} \times \mathbf{Elements}) \rightarrow \mathbf{Elements}$$
$$(e, e') \mapsto \begin{cases} e' & \text{if } N = ns(e) \\ substitute(e'', e''') & \text{if } N \neq ns(e) \end{cases}$$

where e'' and e''' are such that $ns(e'') = N$ and $ns(e''') = (n, (S \setminus \{e\}) \cup \{e'\}))$.

Note that we can ensure that this recursion is well-founded. Indeed, each recursive call of the function $substitute$ is made on the "super-namespace" and the definition 3 ensure

that we will reach the condition $N = ns(e)$. Following the same principle, adding or deleting a model element requires to substitute the initial set to which the model element is added or from which it is deleted. The functions add and $delete$ can thus be defined using the recursive function $substitute$. Note that this shows that *« add »* and *« delete »* are syntactic sugar of *« substitute »*, as mentioned in Section 2.1.2.

Definition 29 (Addition)**.** *Let e be a UML model element and e' the set of additive model elements (which may contain only one element if only one element is added) to add in e, such that $e \in \mathbf{Elements}$ and $e' \subset \mathbf{Elements}$, and assume $ns(e)$ the namespace general representation of e, such that $ns(e) = (n, S)$, with n a name of e and $S \subset \mathbf{Elements}$. An addition of e' in e is defined by a function $add(e, e')$ such that:*

$$add : (\mathbf{Elements} \times \mathbf{Elements}) \rightarrow \mathbf{Elements}$$
$$(e, e') \mapsto substitute(e, e'')$$

where e'' is such that $ns(e'') = (n, S \cup e')$.

Again, as for the definition 23, the *merge* behavior is automatically supported by this definition.

Definition 30 (Deletion)**.** *Let e be a UML model element to delete from the model, such that $e \in \mathbf{Elements}$, and let $N \in \mathbf{Namespaces}$, such that $N = (n, S)$, with n a name of N and $S \subset \mathbf{Elements}$, be the namespace in which e is contained, or the element e itself in its namespace representation if e is a top namespace, such that $N = space(e)$. A deletion of e from N is defined by a function $delete(e)$ such that:*

$$delete : \mathbf{Elements} \rightarrow \mathbf{Elements}$$
$$e \mapsto \begin{cases} \emptyset & \text{if } N = ns(e) \\ substitute(e', e'') & \text{if } N \neq ns(e) \end{cases}$$

where $\emptyset \in \mathbf{Elements}$ is the empty model element and e' and e'' are such that $ns(e') = N$ and $ns(e'') = (n, S \setminus \{e\})$.

Thus, to completely apply a change modeled on a model instance, we execute the following. If the change is allowed, i.e. if the corresponding condition is evaluated to true and the consistency rules, defined in the next section, are fulfilled, we apply one of the definitions given above, which apply the change and integrate it to the rest of the model. We then need to update the tagged values to remove the information associated to the change that occurred. Let s be the stereotype modeling the change and v be the tagged value labelling the change, such that $v \in V$, with $V \equiv v(ref(s))$. The change identified by v is contained in the tagged values of type change of s, at the same position as v. Let k be that position, such that $k \in \mathbb{N}$, it can be calculated by $k = f(V, v)$. Thus let C be the set of values of the tag of type change given by $change(s)$, such that $C \equiv v(change(s))$, the change c to apply can be retrieved by $c = g(C, k)$. New tagged values can then be created for s, such that *{ ref = V' }* and *{ change = C' }*, where $V' \equiv V \setminus \{v\}$, $C' \equiv C \setminus \{c\}$ and the tag change can be a tag change, substitute, add or delete. These two tagged

values can replace the former one using the function $substitute$ defined above. Finally, the function ψ is updated such that $\psi(v) = true$. This is done automatically since v is removed from the instance values. Note that an element p of one of the list of the tags ref used as predicate in a condition of a stereotype of type change can be omitted on the diagram if $\psi(p) = true$.

Note that we do not describe the case of the composite changes and the extensions *« substitute-all »*, *« add-all »* and *« delete-all »*, since those type of evolutions will use the same concepts as the ones described above, but applied as many times as necessary to execute all the changes. For the extensions *« substitute-all »*, *« add-all »* and *« delete-all »*, the pair representing the change will be of the form (e, e') where e is not a model element, but **a set of model element**, such that $e = \{e_1, \ldots, e_n\}$ with $n \in \mathbb{N}$. Applying $substitute\text{-}all(e, e')$ is thus equivalent to applying $substitute(e_1, e'), \ldots, substitute(e_n, e')$. The extensions *« add-all »* and *« delete-all »* will follow the same principle.

UMLseCh models possible evolutions and thus, a same model element could be concerned by more than one change. If a change happens on a model element that was concerned by more than one possible evolution, the other possible changes concerning the same model element should be adapted. One possibility could be to remove those changes, hence to remove all the pair having as first element the element that was modified. Another possibility would be to consider that this new element can still evolve. In this case, the first element of all the pairs that were modeling a change concerning the modified model element has to be replaced by the new model element resulting from the change. This second solution allows the evolutions to evolve themselves with time. Note also that given how we modeled the changes, a roll-back function can be defined easily.

2.5 UMLseCh Formal Semantics

In this section, we describe the UMLseCh abstract syntax, which is an extension of the UMLsec abstract syntax that includes the UMLseCh stereotypes, as well as the results obtained from applying a change on the several diagrams and the rules that should define whether a change preserves the consistency of the diagrams. Note that the abstract syntax of UMLseCh differs lightly from the UMLsec abstract syntax, but remains similar to it. In particular, some of the concepts defined for the behavioral semantics and the execution of the UML diagrams are ignored here since they are not necessary in the context of a change. However, the UMLseCh diagrams are still executable provided that the representations described above are adapted to the UMLsec behavioural semantics concepts and used in the context of the UML Machines. For example, an operation of a Class is simply considered as an operation in the following. However, it is easy to consider it as a message, as it is the case in the UMLsec abstract syntax, since they use the same representation, i.e a 3-tuple $O = (oname, args, otype)$. Other elements will also be extended, but none of these extensions will affect the execution of the diagrams. UMLseCh models can thus be used with the behavioral semantics and UML Machine

rules, defined in [19], and be executed.

The concept of stereotypes is also used differently in the UMLseCh abstract syntax. More concretely, the stereotypes attached to model elements in the UMLsec abstract syntax are elements of the set of stereotypes name, defined as $\mathbf{StereoNm}$ in Section 2.2. For UMLseCh, the stereotypes will be stereotypes instances that belong to the set $\mathbf{Stereotypes}$[15]. Again, this will not be a problem regarding the execution of the UMLseCh diagrams since stereotype instances can easily be considered as stereotype names provided that the tagged values associated to a stereotype at the concrete level of the instance of a model are ignored. In particular, each instance of a UML element has only one instance of a given stereotype name. Again, in the following, by "stereotype", we will always mean "stereotype instance" and the use of stereotype definitions will be mentioned explicitly.

Note that the abstract syntax described in the following defines the representation of the UML elements and diagrams using n-tuples and sets. Therefore, all the changes will be executed by modifying those sets and n-tuples directly. Generally, a set A will be modified by $(A\setminus\{e\})\cup\{e'\}$, $A\cup\{e'\}$ or $A\setminus\{e'\}$, with $e, e' \in \mathbf{Elements}$, for a substitution, an addition or a deletion respectively. Formally, once the set is modified, it can be integrated to the rest of the model with the concepts described in the previous section. The modification of the element of a tuple can be expressed easily as well. We will thus not describe all of the changes explicitly and formally, these applications being trivial. Instead, we describe informally the elements concerned by changes and the changes that could require extra changes. The formal rules ensuring the consistency are also given.

2.5.1 General principles

Some concepts and consistency rules are applicable to each type of diagram defined below and thus are presented here in a general context. At first, note that UMLseCh inherits all of the representation and consistency rules from UMLsec. Therefore, any change applied on an UMLseCh model should preserve those rules and definitions so that the resulting model is an UMLsec (and thus UMLseCh) compliant model. In other words, the principles and consistency rules described in the next sections are not exclusive, but added to the existing UMLsec rules and conditions. To allow the application of a modeled evolution, all of the conditions must be fulfilled.

All of the elements used in UMLseCh diagrams are UML named elements, which are defined as elements with an optional attribute *name* of type String. The name of an element hence cannot be modified, since it is not a NamedElement and thus cannot be used as a tagged value of a stereotype *« substitute »*, *« add »*, *« delete »* and their extensions. This limitation however does not affect the efficiency of the UMLseCh notation since changing the name of an element does not represent an important and likely modification of a model.

The diagrams and model elements use sets in their representation. The sets cannot contain duplicate elements and thus, it is not allowed to add an element that already exists

[15]Note that the set $\mathbf{StereoNm}$ is called $\mathbf{Stereotypes}$ in [19], but is not to be confounded with the set $\mathbf{Stereotypes}$ defined here!

or to substitute an element by an element that already exists. Substituting an element by another element that is already contained in the set would indeed be equivalent to a deletion of the model element initially concerned by the substitution. Applying an addition of a model element that is already in the set would simply leave the model unchanged. Again, adding an element that already exists or substituting an element by an existing one represent undesirable types of evolutions and thus this limitation does not affect the efficiency of the UMLseCh notation. Similarly, if an element has another element in its representation that is a simple element and not a set, it is not allowed to add such an element if the concerned model element already has one. For example, it is not allowed to add a guard on a transition if this transition already has a guard.

Finally, a model element can only have one occurrence of a stereotype definition and this stereotype must have the model element to which it is attached as base class. Thus, when substituting a stereotype s by a stereotype s', or when adding a stereotype s', on a model element E, such that $s, s' \in \mathbf{Stereotypes}$, $E \in \mathbf{Elements}$ and $\mathsf{stereo}(E)$ is the set of stereotypes of E, we verify:

$$\nexists st : st \in \mathbf{Stereotypes}, \tau(st) = \tau(s'),$$

where τ is the function defined in definition 5. The second condition cannot be formally verified since base classes are not defined in our abstract syntax. One should thus always ensure that the modification respects the base class definition of the stereotype placed on the model by the application of the modeled evolution.

2.5.2 Object Diagrams

2.5.2.1 Abstract Syntax of Object Diagrams

For an object, the difference from the UMLsec abstract syntax is that the set $stereo$ of stereotypes name is now a subset of stereotype instances, simply called "stereotypes", as mentioned above. We can thus represent an object as a 6-tuple $O = (oname, cname, stereo, aspec, ospec, int)$ where $oname$, $cname$, $aspec$, $ospec$ and int represent the same elements as for UMLsec and $stereo \subseteq \mathbf{Stereotypes}$ is a set of stereotypes (as opposed to UMLsec where they were stereotype definitions). An interface can also evolve and thus integrate UMLseCh stereotypes. It is hence of the form $I = (iname, ospec, stereo)$ where $iname \in \mathbf{String}$ is the interface name, $ospec$ a set of operation specifications and $stereo \subseteq \mathbf{Stereotypes}$ a set of stereotypes.

Dependencies are also adapted to be able to evolve and thus have a set of stereotype that could potentially contain stereotypes of type change. However, a stereotype definition $stereo \in \{\text{«}\textit{call}\text{»}, \text{«}\textit{send}\text{»}\}$ is defined as in the case of UMLsec. This allows us to verify certain constraint concerning call and send operations defined in the UMLsec formal semantics. Refer to [19] for more details about these constraints. A dependency can thus be defined as a tuple $d = (dname, dep, indep, int, stereo, stereoch)$ where $stereoch \subseteq \mathbf{Change}$ is a set of stereotypes of type change. The other elements of d have the same meaning as for UMLsec.

An object diagram is thus a pair $O = (\mathsf{Objects}(D), \mathsf{Dep}(D))$ given by a set $\mathsf{Objects}(D)$ of

objects and a set $\mathrm{Dep}(D)$ of dependencies. The same conditions as for UMLsec hold here.

2.5.2.2 Application of a Change

For object diagrams, the following model elements are concerned by changes: objects, stereotypes, attributes, operations, dependencies and interfaces. An operation can be substituted by another, added on an object or deleted from an object. However, elements from an operation, namely, the name, the return type and the set of arguments, cannot be changed. This choice is motivated by the fact that such changes concerns small part of the model elements and stereotypes cannot be directly attached to operations. Therefore, modifying those elements directly would require the stereotype attached to the object and modeling the change to precisely target the element to modify. This would request more complex expressions since different operations may have the same representation for those elements. For example, many operations could have Integer as return type. The stereotype would thus need to express precisely which return type Integer has to be modified. Such elements hence cannot be changed directly and the change of the complete operation will be used instead. This means of modeling does not require extra efforts and avoid the obligation to use complex expression to represent the element to modify.

An object can be substituted by another, added on the diagram or deleted from the diagram. In addition, the elements of the object can be modified. Concretely, stereotypes, operations, attributes and interfaces can be substituted, added or deleted. Note that, for the same reasons as the operations, the type of an attribute cannot be modified. The set of operations of an interface can be modified as well. A dependency can be substituted by another, added on the diagram or delete from the diagram. Any of the elements of a dependency can be modified as well. Note that dependencies are dependent to objects and thus, if an object that is a target or a source of a dependency is substituted by another, the dependency must be adapted. This adaptation means to also replace the source or target object by the new one in the dependency.

2.5.2.3 Preservation of the Consistency

When modifying an Object diagram, several consistency rules should be preserved after the modification. In the following, we present conditions that must be fulfilled to allow a change so that it preserves the consistency of the diagram.

Since the name of the objects must be mutually distinct, the following constraint must be verified to allow a substitution of an object e by an object e' in a diagram D, with $oname_{e'}$ the name of e':

$$\nexists o : o \in (Objects(D) \setminus \{e\}) : oname_o = oname_{e'}$$

and to allow an addition of an object e' in D:

$$\nexists o : o \in Objects(D) : oname_o = oname_{e'}$$

where $oname_o$ is the name of o. For obvious reasons, it is not necessary when deleting an object. However, an object o, with $oname_o$ the name of o, cannot be deleted if it is the source or the target of a dependency. Precisely, we verify:

$$\nexists d : d \in Dep(D) : oname_o = dep_d \vee oname_o = indep_d$$

where $d = (dname_d, dep_d, indep_d, int_d, stereo_d, stereoch_d)$.
A substitution of a dependency d by a dependency d' or an addition of a dependency d' is possible only if d' *connects* two existing objects o and o', with $oname_o$ and $oname_{o'}$ the names of o and o' respectively. Formally, before applying the change, we verify:

$$\exists o, o' : o, o' \in Object(D), (dep_{d'} = oname_o) \wedge (indep_{d'} = oname_{o'})$$

where $d' = (dname_{d'}, dep_{d'}, indep_{d'}, int_{d'}, stereo_{d'}, stereoch_{d'})$ is the substitutive or additive dependency. Adding or modifying an operation to an interface requires that this operation also exists in the object o to which the interface belongs. Formally, before each change on the set of operation of an interface int, we check:

$$ospec_{int} \subseteq ospec_o$$

where $ospec_{int}$ and $ospec_o$ are the sets of operations of int and o respectively. In addition, it is not allowed to modify or delete an operation op of an object o if this operation is also defined in an interface i of the set int of interfaces of o. Formally, we have:

$$\forall i : i \in int : \nexists op' : op' \in ospec_i : op' = op.$$

where $i = (iname_i, ospec_i, stereo_i), \forall i \in int$.

2.5.3 Class Diagrams

2.5.3.1 Abstract Syntax of Class Diagrams

Again, Class diagrams are very similar to Object diagrams. A class is defined as an object $C = (oname, cname, stereo, aspec, ospec, int)$ where $oname$ is the empty string.

A class diagram is defined as a pair $D = (\mathsf{Classes}(D), \mathsf{Dep}(D))$ given by a set $\mathsf{Classes}(D)$ of classes and a set $\mathsf{Dep}(D)$ of dependencies. Again, we require that the names of the classes are mutually distinct.

2.5.3.2 Application of a Change

The application of a change in a Class diagram will follow the exact same principles as the ones defined above for Objects diagrams.

2.5.3.3 Preservation of the Consistency

The rules defined for the Objects diagrams also apply here. In addition, the names of the different classes must be mutually distinct. Therefore, to allow a substitution of a class

c by a class c' in a diagram D, with $oname_{c'}$ the name of c', we verify if the following condition is fulfilled:

$$\nexists cl : cl \in (Class(D) \setminus \{c\}) : oname_{cl} = oname_{c'}$$

and for an addition of a class c' in D:

$$\nexists cl : cl \in Class(D) : oname_{cl} = oname_{c'}$$

where $oname_{cl}$ is the name of the class cl.

2.5.4 Statechart Diagrams

2.5.4.1 Abstract Syntax of Statechart diagrams

For statechart diagrams, the only difference with UMLsec abstract syntax is that several elements have a set of stereotypes that can include stereotypes of type change. A state S includes this set of stereotypes and is given by
$s = (\mathsf{name}(S), \mathsf{entry}(S), \mathsf{state}(S), \mathsf{internal}(S), \mathsf{exit}(S), \mathsf{stereo})$, where
$\mathsf{stereo} \subseteq \mathbf{Stereotypes}$ is a set of stereotypes. $\mathsf{name}(S)$, $\mathsf{entry}(S)$, $\mathsf{state}(S)$, $\mathsf{internal}(S)$ and $\mathsf{exit}(S)$ have the same meaning as in UMLsec.

A transition is defined as $t = (\mathsf{source}(t), \mathsf{trigger}(t), \mathsf{guard}(t), \mathsf{effect}(t)\ ,\ \mathsf{target}(t), \mathsf{stereo})$ where $\mathsf{stereo} \subseteq \mathbf{Stereotypes}$ is a set of stereotypes and the other elements of t are the same as in UMLsec.

Statemachines differ from other type of diagrams (such as class or deployment diagrams) by being themselves namespaces or model element that are contained in namespaces. They can also appear more than once in a subsystem. In consequence, the statemachines can have their own set of stereotypes. We thus define a statechart diagram as $D = (\mathsf{Object}_D, \mathsf{States}_D, \mathsf{Top}_D\ , \mathsf{Transitions}_D, \mathsf{stereo})$, given by an object name Object_D providing the context of the statemachine by associating it to another element of the model, a set of states State_D, a top state Top_D, containing all the states of D as substates, possibly in a nested way, a set $\mathsf{Transitions}_D$ of transitions, and a set of stereotypes stereo.

2.5.4.2 Application of a Change

For statechart diagrams, the elements that can be modified, added or deleted are states, transitions and properties of states and transitions. All the types of elements of statechart diagrams are thus concerned by evolutions.

Transitions between states are also dependent on the potential modification of the source or the target state. In particular, if a state is substituted by another state, the transitions having that state as a source or target should be adapted.

2.5.4.3 Preservation of the Consistency

On statemachines, the specific elements concerned by changes that could affect the consistency are states and transitions. When a transition t is substituted by t' or when a transition t' is added, for a statemachine D, the following condition must be fulfilled to ensure that the new transition has correct source and target states:

$$\exists s_1, s_2 : s_1, s_2 \in \mathsf{State}_D, (\mathsf{source}(t') = s_1) \wedge (\mathsf{target}(t') = s_2)$$

In addition, we must ensure that if a transition t is substituted by t', such that $\mathsf{source}(t) \neq \mathsf{source}(t')$ or $\mathsf{target}(t) \neq \mathsf{target}(t')$, the connected states will still have at least one incoming transition and one outgoing transition after the change occurerd. Thus to allow the substitution, we verify the following condition:

$$\exists t_1, t_2, t_3, t_4 : t_1, t_2, t_3, t_4 \in (\mathsf{Transition}_D \setminus \{t\}) \cup \{t'\},$$

$$\begin{array}{ll} ((\mathsf{source}(t_1) = \mathsf{source}(t)) & \wedge \\ (\mathsf{target}(t_2) = \mathsf{source}(t)) & \wedge \\ (\mathsf{source}(t_3) = \mathsf{target}(t)) & \wedge \\ (\mathsf{target}(t_4) = \mathsf{target}(t))) & \end{array}$$

Note that the above condition has to be refined if $\mathsf{source}(t)$ is an initial state or if $\mathsf{target}(t)$ is a final state, since an initial state has no incoming transition and a final state has no outgoing transition. This can easily be done by removing the condition $\mathsf{target}(t_2) = \mathsf{source}(t)$ or the condition $\mathsf{source}(t_3) = \mathsf{target}(t)$ from the above condition, or both in the unlikely case that the transition connects the initial state directly to the final state.

Deleting a transition from a statemachine D is allowed only if the source and target states of that transition have other incoming and outgoing transitions, so that they are not isolated by themselves on the diagram. Assume the transition t to delete and $\mathsf{source}(t) = s_1$ and $\mathsf{target}(t) = s_2$, with $s_1, s_2 \in \mathsf{State}_D$, the following condition thus has to be verified:

$$\begin{array}{ll} (\exists t_1, t_2 : t_1, t_2 \in (\mathsf{Transition}_D \setminus \{t\}) : & (\mathsf{source}(t_1) = s_1 \wedge \mathsf{target}(t_2) = s_1)) \wedge \\ (\exists t_3, t_4 : t_3, t_4 \in (\mathsf{Transition}_D \setminus \{t\}) : & (\mathsf{source}(t_3) = s_2 \wedge \mathsf{target}(t_4) = s_2)) \end{array}$$

Initial and final states are not concerned by this situation.

To be consistent, a statemachine must have at most one initial state per "level". By level, we mean a set of state without the substates or the super-states. To verify this constraint, the following must be fulfilled:

$$\forall S : S \in \mathsf{State}_D, \nexists s_1, s_2 : s_1, s_2 \in \mathsf{states}(S), (s_1 \neq s_2) \wedge (s_1, s_2 \in \mathsf{Initial}_D)$$

Note that this condition also verifies the first level of the statemachine, since this level is represented by $\mathsf{state}(\mathsf{Top})_D$ and $\mathsf{Top}_D \in \mathsf{State}_D$. In addition, an initial state cannot have incoming transitions and a final state cannot have outgoing transitions, which is verified by the constraints $\mathsf{source}(t) \notin \mathsf{Final}_D \cup \mathsf{Top}_D$ and $\mathsf{target}(t) \notin \mathsf{Initial}_D \cup \mathsf{Top}_D$ respectively. Note that a statemachine can have more than one final state. Adding or deleting a state directly is impossible without affecting the consistency of the statemachine. Such a change will

require workarounds as presented in Section 2.1.5.

2.5.5 Sequence Diagrams

2.5.5.1 Abstract Syntax of Sequence diagrams

A lifeline of a sequence diagram is extended from UMLsec abstract syntax by adding a set of stereotypes. A lifeline is thus defined as a 3-tuple (O, C, stereo), given by:

- an object O of class C; and
- a set $\text{stereo} \subseteq \mathbf{Stereotypes}$ of stereotypes

The set of lifelines of a sequence diagram D is called $\text{Obj}(D)$. Connections are extended in the same way with a set of stereotypes. A connection is thus defined as a 5-tuple $l = (\text{source}(l), \text{guard}(l), \text{msg}(l), \text{target}(l), \text{stereo})$ where $\text{stereo} \subseteq \mathbf{Stereotypes}$ is a set of stereotypes and $\text{source}(l)$, $\text{guard}(l)$, $\text{msg}(l)$ and $\text{target}(l)$ have the same meaning as in UMLsec.

As for statemachines, a subsystem can have more than one sequence diagram. They are represented by the construct Interaction in UML [44] and therefore represent namespace as well. We thus add a set of stereotypes to sequence diagrams so that they can be concerned by their own evolution. A sequence diagram then simply defined as a pair $D = (\text{Obj}(D), \text{Cncts}(D))$.

2.5.5.2 Application of a Change

The elements concerned by a change in a sequence diagram are lifelines, connections and messages. By connection, we mean the arrow drawn between lifelines on the diagram, which is the source and target of the connection described in the abstract syntax of sequence diagrams presented here. By message, we mean the message on the arrow, which is the message contained in the connection. A message can be substituted, but cannot be added or deleted directly. To add (resp. delete) a message, it is necessary to add (resp. delete) a connection. When a lifeline is substituted by another lifeline, we assume that all the connections are adapted. The adaptation means that if any connection has the substituted lifeline as a source or target, this source or target in the connection is replaced by the new lifeline.

2.5.5.3 Preservation of the Consistency

To substitute a connection c by a connection c', or add a connection c' on a sequence diagram D, at least one of the two objects, one representing the source object and one target object, must be an object of the sequence diagram. Before applying the substitution or the addition, we thus verify the following rule:

$$\exists o, o' : o, o' \in \text{Obj}(D), (source(c') = o) \vee (target(c') = o')$$

To substitute a lifeline l by a lifeline l', or add a lifeline l' on a sequence diagram D, such that $l' = (O, C, s)$ we must ensure that there is not another lifeline with the same object and class and that the object exists. The first constraint can be expressed easily by:

$$\nexists ls : ls \in \mathsf{Obj}(D), ls = (O', C', s'), O = O' \wedge C = C'.$$

For the second rule, assume an object diagram OD, we verify the following constraint:

$$\exists ob : ob \in \mathsf{Objects}(OD), ob = O.$$

Finally, deleting a lifeline is not allowed if connections have this lifeline as source or target object. Formally, assume the lifeline l to delete from the diagram D, the following constraint must be fulfilled to allow the change:

$$\nexists c : c \in \mathsf{Cncts}(D), (source(c) = l) \vee (target(c) = l).$$

2.5.6 Activity Diagrams

2.5.6.1 Abstract Syntax of Activity diagrams

As for UMLsec, activity diagrams are presented as a special type of statechart diagrams. In particular, any construct of our simplified version of activity diagrams can be expressed using the concepts of statechart diagrams. However, as opposed to statemachines, only one activity diagram is defined per subsystem. Thus an activity diagram is a 3-tuple $D = (\mathsf{States}_D, \mathsf{Top}_D, \mathsf{Transitions}_D)$ given by a finite set of states States_D, the top state $\mathsf{Top}_D \in \mathsf{States}_D$, and a set $\mathsf{Transitions}_D$. Again, the set States_D is disjointly partitioned into the sets $\mathsf{Initial}_D$, Final_D, Simple_D, Conc_D, Sequ_D. A state is extended in UMLseCh by adding a set of stereotypes to it. We have $S \in \mathsf{State}_D$ where $\mathsf{stereo} \subseteq \mathbf{Stereotypes}$ is a set of stereotypes and $\mathsf{name}(S)$, $\mathsf{entry}(S)$, $\mathsf{state}(S)$, $\mathsf{internal}(S)$, $\mathsf{exit}(S)$ and $\mathsf{swim}(S)$ have the same meaning as in UMLsec.

The transitions are also extended by adding a set of stereotypes to it. A transition $t \in \mathsf{Transitions}_D$ is given by:

- the source state $\mathsf{source}(t) \in \mathsf{States}_D$ of t;
- the guard $\mathsf{guard}(t)$ of t;
- the target state $\mathsf{target}(t) \in \mathsf{States}_D$ of t; and
- a set $\mathsf{stereo} \subseteq \mathbf{Stereotypes}$ of stereotypes.

2.5.6.2 Application of a Change

The constructs of activity diagrams are the same as the ones of statechart diagrams, with an additional concept of swim lane. Changes of swim lanes cannot be modeled by the UMLseCh stereotypes and therefore are not concerned here. If a state is substituted by another state, we assume that the substitutive state has the appropriate swimlane.

2.5.6.3 Preservation of the Consistency

The rules are the same as for statechart diagrams. For an addition of a state, however, we ensure that the swimlane specified in the additive state refers to an existing object. Formally, for an activity diagram AD, an object diagram OD and an additive state S, such that $\mathsf{swim}(S) = o$, we verify:

$$\exists o' : o' \in \mathsf{Obj}(OD), o = o'.$$

2.5.7 Deployment Diagrams

2.5.7.1 Abstract Syntax of Deployment diagrams

For deployment diagrams, we extend the nodes and the components so that they include a set of stereotypes. Formally, a component is a 4-tuple
$C = (name, int, cont, stereo)$ where $name$ is the component name, int is a set of interfaces that can possibly be empty, $cont$ is the set of subsystem instances and object names contained in the component, and $stereo \subseteq \mathbf{Stereotypes}$ is a set of stereotypes. A node is a 3-tuple $N = (loc, comp, stereo)$ where $stereo \subseteq \mathbf{Stereotypes}$ is a set of stereotypes and loc and $comp$ have the same meaning as in UMLsec.

We extend links so that they include a set of stereotypes. A link l is of the form $l = (\mathsf{nds}(l), \mathsf{ster}(l))$ where $\mathsf{nds}(l) \subseteq \mathsf{Nodes}(D)$ is a set of arity two containing the nodes being linked and $\mathsf{ster}(l) \subseteq \mathbf{Stereotypes}$ is a set of stereotypes. A dependency is also extended in the same way. Formally it is a 4-tuple $d = (clt, spl, int, stereo)$ where $stereo \subseteq \mathbf{Stereotypes}$ is a set of stereotypes and clt, spl and int have the same meaning as in UMLsec.

As for UMLsec, for every dependency $D = (C, S, I, sd)$ there is exactly one link $L_D = (N, sl)$ such that $N = \{C, S\}$. A deployment diagram is given by
$D = (\mathsf{Nodes}(D), \mathsf{Links}(D), \mathsf{Dep}(D))$ where $\mathsf{Nodes}(D)$ is a set of nodes, $\mathsf{Links}(D)$, is a set of links and $\mathsf{Dep}(D)$ is a set of dependencies.

2.5.7.2 Application of a Change

The elements that can evolve on a deployment diagram are the nodes, the components, the links and the dependencies. When a node is substituted by another one, we assume that the possible links connecting that node to another node are adapted. The same application is assumed for dependencies when substituting a component by another one.

2.5.7.3 Preservation of the Consistency

The substitution of a link e by a link e', or the addition of a link e', on a deployment diagram D requires the source and target of e' to exist on D. This requirement can be verified by

the following constraint:

$$\exists n, n' : n, n' \in \mathsf{Nodes}(D), (source(e') = n) \wedge (target(e') = n'),$$

Similarly, to substitute a dependency d by a dependency d' or to add a dependency d' on a deployment diagram D, we verify:

$$\exists n, n' : n, n' \in \mathsf{Nodes}(D), \exists c : c \in comp_n, \exists c' : c' \in comp_{n'}, (ctl_{d'} = c) \wedge (spl_{d'} = c').$$

where $n = (loc_n, comp_n, stereo_n)$, $n' = (loc_{n'}, comp_{n'}, stereo_{n'})$
and $d' = (clt_{d'}, spl_{d'}, int_{d'}, stereo_{d'})$.

Finally, a node cannot be deleted if a link connects it to another node. In the same way, a component cannot be deleted if a dependency connects it to another component. Formally, before deleting a node n or a component c from a diagram D, we verify:

$$\nexists l : l \in \mathsf{Links}(D), n \in \mathsf{nds}(l),$$
$$\nexists d \in \mathsf{Dep}(D), d = (clt_d, spl_d, int_d, stereo_d), (clt_d = c) \vee (spl_d = c).$$

2.5.8 Subsystem

2.5.8.1 Abstract Syntax of Subsystem

As for UMLsec, by subsystem, we always mean subsystem instance. We extend the representation of a subsystem by adding a set of stereotypes and a set of namespaces to it. Recall that the namespaces are given in their general representation and only provide a means of storing complex substitutive or additive elements. The existing elements of the subsystem, although they are themselves namespaces, are thus not concerned by that set. We define a subsystem as a tuple
$\mathcal{S} = (\mathsf{name}(\mathcal{S}), \mathsf{Op}(\mathcal{S}), \mathsf{Ints}(\mathcal{S}), \mathsf{Ssd}(\mathcal{S}), \mathsf{Dd}(\mathcal{S}), \mathsf{Ad}(\mathcal{S}), \mathsf{Sc}(\mathcal{S}), \mathsf{Sd}(\mathcal{S}), \mathsf{Nms}(\mathcal{S}), \mathsf{stereo})$ is given by:

- the name $\mathsf{name}(\mathcal{S})$ of the subsystem;
- a set $\mathsf{Op}(\mathcal{S})$ of names of offered operations and accepted signals, this set can be empty;
- a set $\mathsf{Ints}(\mathcal{S})$ of subsystem interfaces, this set can be empty;
- a static structure diagram $\mathsf{Ssd}(\mathcal{S})$;
- a deployment diagram $\mathsf{Dd}(\mathcal{S})$;
- an activity diagram $\mathsf{Ad}(\mathcal{S})$;
- for each of the activities in $\mathsf{Ad}(\mathcal{S})$, a corresponding specification of the behavior of objects appearing in $\mathsf{Ssd}(\mathcal{S})$ given by a set $\mathsf{Sc}(\mathcal{S})$ of statechart diagrams, a set

of sequence diagrams $\mathsf{Sd}(\mathcal{S})$, and the subsystems in $\mathsf{Ssd}(\mathcal{S})$. Each diagram $D \in \mathsf{Sc}(\mathcal{S}) \cup \mathsf{Sd}(\mathcal{S})$ has an associated name $\mathsf{context}(D)$. In the concrete syntax, it is written next to it;

- A set $\mathsf{Nms}(\mathcal{S})$ of namespaces containing substitutive or additive model elements; and
- stereo of stereotypes.

The subsystems follow the same criteria and conditions as the ones presented in UMLsec. In addition, we define the following condition to ensure that for each complex change modeled on the instance of a model, there exists a namespace containing the substitutive or additive complex element. Assume the set $\mathbf{TagChange}_{\mathcal{S}}$ of instances of tag of type change on the subsystem $\mathcal{S}$, we have:

$$\forall t \in \mathbf{TagChange}_{\mathcal{S}}, \forall e \in \upsilon(t), \exists ns = (n, elst), ns \in \mathsf{Nms}(\mathcal{S}), n = e$$

2.5.8.2 Application of a Change

In UML, diagrams provide a means to graphically represent systems by grouping together graphical representations of elements of same type and context. However, diagrams in UML are not model elements and therefore do not have a name and associated stereotypes. This means that elements of a diagram are directly contained in packages. In our case, the model elements of the diagrams defined above are directly contained in the subsystem containing the diagrams. Thus if one wants to model general evolution on a diagram, such as the addition of a class, the substitution of all the links of a deployment diagram or the addition of a statemachine, the corresponding stereotypes will be attached to the subsystem. The statechart diagrams and sequence diagrams are nevertheless different since they are considered as model elements, under the construct Statemachine and Interaction respectively. To apply a modification to those diagrams, such as adding an element to it, the corresponding UMLseCh stereotypes can be directly attached to them.

All elements from a subsystem can theoretically be modified by a UMLseCh stereotype attached to that subsystem, provided that the element is precisely identified. However, certain of these modifications can also be modeled by attaching a UMLseCh stereotype to a sub-element containing the evolutive element. This latter solution should be used whenever it's possible since it is more direct and it increases the readability. The following evolution can, on the other hand, be modeled only by a stereotype attached to a subsystem itself: the addition of elements contained in the diagrams defined above; the addition of a statemachine or a sequence diagram and the substitution; addition or deletion of a subsystem operation; signal or interface. Recall that these modifications can only be modeled by attaching UMLseCh stereotypes to the subsystem, but other modifications, such as modifying an operation of a class, can also be modeled in such a way by giving an expression that precisely identifies the sub-element to modify. These evolutions can thus either be modeled by attaching the UMLseCh stereotypes to the subsystem or to the sub-namespace containing the evolutive element. Again the first solution should be used whenever possible.

2.5.8.3 Preservation of the Consistency

All the rules concerning modifications of elements that belong to diagrams described above were defined in the preservation of the consistency of those diagrams. We can however add some rules related to evolutions that could possibly affect the consistency of subsystems. These rules involve statemachines and sequence diagrams, as well as the operations, signals and interfaces of the subsystems.

Since each activity of an activity diagram has one, and only one, associated behavior, being in the form of a statemachine or a sequence diagram, the substitution of this behavior should be substituted by another behavior associated to the same activity. Therefore, before substituting a statemachine or a sequence diagram s of a subsystem $\mathcal{S}$, such that $s \in \mathsf{Sc}(\mathcal{S}) \cup \mathsf{Sd}(\mathcal{S})$, by a statemachine or a sequence diagram s', we verify:

$$\mathsf{context}(s') = \mathsf{context}(s)$$

Provided that the subsystem is in a consistent state before the modification, and in particular that it has a behavior defined for each activity, this condition is sufficient to ensure the consistency of the subsytem after the evolution. If we consider that an activity can temporarily have no behaviour and that the evolution consists in adding that behavior, the following constraint is defined to ensure that an activity has only one associated behavior. Before adding a statemachine sm or a sequence diagram sd on a subsystem $\mathcal{S}$, with $\mathsf{Sc}(\mathcal{S})$ the set of statechart diagrams of $\mathcal{S}$ and $\mathsf{Sd}(\mathcal{S})$ the set of sequence diagrams of $\mathcal{S}$, we verify:

$$\nexists e : e \in \mathsf{Sc}(\mathcal{S}), \mathsf{contex}(e) = \mathsf{contex}(sm')$$

and

$$\nexists s : s \in \mathsf{Sd}(\mathcal{S}), \mathsf{contex}(s) = \mathsf{contex}(sd').$$

If an operation or a signal os of a subsystem $\mathcal{S}$, such that $os \in Op(\mathcal{S})$, is substituted by an operation or signal os', or if an operation or signal os' is added on a subsystem $\mathcal{S}$, we ensure that this operation or signal is defined in the static structure diagram of the subsystem. More precisely, if the static structure diagram is an object diagram OD, such that $\mathsf{Ssd}(\mathcal{S}) = OD$, we verify:

$$\exists ob : ob \in \mathsf{Objects}(OD), \exists i : i \in int_{ob}, os' \in ospec_i.$$

where ob is of the form $(oname_{ob}, cname_{ob}, stereo_{ob}, aspec_{ob}, ospec_{ob}, int_{ob})$ and i is of the form $(iname_i, ospec_i, stereo_i)$. If the static structure diagram is a subsystem $\mathcal{S}'$ such that $\mathsf{Ssd}(\mathcal{S}) = \mathcal{S}'$, we verify:

$$os' \in \mathsf{Op}(\mathcal{S}').$$

Finally, we describe informally some additional rules regarding the statemachines, the sequence diagrams and the deployment diagrams. A component C of a deployment diagram has a set $cont_C$ of objects and subsystem instances. Therefore, we must ensure that those objects and subsystem instances are defined within the subsystem $\mathcal{S}$, more precisely in the static structure diagram $\mathsf{Ssd}(\mathcal{S})$. Statechart diagrams and sequence diagrams use operations for triggering events and in messages respectively. Again, those operations must be defined within the subsystem $\mathcal{S}$ and in particular in the static structure

diagram $\mathsf{Ssd}(\mathcal{S})$.

2.5.9 Consistency of a Composite Change

Verifying the consistency of a diagram in the case of a composite change is slightly different. Indeed, a composite change can be compared to a transaction in a data base [13]. In particular, a composite change must fulfill the constraint of atomicity: the change is applied completely, or not at all. Therefore, the consistency is verified when all the sub-changes are applied on the model and not only one of them. The consistency rules defined above thus cannot simply be reused independently with each sub-change, since the consistency concerns the composite change as a whole. Note that the constraint of isolation and durability are implicitly verified here. No other change can take place while the composite change is applied and the modifications remain on the model. For the sub-changes occurring in parallel, we do not ensure isolation. This means that a user should avoid to model several sub-changes on the same model element if those sub-changes belong to the same composite change.

Two approaches can be considered in the context of a composite change. The first solution would be to apply the change and verify the consistency of the diagrams afterwards. The modification is then conceived and the change validated if the consistency is preserved, otherwise the model is rolled-back to the initial version. This solution however requires to apply the changes first and then roll-back if the consistency is not preserved. We will thus consider another approach following the same principle as the conditions defined above, i.e. to define conditions that will be verified before allowing the change. This approach consists in the following: the condition of each sub-change is verified assuming that the other changes were applied. Note that the changes did not really occur, but the sets concerned by the modifications were adapted to verify the condition. Note also that certain changes will be allowed in the case of a composite change although they were not permitted with the conditions described above. This is for example the case of a deletion of a state in a statemachine, which is forbidden in the case of a normal change because transitions are connected to that state. With a composite change, it can be allowed if other parallel changes consist in deleting the connected transitions in a way that leaves the resulting diagram consistent.

To illustrate the approach described above, we can consider the example shown in Figure 2.15. Deleting the final state would be forbidden for a single change, as defined in the consistency rules of Section 2.5.4. Deleting the transition between the state A and the final state is also forbidden by a condition of Section 2.5.4. However, assuming that a state can be deleted provided that no transitions are connected to it, as mentioned in the previous paragraph, the deletion of the final state can be allowed if the parallel change that consists in deleting the connected transition is applied. This can be formalized by assuming that the parallel change delete-transition happened and thus by adapting the set, such that the condition is defined as follows:

$$\nexists t : t \in (\mathsf{Transitions}_D \setminus t_f), (\mathsf{source}(t) = f) \vee (\mathsf{target}(t) = f)$$

where f is the deleted final state and t_f the transition from the the state A to f. We can see that adapting the set with $\mathsf{Transitions}_D \setminus t_f$ consists in assuming that the parallel

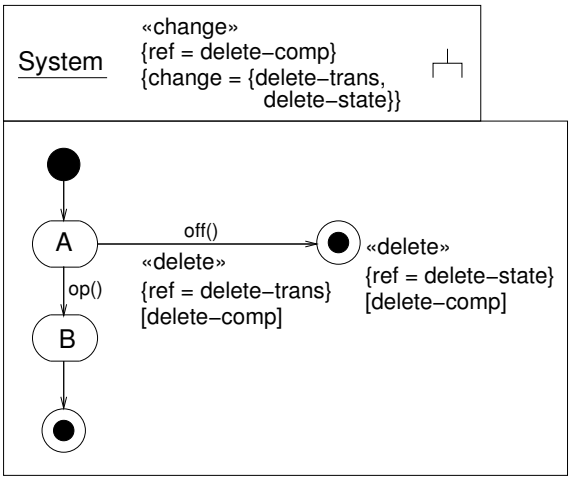


Figure 2.15: Example of an allowed composite change

change which removes t_f was applied. The condition for deleting the transition between A and the final state can be adapted in a similar way. Any condition from the previous sections defining the consistency rules can thus be adapted by simulating the other parallel sub-changes on the concern sets.

Note that simulating the change concretely, in a tool, could require as much resources as applying the change completely. An idea could be to specify a scope for a sub-change, which would define which parallel changes could affect the condition and thus have to be considered. This question however is left as future work. Verifying the consistency of an addition using the *merge* behavior defined in Section 2.1.4 will follow the same principle as the composite changes.

2.6 Related Work

Applying evolution and changes on models evidently requires to execute model transformations. In the following we give an overview of related work on Model transformation and Software evolution in general. To the extent of our knowledge, there is however 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, which is the scope of this Deliverable as we will see in the following chapters.

2.6.1 Model Transformation

Generally, transforming a model consists in taking a source model Ma conforming to a metamodel Mma and to produce a target model Mb conforming to a metamodel Mmb. There exist two types of model transformations: horizontal and vertical. If the level of abstraction of the target model is different from the one of the source model, the transformation is called vertical, otherwise, the transformation is called horizontal.

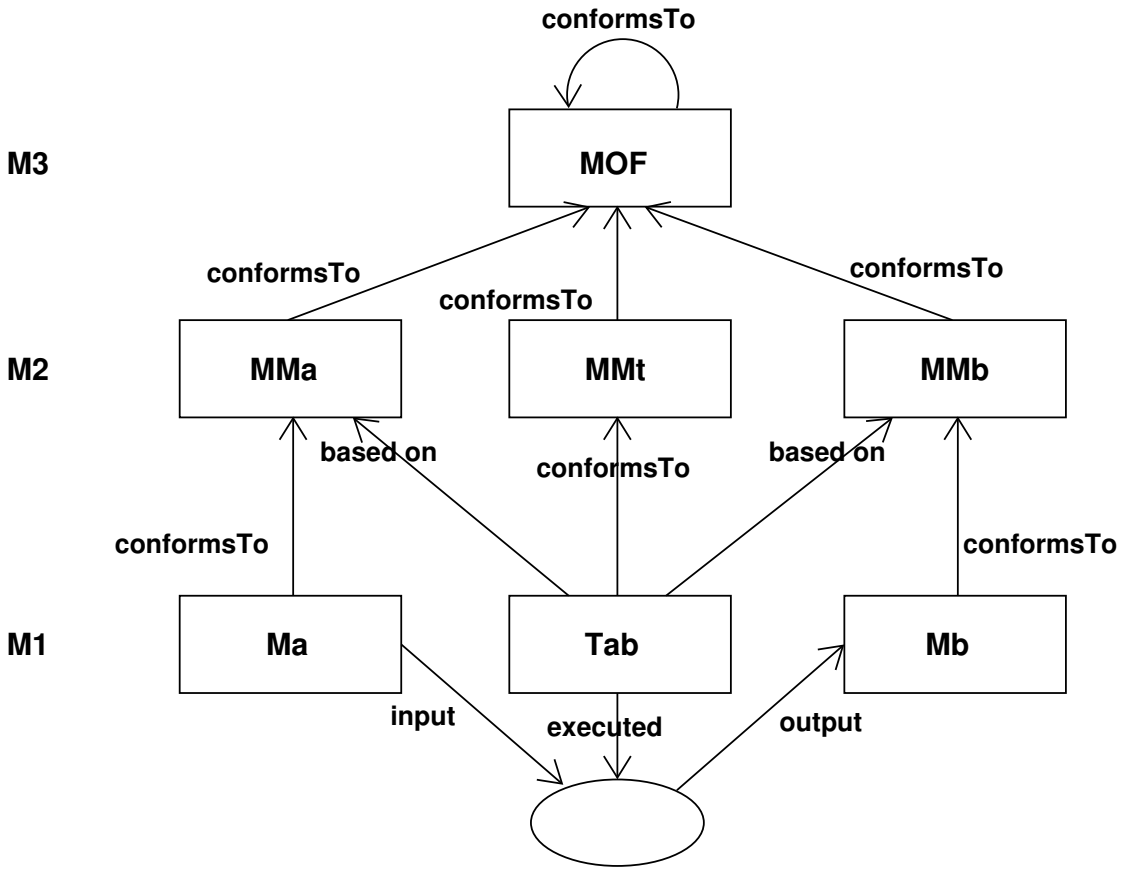


Figure 2.16: Context of ATL Transformation as shown in [17]

As mentioned in [53], the evolution of a model is often supported by model transformation rules written in specific languages. The OMG [14] defined a standard for expressing model transformations, called QVT (Query/View/Transformation) [43]. As mentioned in [17], the growing importance of model transformations led the OMG to publish a QVT request for proposal (RFP) [42]. Several propositions answered that RFP, which then evolved toward a single proposal [42]. Others however continued to develop independently, as for example the ATLAS Transformation Language (ATL) [30]. ATL is a model transformation language that provides a powerful abstract syntax as well as a concrete syntax. It offers the possibility to express model transformations rules for horizontal and vertical transformations from any source metamodel to any target metamodel. Since it was initially developed to answer the QVT RFC issued by the OMG, it shares common requirements with QVT [17]. The concept of model transformation with ATL is shown in Figure 2.16. The idea of software evolution can thus be formulated with ATL. For example, to add an operation to a UML Class, we can define a rule such as the one of Figure 2.17, where 'newOp()' is simply an abstract representation of the added operation to facilitate the example.

Tefkat [31] is another example model of a transformation language, which in this case is based on F-Logic [28]. An example of a Tefkat rule, taken from [31] where the complete

```
module example;
create OUT : UML from IN : UML;
rule addOp {
  from a:UML!Class
  to  b:UML!Class (
        name <- a.name,
        ownedAttribute <- a.ownedAttribute,
        ownedOperation <- a.ownedOperation
                              ->union(a.ownedAttribute->'newOp()')
        )
}
```

Figure 2.17: Simple example of a rule adding an operation in a UML class

```
      CLASS ClsToTbl {
        Class class;
        Table table;
      };
      RULE ClassAndTable(C,T)
        FORALL Class C {
                  is_persistent: true;
                  name: N;
                }
        MAKE Table T {
                name: N;
              }
        LINKING ClsToTbl WITH class = C,
 table = T;
```

Figure 2.18: Example of a transformation using Tefkat

transformation can be found, is shown in Figure 2.18.

For Tefkat as for ATL, they both represent the evolutions with rules using text-based notation. Therefore, they are not adapted to easily represent the model transformations on the diagrams of the high-level models. A graphical representation of the model transformation however would increase the readability. This issue is addressed by UMLX and Mola.

UMLX [55] is a graphical model transformation language that was also developed to answer the QVP RFP [42] issued by the OMG. It is based on the class diagram of UML. More precisely, it extends the class diagram to include notations that support inter-schema transformation. The syntax of the language is shown in Figure 2.19. It is thus possible to graphically represent the evolution of models using this language. Other constructs, such as constraints or multiplicity, can also be used in order to refine the transformation. UMLX hence provides a means to define a similar approach as the one presented in this document. However, several differences exist. UMLX provides an extended notation by adding new constructs and therefore extends the UML metamodel. A discussion about the consequences of extending the UML metamodel is given in Section 2.3. No

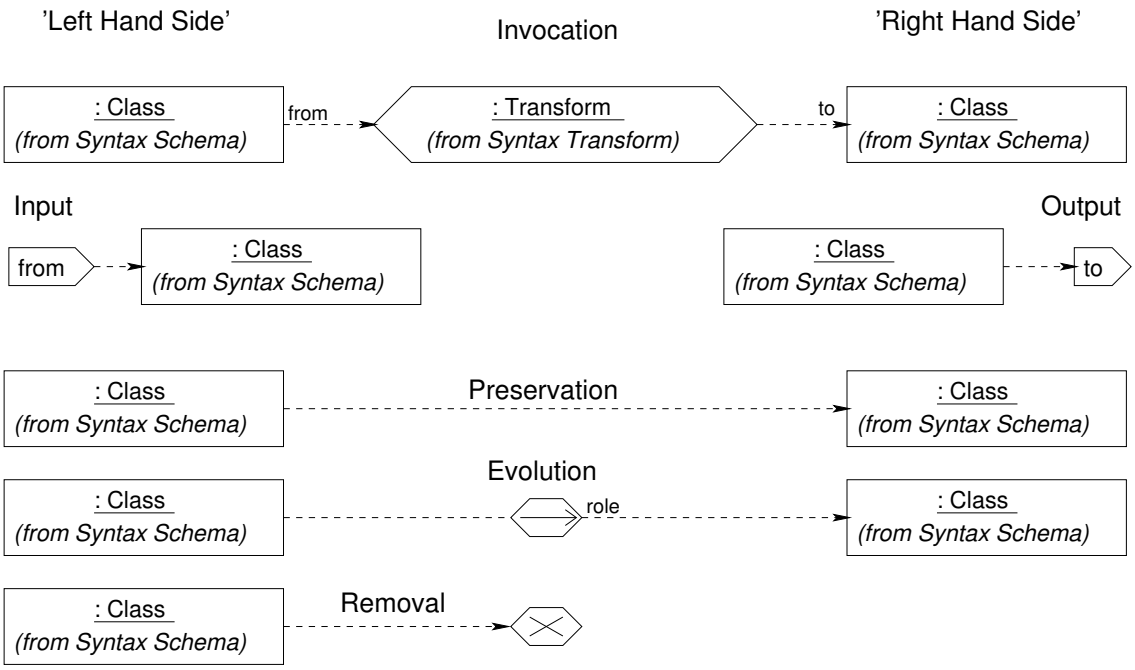


Figure 2.19: UMLX Transformation Syntax [55]

transformation operator exists in UMLX for an addition and thus, adding an element is not explicitly modeled. Instead, transformation shows the result with the new element added on the model. Finally, a main difference with the work described in this deliverable is that UMLX offers a graphical notation for model transformations based on the UML class diagram. This means that the representation will remain separated from the evolving models. For example, if the diagram being transformed is a state diagram or a deployment diagram, the transformation will be represented by an extended UMLX class diagram, but the notation will not be applied directly on the statemachine or deployment diagram. However, as opposed to the notation defined in this document, UMLX is a graphical representation of QVT transformation and thus allows a larger set of possibility, such as vertical transformations or transformations where the target model conforms to a different metamodel than the source model. An example of such transformations can be found in [54], where UMLX is used to model transformations from UML class diagrams to relational data bases (RDBMS).

MOLA (MOdeling transformation LAnguage) [27] is another graphical language for model transformations. It expresses the transformations with special constructs that are similar to structured flowcharts using the concept of pattern matching. A transformation is then represented as a MOLA program, which is a sequence of graphical statements linked by dashed arrows. It also introduces the concept of foreach loop, the most used kind of statement, which is graphically represented by a rectangle with a bold frame. Figure 2.20 shows an example of a MOLA program, described in [27], which builds a new W for each B that is linked to A, links this new W to the corresponding A with the association $roleW$ and assign the concatenation of the parameters of the corresponding A and B to the parameter of W. Note that this transformation occurs at the level of the instances of

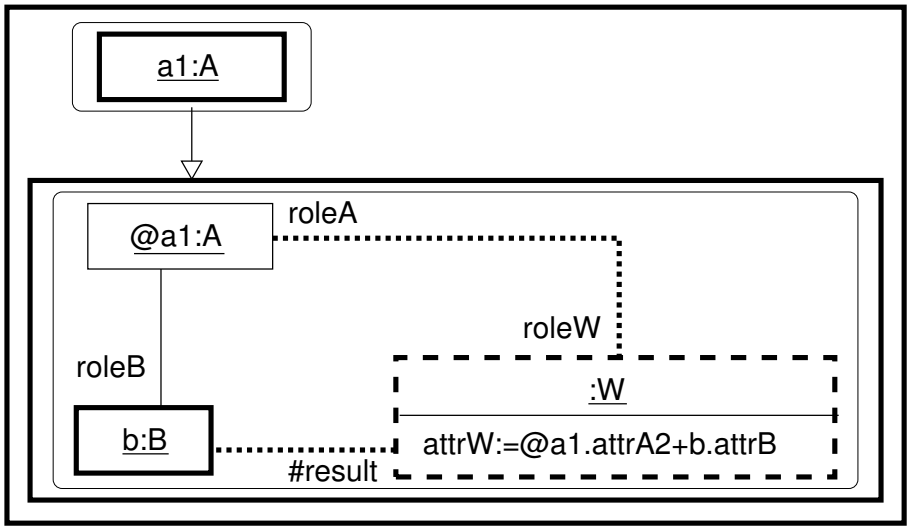


Figure 2.20: Example of a transformation using a MOLA program [27]

the model elements and thus, there might be more than one A or B, which explains the presence of the foreach loop. The classic example of the transformation of UML class diagrams to data base schemas can be found in [26]. The same comparison as the one between the approach defined in this deliverable and UMLX can be made for Mola.

2.6.2 Software Evolution

This section briefly presents the question of software evolution and some related work. Given that the amount of work being rather considerable, it does not aim to precisely describe every technique, approach or solution that exists for software evolution. Instead, we present a short introduction on how the importance of software evolution was raised, and why changes and evolvability of systems is still a challenge. We also mention a few solutions that support the evolution of software and show the difference that exist between those techniques and the approach developed in this Deliverable.

The importance of evolution in the process of developing large software was first discussed in the early seventies [33], after a study of the software process in 1968 [32]. Due to the recurring need of improvements, it was quickly stated that major programs that are commonly used are always incomplete such that they constantly undergo changes and evolution [35]. These modifications can be caused, for instance, by new requirements, changes in the requirements, corrections of errors or suppression of outdated elements. In a period of twenty years, eight laws of software evolution were defined, starting by the first three in [33] to the last two laws, first published in [34]. The first six laws were also revisited in [34]. In addition, it was stipulated in [38] that software keep growing and changing while [46] argued that software age and their quality thus degrades with time.

Although a considerable amount of work and research followed these facts in the past decades, the question of evolution in the software process remains a challenge today [39]. In particular, it is still considered that the evolution should be placed at the center of

the development process and the concept of change should be integrated in the software life-cycle [39]. Other challenges exist around the concept of change and evolution management, refer to [39] for the complete description. Nonetheless, we can present several achievements in the context of software evolution. One of the first and best known good practice that was defined to facilitate software changes and evolution is probably to anticipate change and design the systems in a way that allows evolvability. A key concept of such practice was the modularity of the system, such that a change to one component of the software should not affect (or affect as slightly as possible) the others [45]. However, [48] specified that there is not a single definition of evolvability and instead, defines evolvability as "a composite quality which allows a system's architecture to accommodate change in a cost effective manner while maintaining the integrity of the architecture". This definition is supported by a taxonomy of change which includes four properties: generality; adaptability; scalability and extensibility.

Studying the evolution of software has often been accomplished by an empirical observation of the software throughout its history [39]. However, to be able to interpret the collected data, [39] mentions the necessity of defining a *theory of evolution*, which follows the idea presented in [37] and [36]. On a smaller scale, a classification of 12 different types of software evolution and software maintenance was given in [8]. This taxonomy, focused on the purpose of a change, is refined in [7] where other points of view, such as the *how*, the *when* or the *where*, are considered. In [6], a list of possible changes that could occur on UML diagrams is given, together with an approach to define impact analysis. Another idea is also presented in [12], where understanding the evolution is seen as properly representing the software history. This approach provides a metamodel for software evolution analysis, called Hismo, centered around history as a first-class entity. Although the work presented above allows one to analyze and support evolution, none of the techniques provide a means to explicitly represent and apply changes to the systems.

Techniques to support and apply evolution on software have also been defined at the level of source code. For instance, [29] presents the evolution of rule-based programs and provides an operator suite for the transformation of such programs. This is however limited to rule-based program, such as definite clause programs or SOS specifications. The rules are themselves expressed in code, with for example the Prolog directives that invoke a meta-programming operator add/2. Figure 2.21 illustrates an example of such a transformation. This approach is thus only focused on the code. The self-adaptive software [49] also provides techniques to allow systems to automatically adapt to change in their environment. This approach uses the concept of a feedback loop which allows the system to adjust itself during its execution. However, as opposed to the work presented in this deliverable, this solution treats changes when they occur but do not anticipate them. Both the solutions presented in [29] and [49] focus on the level of source code and hence differ from the technique defined in this deliverable, which aim to represent evolution on models expressed by UML diagrams. Representing evolution by model transformations will be discussed in the next section.

An approach to model transformation and evolution at a high-level of abstraction is given in [52]. The software process is seen as a sequence of evolutions, starting from an empty model in which the developer subsequently adds new elements. Although the approach is developed around UML, the models are generally considered as typed trees where

```
:- add(+storee,valuate).
:- add(-storee,valuate).
```

Figure 2.21: Transformation of a rule-based program [29]

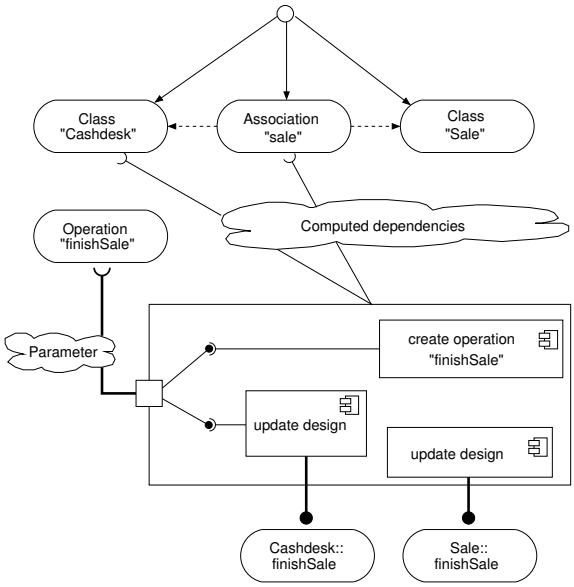


Figure 2.22: Representation of a concrete evolution illustrated in [52]

the nodes are model elements and the directed edges are "owner" relationships. Three types of actions, namely add, update and delete, take a model element as argument and apply the corresponding change. UML component diagrams are then used to graphically represent such evolutions. Concretely, the components represent the actions and the interfaces represent the model elements. A concrete transformation of a UML model is illustrated in Figure 2.22. This transformation consists in placing the method FinishSale of the class Cashdesk in the class Sale and replacing the body of the method FinishSale in Cashdesk by a call to the method moved to Sale. The transformation is also represented on the class diagram, as shown in Figure 2.23. This approach is very similar to the one presented in this deliverable. It covers however some additional concepts, since the level of abstraction of UMLseCh does not include the body of methods. On the other hand, the language that we will define in Chapter 2.1 represents the evolutions more explicitly. Indeed, Figure 2.22 does not clearly indicate how the elements are updated and this information cannot be found on the class diagram of Figure 2.23 either. In addition, UMLseCh does not include graphical representations as the ones of Figure 2.22 and remains compliant to the UML specifications [44]. The solution of [52] and the one described in this deliverable thus target a challenge specifying that "modeling languages should provide more direct and explicit support for software evolution. The idea would be to treat the notion of change as a first-class entity in the language" [39].

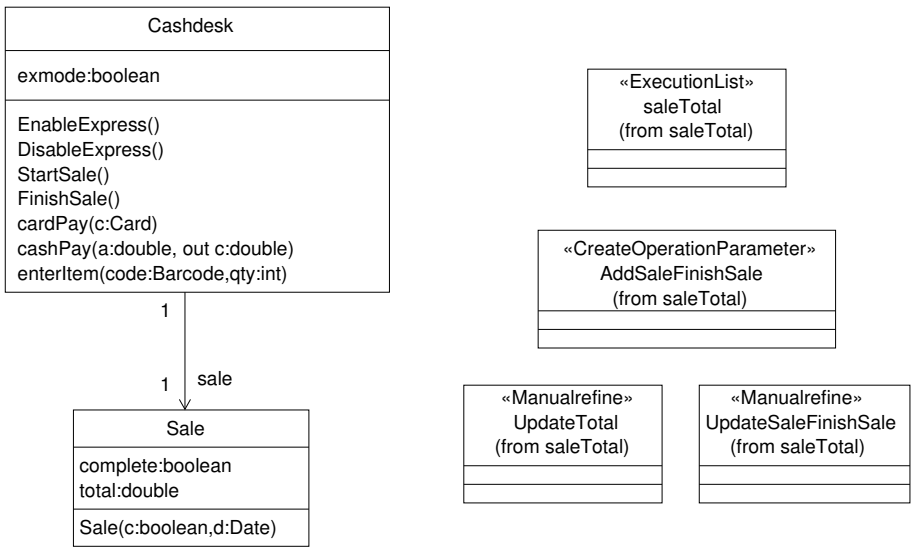


Figure 2.23: Representation of a concrete evolution on the class diagram, as shown in [52]

3 Model-Based Verification under Evolution

Security verification of UML models can be an expensive task (in time and resources) since security properties are often challenging to verify. The evolution of the models through different system versions poses the problem of the re-verification of the desired security properties. This chapter presents results towards addressing this challenge in the context of the UML security extension UMLsec. We investigate the security analysis of UMLsec models with respect to different evolving scenarios, with the main goal of re-using (when possible) the verification results from before the evolution. In particular we present results for both structural and behavioural diagrams. This approach is validated by a tool implementation of these verification techniques that extend the existing UMLsec tool support and that is significantly more efficient than re-checking the complete evolved model, as presented in Chapter 7. In this Chapter however, we discuss the problem of model evolution for security properties expressed in UMLsec independently of the transformation language used to express the evolution. We just assume that the delta between two models can be parsed from the notation (as it is the case for UMLseCh).

3.1 Verification Strategy

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 class or deleting a dependency). In general, given two diagrams A and B which are secure, there need not be a sequence of atomic evolutions transforming A into B such that each of the atomic transformations preserves security (i.e., there may be intermediate diagrams which may not be secure). Therefore the goal of our verification is to allow some modifications to happen *simultaneously*.

Since the evolution is defined by additions, deletions 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, i.e., if it represents a class, we can query for its associated stereotypes, methods, and attributes. For example, the class “Customer" in Fig. 3.1 can be seen as an object with parent the subsystem “Book a flight”. 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

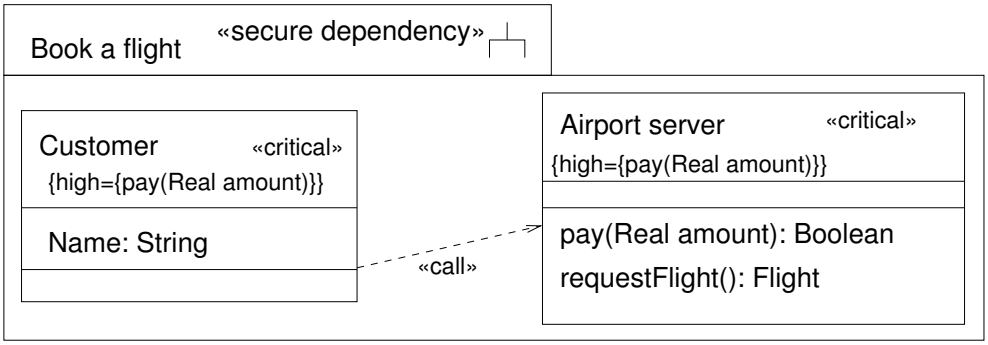


Figure 3.1: Class diagram Annotated with *« secure dependency »* before evolution

between the three sets, more specifically:

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

guarantees one does not delete and add the same model element which would be trivial. Additionally:

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

which prevents adding/deleting a model element present in a substitution (as target or as substitutive element).

As explained above, in general, an "atomic" modification (that is the action represented by a single model element in any of the sets above) could by itself harm the security of the model. So, one has to take into account other modifications in order to establish the security status of the resulting model. We proceed algorithmically as follows: we iterate over the modification sets starting with an object $o \in$ **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 $P(M)$ (and conversely by $\neg P(M)$). To show *soundness* for the security preserving rules R for a property P on diagram M means:

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

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 $P(M[\Delta])$.

We assume the existence of these delta sets to be able to carry out our analysis in the following. One could obtain them by using a transformation specific language such as QVT [3] or by using UMLseCh (see Chapter 2). Alternatively one could compute the difference between an original diagram M and the modified M'. This is nevertheless not central to our discussion, which focuses on the verification of evolving systems rather than on model transformation itself.

3.1.1 Evolving Secure Structural Diagrams

In this section, we discuss evolving structural UMLsec diagrams. We assume that the starting diagram is secure with respect to the properties that can be specified using UMLsec. The idea is to classify all possible single evolutions on model or sub-model elements and to decide whether these modifications preserve the desired security properties of the system. If by themselves they do not, we discuss which simultaneous changes must take place in order to preserve security.

As examples of a structural UMLsec diagrams, we discuss the evolution of class diagrams together with the *« secure dependency »* stereotype and the evolution of deployment diagrams together with *« secure links »*.

3.1.1.1 Class Diagrams

The stereotype *« secure dependency »* requires that for every dependency (*« send »* or *« call »*) between two classes in a class diagram such that in one of both classes a tag specifies a security requirement for a method or attribute
(for example *{ high = {method()} }*resp.*{ secrecy = {method()} },{ integrity = {method()} }*), then the other class has the same tag for this method/attribute as well (for example see Fig. 3.1). It follows that the computational cost associated to verify 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 **Del** of modifications by distinguishing cases on the type of $o \in$ **Del**. We will give explicitly the proofs for the deletion case, afterwards we will only argue informally on why the accepted evolution is sound.

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 $P(M[o, D])$ holds since clearly:

$$P(M[o, D]) : \ \forall C, C' \in M.\text{Classes} \setminus \bar{C} \ (\exists d \in M.\text{dependencies}(C, C')$$

$$\Rightarrow C.\text{critical} = C'.\text{critical})$$

  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 $M.\text{Classes} = (M.\text{Classes} \setminus C_{\bar{C}}) \cup C_{\bar{C}}$ 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 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). We can express this rule as follows:

$$\exists_{\text{o} \in \mathbf{Add}} \ \text{o.critical} \Rightarrow$$

$$\forall_{\text{C} \in \text{o.parent.dependentClasses}} \ (\exists_{\text{o}' \in \mathbf{Add}} \ \text{o}'.\text{parent} = \text{C} \wedge \text{o}'.\text{securitytag} = \text{o.securitytag})$$

  where the method critical returns true if the object is a *«critical»* stereotype, dependentClasses returns the set of classes associated through dependencies to a given class and securitytag returns the security tags associated with the *«critical»* stereotype. 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 suitable additions that preserve this symmetry.

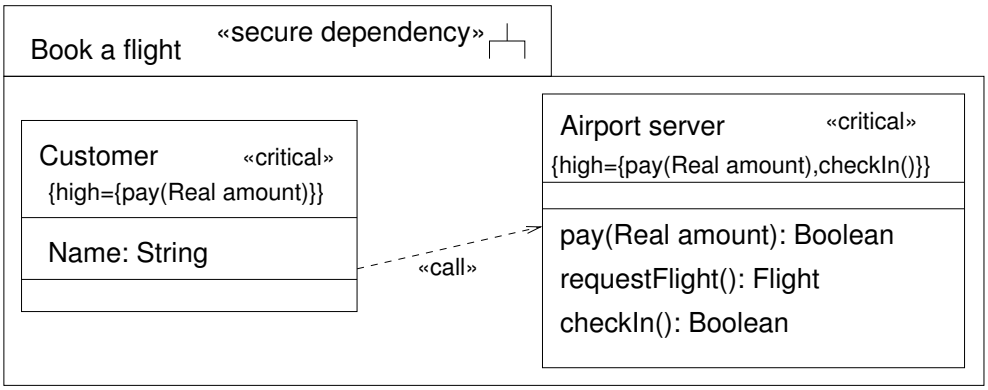


Figure 3.2: Class diagram annotated with *« secure dependency »* after evolution

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() }* by *{ requirement' = method()' }* in class C, 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 clearly preserved.

Example Assume that in a flight booking system (as in the class diagram in Fig. 3.1), we want to add a method within a class diagram that is secure with respect to *« secure dependency »* as depicted in Fig. 3.2. We also mark the new method "checkIn()" as critical for security with the tag {high} in the class "Airport server". This is a security violating evolution since we did not add the same method to the methods tagged within {high} in the "Customer" class.

3.1.1.2 Deployment Diagrams

The UMLsec stereotype *« secure links »* imposes constraints on the physical links between two nodes that are logically connected by a dependency. If for example a *« call »* dependency between nodes C and S is further annotated with the stereotype *« secrecy »* then *« secure links »* (for a default adversary) requires that the physical node between C and S is *« encrypted »* (for details see [19]). We can therefore characterize possible further evolutions of a deployment diagram that fulfils the security requirements given by *« secure links »* by considering the deletion, addition and substitution of physical links, nodes, dependencies and security related stereotypes. We assume that there is at most one physical links between two nodes, and that all physical links are annotated with a UMLsec specific stereotype. Moreover we assume that the initial UMLsec model is valid (i.e secure).

Formally, The pre-condition $P(M)$ holds:

$$\begin{aligned} P(M) :& \forall d \in M.dependencies, \\ & \forall s \in \{<< secrecy >>, << integrity >>, << high >>\} \\ & (s \in d.stereotypes \Rightarrow d.hasLink \wedge (d.link).secureAgainst(s)) \end{aligned} \tag{3.1}$$

whereas

M.dependencies is the set of all dependencies in the given model.

d.stereotypes returns the set of stereotypes attached to the dependency d.

d.link returns the link corresponding to the dependency d
(only one link is assumed to exist at most between two nodes)

secureAgainst(stereotype s) returns TRUE, if the link
fulfills the security requirement given by s

An atomic change o may either maintain the security of the model by and of itself or by applying simultaneous changes $C = \{o_1, ..., o_n\}$. If this is possible, and all changes present in the changeset Δ can and are executed and thus removed from the changeset until Δ is empty, then and only then $P(M[\Delta])$ holds.

Detailed analysis of individual model element and change types After defining the necessary conditions the individual model elements of a deployment diagram are analyzed in regards to different types of change.

Deletion

- **Node** It is assumed that by deleting a node all components and interfaces contained in the node, as well as all associated links are also deleted.
  This leads to the deletion of all associated dependencies d_N.
  The deletion of those dependencies is noted by $D = \{o_1, ..., o_n\}$.
  $P(M[o, D])$ holds, since the following applies:

  $$\begin{aligned} P(M[o, D]) :& \forall d \in M.dependencies \setminus d_N, \\ & \forall s \in \{\ll secrecy \gg, \ll integrity \gg, \ll high \gg\} \\ & (s \in d.stereotypes \Rightarrow d.hasLink \wedge (d.link).secureAgainst(s)) \end{aligned}$$

  This is clearly valid, as a property that is true for a given set is also true for a subset of that set.

- **Component** All associated interfaces and therefore every associated dependency are also deleted, which again leads to a subset of dependencies. As a result the above property clearly holds.

- **Interface** As with components the associated dependencies are deleted, leading to a subset, see above.
- **Dependency** The deletion of a dependency leads to a subset, see above.
- **Link** A link by itself may not be deleted, if the dependencies between the according link nodes have security stereotypes applied to them. If that is the case, *d.hasLink* after deletion would resolve to a false statement negating the property. Ergo the dependencies mentioned have to be deleted as well, which again leads to a subset of dependencies, see above.
- **Tag (Threat)**This completely nullifies the security requirements. Therefore this change is not expanded upon.
- **Stereotype (Dependency)** The deletion of a stereotype applied to a dependency causes a weakening of the security requirements, as the above property depends upon the testing of the applied stereotypes.
- **Stereotype (Link)** This is not allowed or rather not defined. A link with no stereotype can not be used to infer information regarding the security. Thus no positive or even definitive answer regarding the security of the model can be given.

Addition

- **Node** The addition of a node by itself doesn't change the state of the security, as no changes regarding the dependencies or links take place.
- **Component** If the component is the only thing added, meaning no additional associated dependencies, the security is preserved, as no additional stereotypes have to be taken into account.
- **Interface** An interface added by itself, i.e. with no dependencies attached, preserves the security of the model, see above.
- **Dependency (without Stereotypes)** $P(M[o])$ holds, because although the set *M.dependencies* has grown, the set *d.stereotypes* of the new dependency d is still empty.
- **Dependency (with Stereotypes)** see **Stereotype (Dependency)**
- **Link** If $P(M)$ holds as a pre-condition and a link l is to be added between nodes n_1 and n_2:
  - if there hasn't been a link between n_1 and n_2 prior to the change, then there can't be any dependencies with security relevant stereotypes applied to them between n_1 and n_2, else $P(M)$ wouldn't hold because *d.hasLink* would resolve to false. Therefore the link may be added without any simultaneous changes.
  - if a link between n_1 and n_2 already exists, no further link is allowed to be placed between those nodes due to the fact that only one link is allowed between two nodes.

- **Tag (Threat)** The addition of a threat tag has to be followed by a complete re-verification of the model as the return value of the function *secureAgainst(s)* is dependent on the type of threat added.
- **Stereotype (Dependency)** If a stereotype s_{add} of the set of the three security relevant stereotypes is added to a dependency d, the set of stereotypes *d.stereotypes* increases.
  Therefore one has to check, if the following holds:

$$\forall s \in d.stereotypes \cup s_{add} \Rightarrow d.hasLink \wedge (d.link).secureAgainst(s)$$

  - If $(d.stereotypes \setminus s_{add}) \neq \{\emptyset\}$ is true, $P(M)$ holds for $d.stereotypes \setminus s_{add}$. The part of the property that remains to be verified is $(d.link).secureAgainst(s_{add})$. If this is not the case, a simultaneous substitution $S = \{o_1\}$ of the link stereotype with a stereotype conforming to the security requirements given by the threat tag has to be carried out so that $P(M[o, S])$ holds.
  - If $(d.stereotypes \setminus s_{add}) = \{\emptyset\}$ is empty preceding the change, it is also possible that no corresponding link exists. If such is the case, a simultaneous addition $A = \{o_2\}$ has to be carried out, whereas o_2 refers to the addition of a link between the supplier and client nodes of the dependency d. Following that $P(M[o, A])$ holds.
- **Stereotype (Link)** The addition of a stereotype to a link is possible only if no stereotype was attached to the link prior to the change, as a link may not have more than one stereotype attached to it. If $P(M)$ holds, the model M preceding the change had no security relevant stereotypes attached to the dependencies corresponding to the link in question. If such would be the case, the link would also already have a stereotype attached to it.

Substitution

- **Node** If all components and links belonging to the old node are transplanted to the new node, the security requirements do not change. If there are changes to any of the said model elements, see the corresponding section.
- **Component** If all dependencies and interfaces remain the same way with the new component, the security requirements do not change. If that is not the case, see the corresponding section.
- **Interface** If the dependencies connected to the old interface are connected to the new interface in the same way, the security requirements do not change. In other cases see the corresponding section.
- **Dependency (different nodes)** If a dependency d_a between nodes n_1, n_2 is substituted with a dependency d_n between nodes n_3, n_4, whereas n_3 or n_4 is different from n_1 as well as n_2,d_n can be treated like an addition of a dependency, see the corresponding section. The stereotypes of the new dependency may be handled like additions of stereotypes to a dependency.

- **Dependency (same nodes)** If the nodes of the substituted dependency do not change, the only meaningful change is a substitution of the stereotypes attached to the dependency and are described in the corresponding section.
- **Tag (Threat)** If a threat tag is substituted with a different threat tag, the security requirements regarding the model change. This requires a complete re-verification of the model.
- **Stereotype (Dependency)** The stereotype substitute can be handled like the addition of a stereotype to a dependency. See the corresponding section.
- **Stereotype (Link)** If a link stereotype is substituted, for all the dependencies must be checked, whether the security requirements are satisfied.

3.1.2 Evolving Secure Behavioral Diagrams

3.1.2.1 Activity Diagrams

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

Formally:

$$P(M): \forall A \in M.\text{protected} \; \forall \, U \in M.\text{users} \; (U.\text{hasAccess}(A) \Rightarrow \exists R \; ((A,R) \in M.\text{rights} \wedge (R,U) \in M.\text{roles}))$$

As before, we distinguish cases in the evolution to discuss the security preservation. We assume that all activities associated to a user are reachable.

Deletion

- **Activity/vertex**: The deletion of an activity or a vertex clearly does not alter P (independently of the fact that the activity is protected or not).
- **Pair activity/right**: The deletion of a pair (A,R) in {rights} preserves P if together with this deletion the activity A is deleted from all users U with role R.
- **Pair role/user**: The deletion of a pair (R, U) in {roles} preserves P if together with this deletion all activities linked to this role via access rights are deleted for U.

Addition

- **Activity**: The addition of a protected activity A to actor U must be accompanied by the addition of a pair (A,R) and a pair (R,U) if there are no such two pairs for a given R'.

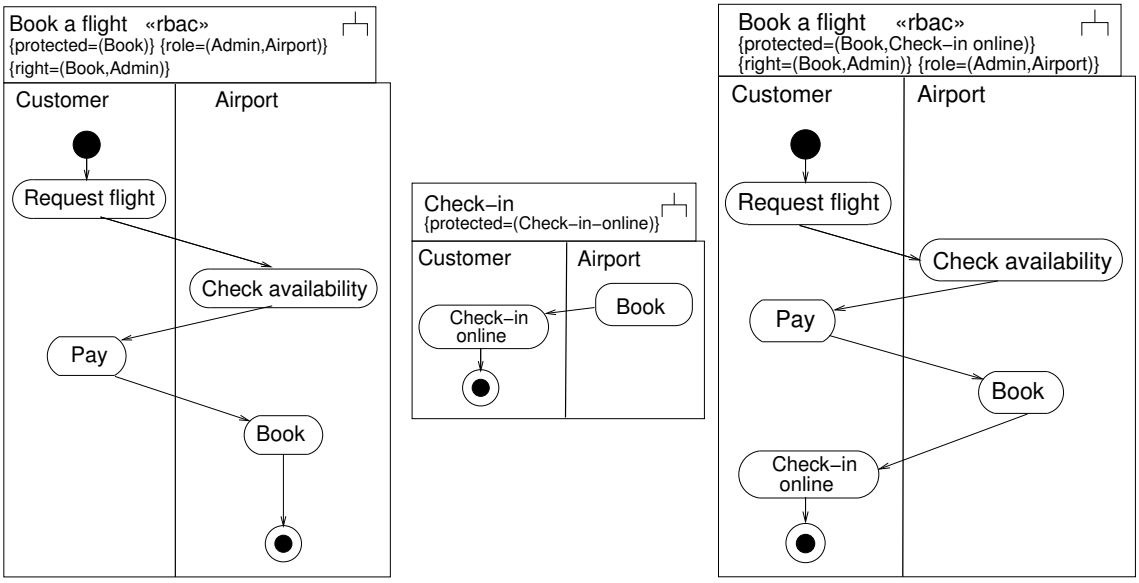


Figure 3.3: Activity diagram annotated with *« rbac »* before evolution (left-hand side), added model elements (middle), and after evolution (right-hand side)

- **Vertex**: Since we assume all activities are reachable, the addition of a vertex does not affect the security properties of the diagram.
- **Pair activity/right** or role/user: The addition of rights or roles to a secure diagram do not compromise by themselves the fulfilment of P.

Substitution

- **Activity**: If a protected activity A is substituted by A' then there must be a simultaneous substitution of A by A' in all pairs in {rights} where A appears.
- **Condition on a vertex**: This is irrelevant for P if we assume the condition is satisfiable.
- **Pair activity/right**: The substitution of a pair (A,R) by a pair (A',R) must be accompanied by a substitution of the activity A by A'. Likewise, the substitution of (A,R) by (A,R') is accepted only when the role R is substituted by R' in the pairs contained in {roles} and {rights}.
- **Pair role/user**: Similar to the previous case.

Example In Fig. 3.3 (left-hand side), we show an activity diagram to which we want to add a new set of activities, introducing a web-check-in functionality to a flight booking system (as in the example in the previous section). The new activity “Check-in online” (middle of Fig. 3.3) is protected, but we do not add a proper role/right association to this activity, thus resulting in a security violating diagram (right-hand side Fig. 3.3).

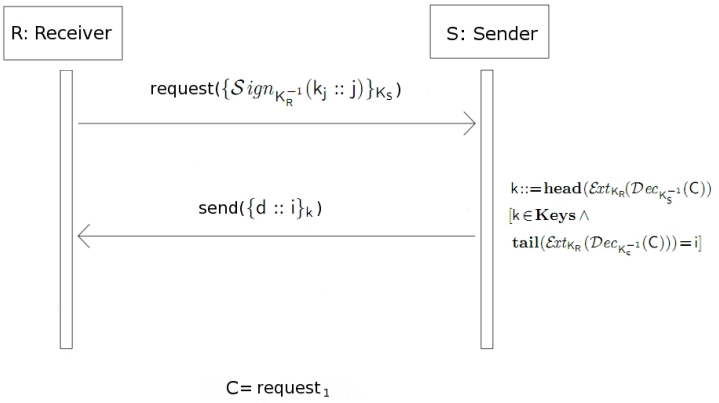


Figure 3.4: Example of a secure channel

3.1.2.2 Sequence Diagrams

In this section we want to analyse the evolution of Sequence Diagrams where a symbol should remain secret to unwanted partners. This case is particularly challenging with respect to the examples presented before, because the message exchange might involve a non-trivial behaviour between at least two entities and the adversary. This adversary is of a default type over an Internet communication channel, as described in the background section. We assume however, that it is of a Dolev-Yao type, that is, he is not able to read messages encrypted with keys he does not possess. In this case we will treat the possible evolutions (addition, deletion and substitutions of parameters in a message and constraints) mostly as "atomic". This means that we will often discuss the consequences of a single change without considering simultaneous other changes that could preserve security as we did in the previous examples. This is due to the complexity of the setting, where the simultaneous evolutions that are security preserving are much larger than in static diagrams. One could enrich these rules in the future with frequent operations that are security preserving. Before describing the possible evolutions and the necessary enforcing rules to preserve the desired security properties, we begin with a motivating example. We will informally describe how the verification takes place, for more details on this formalization see [21].

We consider a secure channel [19] as defined in Fig. 3.4. If we assume that the set of keys consists of:

$$\textbf{Keys} = \{K_R, K_R^{-1}, K_S, K_S^{-1}, k_j, k_a, K_A, K_A^{-1}\}$$

(where k_j, k_a are symmetric and the rest are asymmetric keys) and we denote the knowledge of the adversary by means of a **knows** predicate then the basic knowledge that a default adversary can gain of this message exchange is:

$$\mathsf{knows}(\{\mathcal{S}ign_{K_R^{-1}}(k_j :: j)\}_{K_S}) \wedge$$
$$(\mathsf{knows}(\mathcal{S}ign_{K_R^{-1}}(k_j :: j)) \Rightarrow \mathsf{knows}(\{d :: j\}_{k_j})) \wedge$$

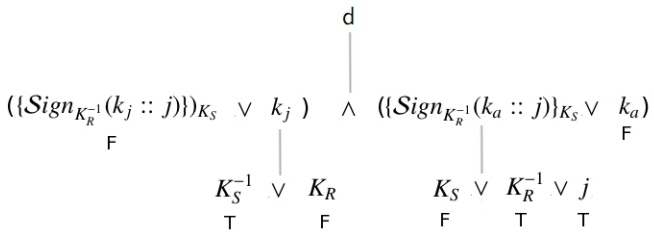


Figure 3.5: Dependency tree depicting the secrecy preservation of the value d

$$(\mathsf{knows}(\{Sign_{K_R^{-1}}(k_a :: j)\}_{K_S}) \Rightarrow \mathsf{knows}(\{d :: j\}_{k_a}))$$

as we can see by solving the constraint on the sender side (one can obtain an automatic translation from Sequence diagrams to such predicates using the UMLsec tool, see [20]).

This knowledge can be enlarged by the adversary by means of encryption, decryption, signature and extraction with the keys in his possession, that we assume are:

$$\mathsf{knows}(K_R) \wedge \mathsf{knows}(K_S) \wedge \mathsf{knows}(k_a) \wedge \mathsf{knows}(K_A) \wedge \mathsf{knows}(K_A^{-1})$$

(since the other are supposed to be private before the execution of the protocol).

We can build a dependency tree of symbols based on the cryptographic operations that an adversary could perform using his knowledge on the system, as shown in Fig. 3.5. We denote with **T** (respectively with **F**) the fact that the secrecy of a given symbol is preserved. Such a tree can be generated automatically using the first order logic formula that represents the knowledge of the adversary by encoding all possible operations he can perform. In this example, all symbols contained in the message exchange are contained in the tree. In general one can generate finitely many such trees to cover all symbols in the sequence diagram (not necessary related to the symbol which is supposed to remain secret) as this will come handy to analyse the evolution.

We can see from the dependency tree, that there are many symbols from which the value of truth of the secrecy of d could be altered, namely:

$$Y = \{K_S^{-1}, K_R^{-1} :: j, k_j, \{Sign_{K_R^{-1}}(k_a :: j)\}_{K_S}, d\}.$$

Shall any of this symbols change its value of truth (i.e. secrecy not preserved), then the secrecy of **d** would be compromised. From this analysis we can conclude that if we foresee to add a message msg to the protocol in a future version of the protocol, we have to be careful not to leak any of the symbols in Y. For example, if we add a parameter msg at any point, it must hold :

$$\neg\mathsf{leaks}(msg, Y)$$

where the predicate $\mathsf{leaks}(x, Y)$ means that from x and all other symbols the adversary can derive from the protocol (as stored in the dependency tree) it is not possible to derive any of the symbols contained in Y.

We now discuss more precisely the evolution of sequence diagrams in general. We assume that the symbol that is supposed to remain secret is also d.

Deletion

- **Parameter** : The deletion of a parameter “shrinks" the knowledge of the adversary instead of enlarging it. Therefore the single deletion of a parameter within a message does not alter the security property of d. However, one should update the dependency tree(s) associated to the Diagram after the deletion.
- **Constraint**: The deletion of a constraint could potentially harm security, since it poses less requirements to gain the knowledge of certain symbols (the ones contained in the message associated to the constraint). In particular, one can alter the dependency tree of the d by re-computing the symbols below the ones that should be returned if the constraint is fulfilled. In our example, if we fully delete the only constraint, then we would have to enlarge the tree under d with all the combinations of $\mathcal{S}ign_{K_R^{-1}}(x :: y)\}_{K_S}$ since we would not check any more whether x is a key or whether y equals j. If we do not limit the number of encryptions an adversary can perform, these combinations are endless: for example we could encrypt with the same key forever in order to generate a value for x, and this could be a valid input for S to return a message.

Addition

- **Parameter**: As explained above, in this case the addition of parameter x to a sequence diagram with set Y of symbols that could alter the secrecy value of d should not leak, together with all the symbols known to the adversary, a symbol contained in Y.
- **Constraint**: The addition of a constraint in a secure Sequence Diagram also shrinks the knowledge of the adversary and does not harm the security of the system. The dependency tree should be updated accordingly.

Substitution

- **Parameter**: In general, the substitution of a parameter could potentially harm the security of d. We consider only the case where a key is substituted through the whole protocol. In this case, if an unknown key for the adversary is substituted by another unknown key that was not already part of the protocol, then security is preserved.
- **Constraint**: Substitution of a constraint is only allowed if, as for the substitution of a parameter, only unknown keys are replaced by new unknown keys.

3.2 Application to the Global Platform

In this section we present the application of the verification strategy to the GP Specification. First we define and analyse two stereotypes tailored specifically for the GP life-cycle

(which we will further use as part of the WP4-WP7 link in Chapter 5). Then we will see how is secrecy preserved for the Secure Channel Protocol 01 and its evolution to SCP 02. Another example of the application of the UMLseCh approach to the POPS Case Study will be introduced in Chapter 4 as part from the integration effort with WP6.

We consider the changes in the Global Platform Specification [2] between version 2.1.1 and 2.2. One important change in the specification regards the Card Life-Cycle, as described in the following subsection.

3.2.1 Card Life Cycle

A GlobalPlatform card provides security services related to information exchanged between the card and an off-card entity. The security level of the communication with an off-card entity does not necessarily apply to each individual message being transmitted but can only apply to the environment and/or context in which messages are transmitted. The concept of the Life Cycle of the card may be used to determine the security level of the communication between the card and an off-card entity.

The GlobalPlatform defines Life Cycle models to control the functionality and security of its components. Card Life Cycle is one of these models. From a GlobalPlatform perspective, the card Life Cycle begins with the state OP_Ready. The other states in this cycle are INITIALIZED, SECURED, CARD_LOCKED and finally TERMINATED.

Card Life Cycle State OP_READY: The state OP_Ready indicates that the runtime environment shall be available and the Issuer Security Domain, acting as the selected Application, shall be ready to receive, execute and respond to APDU commands.

The following functionality shall be present when the card is in the state OP_Ready:

- The runtime environment shall be ready for execution;
- The OPEN shall be ready for execution;
- The Issuer Security Domain shall be the implicitly selected Application for all card interfaces;
- Executable Load Files that were included in Immutable Persistent Memory shall be registered in the GlobalPlatform Registry;
- An initial key shall be available within the Issuer Security Domain.

Card Life Cycle State INITIALIZED: The state INITIALIZED is an administrative card production state. The state transition from OP_Ready to INITIALIZED is irreversible. This state may be used to indicate that some initial data has been populated (e.g. Issuer Security Domain keys and/or data) but that the card is not yet ready to be issued to the Cardholder.

Card Life Cycle State SECURED: The state SECURED may be used by Security Domains and Applications to enforce their respective security policies. The state transition from INITIALIZED to SECURED is irreversible. The SECURED state should be used to indicate to off-card entities that the Issuer Security Domain contains all necessary keys and security elements for full functionality.

Card Life Cycle State CARD _ LOCKED: The card Life Cycle state CARD_LOCKED is present to provide the capability to disable the selection of Security Domain and Applications. The card Life Cycle state transition from SECURED to CARD_LOCKED is reversible. Setting the card to this state means that the card shall only allow selection of the application with the Final Application privilege. Card Content changes including any type of data management (specifically Security Domain keys and data) are not allowed in this state. Either the OPEN, or a Security Domain with Card Lock privilege, or an Application with Card Lock privilege, may initiate the transition from the state SECURED to the state CARD_LOCKED.

Card Life Cycle State TERMINATED: The state TERMINATED signals the end of the card Life Cycle and the card. The state transition from any other state to TERMINATED is irreversible. The state TERMINATED shall be used to permanently disable all card functionality with respect to any card content management and any life cycle changes. This card state is intended as a mechanism for an Application to logically 'destroy' the card for such reasons as the detection of a severe security threat or expiration of the card. If a Security Domain has the Final Application privilege only the GET DATA command shall be processed, all other commands defined in this specification shall be disabled and shall return an error. If an application has the Final Application privilege its command processing is subject to issuer policy. The OPEN itself, or a Security Domain with Card Terminate privilege, or an Application with Card Terminate privilege, may initiate the transition from any of the previous states to the state TERMINATED.

Figure 3.6 illustrates the state transition diagram for the card Life Cycle.

Global view of the Scenario

A mobile network operator MOB propose a SIM card to its customers that will be used for payment as a contactless credit card or for ticketing as a contactless transport card. The USIM will embed a Java Card Virtual machine and a subset of GP.

We consider now the security properties the card Life Cycle using in POPS Scenario:

PA1-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

PA2-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

Evolving Secure State Diagrams

Given a model of the life-cycle (depicting the behaviour of the system as black-box, as we will see in Chapter 5 the above named criteria may be verified thru UMLsec stereotypes, as we will discuss in the following.

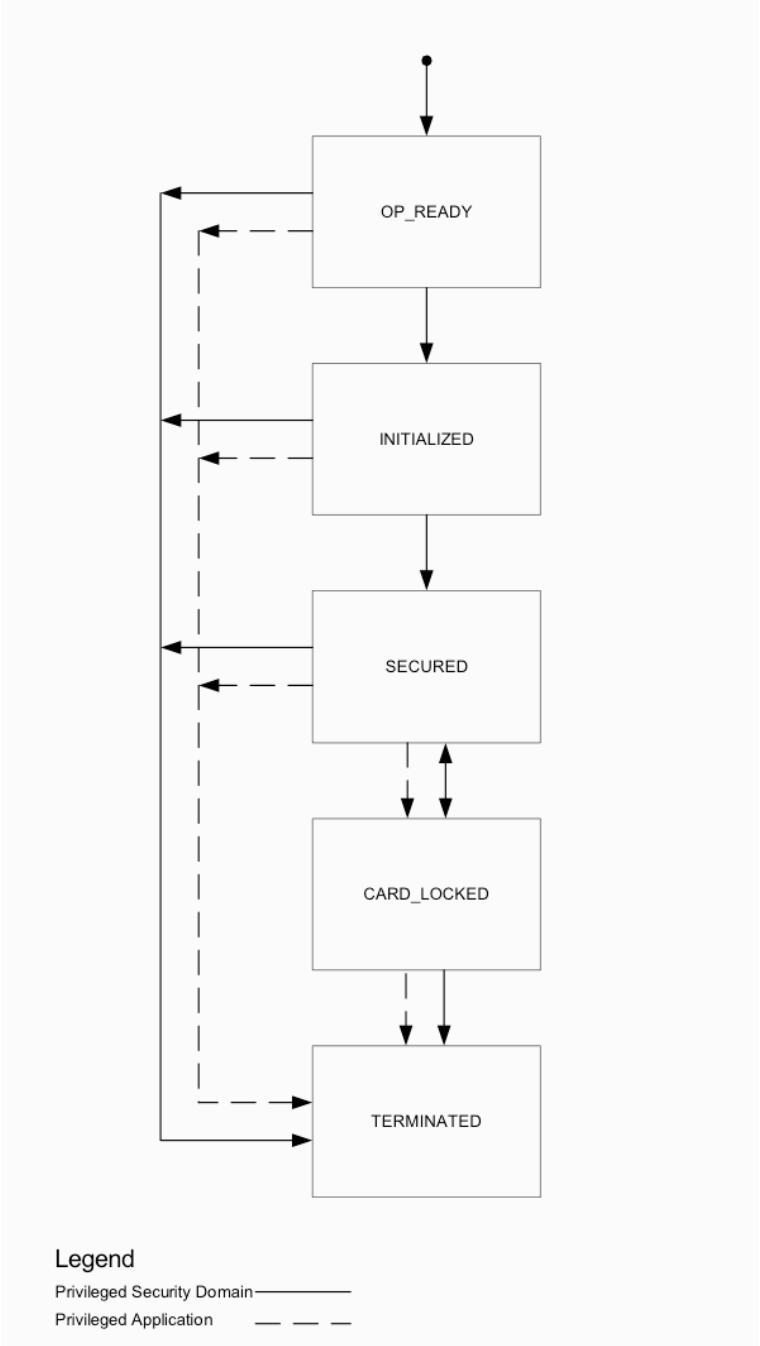


Figure 3.6: Card life cycle state transitions in GP V2.2

The stereotype *« locked-status »* with tag *{ status = Status }* requires that no outgoing transition from a state with label $Status$ exists.

The stereotype *« authorized-status »* with tag *{ status = Status }* and *{ permission = Permission }* requires that every incoming transition to a state with label $Status$ has within its guard the substring $Permission$.

Although we will discuss more details about the usefulness of this verification in the WP4-WP7 link, we can already analyse the consequences of changes in the model with respect to the two stereotypes.

We assume that the security properties in the starting state machine are satisfied. Now we analyze the possible changes on states, transitions and guards with respect to both above named stereotypes as done previously for other stereotypes. This is possible since the nature of the verification is purely statically.

Deletion

- **State** (including all incoming and outgoing transitions) or a transition or a guard with respect to *« locked-status »* does not alter the security property from a state machine.

  Deletion of a state or transition can cause that one or even more states from state machine are not reachable anymore.

- **Guard** with respect to *« authorized-status »*: Let $D = \{o_1, ..., o_n\}$ the set of objects to be deleted . We separate the model into two parts;

  One part consists of the guard o, which must be deleted, its related transition and the target state of this transition. This submodel is called M_1. Let P be the security property with respect to stereotype *« authorized-status »*, P is for each element o from D satisfied if the following condition is fulfilled;

  If deleted guard contains the condition *{ permission = Permission }* and the transition corresponding to this guard has target state which is labeled as "$Status$", the transition itself should be deleted. This is described by the rule:

$$\forall_{o \in D}(o.type = guard \wedge Permission \in o.permission \quad \wedge$$
$$o.parentTransition.targetState.label = Status) \Rightarrow$$
$$\exists_{o' \in D}(o' = o.parentTransition)$$

  The security property P is always satisfied in the other part of the model (defined as $M_2 : M \setminus M_1$) with respect to *« authorized-status »*, because in this part there exists no state with label "$Status$".

  Thus the security property in evaluated model $P(M[o, D])$ results as:

$$P(M[o, D]) \Rightarrow P(M_1[o, D]) \wedge P(M_2[o, D])$$

Addition

- **State** Addition of a new state without any transition does not violate the security property from a state machine.
- **Transition** with respect to *« locked-status »*: Let $A = \{o_1, ..., o_n\}$ the set of objects to be added. We separate the model into two parts;

  One part consists of new Transition o, its source state and target state. This submodel is called M_1. Let P be the security property with respect to stereotype *« locked-status »*. P is satisfied for each o in the set A if the following condition is fulfilled;

  If the source state from new transition o is labeled as $Status$, then the target state from the transition should be labeled as $Status$ too, i.e. the new transition should have the same state as source and target. We can express this rule as follows:

  $$\forall_{o \in A}(o.type = transition \wedge o.sourceState.label = Status) \Rightarrow$$
  $$(o.targetState = o.sourceState)$$

  The security property P is always satisfied in the other part of model (defined as $M_2 : M \setminus M_1$) with respect to *« locked-status »*, because in this part exists no state with label $Status$.

  Thus the security property in evaluated model $P(M[o, A])$ results as:

  $$P(M[o, A]) \Rightarrow P(M_1[o, A]) \wedge P(M_2[o, A])$$

- **Transition** with respect to *« authorized-status »*: Let $A = \{o_1, ..., o_n\}$ the set of objects to be added. We separate the model into two parts;

  One part consists of new Transition o and its target state. This submodel is called M_1. Let P be the security property with respect to stereotype *« authorized-status »*. P is satisfied for each o in the set A if the following condition is fulfilled;

  $$\forall_{o \in A}(o.type = transition \wedge o.targetState.label = Status) \Rightarrow$$
  $$\exists_{o' \in A}(o' = o.guard \wedge Permission \in o'.permission)$$

  The security property P is always satisfied in the other part of model (defined as $M_2 : M \setminus M_1$) with respect to *« authorized-status »*, because in this part exists no state with label $Status$.

  Thus the security property in evaluated model $P(M[o, A])$ results as:

  $$P(M[o, A]) \Rightarrow P(M_1[o, A]) \wedge P(M_2[o, A])$$

- **Guard** Addition of a new guard does not alter the security properties.

Substitution Let $S = \{((o_1, o_1'), ..., (o_n, o_n'))\}$ the set of objects pairs to be substituted.

- **State** with respect to *« locked-status »*: In case that a state o , which is not labeled as $Status$, is substituted with a State o' , which is labeled as $Status$, o' should have no outgoing transition to preserve the secure requirement of the system.
- **State** with respect to *« authorized-status »*: If a state o , which is not labeled as "$Status$", is substituted with a state o' , which is labeled as $Status$, all incoming transitions to o' should have a guard, which contains the condition *{ permission = Permission }* to preserve the secure requirement of the system.
- **Transition** If a transition o is substituted with a transition o', to preserve the secure requirement of the system, o' should have the same rule explained by addition of a new transition.
- **Guard** with respect to *« locked-status »*: Substitution of a guard does not alter the secure properties of the system.
- **Guard** with respect to *« authorized-status »*: If a guard o which contains *{ permission = Permission }* is substituted with a guard o' without this condition, the transition corresponding with o' should be deleted too.

3.2.2 Evolution of the Secure Channel Protocol

We now take as an example within the Global Platform specification the Secure Channel Protocol 02 (SCP02) Authentication Phase seen as an evolution of its predecessor, the SCP01. The SCP01 works as follows: the host initiates the Authentication Phase by sending a random challenge N_H to the smartcard. The card then generates a random challenge N_C and uses a function based on pre-shared keys to generate a symmetric session key with parameters N_H' (we change the challenge symbol since the card may actually receive a manipulated message from an adversary in the communication channel) and N_C. It then signs a cryptogram using the session key and sends it to the host together with the card challenge. The host should be then able to verify that signature by using the same function with both challenges as parameters.

The security of this protocol relies on the pre-shared key generating function. Formally, we can analyze SCP01 in a similar way as done in the Secure Channel Example in Section 3.1.2.2. If we assume that the set of nonces is:

$$\textbf{Nonces} = \{N_C, N_H, N_C^A, N_H^A\}$$

then the secrecy of d depends on the secrecy of $K(N_H, N_C)$ or the secrecy of $K(N_H^A, N_C)$. Therefore the secrecy of d is guaranteed by the assumption that the adversary doesn't know $K(x, y)$ for any $x, y \in$ **Nonces**.

In the SCP02 an extra parameter is given to the generating function: a session sequence number seq. This change in the protocol was introduced to to limit the lifetime of the (pre-shared) keys between host and card. We can model this by using one application of *« substitute-all »* and one application of *« add »* as shown in Figure 3.7.

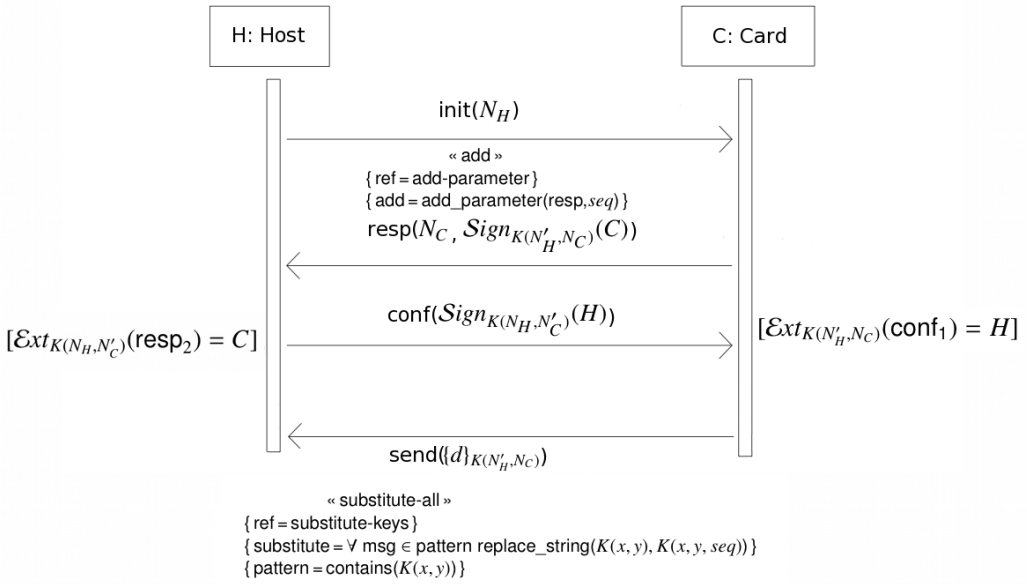


Figure 3.7: Secure channel protocol evolution

We can therefore reuse the confidentiality result of our analysis of SCP01 since the addition of the symbol seq does not alter the dependency tree of d, and we can reflect the application of *« substitute-all »* in the dependency tree by simply substituting the corresponding symbols and obtaining:

$$\mathsf{knows}(d) \Leftrightarrow \mathsf{knows}(K(N_H, N_C, seq) \vee \mathsf{knows}(K(N_H^A, N_C, seq)).$$

Again, since the assumption is:

$$\forall\, x, y \in \textbf{Nonces}, seq \in \mathbb{N}\ \neg\mathsf{knows}(K(x, y, seq))$$

then secrecy of d is preserved in SCP02.

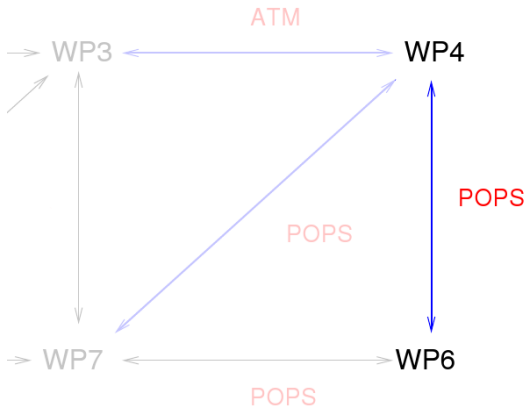
Authenticity is also enforced for both the host and the client by the random challenges: even if the adversary records a message exchange for a given session between the card and a host, a new session will have new values for N_H and N_S, making the old signatures useless for forging the identity of the host/card. A formal argument similar to the one done for the secrecy of d can be done by means of the correspondence predicates **init**(x) and **ok**(x) inserted appropriately in conjunction with the **knows** predicates. This correspondence predicates state that the node x has started the communication and that node y has certified the authenticity of x respectively. The conjecture to show in order to prove authenticity of the protocol (for example for the card C) is then:

$$ok(C) \Rightarrow init(C).$$

To save space, we don't show the details of this argument, but we highlight that the core

of the proof is the fact that the adversary cannot forge the signature of either the Card or the Host because of the secrecy of all possible keys generated by the function $K(x, y)$. This way he is forced to wait until the card (respectively the host) generates a signature, triggering the correspondence predicate **init**. Authenticity is thus preserved by the above mentioned evolution because the protocol maintains the secrecy of all keys generated by the extended function $K(x, y, z)$.

4 Model-based and Code-based Verification



This chapter presents the common work done as an integration effort between Work Packages 4 (Model-based verification) and 6 (Code-based verification). This work is based on the POPS case study (common to both Work Packages) and in particular on the GP specific *control-flow* property and on classical *non-interference*. These correspond to the "Information Protection" property and the change requirement "Software Update" of the POPS case study as defined in [50].

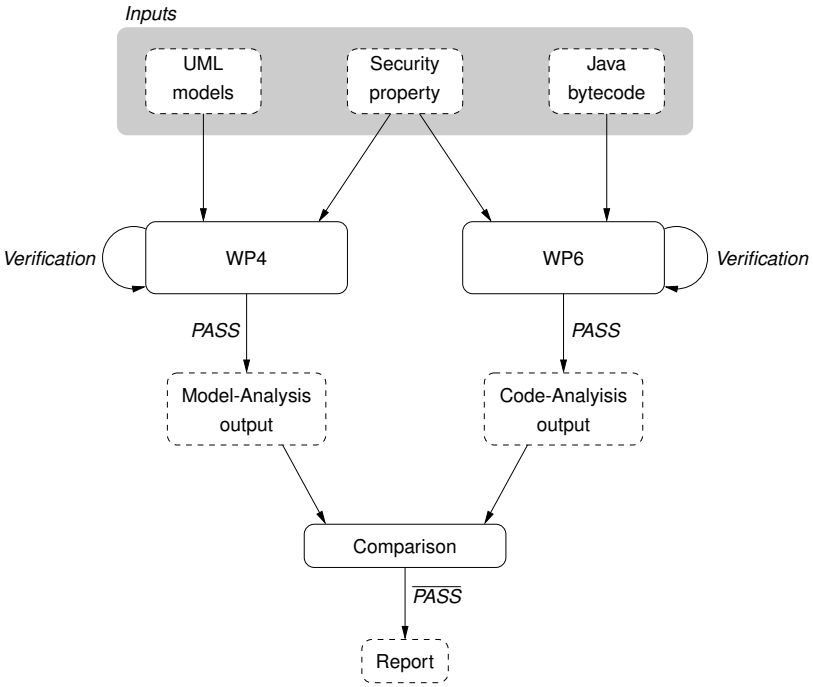


Figure 4.1: General process

The general idea behind this work is depicted in Figure 4.1. An application (or set of applications) modeled in UML can be verified against a security property by means of the WP4 techniques at the design phase, some output being produced after the verification. The same property can be verified on an implementation of the same application (set of) and the output produced by this process can be then *compared* to the output produced at design time. One can then report on the differences, thus for example informing about interesting inconsistencies as we will see in the following.

The Change aspect The techniques used on both Work Packages to deal with *evolution* can be then applied in parallel to iterate this process over the application life-time. This works similarly to the process depicted in Figure 4.1, where the inputs are evolved UML models, security properties and byte code. The verification under evolution techniques developed in WP4 and WP6 are used for dealing respectively with evolving models and evolving bytecode. After the verification has taken place, instead of comparing the whole model and code analysis output artifacts we can compare only the modified outputs, following the overall SecureChange approach. The precise definition of this comparison for the cases of control flow and non-interference introduced in this chapter is matter of current work and will be reflected in subsequent deliverables.

4.1 Control flow analysis

In this section we describe the verification process associated with the stereotype *« controlflow »*. Intuitively, this stereotype enforces that a sequence diagram describing the behavior of a set of methods (their classes and their methods) has no illicit control flows between applications. This stereotype establishes an equivalence at the model with the direct and transitive control flow models of the WP6 working at the code level described in the deliverable D6.3 (Chapters 3 and 4). In addition to provide the model-based verification process, we also provide some basic artifacts pushed to the code-based verification process to permit a basic coherency verification between the design level and the implementation level.

4.1.1 Model-based control flow analysis

4.1.1.1 UMLsec stereotype

A sequence diagram SD is a directed graph $(\textbf{Methods}, \textbf{Messages})$ that models exchanges of messages between methods of instances, where **Methods** is a set of fully qualified method names $A.C.m$ corresponding to method calls in the sequence diagram, $A \in$ **Applications** is an application, C is a class of application A and m is a method that can be called on C either implemented in C or inherited, and **Messages** $\subseteq$ **Methods** $\times$ **Methods** is a set of tuples $(A.C.m, A'.C'.m')$, also written $A.C.m \rightarrow A'.C'.m'$, that denote invocation of method $A'.C'.m'$ in method $A.C.m$.

Three tags are attached to the *« controlflow »* stereotype:

- services $\subseteq$ **Methods** the set of services shared by applications, i.e. methods that can be invoked by other applications;
- transitive $\in \{\text{yes}, \text{no}\}$ is a mandatory flag to precise if the control flow policy is to be applied on direct method calls only (value no) or if the verification has to take into consideration transitive method calls (value yes);
- policy $\subseteq$ **Applications** $\times$ **Methods** the set of control flow policy rules where each tuple $(A, B.C.m)$ denotes that an application A is authorized to invoke the method $B.C.m$.

The Figure 4.2 shows an example of sequence diagram inspired from the POPS use case annotated with *«controlflow»* stereotype. This example consists in three applications, two of them having shared services to permit the others to invoke them. JTicket is a simple ticketing application that, when needed, is refilled using money stored in EPurse application. When the amount of money requested to the EPurse is not available, the EPurse tries to obtain more credits from the EMV application. The annotated non-transitive control flow policy permits the JTicket to use EPurse's services, and the EPurse to use EMV's services.

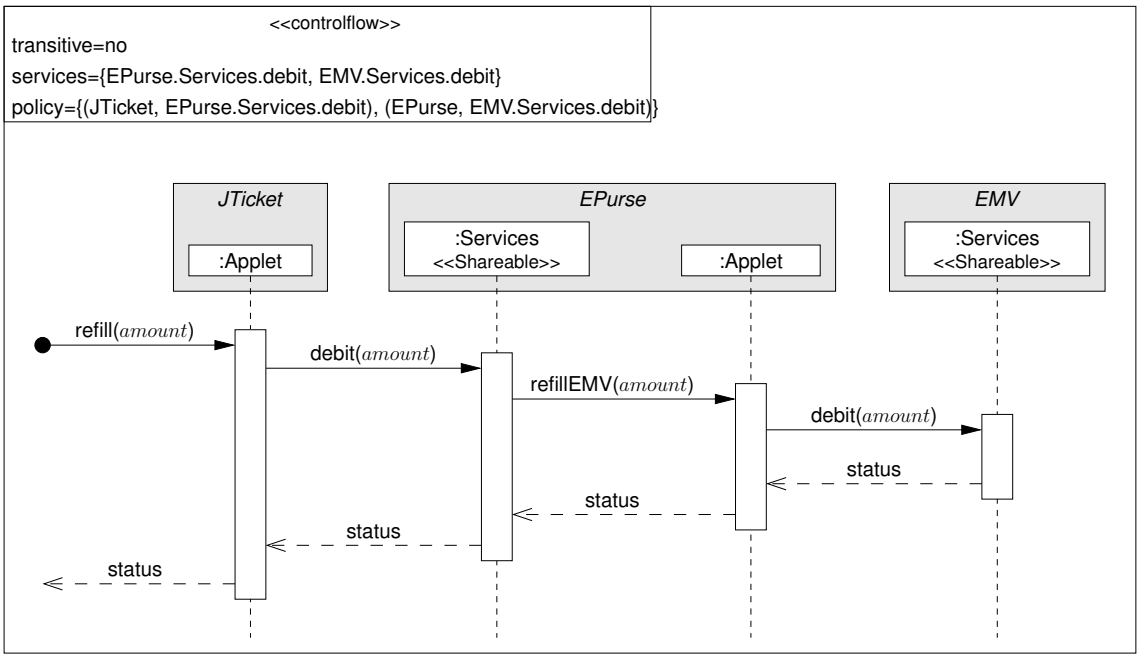


Figure 4.2: Example of sequence diagram annotated with control flow policy.

4.1.1.2 Verification process

For the non-transitive case, the absence of illicit control flow property P is verified in a sequence diagram $SD = (\textbf{Methods}, \textbf{Messages})$ annotated with *«controlflow»* stereotype

for the non-transitive case if:

$$P(SD) : \forall(A.C.m, A'.C'.m') \in \textbf{Messages},$$
$$A \neq A' \implies (A'.C'.m' \in \text{services} \wedge \exists(A, A'.C'.m') \in \text{policy})$$

The example displayed in Figure 4.2 is free of illicit control flow as the two inter-application method calls, debit of EPurse from JTicket and debit of EMV from EPurse, are authorized by in the policy. In addition, there is no inter-application method call involving a method not part of shared services.

For the transitive case, the previous property must also be verified to guarantee the absence of illicit control flows, but it is not enough to guarantee the absence of illicit transitive calls. To simplify the verification of transitive calls, we compute for each $A.C.m \in$ **Methods** the smallest set $\mathcal{I}(A.C.m) \subseteq$ **Methods** of methods that invoke it directly or indirectly:

$$\mathcal{I}(A.C.m) = \mathcal{I}^{direct}(A.C.m) \cup \mathcal{I}^{indirect}(A.C.m)$$

with

$$\mathcal{I}^{direct}(A.C.m) = \{A'.C'.m' \mid (A'.C'.m', A.C.m) \in \textbf{Messages}\}$$
$$\mathcal{I}^{indirect}(A.C.m) = \bigcup_{A'.C'.m' \in \mathcal{I}^{direct}(A.C.m)} \mathcal{I}(A'.C'.m')$$

The absence of illicit control flow property P is verified in a sequence diagram $SD =$ (**Methods**, **Messages**) annotated with *«controlflow»* stereotype for the transitive case if:

$$P(SD) : \forall A'.C'.m' \in \textbf{Methods}, \forall A.C.m \in \mathcal{I}(A'.C'.m'),$$
$$A \neq A' \implies (A'.C'.m' \in \text{services} \wedge \exists(A, A'.C', m') \in \text{policy})$$

The example displayed in Figure 4.2 is not free of illicit control flow if we assume that the control flow policy is transitive because the debit method of EMV is indirectly called by the refill method of JTicket, which is not permitted by the control flow policy.

4.1.1.3 Evolution strategy

UML models annotated with the *«controlflow»* stereotype do not need to be completely re-checked on each change (addition, deletion or substitution of entities). As part of UMLsec security extensions, we provide a way to incrementally deal with changes relying on the delta ($\Delta = (\textbf{Add}, \textbf{Del}, \textbf{Subs})$) between the original sequence diagram SD and its new version $SD[\Delta]$, as described in Chapter 3. Addition and deletions are dealt equivalently by WP6 methods on impacted byte code, as depicted in Chapters of D6.3 and D6.4 related to direct and transitive control flow models. Subsitutions are not exactly considered as is by WP6, but rather considered as two atomic changes: a removal followed by an addition.

Addition

- **Message**: The addition of a message $(A.C.m, A'.C'.m')$ in SD just requires verifying that $A'.C'.m'$ can be invoked from application A in the non-transitive case for P to hold

$$\exists(A, A'.C'.m') \in SD[\Delta].\text{policy}$$

and that any application that is authorized to call $A.C.m$ is also authorized to call $A'.C'.m'$ in the transitive case for P to hold

$$\exists(A, A'.C'.m') \in SD[\Delta].\text{policy} \wedge \forall(B, A.C.m) \in SD[\Delta].\text{policy}\ \exists(B, A'.C'.m') \in SD[\Delta].\text{policy}$$

If $A \neq A'$, then it is also mandatory to check that $A'.C'.m'$ is a member of the set $SD[\Delta]$.services.

- **Tag services**: Nothing needs to be verified on the addition of a method in services.
- **Tag policy**: The addition of a policy rule $(A, B.C.m)$ does not require any re-verification. If this rule is to be added, it means that it does not appear yet in the set of policy rules SD.policy, and that $B.C.m$ is never invoked from A directly or transitively simply because the SD is secure, and therefore clearly remains secure after this modification is applied.

Deletion

- **Message**: The removal of a message in the sequence diagram SD cannot make it insecure, which is clear from the definition of $P(SD)$.
- **Tag services**: The removal of a service $A.C.m$ in SD.services can be rejected if there exists a message $(A'.C'.m', A.C.m)$ in $SD[\Delta]$ such that $A \neq A'$ as it is not permitted to call a method $A.C.m$ from an application A' if $A.C.m$ is not a service shared by A.
- **Tag policy**: The removal of a policy rule $(A, B.C.m)$ requires to verify that A does not invoke $B.C.m$ (directly or transitively according to the value of $SD[\Delta]$.transitive) but also any method that invoke $B.C.m$ (directly or transitively also) does not permit to be invoked by A according to the rules in $SD[\Delta]$.policy. For the non-transitive case, it means to verify the following statement:

$$\forall A'.C'.m' \in \mathcal{I}^{direct}(B.C.m), A \neq A' \vee \nexists(A, A'.C'.m') \in SD[\Delta].policy$$

And for the transitive case:

$$\forall A'.C'.m' \in \mathcal{I}(B.C.m), A \neq A' \vee \nexists(A, A'.C'.m') \in SD[\Delta].policy$$

Substitution

- **Message**: The substitution of a message $(A_1.C_1.m_1, A'_1.C'_1.m'_1)$ by a message $(A_2, C_2.m_2, A'_2.C'_2.m'_2)$ cannot be dealt more efficiently that the two atomic modifications in sequence: remove $(A_1.C_1.m_1, A'_1.C'_1.m'_1)$, and then add $(A_2, C_2.m_2, A'_2.C'_2.m'_2)$.
- **Tag services**: The substitution of a service $A.C.m$ by a service $A'.C'.m'$ also cannot be dealt more efficiently than the removal the service $A.C.m$ followed by the addition of $A'.C'.m'$.
- **Tag policy**: The substitution of a policy rule (A, m) by a policy rule (A', m') also cannot be dealt more efficiently than the removal the service $A.C.m$ followed by the addition of $A'.C'.m'$.
- **Tag transitive**:
  - from yes to no: nothing to verify as all properties verified in the non-transitive case are also verified in the transitive case;
  - from no to yes: this change requires to completely reverify the whole model.

4.1.2 From model to code

A sequence diagram free of illicit control flows contains some information that can be pushed to code-based verification. These information are inter-method calls expected described in the model and thus expected to occur in the code. Given the **Messages** set of the *« controlflow »* stereotype, code-base verification can check that expected method calls occur where expected, ensuring a basic coherency with the corresponding model.

Several situations can occur during verification of expected method calls at the code level:

- $A'.C'.m'$ is invoked from $A.C.m$ and $(A.C.m, A'.C'.m') \in$ **Messages**, so the code is coherent with the model;
- $A'.C'.m'$ is invoked from $A.C.m$ but $(A.C.m, A'.C'.m') \notin$ **Messages**, so the model is lacking refinement or the code is incorrect;
- $A'.C'.m'$ is not invoked from $A.C.m$ while $(A.C.m, A'.C'.m') \in$ **Messages**, so the code is not compliant with the model or incomplete.

To avoid a large number of incoherency occurrences, especially those related to internal methods not intended to be modeled in sequence diagrams, only the subset **Messages** related to calls to shared services ($A'.C'.m' \in$ services or $A \neq A'$ in the previous rules) can be considered.

4.2 Model-based Information-flow analysis

In this section we describe preliminary work for an interaction between model and code-based verification aimed at detecting unwanted *information flows* within applications.

In particular, WP6 uses byte-code static verification techniques for dealing with non-interference [11]. This verification computes *signatures* to decide on the presence of down-flows. We will introduce a novel stereotype extending UMLseCh that performs a similar static check on *UML state-charts* and that computes the same signatures. The general idea of the integration is to compare the outputs (signatures) produced at the model level with those produced at the code-level, as summarized in Fig. 4.3.

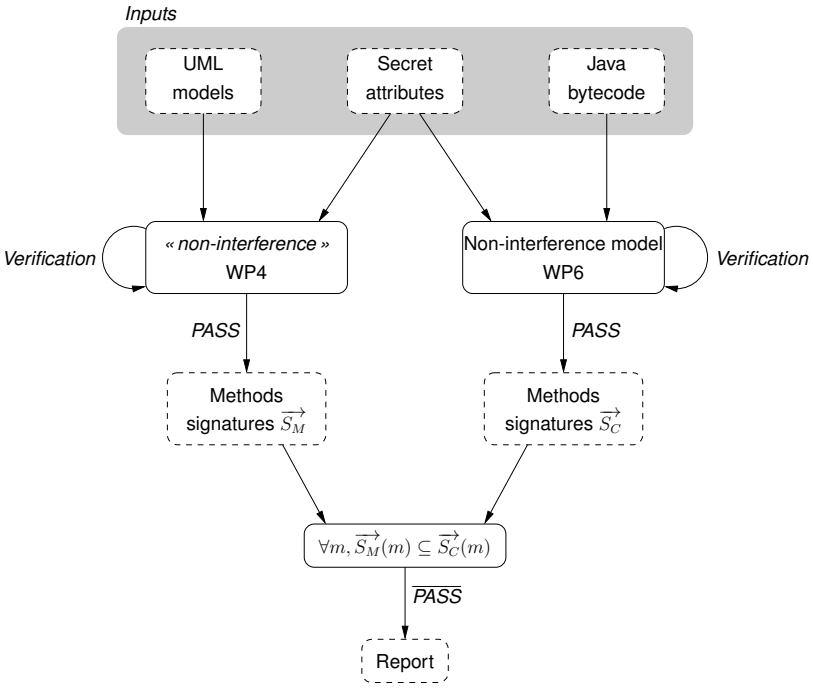


Figure 4.3: General process

4.2.1 Model level

There already exist a stereotype *« no down-flow »* in UMLsec for deciding whether a state-chart has down flows with respect to a set of marked attributes [18] (see Figure 4.4). This stereotype is defined based on a behavioral semantics of state-charts. Currently there exists no tool support for this security check in the UMLsec Tool Suite. In the following we want to propose a new stereotype (*« non-interference »*) whose semantics are inspired in the statically analysis done at the byte-code level in WP6.

The motivation to extend UMLseCh to include this static analysis of non-interference is two- fold: on the one hand a statically analysis would be easier to implement than the behavioral check of *« no down-flow »*; on the other hand the artifacts used to decide the presence of flow ("signatures", see [10] and D6.3) would be common both to the model and the code level, allowing for a comparison of the verification results on the design level and on its implementation.

Semantics Intuitively, *« non-interference »* enforces that a group of state-charts describing the behavior of a class and its methods has no down-flows from attributes marked with

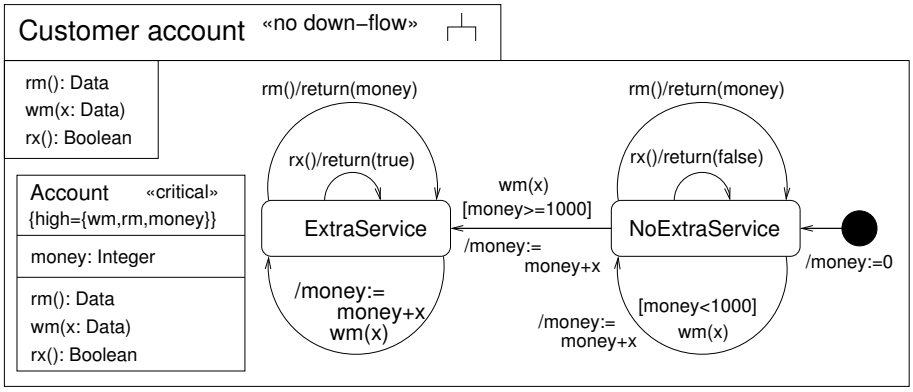


Figure 4.4: Example of the current no down-flow stereotype

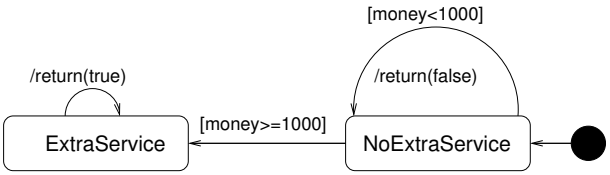


Figure 4.5: Explicit behavior of the rx() method

the tag {secret} to other attributes.

To be able to use the existing *« no down-flow »*, method calls are treated as *events* on a single state-chart, as depicted in the example in Figure 4.4.

In order to be able to perform a statically analysis similar to the one done at the byte code level (that is, in order to compute the same *signatures*) we need the **explicit** behavior of each method. That is, we need as many state-charts as methods for each class. For example, the behavior of the method rx() of Figure 4.4 is depicted on Figure 4.5.

We want to compute signatures of the form:

$$o^{p,s} \xrightarrow{f} s^{p,s}$$

where $o, p \in \{R, p_i, \text{this}\}$ (return value of a method, parameters of a method or local variables of the class), p, s state for secret or public and $f \in \{i, v, r\}$ is the type of the flow relation, i for implicit flows, v for value flows and r for reference flows.

The diagram M will have down flows only if there exist a signature of the form:

$$o^{p} \xrightarrow{f} s^{s}$$

that is a flow from a public value to a secret value.

The core idea is that those signatures can be computed by statically analyzing the *transitions* between states. Since we assume that transitions do not contain events, they only

consist of pairs of the form (G, A) (guard and action). We can then think of them as if constructs of an imperative programming language of the form if G then A. Given this set of state-charts for a class (or set of classes) it is possible thus to extract the signatures as done for the java-code in [9], since at this point it would possible to generate a (very high-level but complete) code of the application modeled in UML.

Soundness The soundness of the static analysis of non-interference relies on abstract memory graphs [11] whereas the soundness of the existing *« no down-flow »* relies on a more classical input/output process-like definition adapted to the behavioral semantics of state-charts. At the present time, we conjecture that these definitions are equivalent, and thus the static analysis at the UML level is sound with respect to the UMLsec behavioral semantics [19].

4.2.2 Model Level vs. Code Level

Similarly as for *« control flow »*, a set of state-chart diagrams free of down flows can 'push' some information to code-based verification. The signatures produced at model-based verification can be compared with the ones computed during code-based verification. In general, since the code refines the model, it is expected that the signatures computed for each method at the code level *exceed* (or are at least the same) the ones produced at the model-level. That is, if Model M has $\overrightarrow{S_M}$ signatures and code C has $\overrightarrow{S_C}$ signatures to check is:

$$\forall\, m \;\in\; \text{Methods(M)}\; \overrightarrow{S_M}(m) \subseteq \overrightarrow{S_C}(m)$$

If this test is not successful for some m, there are two possibilities to be reported:

- The intersection $\overrightarrow{S_M}(m) \cap \overrightarrow{S_C}(m)$ is not empty and $\overrightarrow{S_C}(m) \subset \overrightarrow{S_M}(m)$. In this case, the code is lacking some functionality specified in the model, which contradicts the traditional notion of refinement.
- The intersection $\overrightarrow{S_M}(m) \cap \overrightarrow{S_C}(m)$ is strictly contained in both signature sets (it might be the empty set).In this case the model and the code are incoherent.

Change aspect Although we have not yet analyzed the *« non-interference »* stereotype for its behavior under model evolution (as opposed to the *« controlflow »* case in the previous section), we believe it will be not difficult to reproduce the techniques developed for evolving byte code analysis on the model level as done in WP6.

As stated in the introduction of this chapter, once the verification of the changes is done in parallel one could compare the signatures that actually changed. Suppose we compose a set of verified applications with new applications (thus we *add* new elements to model and code). The changes on the model level are given by a Δ as follows:

$$M[\Delta] = (M \otimes \Delta)$$

The changes on the code level are given by δ:

$$C[\delta] = (C \otimes \delta)$$

Since we are adding new applications, we can safely assume:

$$\mathsf{Class}(M) \cap \mathsf{Class}(\Delta) = \mathsf{Class}(C) \cap \mathsf{Class}(\delta) = \emptyset.$$

We are interested in the signatures that actually changed, so let $\overrightarrow{S'_M}$ and $\overrightarrow{S'_C}$ the new signatures on the model resp. code-level:

$$M[\Delta] \Rightarrow \overrightarrow{S'_M},\ C[\delta] \Rightarrow \overrightarrow{S'_C}$$

Our assumption was:

$$\forall\, m \ \in\ \mathsf{Methods(M)}\ \overrightarrow{S_M}(m) \subseteq \overrightarrow{S_C}(m)$$

Then it follows:

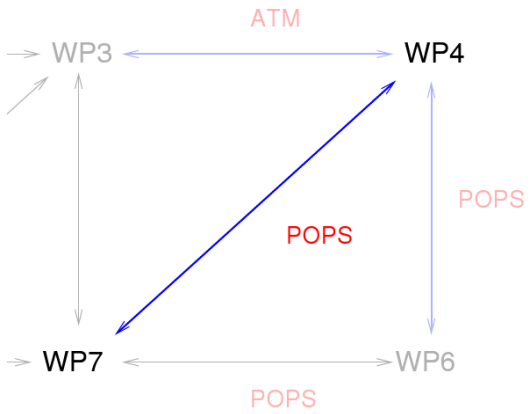
$$\forall\, m \ \in\ \mathsf{Methods(M)}\ \overrightarrow{S'_M}(m) \subseteq \overrightarrow{S'_C}(m)$$

since the signatures for the pre-existing methods remain unmodified.

To check is only:

$$\forall\, m \ \in\ \mathsf{Methods}(\Delta)\ \overrightarrow{S'_M}(m) \subseteq \overrightarrow{S'_C}(m).$$

5 Model-based Testing and Model-based Verification



We describe in this chapter the connection between Work Package 4 (model verification) and WP7 (model-based testing). Based on the Global Platform life-cycle (POPS), this link shows how model-based testing for evolving systems can benefit from the techniques developed in WP4. The general requirement considered is '*Specification Evolution*' and the common property is '*Life-cycle consistency*' [50]. From a methodological point of view the two approaches are complementary, and address a good practice of software engineering: in order to build relevant model based tests, it is mandatory to count on a correct model in the first place.

Typically the models verified against security properties (using for example the UMLsec profile) are explicit models of the *system* design, whereas model-based testing is based on models describing the expected behaviour of an application, seen thus as a *black-box*. The contribution of this common work is the application of the model verification techniques of security properties to the models used in the testing domain, both with and without evolution. With these results the testing engineers can ensure they use a *correct* model with respect to the security properties they are interested in. The integration is based on the POPS case study on the change requirement "Specification Evolution" and the "Information protection" property of the Global Platform (see [50]).

This chapter is organized as follows. We first introduce the general process of coupling model-based testing and model verification in Section 5.1. We then present how the integration works at the levels of *security* (5.2) and *evolution* (5.3).

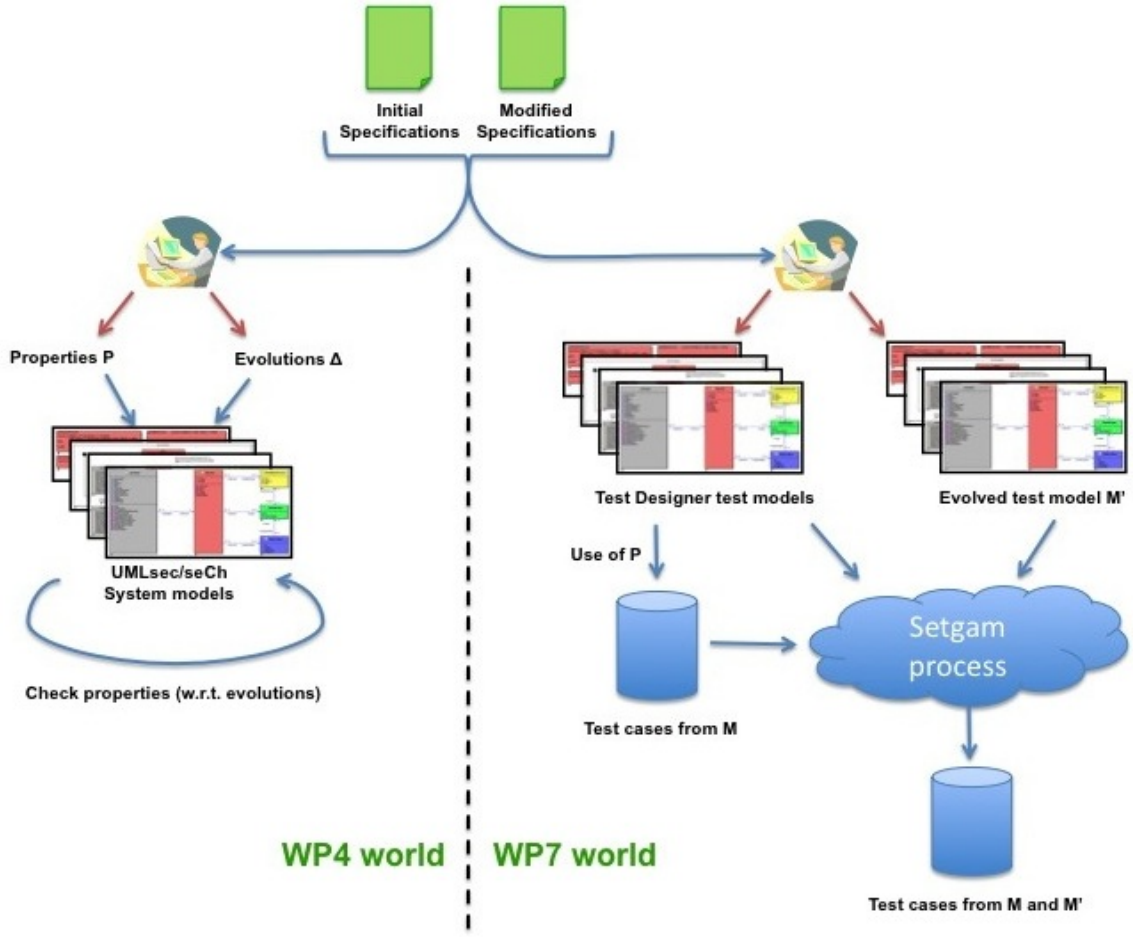


Figure 5.1: WP4 and WP7 approaches without integration

5.1 General process

In the context of the SecureChange project, the WP7 aims at providing a means for taking into account the evolutions and the security in the software validation process. To achieve that, WP7 proposes two solutions to address these issues. First, a dedicated test generation process, based on user-defined test scenarios, is defined. Second, a special process based on an differential analysis of models makes it possible to focus the test generation effort on a subset of the software without sacrifice the overall validation of the whole software.

Regarding WP4 and WP7 contributions to the project, before integration, the respectives processes can be summarized as depicted in Fig. 5.1.

The initial and modified specifications are used independently by the two engineers. On the WP4 side, the system engineer designs a conception model used for system design. He includes the security properties, that can be checked on the model, using the UMLsec approach. During the system's life time, he updates the design with a set of changes to be applied on the model, and the UMLseCh process checks that the evolutions preserve the

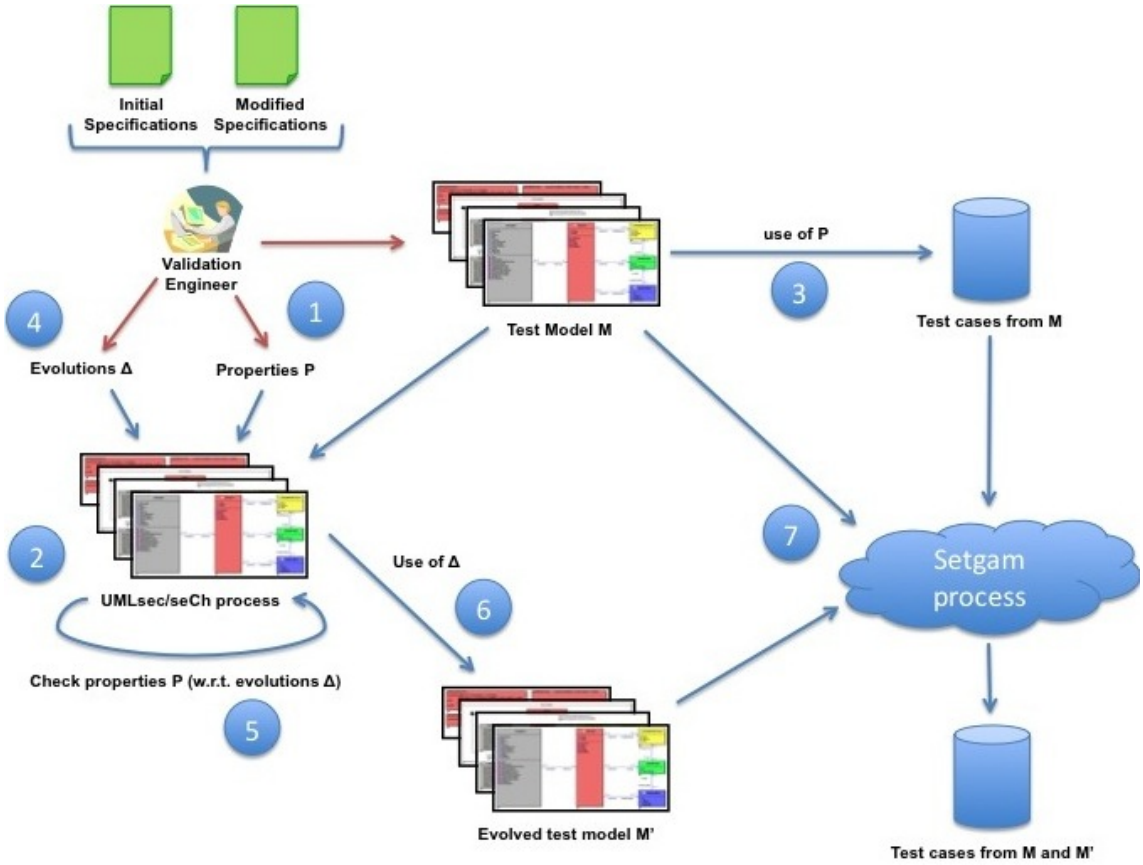


Figure 5.2: WP4 and WP7 integration

security properties. On the WP7 side, the validation engineer designs a test model, and writes test scenarios that are used to produce test cases exercising security properties. Notice that the security properties considered for testing are expressed at a different abstraction level w.r.t. the properties that are verified by UMLsec. When an evolution occurs, he propagates these evolutions to the test model, that is used in the SetGam method (WP7 contribution to SecureChange), along with the original model and the tests that were produced from that model. Notice that the SetGam method starts by computing the differences between the two models.

Our integration proposal is as follows. It relies on the complementarity of verification and testing approaches. Namely, the model that is used for test generation has to be validated regarding the security properties that are considered. If not, the model may authorize an incorrect behavior, and, more importantly, the tests that will be produced will expect the System Under Test to present the same behavior as the model. As a consequence, a faulty implementation would be considered as correct (regarding the execution of the tests), and a correct implementation would be declared as faulty. It is thus mandatory to ensure that the test model respects the security properties on which the tests are based.

The collaboration between WP4 and WP7 is summarized in Fig. 5.2. First, a validation engineer designs a test model and some properties (step 1). He uses the UMLsec ap-

proach (step 2) to validate the model against the security properties, so as to make sure that the model respects the considered properties. Once the model is declared correct, it can be used to produce test cases exercising the properties (step 3). When an evolution occurs, the evolutions are described as the set of changes between the two model versions (step 4). The UMLseCh process is then in charge of validating the evolutions w.r.t. the security properties (step 5). Once the preservation of the security properties by the evolutions is ensured, the set of evolutions is used to automatically compute the evolved version of the test model (step 6). The SetGam process can then be applied to produce test cases validating the evolutions w.r.t. the security properties.

The outcome of the integration is the following:

- WP4 enhances its verification activities not only to conception models, but also to test models, and the kind of properties that are verified on the model. In addition, both approaches are now used by the same actor (the validation engineer).
- WP7 benefits from the already existing computation of the differences that is helpful at two levels: (i) it makes it possible to automatically compute the new version of the model, and (ii) it avoids computing the model differences at the beginning of the SetGam process.
- The complementarity between verification and testing is clearly shown.

We now present in the following sections how integration is concretely done.

5.2 Security

In this section we show how the security Model-Based Testing (MBT) process benefits from model verification (see 3). We first recall the considered test generation process and then show the proposed verification approach based on the case study. In particular, we consider the following two properties:

Property 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**"

Property 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**"

5.2.1 Background of Test Generation Process

As explained previously, our test generation process aims at generating model-based tests. These test cases are thus computed by animating the model, i.e. by simulating its execution through its formal description.

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

Figure 5.3: Syntax of the TestDesigner scenarios language

In order to build security-based test cases, we rely on the use dedicated test scenarios that describe either nominal test cases, aiming at illustrating the considered property (i.e. the preservation of secrecy, the denial of an access to a specific security asset, etc.), or aiming at checking the robustness of the system towards security.

To describe scenarios, a dedicated scenario language has been implemented as a plug-in of TestDesigner, version $4.1.2$. The syntax of the scenario language is given in Fig. 5.3.

Roughly speaking, the language makes it possible to design test scenarios as a sequence of steps, each step being composed of a set of operations (possibly iterated at least once, or many times) and aiming at reaching a given target (a specific state, the activation of a given operation, etc.).

Example 1 (Scenario example)**.** *Consider an example from the GlobalPlatform case study. One wants to test a security property of access control specifying that when the card is put in the terminated state by means of a privileged application, then it should not be possible to revert to another state.*

A test scenario exercising this property would act as follows. First, it selects an application that has the appropriate privileges (i.e. those required to change the status of the card).

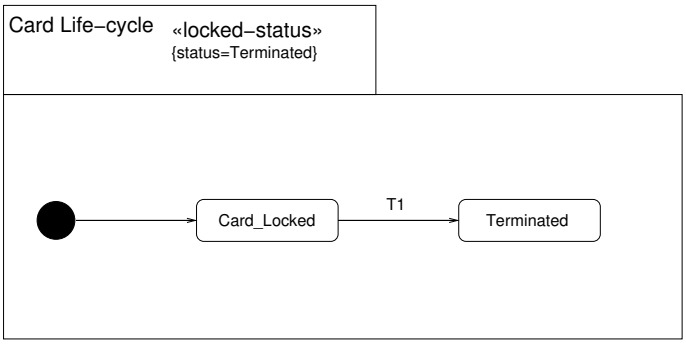


Figure 5.4: Fragment of the card life-cycle

Then, it applies the dedicated operation that sets the card state to terminated. Finally, it ensures that the card status can not be changed by checking if all operation activations possibly lead to a change of state.

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

5.2.2 Correctness verification

In order to check whether the model used for the test generation is correct in the first place, one could use the stereotypes defined for UMLseCh in Section 3.2. We recall here briefly the definition:

« locked-status » together with the tag {status} applied to a state-chart diagram checks whether there are any transitions going out of the status matching the identifier contained in the {status} tag. If there are, the check fails, otherwise it passes. This corresponds to Property 1.

For example, in Figure 5.4, one can statically check for a fragment of the Card-life cycle that there are not outgoing transitions from the Terminated status to any other status. This corresponds to the Example 1 of the previous subsection.

« authorized-status » together with tags {status} and {permission} checks whether the status specified in {status} is reachable by a path not containing a permission as specified in {permission} in the last transition (by parsing the guards associated to the transitions). This enforces Property 2.

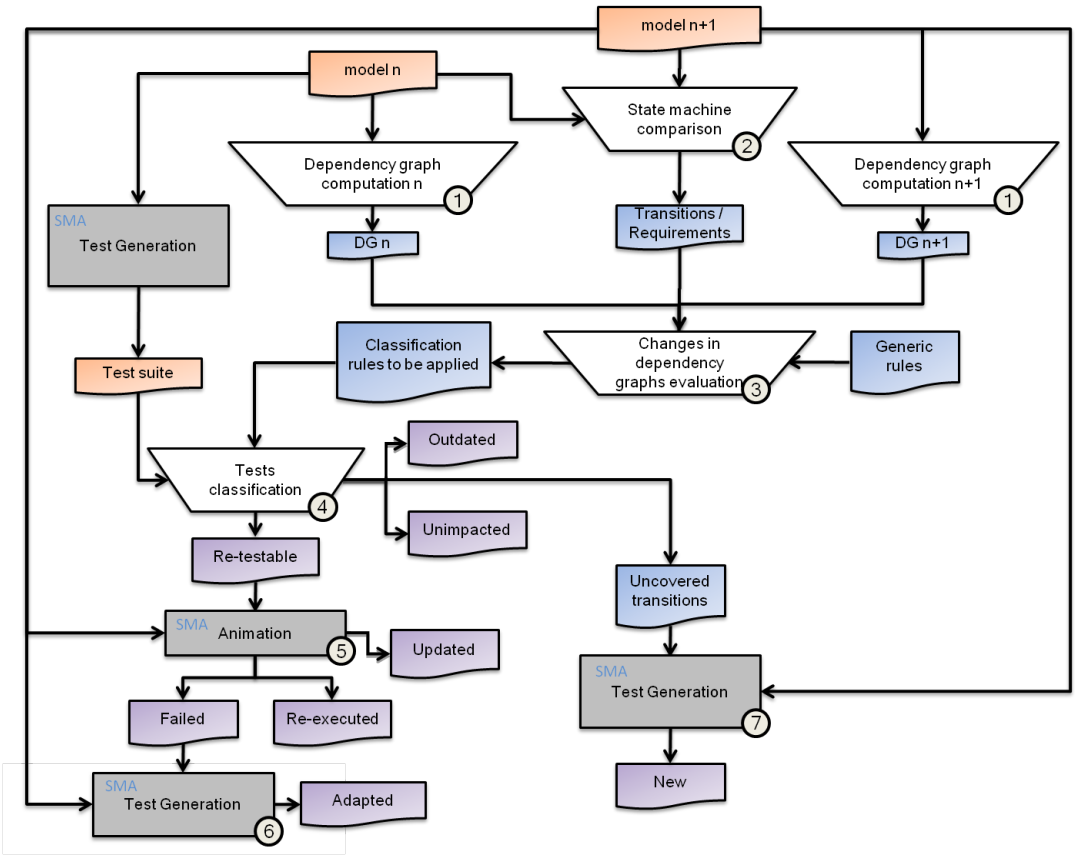


Figure 5.5: The SetGam process

In case any of these two checks fails in the Models used for testing generation, the properties the testing engineer wants to check on the system would be obviously wrong from the beginning, since the expected behavior of the system would be contradicting the properties.

5.3 Evolution

The second integration point concerns the evolutions of the model. The UMLseCh approach considers a set of possible evolutions for which the preservation of security properties has to be ensured.

5.3.1 Test Generation Process under Evolution

The test generation process regarding evolutions is as follows. It takes as input two formal models, one representing the system before evolution and another representing the same system after the evolution took place. It also considers a set of test cases applicable/computed from the original model.

The WP7 process, called SetGaM, is depicted in Fig. 5.5. It starts by a dependency analysis of the original and the evolved model. This dependency analysis aims at identifying data and control dependences inside the model (1). Then, the models statecharts are compared (2) to identify the changes between them and their impact w.r.t. the existing test suite (3). Then test cases from the original test suite are classified (4), with the help of model animation (5), so as to identify:

- outdated tests, that are no longer relevant w.r.t. the new version of the software (to be used for stagnation testing)
- unimpacted tests, that do not cover evolved parts of the system/requirements and are thus still relevant w.r.t. the new version (to be used for regression testing)
- updated tests, that are an updated version of an existing test case for which the oracle had to be re-computed (to be used for evolution testing).
- adapted tests, that cover already existing part but because of the evolution they failed, and for which we need to recompute the test sequence.
- re-executed tests, that cover evolved parts of the system/requirements and need to be recomputed (to be used for evolution testing) (6)
- new tests, that have to be computed for new parts of the software that did not exist before (7).

5.3.2 Benefiting from the UMLseCh approach

For the stereotypes *« locked-status »* and *« authorized-status »*, a sound decision procedure for evolving state-chart diagrams has been described in Section 3.2 of this Deliverable. One could therefore check whether the evolution depicted in the example in Figure 5.6 respects *« locked-status »*. In this example a new transition ($T2$) is added to the Status Card_Locked, and the Guard of transition $T1$ is substituted with a new guard.

This way the testing engineer can model expected evolutions in the testing model and testing for correctness of the evolved models without re-running the verification from scratch and thus benefiting from the WP4 methodology.

Moreover, one can further exploit the UMLseCh notation to compute the delta between a model and its evolution, and give it as an input to the WP7 testing generation methodology. On one hand, WP7 evolution testing approach considers two models and computes their differences. On the other hand, UMLseCh can return a set of possible evolutions for a considered model.

Thus, our integration work consists at this level consists in:

- working on a common study, which is the Global Platform Life Cycle scope
- using UMLseCh to specify one evolution of the model (a Δ between the original and evolved model depicted in Fig. 5.5).
- export this evolution into an XML file to provide it as an input for the WP7 process.

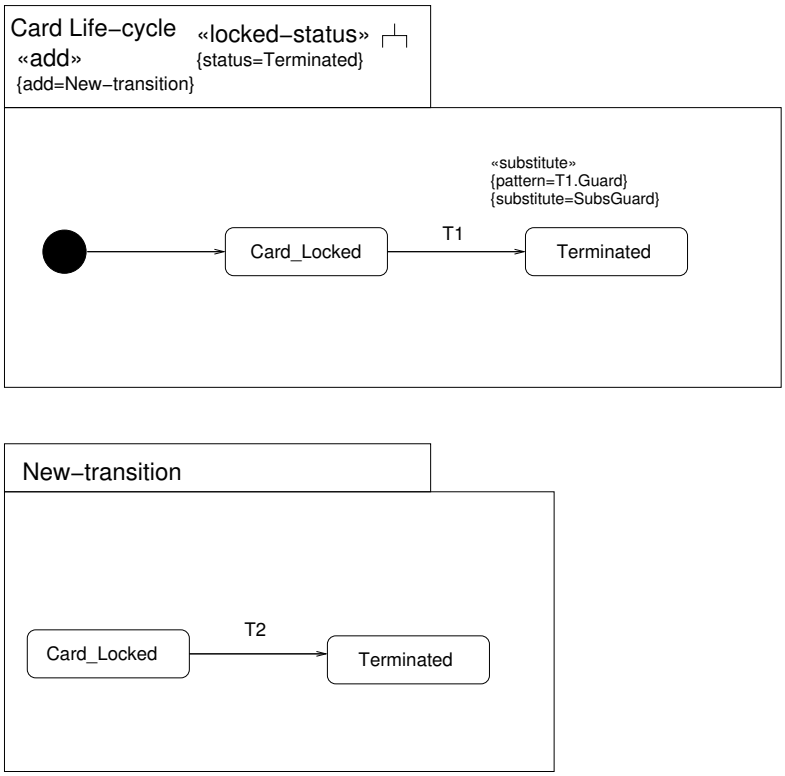


Figure 5.6: An evolving fragment of the card life cycle model

UMLseCh can send a set of possible evolutions for a given model:

- the addition of a new model entity (class, state, etc.)
- the deletion of an existing model entity
- the substitution of one model entity by another new one.

These additions, deletions and substitutions are specified in dedicated stereotypes in the corresponding UML diagram: *add, del and substitute/substitute-all*. Thus, for our case study scope with respect to the defined security properties in 5.2 and the evolution, the UMLSeCh model is defined as depicted on the figure 5.6.

Notice that the two approaches consider models that are designed using different UML modeling tools (ArgoUML for UMLseCh vs. IBM Rational Software Architect for TestDesigner). The switching from one of the notations to the other is not considered, since such activity is highly time-consuming and would not respond to a concrete need for integration. Thus, our integration solution has to be the less invasive as possible in terms of adaptation of the existing tools. That is why we have decided to create a common XML exchange file, containing the information about each statechart evolution *(Addition, Deletion, Substitution)*.

Addition: We have created the XML marker *<add>* to present addition of a transition. As described at the File 1 we can specify its *the name, the source, the target, the event,*

File 1: <add>,<del>,<sub> XML markers

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

the action and the guard.

Deletion: The XML marker *<del>* represents the deletion of a transition. As depicted in the File 1 we put the set of elements that are deleted w.r.t the evolution.

Substitution: Finally, all smallest substitutions (XML marker *<sub>*) are gathered as a set of substitutions (marker *<sub-all>*). To simplify the process of substitution for the engineer, we give the name of the concerned transition, than we give the element that is changed: source, target, event, guard or action of the transition, and the information about what to substitute. As given in the example 1, the concerned transition is *t3* and we need to make a substitution of the guard *G2* and the transition's target *Ty*.

With these XML marks we are able to create the XML exchange file and we can easy note the additions, deletions and substitutions in a statechart. In the Appendix 5.4 you can find the file corresponding to the Global Platform scope results of the applied approach. Moreover, you have an example of addition of the transition *setStatus* and substitution of the guard of the transition *setStatus*.

Thus, having Δ it will be possible for WP7 to:

- build the new model resulting from the described evolution,

- avoid computing the difference between the models, since it will be directly provided by Δ,
- apply the rest of the methodology without any interference.

5.4 The GP Life-Cycle

Until now we have shown an example of a state-chart inspired in the GP Life-Cyle. In this section we present a fragment of the delta XMI for the actual Model of the GP Application in Listing 2. This delta represents the change from the life-cycle model for testing done by WP7 for Versions 2.1 and 2.2 of the GP, a fragment of which are represented in Figures 5.7 and 5.8.

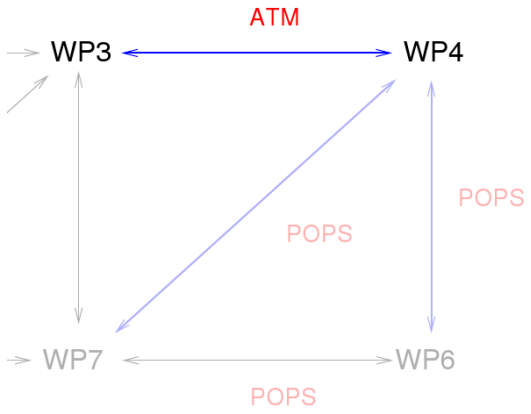
Figure 5.7: Fragment of the life-cycle in V2.1 of the GP

Figure 5.8: A new transition is introduced between Card_Locked and Terminated in V2.2 of the GP

File 2: *delta* xml exchange file for GlobalPlatform sub-scope

```
<add>
   <element type='transition'
            name='setStatusCardLockedToTerminated_privilegedApp'>
   <event name='APDU_setStatus' />
   <source name='Card_Locked' />
   <target name='Terminated'/>
   <guard>
            self.lcs->exists(lc : LogicalChannel |
            ...
            lc.selectedApp.privileges.cardTerminate = true and
            lc.selectedApp.privileges.securityDomain = false
            )
        ...
   </guard>
   <action>
   ...
   self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS
   /**@AIM: FROM_CARD_LOCKED */
   /**@REQ: APDU_SETSTATUS_SUCCESS_APP_CARD_LOCKED_TO_TERMINATED */
   </action>
   </element>
</add>
<sub-all>
   <sub type='transition'
            name='setStatusCardLockedToTerminated_privilegedSD'>
             <guard>
             self.lcs->exists(lc : LogicalChannel |
             ...
             lc.selectedApp.privileges.cardTerminate = true and
             lc.selectedApp.privileges.securityDomain = true
              )
             ...
            </guard>
   </sub>
</sub-all>
```

6 Integration of Thales Security DSML with UMLseCh and application to the ATM Use Case



This chapter contains the integration link presenting a connection between the modeling and verification techniques developed by WP4 with WP3 (Requirements) based on the ATM Case Study. A risk analysis done with the Thales Security DSML gives high-level security requirements, which are reflected in the System Design and analyzed by means of the UMLseCh approach. The general requirement considered is '*Organizational Level Change*' and the properties considered are '*Information Access*' and '*Information Protection*' [50]. Both Security DSML and UMLseCh provide a Domain Specific Modelling Language that captures security concepts and enables to annotate a model design.

Security DSML, developed at Thales after EU-FP6 Modelplex project shall be regarded as a security viewpoint of a system model design tool in the sense where viewpoint is intended as a technology to provide non functional properties tooling integrated to a system engineering workbench. This technology is the focus on French research project Movida (ANR – Call 8). In the last progress of the work for Secure Change project, Security DSML has been integrated as a security viewpoint above model design tool Papyrus UML. Security DSML focuses on a risk management process at system design phase.

UMLseCh, developed within Secure Change under the lead of TUD, is a formal verification tool that permits to validate formally security properties of a design model.

The example developed below shows how the two tools can enrich a system design process by providing argumentation for the security design of the system. The second section explains the overall process and terminologies used in the two tools.

The third section describes the ATM use case scenario at Organisational Level Change

by the introduction of the Arrival Manager (AMAN) and the Sequence Manager (SQM).

The fourth section shows how Security DSML is used to study the security risks of a design model and how this risk management process produces security requirements as output.

The fifth section shows how the security requirements produced by means of the Security DSML can be captured in UMLseCh formal language and used to validate the consistency of the system model with the new requirements.

Security DSML and UMLseCh demonstration overall process and terminologies

Starting from a model design, Security DSML enables to conduct a risk analysis. The risk management phase following it produces Security Objectives, which are in their turn refined in Security Requirements. These security requirements lead to an evolution of the model since security solutions shall be implemented to complete or transform the model. This is how security engineering and security evolution as studied through Secure Change project shall improve Thales system engineering methodology.

After having proceeded to model evolution thanks to a security analysis study with Security DSML tools, UMLseCh tooling provides a nice-to-have tool to validate model consistency with some of the security requirements produced as output of the security analysis. The purpose of UMLseCh indeed is the validation of security properties of activity diagrams. Actually, some Security Objectives and Security Requirements correspond to security properties that can be expressed using UMLseCh annotations.

In figure 6.1, in the first step, the system architect starts modelling the business architecture or processes.

The risk manager, in the second step, gets the model started by the System Architect, and analyses the risks at business and service level. He updates the risk model which annotates the system model. In order to cover the risks, he defines new security objectives and security requirements, and propagates these requirements to the requirements models.

In the third step, the requirements engineer then gets the requirements, and assesses them.

Once the requirements are accepted, the System Engineer, in the fourth step, translates them into solutions, and models the Logical architecture and the Physical architecture.

As the system engineer, the security engineer, in the fifth step, translates the security requirements into solutions, and changes the Logical architecture and the Physical architecture in order to show the security measures to be implemented.

The sixth step is the verification by the UMLseCh tool, of the consistency of the model.

In the seventh step, the system architect proceeds to the overall mitigation and assessment of the system model.

In the example described below, both tools start with similar activity diagrams of a process. The use case focuses on Role Base Access Control specification. This detailed

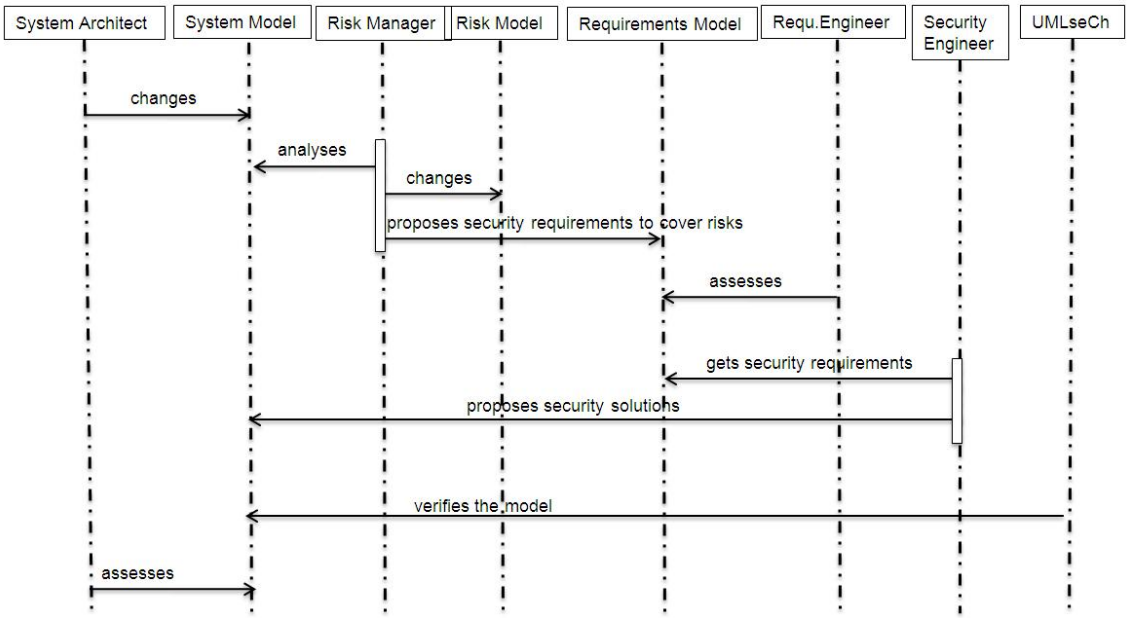


Figure 6.1

requirement can be expressed with both languages. Role Base Access Control rules expressed as Security Requirements with Security DSML can be captured with UMLseCh formal notation, and thus validated towards the activity diagram. In that way, UMLseCh enables consistency checking validates the output of some of the security requirements expressed by means of Security DSML.

Figure 6.2 presents a more complete mapping between the types of concepts used in both languages.

The first column at the right shows UMLseCh stereotypes. The second column is used to describe eventual UMLseCh tags.

The following columns to the rights indicate to which Security DSML concepts they correspond. The third column refers to Thales segmentation of its security model. The fourth column shows to which Security DSML concept or meta-class the UMLseCh stereotype or tag corresponds. A fifth column is needed to note which instance of Security DSML concept the UMLseCh stereotype or tag corresponds. As often the semantic for a security concept in UMLseCh is not the same in Security DSML (a same word does not mean the same with regard to UMLseCh or with regard to Security DSML default ontology, we proposed to create new concepts in Security DSML as referring to UMLseCh ones. A comment explicates this in the last column.

6.1 ATM use case scenario at Organizational Level Change

The purpose of the following sections is to present a change scenario and demonstrate how UMLseCh and Thales' security DSML can be leveraged to implement it in a secure

Mapping some security properties from UMLseCH to Security DSML

UMLseCh stereotypes	UMLseCh tags	DSML Models	DSML Elements	DSML Elements instances	Comment
critical		Risk	Damage condition		
secrecy			Security criterion (Need)	confidentiality	Confidentiality of the data
integrity			Security criterion (Need)	UMLseCh_Integrity	Integrity of the data. Warning DSML integrity and UMLseCh integrity are not similar
high			Security Objective Level	UMLseCh_high	Security Objective Level values are {not applicable, low, medium, high, critical}
Internet		Context	Target		
encrypted			Target		
LAN			Target		
wire			Target		
smard card			Target		
POS device			Target		
issuer node			Target		

Mapping some security properties from UMLseCH to Security DSML

UMLseCh stereotypes	UMLseCh tags	DSML Models	DSML Concepts	DSML Concepts values	Comment
secure links		Requirement	Security objective	UMLseCh_SecureLink	The link is secure according to Threat T
	adversary				adversary values are default, insider
secure dependency			Security objective	UMLseCh_SecureDependency	The perimeter has the security objectives expressed by the stereotype critical
data security			Security objective	UMLseCh_DataSecurity	The data has the security objectives expressed by the stereotype critical or according to Threat T
	adversary				
	integrity				
	authenticity				
	secrecy				
provable			Security objective	UMLseCh_Provable	The action shall be traceable.
	action				
	cert				
	adversary				
guarded access			Security objective	UMLseCh_GuardedAccess	
guarded			Security objective	UMLseCh_Guarded	
	guard				
fair exchange (for use case)				UMLseCh_FairExchange	any transaction should be performed in a way that prevents both parties from cheating
	start				
	stop				
	adversary				
fair exchange (for activity diagrams)				N/A	
no down-flow			Security objective	UMLseCh_NoDownFlow	It enforces sensitive data tagged with the tag {high} to be associated with the stereotype «critical».
	high				
no up-flow			Security objective	UMLseCh_UpDownFlow	
	high				
rbac			Security requirement	Role Based Access Control	A given perimeter is protected by RBAC. Authorization are granted for given rights to given roles.
	protected				
	role				
	right				

Figure 6.2: Mapping between UMLseCh and Thales Security DSML.

way. This section describes the organizational change of the ATM use case, namely the introduction of an Arrival Manager, from a role-based access control angle. Section 6.2 realizes a risk management process of the change in the Thales environment, while Section 6.3 explains how UMLseCh can be used to validate the results of the risk management process. This approach can contribute to the design of an Arrival Manager secure interface for the different Air Traffic Controllers (ATCOs) roles in an ACC.

The Arrival MANager (AMAN) is an aircraft arrival sequencing tool helping to manage and better organize the air traffic flow in the approach phase.

Arrival Management is a very complex process, involving different actors. A high level description of the Arrival Management process could be:

- Setting Goals (e.g. maximum usage of runway capacity, minimizing noise or fuel consumption).
- Creating a plan to meet the goals.
- Monitoring the conformance to the plan.
- Adjusting/updating the plan if necessary.

Before the introduction of the AMAN, the sequence creation and adjustment was carried out by the Sector Team, in particular by the Tactical Controller with the Planner Controller support. The main functionalities of the AMAN are

- The creation of an arrival sequence using 'ad hoc' criteria.
- The management and modification of the proposed sequence.
- The provision of data to the HMI to allow controllers to implement the proposed sequence.
- The support of runway allocation at airports with multiple runway configurations.
- The generation of advisories on: (1) Time to lose or gain, (2) Speed, (3) Top-of-descent, (4) Track extension, holding.

The computation of the sequence is carried out no more by the ATCOs, but by the AMAN tool itself. Moreover, a new role in the ACC has been introduced: the Sequence Manager (SEQ MAN), that will monitor and modify the sequences generated by the AMAN and will provide information and updates to the Sectors' Teams. After AMAN introduction, ATCOs have different privileges according to their role. For example, the Sequence Manager can modify the sequence of arrivals provided by the AMAN, while the Tactical (TCC) and Planner (PLC) can only view it. Thus, the AMAN tool needs different functionalities and subsequent access rules for different ATCOs roles. ATM Engineers customizing the AMAN for an ACC have the problem to design suitable Role-Based Access Control and Users Interfaces for the AMAN tool.

Those roles and activities can be presented in a semi-formal way in the form of a UML activity diagram. Figure 6.3a presents the roles of the PLC and the TCC before the

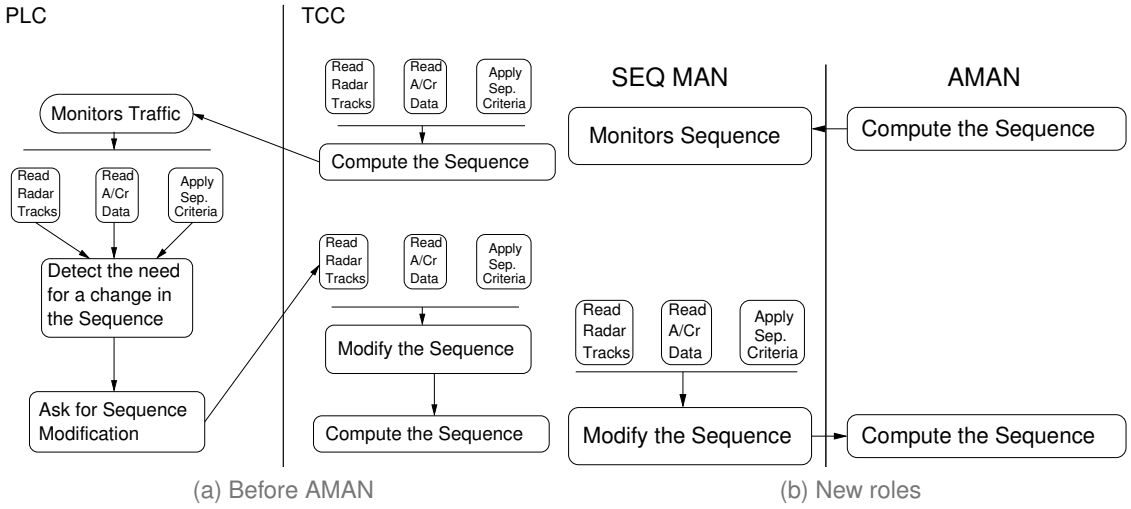


Figure 6.3: Roles and activities.

introduction of the AMAN. Figure 6.3b presents the new roles of the SEQ MAN and the AMAN. Crucial activities such as 'Compute the sequence' and 'Modify the sequence' have been reassigned. Our study will concentrate on the task of maintaining a secure role-based access control for them.

6.2 Risk Management with Security DSML

The purpose of this step is to perform a security Risk Management study on a system model related to the ATM use case described above.

System Model tool Papyrus UML is used to model the system. Here, the focus is put on the computation of the aircraft (A/C) sequence and its modification.

The overall methodology used can be summarized with the following process (see figure 6.4):

1. A risk analysis is made on a design model, which leads to a Risk change request.
2. Security Objectives and Requirements are defined in order to cover the risks.
3. Since Requirements concern the introduction of the AMAN, a Design change request is performed in order to introduce the two new roles and their respective activities.

The first sub-section describes the activities modelled with Papyrus, the second sub-section show the step of the risk analysis, the third sub-section indicates the need for a change in the process. Section 6.3 shows how UMLseCh can prove that the Design Change Requests proposed were incomplete, which leads to additional Design Change Request.

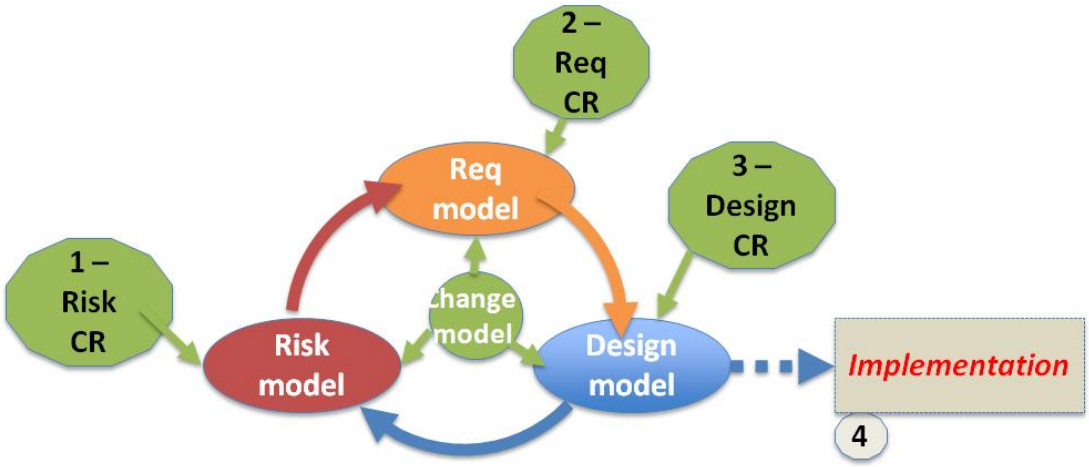


Figure 6.4: Overall Thales process.

6.2.1 Activities modelling with Papyrus

Activity diagrams are used under Papyrus to describe the activities of the ATM scenario accomplished by different roles of Air Traffic Controllers such as Tactical Controller (TCC), Planning Controller (PLC).

After the introduction of the Arrival Manager (AMAN) automatic computation engine, two roles are added, the AMAN and the Sequence Manager (SQM).

Before the introduction of the AMAN, the activities are the following:

- **PLC monitors the traffic:**
  PLC monitors traffic on his/her Controller Working Position.
- **PLC detects the need for a change in Sequence:**
  by means of the sub-activities performed in parallel such as:
  - Read Radar Tracks
  - Read A/C Data
  - Apply Separation Criteria[1]

  PLC detects the need for a change in Sequence.
- **PLC asks for a Sequence Modification:**
  After having detected the need for a change in the Sequence, PLC asks TCC for a Sequence modification.
- **TCC computes the Sequence:**
  by means of the sub-activities performed in parallel such as:

[1]Separation criteria shall be applied by Air Traffic Controllers in order to guaranty a safe separation of the aircraft in a sequence of arrivals.

  - Read Radar Tracks
  - Read A/C Data
  - Apply Separation Criteria[1]

  TCC computes the aircraft sequence.

- **TCC modifies the Sequence:**
  by means of the sub-activities performed in parallel such as:

  - Read Radar Tracks
  - Read A/C Data
  - Apply Separation Criteria

  TCC modifies the aircraft sequence.

6.2.2 Risk analysis performed with Security DSML

The overall process for the risk analysis is summarized in figure 6.5.
The activities performed are:

- Identifying essential elements, aka identifying the perimeter of the study
- Analysis of the damages
- Determination of the targets
- Determination of the vulnerabilities
- Analysis of the threats
- Definition of the risks
- Definition of the confinement zones
- Definition of the Security Objectives
- Definition of the Security Requirements

6.2.2.1 Perimeter of the study / Identify Essential elements

The Security DSML is used over Papyrus design tool in order to perform a risk analysis. The perimeter considered for the risk analysis consists of the activity diagrams drawn with Papyrus.
The activities considered are the following:

- **PLC monitors the traffic**
- **PLC detects a need for a change in the Sequence**

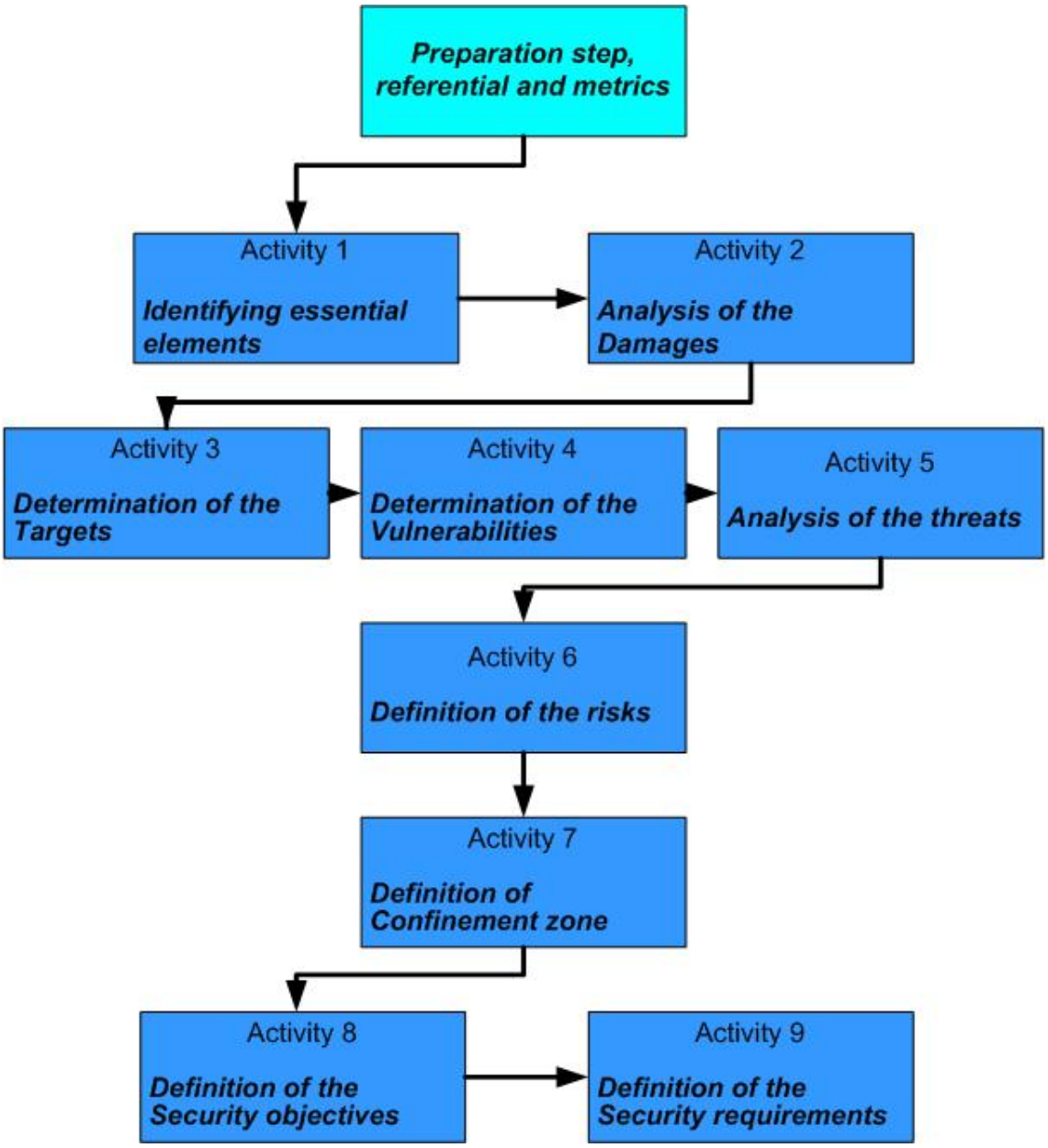


Figure 6.5: Overall risk analysis process.

- **PLC asks for a Sequence modification**
- **TCC computes the Sequence**
- **TCC modifies the Sequence**

The risk analysis performed with WP5 with the domain experts from Deep Blue provides relevant inputs to analyse these activities.

6.2.2.2 Analysis of the Damages

A list of possible damages is identified in relation with the activities above, with an impact level. For each damage of the list below, the impacted activities are indicated.

- **Loss of information provisioning to/from ATCOs**: Critical
  - TCC computes the Sequence
- **Failure in the provisioning of correct arrival information**: High
  - TCC modifies the Sequence
- **Failure in the provisioning of optimal arrival information**: Medium
  - PLC detects a need for a change in the Sequence
  - PLC asks for a Sequence modification
  - TCC computes the Sequence
  - TCC modifies the Sequence

6.2.2.3 Determination of the targets

The targets are the activities.

6.2.2.4 Determination of the Vulnerabilities

Vulnerabilities identified on the different actors and activities are the following. For each vulnerability of the list below, the related activities are indicated.

- **High coordination workload**
  - PLC detects a need for a change in the Sequence
  - TCC computes the Sequence
- **Non-compliance of ATCO with procedures**
  - PLC detects a need for a change in the Sequence
  - TCC computes the Sequence

- **Stress, concentration problems, health conditions, etc.**
  - TCC computes the Sequence
- **Lack of routines for avoiding multitasking**
  - TCC computes the Sequence
- **Overload of traffic; high workload**
  - TCC modifies the Sequence

6.2.2.5 Analysis of the Threats

Threats are listed below with a probability level. For each threat, the related activities are indicated.

- **ATCO mistake**: High
  - PLC detects a need for a change in the Sequence
  - TCC computes the Sequence
- **Non-compliance of ATCO with procedures**: Low
  - PLC detects a need for a change in the Sequence
  - TCC computes the Sequence
- **TCC unavailability**: Low
  - TCC computes the Sequence
- **TCC overloaded**: High
  - TCC computes the Sequence
- **ATCO fails to manually update the system**: Medium
  - TCC modifies the Sequence

6.2.2.6 Definition of the Risks

The following risks are identified, with Severity and Opportunity and overall risk level. For each risk, the activity, vulnerability and threat are indicated.

B17 Failure in the provisioning of correct or optimal arrival information (stabilization or coordination of sequence) due to ATCO mistakes

- Severity: High, Opportunity: High, overall risk level: **High**
- Activity 1: PLC detects a need for a change in the Sequence; Damage 1: Failure in the provisioning of correct or optimal arrival information; Vulnerability 1: High coordination workload ; Threat 1: ATCO mistake

- Activity 2: TCC computes the Sequence; Damage 2: Failure in the provisioning of correct or optimal arrival information ; Vulnerability 2: High coordination workload ; Threat 2: ATCO mistake

B16 Failure in the provisioning of correct or optimal arrival information due to non-compliance of ATCO with procedures

- Severity: High , Opportunity: Low, overall risk level: **Medium**
- Activity 1: PLC detects a need for a change in the Sequence; Damage 1: Failure in the provisioning of correct or optimal arrival information; Vulnerability 1: Non-compliance of ATCO with procedures; Threat 1: Non-compliance of ATCO with procedures
- Activity 2: TCC computes the Sequence ; Damage 2: Failure in the provisioning of correct or optimal arrival information; Vulnerability 2: Non-compliance of ATCO with procedures; Threat 2: Non-compliance of ATCO with procedures

B13 Tactical Controller (TCC) becomes unavailable during arrival management process due to his/her physical/mental condition

- Severity: Critical , Opportunity: Low, overall risk level: **High**
- Activity: TCC computes the Sequence; Damage: Loss of information provisioning to/from ATCOs; Vulnerability: Stress, concentration problems, health conditions, etc.; Threat: TCC unavailability

A12 TCC fails to provide arrival information to all relevant recipients simultaneously due to communication overload (radio with A/C, voice with PLC)

- Severity: Critical , Opportunity: High, overall risk level: **Critical**
- Activity TCC computes the Sequence; Damage: Loss of information provisioning to/from ATCOs; Vulnerability: Lack of routines for avoiding multitasking; Threat: TCC overload

B18 ATCO fails to manually update the system which leads to the provisioning of inconsistent data

- Severity: Medium, Opportunity: Medium, overall risk level: **Medium**
- Activity: TCC modifies the Sequence; Damage: Failure in the provisioning of correct arrival information; Vulnerability: Overload of traffic; high workload; Threat: ATCO fails to manually update the system

The definition of these new risks implies the emission of a Risk Change Request for these risks to be considered in the secure engineering process.

6.2.2.7 Definition of the confinement zone

In the context of this risk analysis performed over activities of a process, this activity corresponds into deciding what level of risk is considered as unacceptable and therefore the type of risk management shall be put in place.

In the domain of Air traffic control, risks with risk level from critical to medium shall be managed.

6.2.2.8 Definition of the Security Objectives

According to the risk analysis performed within WP5, a list of security objectives was listed as requirements from the domain experts.

- **ATM_SEC_1**: The system shall maintain consistency between data and presented information.
- **ATM_SEC_2**: The system shall be equipped with suitable security mechanisms to prevent from corruption of data.
- **ATM_SEC_3**: The system shall be equipped with suitable security mechanisms to prevent from accidental loss of data.
- **ATM_SEC_4**: The system shall be equipped with suitable security mechanisms to prevent from intentional loss of data.
- **ATM_SEC_5**: The system shall guarantee the integrity and confidentiality of data against illicit attempt to obtain access to such data.

In order to cover the risks defined through the Risk analysis process, the following additional Security Objectives are defined. For each Security Objective, the related risks are indicated:

O1 The Sequence shall be computed automatically by an Arrival Manager system.

- **B17** Failure in the provisioning of correct or optimal arrival information (stabilization or coordination of sequence) due to ATCO mistakes
- **B16** Failure in the provisioning of correct or optimal arrival information due to non-compliance of ATCO with procedures
- **B13** Tactical Controller (TCC) becomes unavailable during arrival management process due to his/her physical/mental condition
- **A12** TCC fails to provide arrival information to all relevant recipients simultaneously due to communication overload (radio with A/C, voice with PLC)

O2 The update of the system should be handled through a dedicated role of Sequence Manager.

- **B17** Failure in the provisioning of correct or optimal arrival information (stabilization or coordination of sequence) due to ATCO mistakes

B18 ATCO fails to manually update the system which leads to the provisioning of inconsistent data

All the risks are covered by at least one security objective.

6.2.2.9 Definition of the Security Requirements

The following security requirements are defined in order to refine the security related objectives as indicated.

- **Req01**: Access Rights to AMAN on Computing the Sequence

  O1 The Sequence shall be computed automatically by an Arrival Manager system

- **Req02**: Access Rights to SQM on Monitoring the Sequence

  O2 The update of the system should be handled through a dedicated role of Sequence Manager

The definition of these new security requirements implies the emission of a Requirements Change Request for these requirements to be taken into account in the security engineering process.

This is the end of the risk analysis.

6.2.3 Requirements lead to a change in the design

The previous risk analysis process leads to a requirement change request which contains two requirements:

- **Req01**: Access Rights to AMAN on Computing the Sequence
- **Req02**: Access Rights to SQM on Monitoring the Sequence

In order to cover these two requirements, a Design Change Request shall be emitted. The following two activities shall be

- **Modify the Sequence by the SQM:** This activity replaces the modification of the sequence by TCC described above by the same activity handled by the dedicated role of the Sequence Manager.
- **Compute the Sequence by AMAN:** This activity replaces the computation of the sequence by TCC described above by an automatic process performed by the AMAN.

6.2.4 Introduction to next step

The following section describes how UMLseCh tool is used to verify the new version of the design model. We will see that the Design change request proposed in the previous section is not sufficient to lead to a consistent status, with regards to all the Security requirements, especially

- **ATM_SEC_4**: The system shall be equipped with suitable security mechanisms to prevent from intentional loss of data.
- **ATM_SEC_5**: The system shall guarantee the integrity and confidentiality of data against illicit attempt to obtain access to such data.

and with regards with the new Role Based Access Control rules established in order to guaranty **Req1** and **Req2**.

6.3 Model consistency with UMLseCh

The previous section illustrates the process of deriving security requirements through a risk analysis to guarantee security objectives. Although backed up by standard methodologies which include provisions for ensuring that risks are covered by requirements, such an analysis remains non formal and lacks a measure of completeness. Specifically, the process may fail

- if some risks are not identified by the analyst,
- or if the set of requirements fails to cover the risks with respect to the security objectives,
- or if the requirements are not fully implemented in the system under consideration.

There are no generic ways to tackle those shortcomings, but in specific cases other methods may complement the risk analysis by providing a means to verify its completeness, as seen in figure 6.6. This point is illustrated in this section through the use of the role-based access control facilities of UMLseCh.

UMLsec includes an *« rbac »* stereotype of subsystems that contain an activity diagram. It formalizes role-based access control through the marking of some activities as protected, the assignment of roles to actors and of protected activities to roles. This is done with the tags {protected}, {role}, and {right} respectively. The semantics specify that every protected activity may only appear in the swim-lane of an actor with a role equipped with the right to perform the activity. For the sake of simplicity, we omit roles in this presentation and assign protected activities to actors directly. Such an additional indirection is unnecessary in the simple case presented here.

The simple scenario presented in this section consists of the following steps:

1. The introduction of the AMAN is implemented as a modification of the activity diagram presented before, as specified by the Design Change Request described in

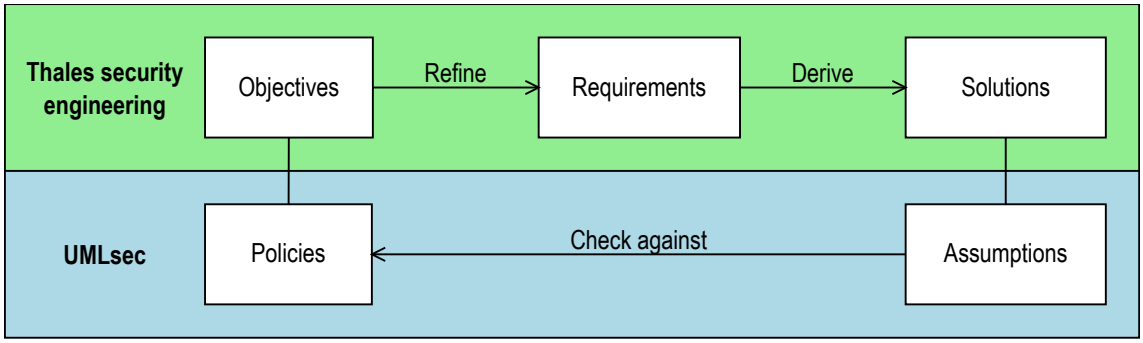


Figure 6.6: Integration between the Thales security engineering and UMLsec.

the previous section. From the point of view of UMLsec, this corresponds to a set of changes in the form of UMLseCh tags.

2. The new activity diagram does not respect the constraint associated with the stereotype *« rbac »*. A flaw is identified in the risk analysis.

3. Additional changes are applied, thus resulting in an *« rbac »*-compliant activity diagram describing the situation after the introduction of the AMAN.

The two requirements identified in the previous section are represented in UMLseCh by the addition of the following information:

- the right to perform the protected activity ‘Compute the sequence’ is transferred from actor TCC to actor AMAN,
- the right to perform the protected activity ‘Modify the sequence’ is transferred from actor TCC to actor SEQ MAN,

Figure 6.7 presents the activity diagram resulting from this rather naive modification. The UMLseCh stereotypes *« add »* and *« del »* self-describe the history of changes: the *« right »* tag is deleted as TCC loses their access rights, while the AMAN is added, with proper rights assigned to AMAN and SEQ MAN.

The mistake is obvious: while the risk analysis correctly inferred the need to strip the TCC from their access rights, the new requirements did not specify to update their activities accordingly. This is witnessed in UMLsec in that the constraint associated with the *« rbac »* stereotype is not satisfied any more. Both protected activities ‘Compute the sequence’ and ‘Modify the sequence’ appear in the swim-lane of actor TCC although the pairs $(\text{TCC}, \text{‘Compute the sequence’})$ and $(\text{TCC}, \text{‘Modify the sequence’})$ are no longer listed under {right}.

The additional changes necessary to recover the compliance to *« rbac »* are presented in figure 6.8. There, all activities by TCC are removed, only to be replaced with two new activities ‘Monitors traffic’ and ‘Agrees’ and transitions.

In this scenario, the *« rbac »* stereotype of UMLseCh provides a simple way to express a static constraint on a model. Such constraints provide consistency checks that can

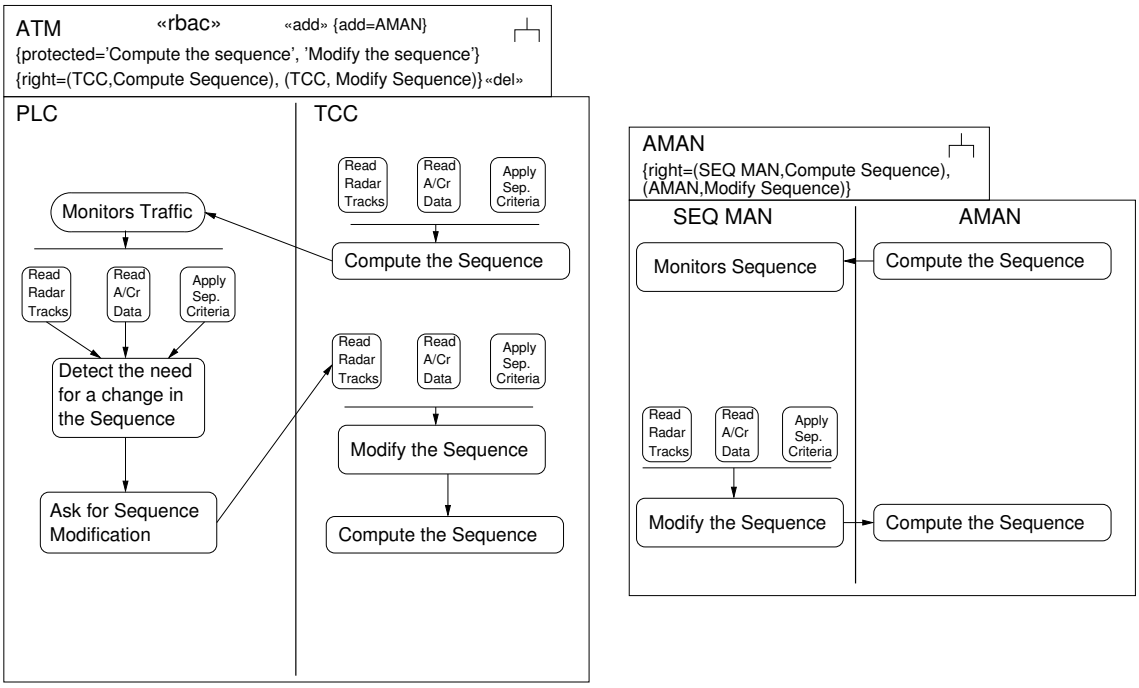


Figure 6.7: Activity diagram after applying the Design Change Request.

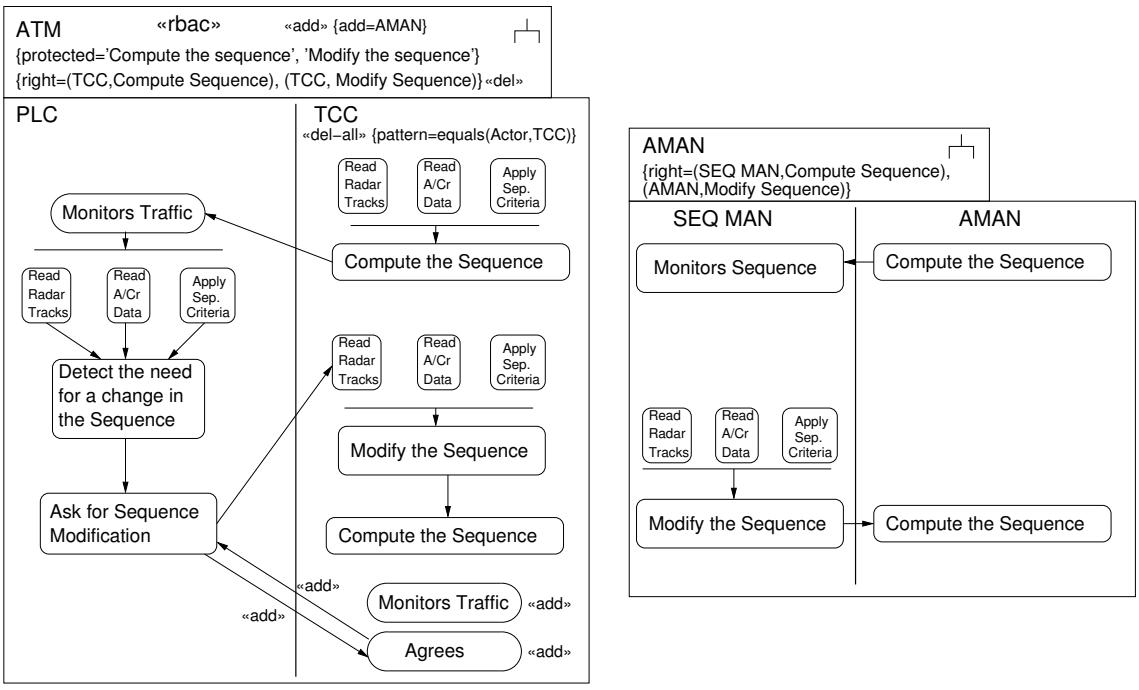


Figure 6.8: Activity diagram after applying more changes to recover compliance to *« rbac »*.

routinely and automatically be run on the model during its development. This is especially useful in the context described here, as the risk analyst and system designer are likely to be distinct persons who collaborate using distinct languages and tools.

7 Tool support for Model-Based Verification in Evolving systems

7.1 Introduction

Understanding the security goals provided by software making use of cryptography is one of the major challenges with security-critical systems. Any support to aid secure systems development is thus dearly needed. Towards this goal, the security extension UMLsec for the Unified Modeling Language allows us to include security requirements as stereotypes with logical constraints.
In this chapter we present automated tool-support for the analysis of UMLsec and UMLseCh models against security requirements by checking the constraints associated with the UMLsec stereotypes. Besides presenting a general, extensible framework for implementing verification routines for the constraints associated with security-critical UML stereotypes under evolution, we focus on a plugin that performs a static check for 'Secure Dependency' to verify security properties of UMLseCh models.
The plugin provides two functionalities, illustrating the general approach for dealing with security properties in evolving diagrams. First we have a 'UMLseCh Notation Analyser', that gets the evolution information from a UML model and keeps it in a data structure. This data structure contains all the important information of the delta of changes we need for further checks.
An additional functionality is the 'UMLseCh Static Check', which checks whether evolution preserves the 'Secure Dependency' property on the model.

7.2 UMLsec Framework

UMLsec is an extension of UML aimed to model security requirements and verifies the fulfillment of these requirements. It defines a notation as well as formal representations of the security concepts providing techniques to verify whether the security requirements are respected. The analysis of the security of a system consists in representing the execution of this system together with an attacker. This section introduces the UMLsec Tool Architecture.

7.2.1 Architecture

UML offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context. As the de-facto standard in industrial modeling, a large number of developers is trained in UML. Compared to previous industrial notations with a user community of comparable size, UML is relatively precisely defined. A number

of tools are being developed to assist the every-day work using UML. The UMLsec framework provides automatic verification plugins of UML models for critical requirements. In particular, it includes automated analysis of UMLsec models for the security requirements included as stereotypes. The input is a .zargo or .xmi file containing UML diagrams created with the UML tool ArgoUML. A particular focus of many of the verification plugins within this tool architecture is on security-critical systems.
The tool support for the automated verification of constraints associated with UMLsec stereotypes by given UML models is constantly under development. The UMLsec framework provides a programming environment for different UML verification tools, that encapsulates an MDR repository for loading, parsing, editing and storing UML models.

The Figure 7.1 depicts the UMLsec tool Architecture. The framework defines interfaces, which are be implemented by the tool. This provides an easy extension mechanism for including new plugins to the tool, as we will describe in more detail in Section 7.3.

7.2.2 MDR library

Adequate tool support for security validation is important. There are a number of tools that work with UML 1.4 models and store them in XMI format. XMI is a powerful standard that makes it possible to exchange models between different tools. UMLsec can import XMI data into the netbeans Meta data repository, as summarized in Figure 7.2.

The MDR is customized to the particular model type through a metamodel in XML format that is loaded into the repository. It will then create a storage customized for this type of model and generate JMI (Java Metadata Interface) definitions for the application (in our case the UMLsec framework) to access the model. The data extracted from the model will then be processed by the UMLsec framework. Figure 7.3 gives an overview of the complete UMLsec tool suite. As soon as the model is loaded, the different UMLsec checkers can be called to perform an analysis of the model.

7.3 Plugin development

To develop a plugin for the UMLsec framework two classes are needed. One is the Modul-class and the other is the Check-class. The Modul-class contains methods that are needed by the framework. In this class one defines a command and a parameter. The commands implement new functionality in the plugin-window. For each parameter defined here, the UMLsec framework asks for a value to be given as input to the verification algorithm implemented. The Modul-class implements 'IVikiToolBase' and 'IVikiToolConsole'. The Check-class contains the 'check-method', in which the algorithm is actually implemented. The check-class inherits from 'StaticCheckerBase'.

Finally, a modification takes place in the 'SystemVerificationLoader' class. The array 'IVikiToolBase []tools' needs a new entry with the new plugin und a new index must be created. This makes the new plugin accessible from the Plugin menu list of the tool.

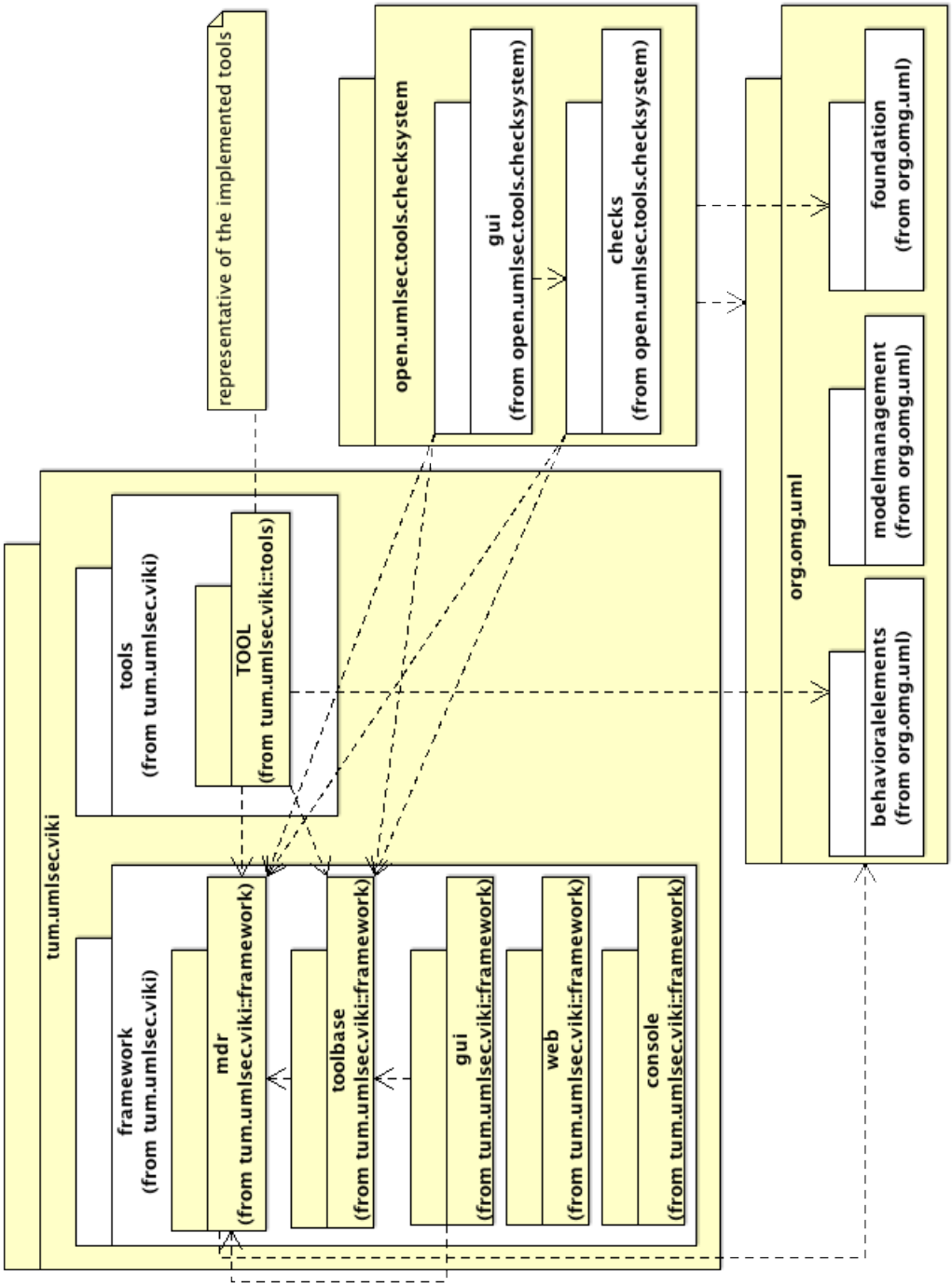


Figure 7.1: Framework architecture

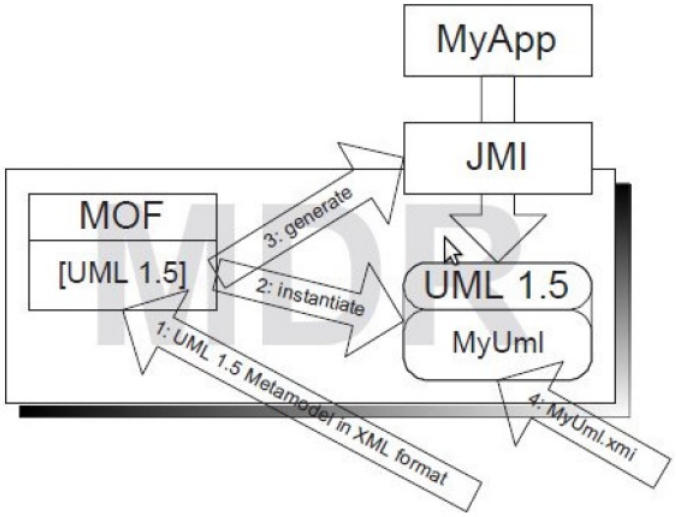


Figure 7.2: MDR library

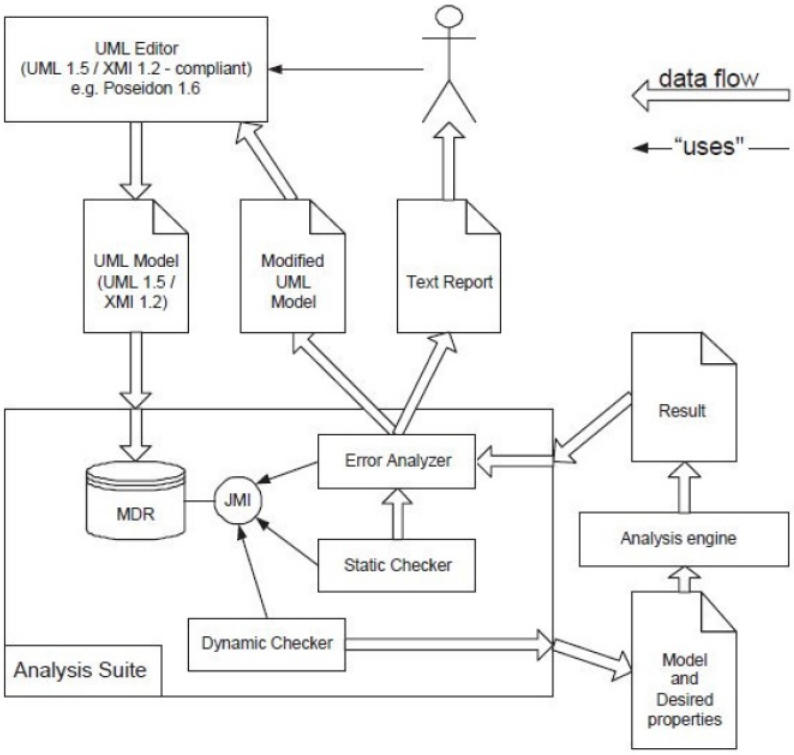


Figure 7.3: UMLsec tool suite

7.4 UMLseCh

The UMLseCh profile concerns all of UML. Figure 7.4 shows the UMLseCh profile. The tag ref is a DataTag and the tags substitute, add and delete are all ReferenceTags. UMLseCh models possible future changes. At the concrete level, i.e. in the tool, this value is either the model element itself if it can be represented with sequence of characters, or a namespace containing the model element.

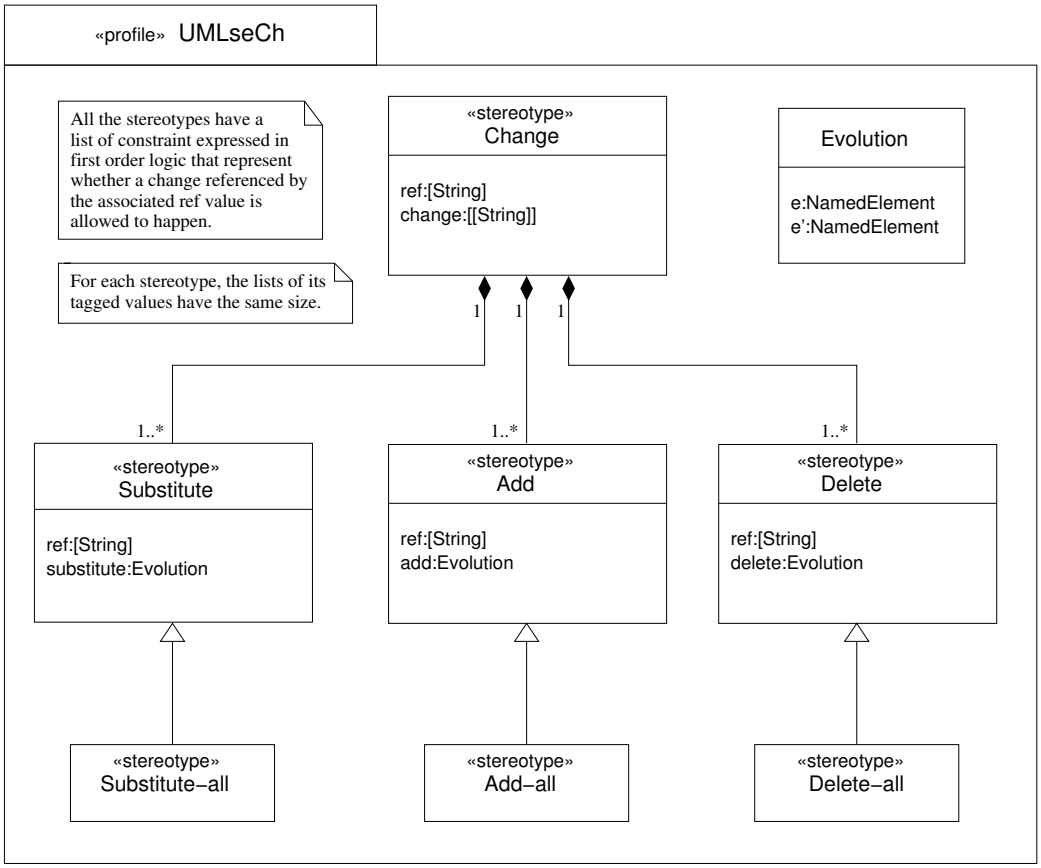


Figure 7.4: The UMLseCh profile

Although UMLseCh could be used alone as an evolution modeling language, it is initially intended to model the evolution in a security oriented context. It is thus an extension of UMLsec and requires the UMLsec profile as prerequisite profile.
For a more complete description of the UMLseCh Notation see Section 2.1.2.

7.5 UMLseCh Framework extension

Typically, it is the responsibility of each plugin of the UMLsec framework, to do the required calculations for the analysis based on the security-related stereotypes independently. For the UMLseCh annotated models however, we want to outsource the computation of the delta, since this is common to all of our UMLseCh plugins and analysis

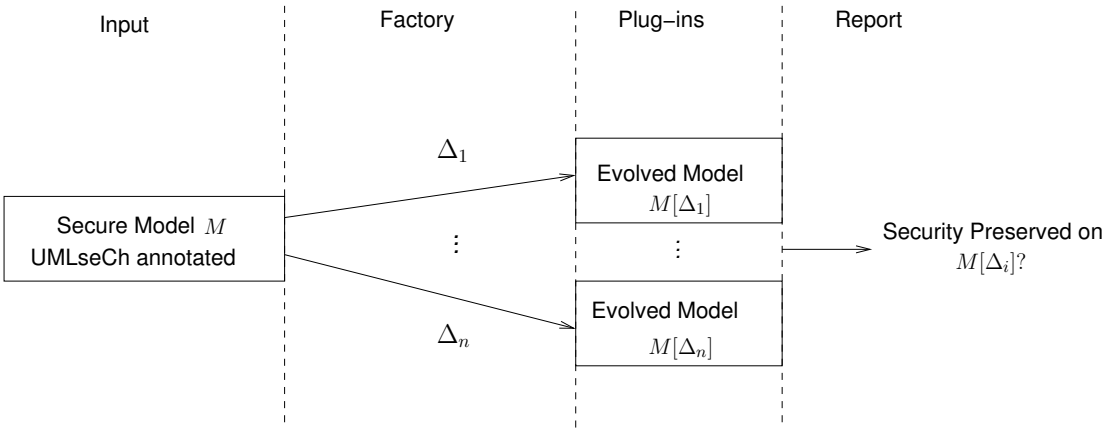


Figure 7.5: UMLseCh factory

algorithms. We introduce thus a new layer in the UMLsec framework, which can be used by any UMLseCh analysis plugin by means of a *factory*. This allows us by design to clone the delta information every time a plug-in requires it, without having to re-parse the UMLseCh diagrams. Additionally, much of the functionality to deal with model correctness (not security) and model/delta transformation in memory for the analysis algorithms can be also implemented within the factory. Figure 7.5 summarizes this idea: The generation of the delta is outsourced into the Factory layer, every plugin gets the delta(s) from the factory.

To get a better access to the UML-elements of a model and more functionalities, and to represent the atomic changes within the delta, we have defined a metamodel for the UML diagrams for which security diagrams are been developed. This is mainly because the functionality provided by the MDR container is rather rudimentary and leads to efficiency problems and unnecessary implementation efforts.

For example, a fragment the implemented metamodel we use for Class Diagrams is shown in Figure 7.6.

7.6 Tool Demo Story

The "UMLseCh Static Check" plugin [1] verifies if a model is still secure with respect to the *« secure dependency »* stereotype after evolution by using the verification techniques described in Chapter 3.

We briefly recall the definition of this stereotype:

« secure dependency » requires that for every dependency (*« send »* or *« call »*) between two classes in a class diagram such that in one of both classes a tag specifies a security requirement for a method or attribute (for example *{ high = {method()} }* resp. *{ secrecy = {method()} }*, *{ integrity = {method()} }*), then the other class has the same tag

[1] All the files used in this example can be found in: `http://inky.cs.tu-dortmund.de/main2/jj/umlsectool/manuals_new/UMLseCh_Static_Check_SecureDependency/index.htm` where also a screen-cast is available.

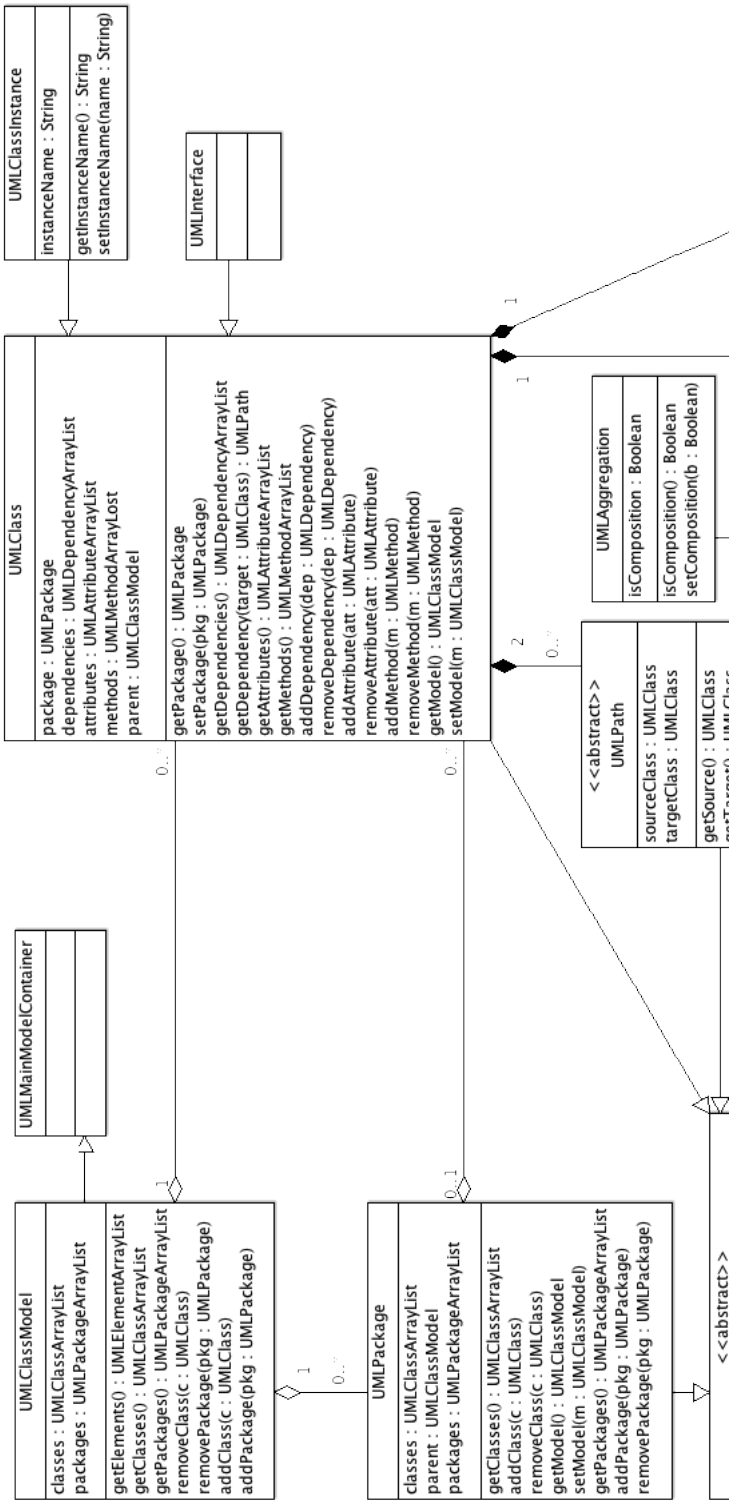


Figure 7.6: Fragment of the metamodel of a UML class diagram

for this method/attribute as well.

To perform the verification, the tool uses three vectors add, delete and substitute to store the delta information. During the check the vectors are verified for consistency, for example it is checked whether the substitute vector contains model elements which are already in the add or delete vector. These consistency checks are common to all UMLseCh plugins and are performed within the factory. After this, the actual check for secure dependency starts.
For this demonstration the class diagram as depicted in Figure 7.7 is constructed in ArgoUML.

7.6.1 Secure Dependency Model

The diagram in Figure 7.7 consists of two classes A and B with attached stereotypes. The stereotypes for class A are *« critical »* and *« substitute »* and for class B *« critical »*,*« add »*, *« substitute »* and *« delete »*. Between these classes exists a call-dependency. We inspect the values of the stereotypes in both classes.

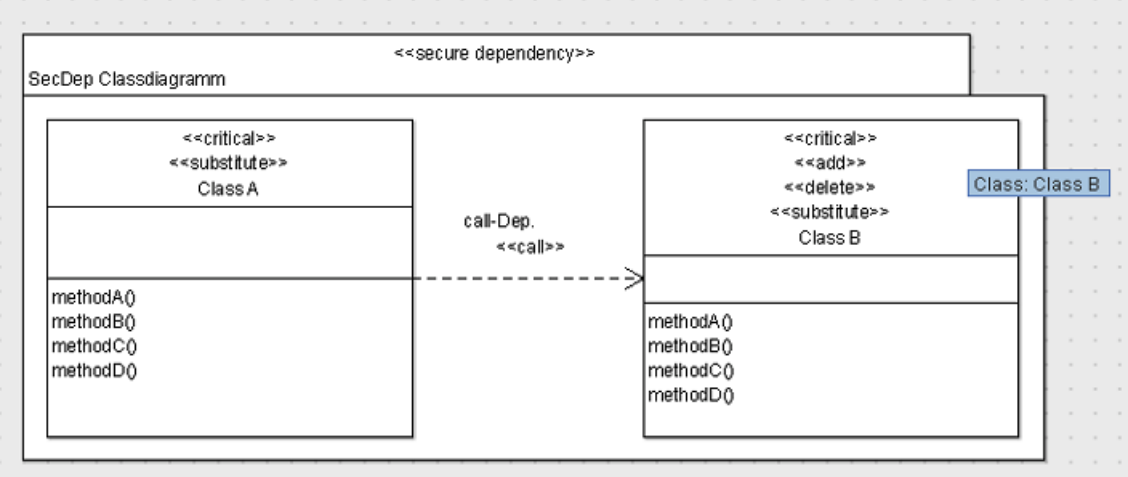


Figure 7.7: Argo diagram

Target: Class (Class A) TD

Kennzeichen	Wert
substitute	integrity=methodC
pattern	secrecy=methodC
secrecy	methodC
secrecy	methodD

Figure 7.8: Tags and stereotypes class A

Figures 7.8 and 7.9 show the tags and stereotypes of the classes A and B. For instance, the stereotype *« del »* modifies the UML-model and deletes the tagged value *{ secrecy = methodD }* from Class B. This operation is not performed on Class A, which maintains this tagged value.

Target: Class (Class B) TD

Kennzeichen	Wert
add	secrecy=methodA
add	secrecy=methodB
substitute	integrity=methodC
pattern	secrecy=methodC
delete	secrecy=methodD
secrecy	methodC
secrecy	methodD

Figure 7.9: Tags and stereotypes class B

7.6.2 Tool input: Example XMI

The important informations of the UML-model are stored in a XMI-format. The algorithm uses the information contained in special tags and stereotypes. Listing 7.1 shows a fragment of the XMI for the class A and the dependency with a Stereotype.

```
<UML:Class xmi.id = '-64--88-0-101-da8cac0:129602bab38:-8000:0000000000000DDB'
    name = 'Class A' visibility = 'public' isSpecification = 'false'
        isRoot = 'false'
    isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
.
.
.
    <UML:Namespace.ownedElement>
      <UML:Dependency xmi.id = '-64--88-0-101-da8cac0:129602bab38
          :-8000:0000000000000DDD'
        name = 'call-Dep.' isSpecification = 'false'>
        <UML:ModelElement.stereotype>
          <UML:Stereotype xmi.idref = '-64--88-0-101-da8cac0:129602
              bab38:-8000:0000000000000DDE'/>
        </UML:ModelElement.stereotype>
        <UML:Dependency.client>
          <UML:Class xmi.idref = '-64--88-0-101-da8cac0:129602bab38
              :-8000:0000000000000DDB'/>
        </UML:Dependency.client>
        <UML:Dependency.supplier>
          <UML:Class xmi.idref = '-64--88-0-101-da8cac0:129602bab38
              :-8000:0000000000000DDC'/>
        </UML:Dependency.supplier>
      </UML:Dependency>
    </UML:Namespace.ownedElement>
.
.
.
  </UML:Class>
```

Listing 7.1: Tags and Stereotype in the XMI-file

However, this is transparent to the user, who can directly load the .zargo file of the example above to the tool. Once loaded, the user can choose to perform the UMLseCh Secure Dependency checks by selecting the plug-in from the plug-ins menu list, as seen in Figure 7.10.

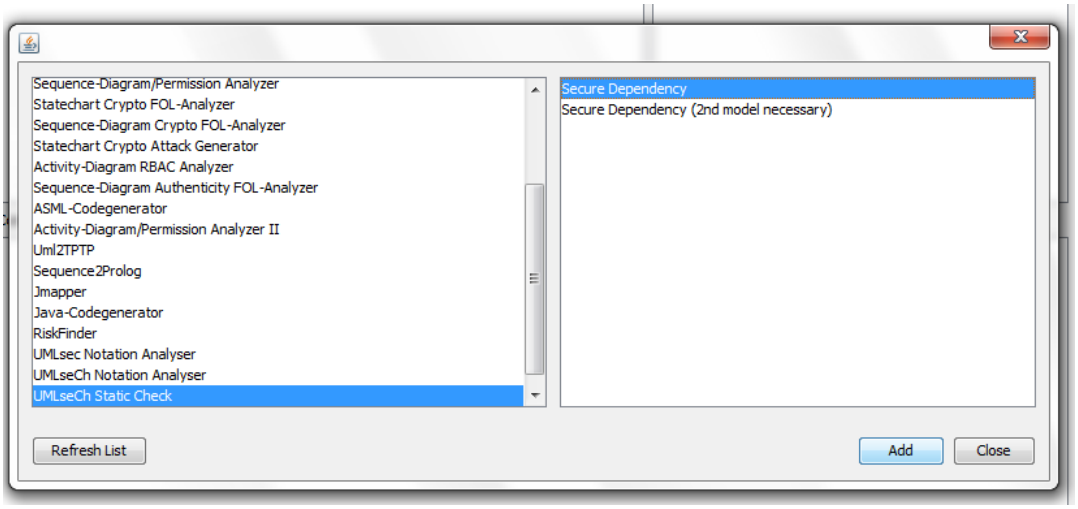


Figure 7.10: Selecting the UMLseCh plug-in

7.6.3 Results of Analysis

The algorithm checks the delete-, add- and the substitute-vector for classes and the belonging dependency. Figure 7.11 shows the logging output of the plugin, when checking for the deleted model elements. Since this operation is not symmetric for the classes under consideration, *« secure dependency »* is violated.

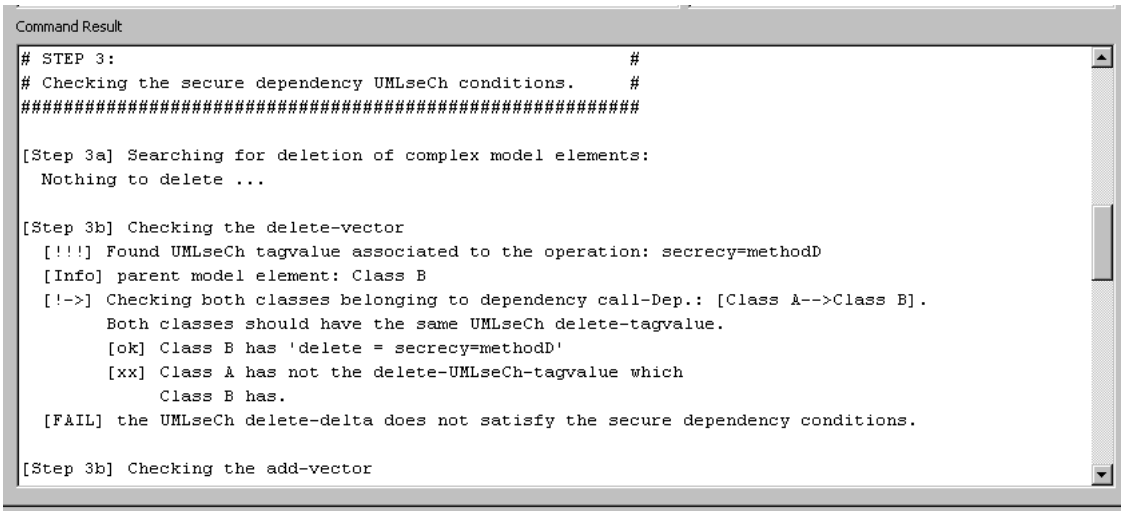


Figure 7.11: The property is violated

Figure 7.12 shows the result summary of the plugin. Since the check of the deleted items failed, the evolution is considered as not security-preserving.

Command Result

```
  [!->] Checking both classes belonging to dependency call-Dep.: [Class A-->Class B].
        Both classes should have the same UMLseCh add-tagvalue.
        [ok] Class A has 'substitute = integrity=methodC' and 'pattern = secrecy=methodC'
        [ok] Class B has 'substitute = integrity=methodC' and 'pattern = secrecy=methodC'
  [SUCC] the UMLseCh substitution-delta satisfies the secure dependency conditions

[Step 3d] Analyse all vector-check results:
  [FAIL] The secure dependency UMLseCh conditions are violated.

############################################################
# STEP 4:                                                  #
# Conclusion:                                              #
############################################################

[FAIL] 'Secure Dependency' conditions are
       violated after UMLseCh evolution. See above
       for more informations.
```

Figure 7.12: Result summary

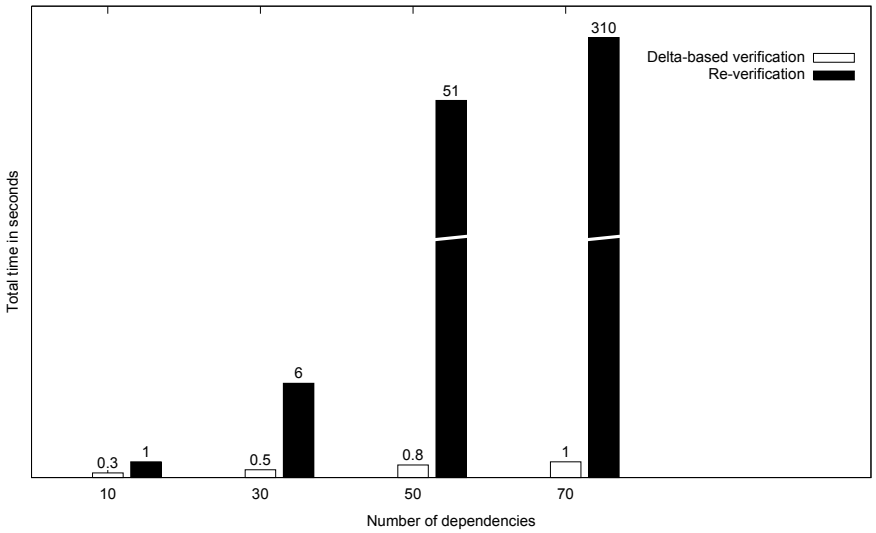


Figure 7.13: Running time comparison of the verification

Efficiency gain

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 number of dependencies. In Fig 7.13 we present a comparison between the running time of the verification by running trivial re-verification and by running the UMLseCh plugin[2] on a class diagram where only 10% of the model elements were modified.

[2] On a 2.26 GhZ dual core processor

8 Conclusions

In this work we introduced a formal foundation for the UMLseCh notation described in D4.1, and discussed how the modifications introduced by this notation can be analyzed for several security properties. We also consider the problem of model *consistency* after evolution. We considered selected classes of model evolutions such as addition, deletion, and substitution of model elements based on structural and behavioral UMLsec diagrams. Assuming that the starting UMLsec diagrams are secure, which one can verify using the UMLsec tool framework, our goal is to reuse these existing verification results to minimize the effort for the security verification of the evolved UMLsec diagrams. We achieved this goal by 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. These conditions are used currently as a basis to extend the existing UMLsec tool framework by the ability to support secure model evolution.
This extended tool should help the development of evolving systems by warning of possible security violating modifications of secure models. We also show that the implementation of the techniques described in this deliverable lead to a significant efficiency gain compared to the trivial re-verification of the entire model.

Bibliography

[1] http://www.omg.org/technology/documents/profile_catalog.htm.

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

[3] Object management group qvt specification, 2008. `http://www.omg.org/spec/QVT/1.0/PDF/`.

[4] A. Bauer and J. Jürjens. Runtime verification of cryptographic protocols. *Computers and Security, vol. 29, 2010, pp. 315–330*, 2010.

[5] A. Bauer, J. Jürjens, and Yijun Yu. Run-time security traceability for evolving systems. *The Computer Journal, Oxford Univ. Press*, 2011.

[6] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 256, Washington, DC, USA, 2003. IEEE Computer Society.

[7] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September/October 2005.

[8] N. Chapin, J. E. Hale, J. Fernandez-Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[9] D. Ghindici, G. Grimaud, and I. Simplot-Ryl. Embedding verifiable information flow analysis. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*, PST '06, pages 39:1–39:9, New York, NY, USA, 2006. ACM.

[10] D. Ghindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 32–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] D. Ghindici, I. Simplot-Ryl, and J.-M. Talbot. A sound analysis for secure information flow using abstract memory graphs. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 5961 of *Lecture Notes in Computer Science*, pages 355–370. Springer Berlin / Heidelberg, 2010.

[12] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution: Research articles. *J. Softw. Maint. Evol.*, 18(3):207–236, 2006.

[13] J. Gray. The transaction concept: Virtues and limitations, 1981.

[14] The Object Management Group. http://www.omg.org/.

[15] S.H. Houmb, S. Islam, E. Knauss, J. Jürjens, and K. Schneider. Eliciting security requirements and tracing them to design: An integration of common criteria, heuristics, and UMLsec. *Requirements Engineering Journal (REJ), vol. 15(1), 2010, pp. 63-93. Special Issue on Security Requirements Engineering*, 2010.

[16] S. Islam, H. Mouratidis, and J. Jürjens. A framework to support alignment of secure software engineering with legal regulations. *Journal of Software and Systems Modeling (SoSyM), 2011, Special Issue on Non-Functional Properties in Domain-Specific Modeling*, 2011.

[17] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM.

[18] J. Jürjens. Security in UML. In *4th International School on Foundations of Security Analysis and Design (FOSAD 2004)*, 2004.

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

[20] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE*. IEEE Computer Society, 2005.

[21] J. Jürjens. A domain-specific language for cryptographic protocols based on streams. *J. Log. Algebr. Program.*, 78(2):54–73, 2009.

[22] J. Jürjens. Model-based security engineering with UML (keynote). *26th IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010)*, Madrid, 21-25 Sep. 2010.

[23] J. Jürjens and B. Nuseibeh. Software engineering for secure systems. *33rd International Conference on Software Engineering (ICSE 2011), Technical Briefing. ACM 2011*, 2011.

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

[25] J. Jürjens and H. Schmidt. Model-based secure software development (keynote). *8th International Workshop on Security in Information Systems (WOSIS 2011), 13th International Conference on Enterprise Information Systems (ICEIS 2011)*, Beijing, June 2011.

[26] A. Kalnins, J. Barzdins, and E. Celms. Basics of model transformation language MOLA.

[27] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In *in: Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004*, pages 14–28, 2004.

[28] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, 1995.

[29] R. Lämmel. Evolution of rule-based programs, 2004.

[30] The ATLAS Transformation Language. http://www.eclipse.org/ATL/.

[31] M. Lawley and J. Steel. Practical declarative model transformation with tefkat.

[32] M. M. Lehman. The programming process. In *IBM Res. Rep. RC 2722*, IBM Res. Centre, Yorktown Heights, September 1969.

[33] M. M. Lehman. Programs, cities, students, limits to growth? In *Inaugural Lecture, in Imperial College of Science and Technology Inaugural Lecture Series*, volume 9, pages 211–229, 1974.

[34] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[35] M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 350–357, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[36] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 70–74, New York, NY, USA, 2001. ACM.

[37] M M Lehman, J F Ramil, and G Kahen. Evolution as a noun and evolution as a verb, 2000.

[38] 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: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[39] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[40] L. Montrieux, J. Jürjens, Yijun Yu C.B. Haley, P.-Y. Schobbens, and H. Toussaint. Tool support for code generation from a UMLsec property. *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), pp. 357-358*, 2010.

[41] H. Mouratidis and J. Jürjens. From goal-driven security requirements engineering to secure design. *International Journal of Intelligent Systems (IJIS) (Wiley Interscience), Volume 25, Issue 8, pages 813-840. (Special issue on "Goal-Driven Requirements Engineering".)*, August 2010.

[42] OMG. Mof 2.0 query/views/transformations rfp, 2002.

[43] OMG. *MOF QVT Transformation Specification*. Object Modeling Group, 2005.

[44] OMG. *Unified Modeling Language: Superstructure, version 2.2*. Object Modeling Group, February 2009.

[45] D. L. Parnas. On the criteria to be used in decomposing systems into modules. pages 139–150, 1979.

[46] D. Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[47] A. Pironti and J. Jürjens. Formally-based black-box monitoring of security protocols. *International Symposium on Engineering Secure Software and Systems (ESSOS 2010), Lecture Notes in Computer Science 5965, Springer, pp. 79-95*, 2010.

[48] D. Rowe, J. Leaney, and D. Lowe. Defining systems evolvability - a taxonomy of change. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:0045, 1998.

[49] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[50] Secure Change Consortium. Selected Change Requirements and Security Properties. `https://trinity.disi.unitn.it/bscw/bscw.cgi/d255281/Case_Study_Requirements_v3.0.pdf`, 2010.

[51] SecureChange. Deliverable 4.1, 2010. `http://securechange.eu/sites/default/files/deliverables/D4.1_Security_Modelling_Notation_for_Evolving_Systems.pdf`.

[52] V. Stolz. An integrated multi-view model evolution framework.

[53] Yu Sun. Supporting model evolution through demonstration-based model transformation. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 779–780, New York, NY, USA, 2009. ACM.

[54] E. D. Willink. A concrete UML-based graphical transformation syntax: The UML to RDBMS example in UMLX, 2003.

[55] E. D. Willink. UMLX: A graphical transformation language for MDA, 2003.

A Evaluation Summary

In this appendix we will give some initial arguments to explain to which degree WP4 complies with the applicable General Scientific Criteria (as defined in D1.2) up to Year 2. More comprehensive success criteria will be given in Year 3 and the Industrial Criteria will be then also validated by WP1.

Internal Criteria: Modeling Languages

- Consistency rules of constructs:
  Precise rules regulates syntax of artifacts so that not all possible syntactic combinations are possible
  Arbitrary syntactic combinations are not allowed, since there is a precise syntax/semantics defined in Chapter 2.
- Computer Aided support of consistency
  The plugins for UMLseCh under implementation partially support consistency (besides their main goal: security). The rules for consistency of UML Models under evolution are all implementable, as defined in Chapter 2. However, this is not the main goal of the plugins and they will not check all possible consistency problems, due to their prototypical nature.
- Formal or operational consistency of constructs
  There is a formal or operational semantics behind the constructs that determines which behaviors are formally acceptable/practically executable
  This is also defined in this Deliverable for the UMLseCh notation in Chapter 2
- Formal characterization of constructs
  There is a clear characterization of the constructs so that is possible to characterize which behavior is not expressible in the syntax
  This follows from the precise semantics definition.
- Local Usability of construct
  User does not need to understand other artifacts than the one he needs to model the specific aspects
  The user only needs to understand the UMLseCh notation to model evolutions.

Internal Criteria: Algorithms

- Precise computation:
  The mechanisms of computations are well defined and the result of the computation are clearly defined and interpretable
  The mechanisms of computations are algorithms, and are currently being implemented. The result of the computation is fundamentally a boolean, reporting on the

security of the evolved model(s). Further reporting in the output refers to specific problems encountered and is clearly defined.

- Effective computation
  There is a clear cut justification on how this computation can be effective on current architectures or the limitations that needs to be overcome
  The computation is decidable and we give arguments on its soundness on Chapter 3. However, we do not aim at completeness.

- Computer-Aided Computation
  There is a fully automatic or interactive implementation of the algorithms
  There is a fully automatic implementation of the algorithms for *« secure dependency »* and other plugins are being currently developed.

- Formal or operational evidence of efficiency
  There is a formal or operational evidence of the efficiency of these algorithms w.r.t. naive algorithm. For instance, it is not possible to just argue informally that 'Testing the difference' cost less than 'Testing the whole' there should be numbers detailing this result at least for the practical case
  There is an informal argument on the efficiency for local enough properties in the Introduction of the deliverable and there is evidence based on measurements over large UML models of the algorithm based on the difference versus the trivial reverification on Chapter 7.

Additionally, WP4 has suggested the following Criteria specific to the Algorithms.

Internal Criteria: WP4 Specific Although the soundness of the algorithms presented is off course an important feature, we would also like to measure how "good" their coverage is with respect to the possible *deltas* expressible by UMLseCh. We begin by noting that there are three main kinds of objects according to the evolution they represent:

- Deletions
- Additions
- Substitutions

So a first measurement criterion is: *Does the algorithm allow evolutions of all kinds?*.

By this we mean that the algorithm will consider valid at least some delta including additions, substitutions and deletions with respect to the given security requirement (as opposed to "failing per default", which is trivially sound).

Additionally, every object in the delta has a UML type, for example "method" (in an evolving class diagram) or "transition" (in an evolving state-chart diagram). Since none of the security requirements considered in this work pose any specific restriction on the UML type of the model elements contained in a diagram (all types are allowed), then a second criterion is: *Does the algorithm allow evolutions of all UML types?* (where by UML types

we mean the model element types definable for the specific diagram under consideration).

So if an algorithm meets these two conditions, we know it will allow at least one evolution involving additions, substitutions and deletions of model-elements of all types for a given diagram. The next natural question is: *How many evolutions of a given kind and UML type does the algorithm allow?*.

We can answer positively to these questions for all the algorithms presented by showing that:

- The algorithms deal with additions, deletions and substitutions.
- The algorithms allow evolutions of all UML types for the diagram they consider.
- The algorithms allow an arbitrary number of evolutions of a given kind and UML type.

Note that this does not mean that the algorithms allow *all* possible evolutions, they simply allow an arbitrary number of evolutions of a given kind and type, if some conditions are satisfied.

Another interesting question is: *Given two models M and M' satisfying a security property P, is there at least one evolution Δ such that $M' = M[\Delta]$ and (M, Δ) is accepted by the decision algorithm?*.

If we can show that given any model element M satisfying P we can find an evolution Δ_1 consisting only of deletions such that $M[\Delta_1] = \emptyset$, and that starting from the empty model $\emptyset$ we can reach any M' satisfying P by Δ_2 then we can answer affirmatively to the previous question if both Δ_1 and Δ_2 are allowed by the algorithm (since then $\Delta = (\Delta_1, \Delta_2)$ will be allowed).

We claim that this property holds for all of the algorithms presented with the exception of the decision procedure dealing with *secrecy*, since we can at least reach an empty model and reconstruct any model respecting the decision procedure defined in Chapter 3.

B The Secure-Dependency Plugin

In this appendix we show in more detail the actual status of the implementation effort for the SecureDependency plug-in. We will continue this task (T4.3) until M30.

This documentation deals with a program which can examine weather a model is secure or not. The useability will be improved and restored and a part of UMLseCh will be implemented as described in Section 3. The first part shows how to extend the program with plugins. The second part discusses three new plugins. The first plugin is able to dump all model elements. The second one is specialized on security relevant Information which is stores in a data structure that is also described in the paper mentioned above. The third plugin implements a static check for "secure Dependency" and defines it. After this we will describe the UMLTypeScanner, which has been developed to ease the use of loaded models. Last but not least changes to the GUI will be explained broadly.

B.1 Generation of a plugin template

This chapter is about implementation of a template for a new plugin for the program "'UMLsec Verification Plugin"'. To avoid any confusion according the naming, we will use the same terms as in the source code. For this purpose the image 2.3 will be described.

It has been clicked on "'Tools"' on the menu bar On the GUI of UMLsec System Verification Plugin and "'Add"'. As a result a nameless selection window has been opened which allows us to integrate "'Commands"' for verification. On the left of this selection window actual tools are listed and on the right the corresponding commands for the verification of UMLsec diagrams.

The Tools are implemented in Java. This Jave files are stored in different folders for the sake of clarity. For structuring each folder contains a folder "'checks"', which includes the command java files with the implemented check methods. It is also possible to abstract several verification mechanisms in one command-java-file and change the call to the check methods in the tool-java-file.

Now we will introduce a template for a plugin which is a command that poses a minimal framework for such a Tool. It consists of a java-command file that contains an empty verification mechanism which can be later specified by the developer. For the proper integration of such a tool several steps have to be considered. The Tool has to know its commands and has to be known by the UMLsec verification plugin. In the first step we will create a folder structure which is the foundation for our tool and explain it. In the second step we will go into detail on the content of the tool java file. A template code is included and tips for choosing your domain name will be given. Step 3 deals with the template code in the command java file. Last but not least, in step 4 we have to create the tool as objects, insert them into arrays and allocate necessary ids.

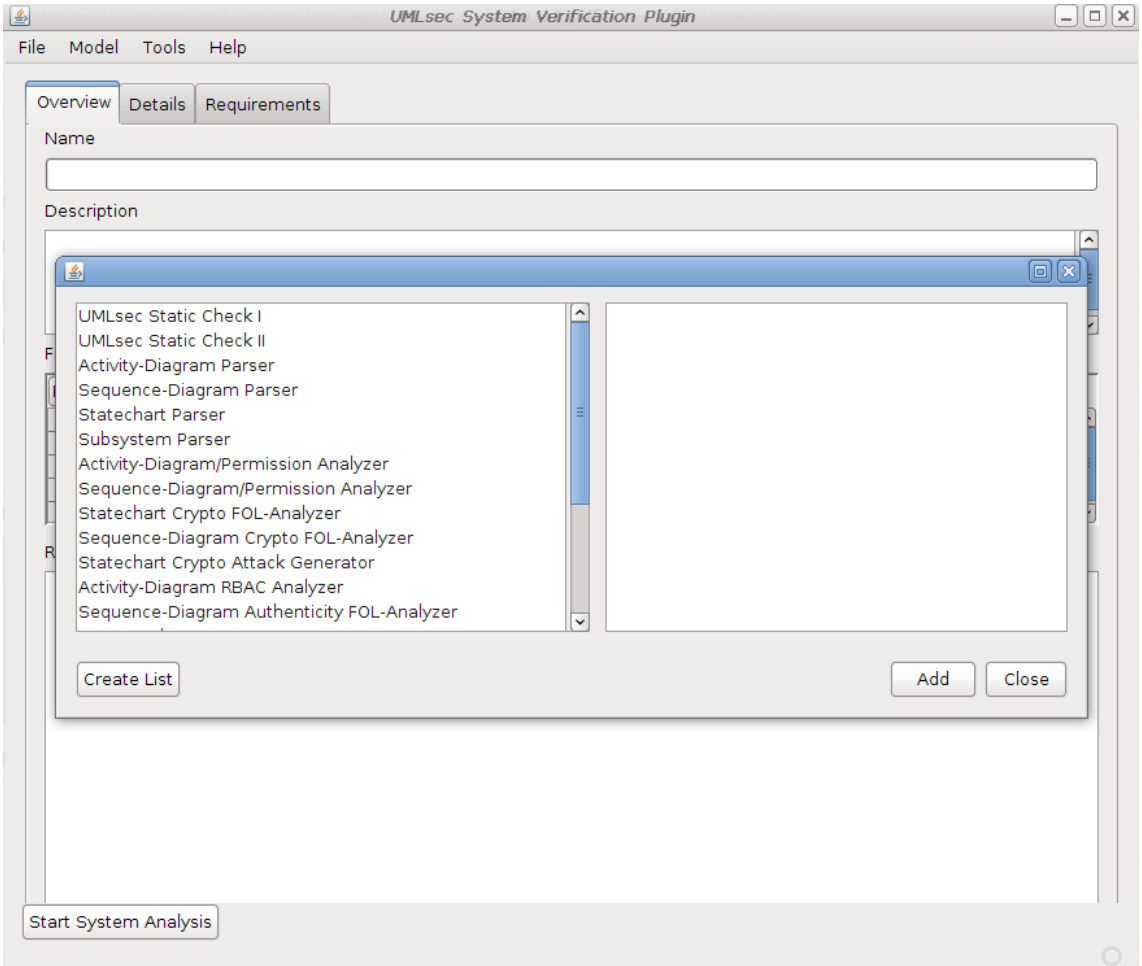


Figure B.1: UMLsec Tool

B.1.1 Step 1: Create tool folder

1. Change to folder `src\uml\umlsec\viki\tools\`
2. Create folder with an expressive name for the tool. In the following we will call it `[ToolFolder]`.
3. Create a subfolder in this folder and name it "‘checks"’.

B.1.2 Step 2: Create tool java file

1. Create file in the folder `[ToolFolder]` and name it meaningfully. The file name without "‘.java"’ is called `[ToolName]`.
2. The content of this file is described in listing B.1.
3. All variables and contents with the infix “Exemple“ can be chosen arbitrary but consistently and according to conventions.

```java
// define the package
package tum.umlsec.viki.tools.[ToolFolder];

// important imports
import java.util.Iterator;
import java.util.Vector;
import tum.umlsec.viki.framework.ILogOutput;
import tum.umlsec.viki.framework.ITextOutput;
import tum.umlsec.viki.framework.mdr.IMdrContainer;
import tum.umlsec.viki.framework.toolbase.CommandDescriptor;
import tum.umlsec.viki.framework.toolbase.IVikiToolBase;
import tum.umlsec.viki.framework.toolbase.IVikiToolConsole;
import tum.umlsec.viki.framework.toolbase.IVikiToolGui;
// is needed for web-support
import tum.umlsec.viki.framework.toolbase.IVikiToolWeb;

// import your check methods classes
import tum.umlsec.viki.tools.[ToolFolder].checks.*;

// class to integrate the tool into the framework
public class ToolEvolutionCheck implements IVikiToolBase, IVikiToolConsole {

        // parameters to identify check methodes
        public static final int CID_EXAMPLECHECKNAME = 1;
        // ...

        // depth of the check methods ids (not sure)
        public static final int CPID_DEPTH = 1;
        Vector commands = new Vector();
        IMdrContainer mdrContainer;
        boolean returnValue=false;
        Vector parametersEmpty = new Vector();

        // method to return the instance that implements the interface for the
            console
        public IVikiToolConsole getConsole() { return this; }

        // method to return the instance that implements the interface for the
            GUI
        public IVikiToolGui getGui() { return null; }

        // method to return the instance that implements the interface for the
            web
        public IVikiToolWeb getWeb() { return null; }

        // method to return the instance that implements the basic functions of
            the tool
        public IVikiToolBase getBase() {return this;}

        // method to return the name of the tool
        public String getToolName() {return "Example Tool Name";}

        // method to return the desription of the tool
        public String getToolDescription() {return "Example Description";}

        // method to initialize the tool
        public void initialiseBase(IMdrContainer _mdrContainer){
                mdrContainer = _mdrContainer;
                commands.add(cmdALL);
        }

        // method to initialize the console
        public void initialiseConsole(){}

        // method to return the possible commands of the console
```

```
62          public Iterator getConsoleCommands() {
63                  return commands.iterator();
64          }
65
66          // method to execute the chosen commands of the console
67          public void executeConsoleCommand(CommandDescriptor _command, Iterator
                _parameters, ITextOutput _mainOutput, ILogOutput _auxOutput) {
68                  // switch to the ID of the method which is selected
69                  switch(_command.getId()) {
70
71                  // static variable with check method ID
72                  case CID_EXAMPLECHECKNAME:
73
74                          // calls the check method with necessary parameters and
                                returns the result of the check
75                          returnValue = new [CommandName]().check(mdrContainer,
                                _parameters, _mainOutput);
76                          break;
77
78                  default:
79                          // default return value
80                          returnValue = false;
81                  }
82          }
83
84          // setting properties of the check methods
85          CommandDescriptor cmdALL = CommandDescriptor.CommandDescriptorConsole(
                CID_EXAMPLECHECKNAME, "Example Check Name", "Example Check
                Description", true, parametersEmpty);
86
87  }
```

Listing B.1: Quellcode der Tool-Java-Datei

B.1.3 Step 3: Create a Command-Java-File

1. Create a Java file in the previous created folder `[ToolFolder]\checks\` Ordner and choose a meaningful name. The file without ".java" is called `[CommandName]`.

2. The template code from listing B.2 belongs to this class

```
1  // define the package
2  package tum.umlsec.viki.tools.[ToolFolder].checks;
3
4  // imports
5  import java.util.*;
6  import org.omg.uml.foundation.core.*;
7  import tum.umlsec.viki.framework.ITextOutput;
8  import tum.umlsec.viki.framework.mdr.IMdrContainer;
9  import tum.umlsec.viki.tools.checkstatic.StaticCheckerBase;
10
11 // [CheckingClass] extends the StaticCheckerBase - forces to implement the "check
       "-method
12 public class [CommandName] extends StaticCheckerBase {
13          // output textbox stream
14          ITextOutput textOutput;
15
16          // check method
17          public boolean check(IMdrContainer _mdrContainer, Iterator _parameters,
                ITextOutput _textOutput) {
```

```
18                  // check something
19                  return true; // or return false
20          }
21  }
```

Listing B.2: Command-Java-file source code

B.1.4 Step 4: Integrate tool into main program

1. file: `tum.umlsec.viki.framework.Loader.java`
   - After importing all other tools we import:
     import tum.umlsec.viki.tools.[ToolFolder].[ToolName];
   - We have to add an entry into IVikiToolBase:
     new [ToolName]()
2. file: `tum.umlsec.viki.framework.web.LoaderWeb.java`
   - After importing all other tools we import:
     import tum.umlsec.viki.tools.[ToolFolder].[ToolName];
   - insert code from listing B.3 into method `InitialiseWebFramework()`.

```
1 // increment index
2 tools = new IVikiToolBase[21];
3 // Increment the indexbounds
4 toolsWeb = new InstalledWebToolDescriptor[21];
5 // Insert tool into the array
6 tools[Indexgrenze-1] = new [ToolName]();
```

Listing B.3: Code-extension - method `InitialiseWebFramework()`

3. file: `open.umlsec.tools.checksystem.gui.SystemVerificationLoader.java`
   - Once again after importing all other tools we import:
     import tum.umlsec.viki.tools.[ToolFolder].[ToolName];
   - method `SystemVerificationLoader()` has to be modified:
     - `VikiToolBase`-Array: `new [ToolName]()`
     - variable `TOOL_IDX_EXAMPLETOOLNAME` has to be incremented
     - also increment Variablee `TOOL_IDX_MAX`
4. file: `open.umlsec.tools.checksystem.checks.ToolSelector.java`
   - One last time we have to import the tool after importing all other tools:
     import tum.umlsec.viki.tools.[ToolFolder].[ToolName];
   - Now we create signature variable:
     private ToolSignature [ToolSignatureName] =
     new ToolSignature(SystemVerificationLoader.
     TOOL_IDX_EXAMPLETOOLNAME);

- insert the following code into the method **public void** clearAll (){...}

```
[ToolSignatureName].resetQueryResult();
```

B.2 Tool UMLsec Notation Analyser: Command DumpAllModelElements

As we already discussed a template for a plugin, we will now only explain functionality of the check method of command `DumpAllModelElements` which is in `Umlsec Notation Analyser`. For this we use listing B.4.

```
1         // DumpAllModelElements check-method
2     public boolean check(IMdrContainer _mdrContainer, Iterator _parameters,
          ITextOutput _textOutput) {
3         //GUI-Outputcontainer for check-messages
4         textOutput = _textOutput;
5
6         CorePackage corePackage ;
7         corePackage = _mdrContainer.getUmlPackage().getCore();
8
9         UmlTypeScanner umlident = new UmlTypeScanner();
10
11        //-----------------------------------------------------
12
13        //None of the diagramtypes could be identified
14        if (!umlident.identifiable(corePackage)) return false;
15
16        //-----------------------------------------------------
17
18        textOutput.writeLn("Modeltyp: " + umlident.diagramType(corePackage));
19        textOutput.writeLn("");
20
21        textOutput.writeLn("====[ Dumping all model elements ]====");
22
23        String act_elem = "";
24        for(Iterator it = corePackage.getModelElement().refAllOfType().iterator()
              ;it.hasNext();){
25          ModelElement me = (ModelElement) it.next();
26          String name = umlident.modelElementType(me.getClass().getName());
27          if(!name.equals(act_elem)){
28            act_elem = name;
29            textOutput.writeLn("");
30            textOutput.writeLn("--- All " + act_elem + "s ---");
31          }
32          if(name.equals("TaggedValue")){
33            TaggedValue tagValue = (TaggedValue) me;
34            textOutput.writeLn("- " + umlident.modelElementType(tagValue.
                  getModelElement().getClass().getName()) + ":" + tagValue.
                  getModelElement().getName() + " | " + tagValue.getType().getName
                  () + " | Value: " + tagValue.getDataValue().iterator().next().
                  toString());
35          }else{
36            if(me.getName() == null || me.getName().equals("")){
37              textOutput.writeLn("- " + act_elem + " with no name");
38            }else{
39              textOutput.writeLn("- " + me.getName());
40            }
41          }
42        }
43
44        return true;
45    }
```

Listing B.4: Check-method `DumpAllModelElements`

This is a method which returns like all check methods boolean as return type. This symbolizes the success state of the check method. If the value is true the check was successful otherwise it is false. The method is called with some parameters which has been passed to the tool previously. This parameters are `IMdrContainer` which contains models, iterator parameters and a output text box `ITextOutput`. The output text box is located on the GUI and shows the statues of the check method. In line 4 the reference of the passed text box is assigned to the variable `textOutput`. In line 6 a variable `corePackage` of type `CorePackage` is created. After that the core of `UmlPackage` of `IMdrContainer` is assigned to this variable. Further on an `UmlTypeScanner umlident` is created which identifies diagram type and model element classes. The class `UmlTypeScanner` will be discussed on page 30 in chapter 7. Line 14 examines if a diagram is clearly identifiable. Is this not the case the check will be terminated without success. At this point the first output begins. The diagram type and a new line will be displayed. An output follows which indicates that henceforth model elements will be dumped (line 21). In line 23 an empty string `act_element` is created. We will explain the use of this variable in a moment. Primary the for-loop is entered which displays one by one all elements of the model group. In the loop header an iterator is initialised with the iterator provided by the core. All elements are grouped and stored in this iterator. This is important for the algorithm so that all groups are listed just once. The condition `it.hasNext()` checks if the iterator provides further elements. A counter is not needed therefore this part remains empty. In line 25 a new element is taken from the iterator. The local string name gets as value the type of the element. The if statement from line 27 till line 31 is important for the title of following elements. The condition proves if `string name`, which contains the type of element is not equal with `string act_elm` that is initialised in line 23. Are both strings equal the part which prints a new headline will be skipped as the element belongs to the current group. Otherwise the `string act_elm` gets the value of `string name`. An empty string will be printed followed by new headline of the current element group. Line 32 proves if current element group is “Taggedvalue“. In that case value and element with type will be printed. From line 35 on the else part will be discussed. In line 36 is proved whether an element is `null` or ““. If ““ is `true` the message “an element without name has been found“ will be printed otherwise just the name of the element. At least the method returns `true`.

B.3 Datastructure StructUMLseChDelta

In chapter 3 is a concept represented which describes
“Secure Dependency“- check for delta. For this purpose we need an additional class that provides data structure for three vectors (“add“,“del“,“subs“). You can find it in listing 4.1.

```
1 //Containerclass for UMLseCh-Delta
2 package tum.umlsec.viki.tools.umlseChNotationAnalyser.checks;
3
4 import java.util.Iterator;
5 import org.omg.uml.foundation.core.*;
6 import tum.umlsec.viki.UmlTypeScanner;
7
```

```
8
9  public class StructUMLseChDelta {
10     private ModelElement element;
11     private String value;
12     private UmlTypeScanner umlident = new UmlTypeScanner();
13
14     // Constructor
15     public StructUMLseChDelta(ModelElement _element, String _value) {
16         this.element = _element;
17         this.value = _value;
18     }
19
20
21     // Value methods
22     public String getValue() {
23         return this.value;
24     }
25
26     public String[] getValues() {
27         String[] temp;
28         temp = this.value.split(",");
29
30         //deleting pre- or suffix spaces in the array
31         for (String item: temp) item.trim();
32
33         return temp;
34     }
35
36     public boolean setValue(String value) {
37         this.value = value;
38         return true;
39     }
40
41     // ModelElement methods
42     public ModelElement getModelElement() {
43         return this.element;
44     }
45
46     public boolean setModelElement(ModelElement element) {
47         this.element = element;
48         return true;
49     }
50
51     // ID method
52     public String getID() {
53         return umlident.modelElementID(this.element);
54     }
55
56     // Parent & path methods
57     public ModelElement getParent() {
58       TaggedValue tv = (TaggedValue) this.element;
59       return tv.getModelElement();
60     }
61
62     public String getPath() {
63       TaggedValue tv = (TaggedValue) this.element;
64       return("[" + umlident.modelElementType(tv.getModelElement().getClass().
             getName()) + "] " + tv.getModelElement().getName() + " -> [TaggedValue]
              " + tv.getType().getName());
65     }
66
67     // Pattern method (only for Subs)
68     public ModelElement getPattern(){
69       TaggedValue tv = (TaggedValue) this.element;
70       ModelElement me = tv.getModelElement();
71       for(Iterator iter = me.getTaggedValue().iterator(); iter.hasNext();){
```

```
72          TaggedValue pat = (TaggedValue) iter.next();
73          if(pat.getType().getName().equals("pattern") && pat.getModelElement().
                equals(me)){
74            return pat;
75          }
76        }
77        return null;
78      }
79
80      // Typ method
81      public String getType(){
82          return umlident.modelElementType(this.element.getClass().getName());
83      }
84
85      public boolean equals(StructUMLseChDelta item){
86        if(this.value.equals(item.getValue())
87                && this.getParent().equals(item.getParent())
88                && this.getType().equals(item.getType())){
89                return true;
90        }else{
91          return false;
92        }
93
94
95      }
96  }
```

Listing B.5: `StructUMLseChDelta`

B.4 Tool UMLseCh Static Check: Command SecureDependency

By using data structure `StructUMLseCHDelta` from chapter 4 it is possible to execute the "SecureDependancy"-check on delta. For that we need vector "add","del" and "subs". The output occurs in `ITextOutout`. The `IMdrConteiner` is a global variable because so it doesn't has to be passed each time through parameters. You can find this code in Listing 5.1

```
1 //Output Textbox Stream
2 ITextOutput textOutput;
3 IMdrContainer mdrContainer;
4
5 //datastructure for UMLseCh-Delta
6 Vector<StructUMLseChDelta> add = new Vector();
7 Vector<StructUMLseChDelta> del = new Vector();
8 Vector<StructUMLseChDelta> subs = new Vector();
```

Listing B.6: `SecureDependency` - global variables

The following Listing B.7 contains code of the main check method, which calls method for filling the vectors with delta elements(listing: B.8), method for checking preconditions (listing: B.9 and B.10) and method for checking the secure dependency (listing: B.11). When all checks were successful the concatenation is generated which delivers the results.

```
1 // Secure Dependency check-method
2 public boolean check(IMdrContainer _mdrContainer, Iterator _parameters,
      ITextOutput _textOutput) {
3     boolean fillbool, checkAbool, checkBbool, checkCbool;
4     this.textOutput = _textOutput;
5     this.mdrContainer = _mdrContainer;
```

```
6
7       textOutput.writeLn("####[ UMLseCh Secure Dependency check ]####");
8       textOutput.writeLn();
9       textOutput.writeLn("
            ############################################################");
10      textOutput.writeLn("# STEP 1:
            #");
11      textOutput.writeLn("# Filling the delta-verctors: add, delete and substitute
            #");
12      textOutput.writeLn("
            ############################################################");
13      textOutput.writeLn();
14
15      fillbool = this.fillVectors();
16
17      textOutput.writeLn();
18      textOutput.writeLn("
            ############################################################");
19      textOutput.writeLn("# STEP 2a: Check the Pre-Condition A:
            #");
20      textOutput.writeLn("# Checking that the same modelelement is not rather
            #");
21      textOutput.writeLn("# added and deleted by UMLseCh at the same time.
            #");
22      textOutput.writeLn("
            ############################################################");
23      textOutput.writeLn();
24
25      checkAbool = this.checkA();
26
27      textOutput.writeLn();
28      textOutput.writeLn("
            ############################################################");
29      textOutput.writeLn("# STEP 2b: Check the Pre-Condition B:
            #");
30      textOutput.writeLn("# Checking that no element of a substition is an element
            #");
31      textOutput.writeLn("# of the add or delete vector.
            #");
32      textOutput.writeLn("
            ############################################################");
33      textOutput.writeLn();
34
35      checkBbool = this.checkB();
36
37      textOutput.writeLn();
38      textOutput.writeLn("
            ############################################################");
39      textOutput.writeLn("# STEP 3:
            #");
40      textOutput.writeLn("# Checking the secure dependency UMLseCh conditions.
            #");
41      textOutput.writeLn("
            ############################################################");
42      textOutput.writeLn();
43
44      checkCbool = this.checkC();
45
46      textOutput.writeLn();
47      textOutput.writeLn("
            ############################################################");
48      textOutput.writeLn("# STEP 4:
            #");
49      textOutput.writeLn("# Conclusion:
            #");
```

```
50      textOutput.writeLn("
            ############################################################");
51      textOutput.writeLn();
52
53      if (fillbool && checkAbool && checkBbool && checkCbool){
54        textOutput.writeLn("[SUCC] 'Secure Dependency' conditions are");
55        textOutput.writeLn("       given after UMLseCh evolution. See above");
56        textOutput.writeLn("       more informations.");
57        return true;
58      }else{
59        textOutput.writeLn("[FAIL] 'Secure Dependency' conditions are");
60        textOutput.writeLn("       violated after UMLseCh evolution. See above");
61        textOutput.writeLn("       more informations.");
62        return false;
63      }
64  }
```

Listing B.7: `SecureDependency` - main check-method

The following listing shows how the vectors is filled.

```
1   //-------------------------------------------------------------------------
2   private boolean fillVectors(){
3       CorePackage corePackage;
4       corePackage = mdrContainer.getUmlPackage().getCore();
5
6       // Filling the vectors with all UMLseCh relevant TaggedValues
7       TaggedValueClass taggedValueClass = (TaggedValueClass)corePackage.
            getTaggedValue();
8       for (Iterator iter1 = taggedValueClass.refAllOfClass().iterator(); iter1.
            hasNext();){
9           TaggedValue tagValue = (TaggedValue) iter1.next();
10
11          if(tagValue!=null && tagValue.getType()!=null && tagValue.getType().
                getName()!=null && (tagValue.getType().getName().equals("ref")
12              || tagValue.getType().getName().equals("pattern")
13              || tagValue.getType().getName().startsWith("substitute")
14              || tagValue.getType().getName().startsWith("add")
15              || tagValue.getType().getName().startsWith("delete")
16              || tagValue.getType().getName().startsWith("["))){
17
18              //adds the elements to the delta-datastructure
19              if (tagValue.getType().getName().startsWith("add")) add.addElement(
                    new StructUMLseChDelta(tagValue, tagValue.getDataValue().iterator
                    ().next().toString()));
20              if (tagValue.getType().getName().startsWith("substitute")) subs.
                    addElement(new StructUMLseChDelta(tagValue, tagValue.getDataValue
                    ().iterator().next().toString()));
21              if (tagValue.getType().getName().startsWith("delete")) del.addElement
                    (new StructUMLseChDelta(tagValue, tagValue.getDataValue().
                    iterator().next().toString()));
22          }
23      }
24
25      // Show an overview of found UMLseCh TaggedValues
26      for(StructUMLseChDelta item: add){
27          for(String ausgabe: item.getValues()){
28              textOutput.writeLn("[add] " + ausgabe);
29          }
30      }
31      for(StructUMLseChDelta item: del){
32          for(String ausgabe: item.getValues()){
33              textOutput.writeLn("[del] " + ausgabe);
34          }
35      }
36      for(StructUMLseChDelta item: subs){
```

```
37          for(String ausgabe: item.getValues()){
38              textOutput.write("[sub] " + ausgabe);
39          }
40          TaggedValue pat = (TaggedValue) item.getPattern();
41          if(pat != null) textOutput.writeLn("  [pattern] " + pat.getDataValue().
                iterator().next());
42      }
43
44
45      return true;
46  }
```

Listing B.8: `SecureDependency` - fill vectors

Using the already known scheme in line 8 till 23 all tagged values will be passed and parsed in Line 11. If the tagged value is “add“,“deleted“ or “substitute“ it will be stored together with a value in the according vector in the data structure `StructUMLseChDelta`. Then from line 25 the Output follows.

On the basis of [models2010-Paper] the first precondition (listing B.9) is checked. This check makes the algorithm more stable and identifies previously eventual errors. The proved formula is:

$$\nexists o, o'(o \in \textbf{Add} \wedge o' \in \textbf{Del} \wedge o = o') \tag{B.1}$$

```
1  //-------------------------------------------------------------------------
2  private boolean checkA(){
3      for(StructUMLseChDelta item: add){
4          for(StructUMLseChDelta item2: del){
5              if (item.equals(item2)){
6                textOutput.writeLn("[FAIL] add-vector-element with the value '" +
                     item.getValue() + "' and parent");
7                textOutput.writeLn("       '" + item.getParent().getName() + "'
                     equals an delete-vector-element with the same");
8                textOutput.writeLn("       value and parent!");
9                return false;
10             }
11         }
12     }
13     textOutput.writeLn("[SUCC] There is no evidence of conformity between the add
           - and the delete-vector.");
14     textOutput.writeLn("       The check was succesful.");
15     return true;
16 }
```

Listing B.9: `SecureDependency` - precondition-check A

The second part of the condition that will be analysed (listing B.10) is the following:

$$\nexists o, o'(o \in \textbf{Add} \vee o' \in \textbf{Del}) \wedge ((o, o') \in \textbf{Subs} \vee (o', o) \in \textbf{Subs}) \tag{B.2}$$

In the first part (line 24 ff.) it is checked if added elements are substitute at the same time. In accordance to this the second part (line 26 ff.) checks if a deleted element is also substitute.

```
1  //-------------------------------------------------------------------------
2  private boolean checkB(){
3
```

```
4   for(StructUMLseChDelta item2: subs){
5     TaggedValue pat = (TaggedValue) item2.getPattern();
6
7     // Check Add - Subs
8     for(StructUMLseChDelta item: add){
9       if(pat != null){
10        if(item.getValue().equals(pat.getDataValue().iterator().next())){
11          textOutput.writeLn("[FAIL] The pattern of a substitute-vector-element
                with the value");
12          textOutput.writeLn("       '" + item.getValue() +  "' and the parent '"
                 + item.getParent().getName() + "'");
13          textOutput.writeLn("       equals an add-vector-element with same value
                 and parent");
14          return false;
15        }
16        if(item.getValue().equals(item2.getValue())){
17          textOutput.writeLn("[FAIL] substitute-vector-element with the value '"
                + item.getValue() +  "'");
18          textOutput.writeLn("       and the parent '" + item.getParent().getName
                () + "' equals an add-vector-element");
19          textOutput.writeLn("       with same value and parent");
20          return false;
21        }
22      }
23    }
24
25    // Check Del - Subs
26    for(StructUMLseChDelta item: del){
27      if(pat != null){
28        if(item.getValue().equals(pat.getDataValue().iterator().next())){
29          textOutput.writeLn("[FAIL] The pattern of a substitute-vector-element
                with the value");
30          textOutput.writeLn("       '" + item.getValue() +  "' and the parent '"
                 + item.getParent().getName() + "'");
31          textOutput.writeLn("       equals an delete-vector-element with same
                value and parent");
32          return false;
33        }
34        if(item.getValue().equals(item2.getValue())){
35          textOutput.writeLn("[FAIL] substitute-vector-element with the value '"
                + item.getValue() +  "'");
36          textOutput.writeLn("       and the parent '" + item.getParent().getName
                () + "' equals an delete-vector-element");
37          textOutput.writeLn("       with same value and parent");
38          return false;
39        }
40      }
41    }
42  }
43  textOutput.writeLn("[SUCC] There is no evidence of conformity between the
        substitute-vector, its patterns and");
44  textOutput.writeLn("       the delete- or add-vector. The check was succesful."
        );
45  return true;
46 }
```

Listing B.10: `SecureDependency` - precondition-check B

The check method returns **false** when conditions in one of the elements are violated. This affects the whole check. If the check is successful the method returns **true**.

```
1 for(StructUMLseChDelta additem: add){
2       if(umlident.isElementType(this.mdrContainer.getUmlPackage().getCore(),
            umlident.umlseChOperation(additem.getValue()), "Operation")){
```

```
3           textOutput.writeLn("  [!!!] Found UMLseCh tagvalue associated to the
                operation: " + additem.getValue());
4           textOutput.writeLn("  [Info] parent model element: " + additem.
                getParent().getName());
5           //Check that the Tag is a TagValue of an UmlClass
6           if(umlident.modelElementType(additem.getParent().getClass().getName()).
                equals("UmlClass")){
7             for(Iterator iter_dep = this.mdrContainer.getUmlPackage().getCore
                  ().getDependency().refAllOfClass().iterator(); iter_dep.
                  hasNext();){
8               Dependency depend     = (Dependency) iter_dep.next();
9               UmlClass client       = (UmlClass) depend.getClient().iterator
                    ().next();
10              UmlClass supplier     = (UmlClass) depend.getSupplier().
                    iterator().next();
11              Stereotype depend_stereo  = (Stereotype) depend.getStereotype()
                    .iterator().next();
12              String depend_stereo_value = "";
13              if(depend_stereo != null) depend_stereo_value = depend_stereo.
                    getName();
14              //Only call and send dependencies
15              if(  (depend_stereo_value.equals("call") || depend_stereo_value
                    .equals("send"))
16                && (client.equals(additem.getParent()) || supplier.equals
                      (additem.getParent())) ){
17                boolean bool_client   = false;
18                boolean bool_supplier = false;
19                textOutput.writeLn("  [!->] Checking both classes
                      belonging to dependency " + depend.getName() + ": ["
                      + client.getName() + "-->" + supplier.getName() + "].
                      ");
20                textOutput.writeLn("        Both classes should have the
                      same UMLseCh add-tagvalue.");
21                //Check that in the other Class the security Tag is also
                      removed
22                for(StructUMLseChDelta additem2: add){
23                  //The other class has to delete the same securityTag
                        for the same method ...
24                  if(additem.getValue().equals(additem2.getValue()) &&
                        additem2.getParent().equals(client)){
25                    textOutput.writeLn("        [ok] " + client.
                          getName() + " has 'add = " + additem.getValue
                          () + "'");
26                    bool_client = true;
27                  }
28                  if(additem.getValue().equals(additem2.getValue()) &&
                        additem2.getParent().equals(supplier)){
29                    textOutput.writeLn("        [ok] " + client.
                          getName() + " has 'add = " + additem.getValue
                          () + "'");
30                    bool_supplier = true;
31                  }
32                }
33                for(StructUMLseChDelta delitem2: del){
34                  if(supplier.equals(additem.getParent()) && delitem2.
                        getValue().equals(client.getName())){
35                    textOutput.writeLn("        [ok] Because " +
                          client.getName() + " and all its dependencies
                           will be deleted");
36                    bool_supplier = true;
37                  }
38                  if(client.equals(additem.getParent()) && delitem2.
                        getValue().equals(supplier.getName())){
39                    textOutput.writeLn("        [ok] Because " +
                          supplier.getName() + " and all its
                          dependencies will be deleted");
```

```
40                                  bool_supplier = true;
41                              }
42                            }//-End of Check that in the other ...
43                            if(!bool_client){
44                              textOutput.writeLn("        [xx] " + client.getName() +
                                    " has not the add-UMLseCh-tagvalue which");
45                              textOutput.writeLn("             " + supplier.getName()
                                    + " has.");
46                            }
47                            if(!bool_supplier){
48                              textOutput.writeLn("        [xx] " + supplier.getName()
                                    + " has not the add-UMLseCh-tagvalue which");
49                              textOutput.writeLn("             " + client.getName() +
                                    " has.");
50                            }
51                            bool_add = bool_add & bool_client & bool_supplier;
52                        }
53                    }
54              }else{
55                    textOutput.writeLn("[ERROR] UMLseCh delete-tagvalue referring to
                          an operation but its parent is no class!");
56              }//-End of Check that the Tag is a TagValue of an UMLClass
57          }
58  }
59  // Conclusion of Iterating over als del-Deltas
60  if(bool_add) textOutput.writeLn("  [SUCC] the UMLseCh add-delta satisfies the
        secure dependency conditions");
61  else textOutput.writeLn("  [FAIL] the UMLseCh add-delta does not satisfy the
        secure dependency conditions");
```

Listing B.11: `SecureDependency` - check C

This part runs through the add vector which has been filled with the ‘add‘ part of UMLseCh delta, as shown in listing B.8. Inside this iteration (line 2-57) only vector entries are considered that want to change security properties of a method (line 2). In such case they will be first shown to the user through output (line 3,4) and then the algorithm proves if the found elements belongs to a class at all (line 6). If they don’t an error occurs. One of the secure dependency conditions is that classes which are linked by $<< call >>$ or $<< send >>$ dependencies need the same tags. So it is a task of the algorithm to find this dependencies. For this there it uses an iteration (line 7) and an if-statement (15,16). New created variables between the for-loop and the if-statement are useful for an easier handling of the found connections. When we are within the if-statement it is clear that a $<< call >>$ or $<< send >>$ dependency has been found. That means that the regarded add vector element belongs to one end of the dependency. Now two further iterations (line 22-32 and 34-42) have to find out if the class at the other end of the dependency has also the tagged value or if it is in the delete-vector. Both options would fulfil the condition of ‘secure dependency‘. After this check the found elements are displayed and the if-statements as well as the for-loops are exited.
We would like to point out that there are cases which can not be covered by this algorithm. As it is not possible to load a second diagram as delta you can not add or substitute whole classes, dependencies and frames which enclose several elements. The notation for adding classes via UMLseCh tagged values into ArgoUML and VikiTool is not applicable yet. The professorship is already working on solutions for this problem.

B.5 UMLseCh Notation Analyser: DumpAllUMLSeChElements

`DumpAllUMLSeChElements` check method is more complex than the one of B.2. This method belongs to UMLseCh Notation Analyser Tool. This chapter explains the functionality with the help of Listing B.12.

```
1  public boolean check(IMdrContainer _mdrContainer, Iterator _parameters,
       ITextOutput _textOutput) {
2    //counters
3    int tag_found = 0;
4    int stereo_found = 0;
5    int dep_stereo_found = 0;
6    int tag_exist = 0;
7    int stereo_exist = 0;
8    int dep_stereo_exist = 0;
9
10   //datastructure for UMLseCh-Delta
11   Vector<StructUMLseChDelta> add = new Vector();
12   Vector<StructUMLseChDelta> del = new Vector();
13   Vector<StructUMLseChDelta> subs = new Vector();
14
15   //dump
16   textOutput = _textOutput;
17   textOutput.writeLn("=================== Dumping all UMLseCh elements...");
18
19   CorePackage corePackage ;
20   corePackage = _mdrContainer.getUmlPackage().getCore();
21
22   UmlTypeScanner umlident = new UmlTypeScanner();
23
24   // list all the tagged values
25   textOutput.writeLn ("======= All UMLseCh Tagged Values");
26   TaggedValueClass taggedValueClass = (TaggedValueClass)corePackage.
         getTaggedValue();
27   for (Iterator iter1 = taggedValueClass.refAllOfClass().iterator(); iter1.
         hasNext();){
28     TaggedValue tagValue = (TaggedValue) iter1.next();
29     tag_exist += 1;
30     if(tagValue!=null && tagValue.getType()!=null && tagValue.getType().getName()
           !=null && (tagValue.getType().getName().equals("ref")
31         || tagValue.getType().getName().equals("pattern")
32         || tagValue.getType().getName().startsWith("substitute")
33         || tagValue.getType().getName().startsWith("add")
34         || tagValue.getType().getName().startsWith("delete")
35         || tagValue.getType().getName().startsWith("["))){
36
37       //adds the elements to the delta-datastructure
38       if (tagValue.getType().getName().startsWith("add")) add.addElement(new
             StructUMLseChDelta(tagValue, tagValue.getDataValue().iterator().next().
             toString()));
39       if (tagValue.getType().getName().startsWith("substitute")) subs.addElement(
             new StructUMLseChDelta(tagValue, tagValue.getDataValue().iterator().
             next().toString()));
40       if (tagValue.getType().getName().startsWith("delete")) del.addElement(new
             StructUMLseChDelta(tagValue, tagValue.getDataValue().iterator().next().
             toString()));
41
42       textOutput.writeLn ("Tagged Value: " + tagValue.getType().getName() + " = "
              + tagValue.getDataValue().iterator().next().toString());
43       textOutput.writeLn ("Model Element: " + umlident.modelElementType(tagValue.
             getModelElement().getClass().getName()));
44       textOutput.writeLn ("Model Element Name: " + tagValue.getModelElement().
```

```
          getName());
45      tag_found += 1;
46    }
47  }
48
49  // list all stereotypes
50  textOutput.writeLn ("======= All UMLseCh Stereotypes");
51  StereotypeClass stereotypeClasses = (StereotypeClass)corePackage.getStereotype
        ();
52  for (Iterator iter = stereotypeClasses.refAllOfClass().iterator(); iter.hasNext
        ();) {
53    Stereotype stereotype = (Stereotype) iter.next();
54    stereo_exist += 1;
55    if(stereotype.getName().startsWith("substitute") || stereotype.getName().
          startsWith("add") || stereotype.getName().startsWith("delete")){
56      textOutput.writeLn (stereotype.getName());
57      stereo_found += 1;
58    }
59  }
60
61  // list all stereotypes of dependencies
62  textOutput.writeLn ("======= All UMLseCh Stereotypes of Dependencies");
63  //first list all the dependecies
64  DependencyClass dependencyClass_S = corePackage.getDependency();
65  for (Iterator iter2 = dependencyClass_S.refAllOfClass().iterator(); iter2.
        hasNext();) {
66    Dependency dependency_S = (Dependency)iter2.next();
67    //list all the stereotypes of every dependency
68    for (Iterator iter3 =dependency_S.getStereotype().iterator(); iter3.hasNext()
          ;) {
69      Stereotype stereotype_D = (Stereotype) iter3.next();
70      dep_stereo_exist += 1;
71      if(stereotype_D.getName().startsWith("substitute") || stereotype_D.getName
            ().startsWith("add") || stereotype_D.getName().startsWith("delete")){
72        textOutput.writeLn (stereotype_D.getName());
73        dep_stereo_found += 1;
74      }
75    }
76  }
77  textOutput.writeLn ("The Command found " + tag_found + " Tagged Values, " +
        stereo_found + " Stereotypes and " + dep_stereo_found + " Dependency-
        Stereotypes belonging to UMLseCh,");
78  textOutput.writeLn ("but there are " + tag_exist + " Tagged Values, " +
        stereo_exist + " Stereotypes and " + dep_stereo_exist + " Dependency-
        Stereotypes in general.");
79
80  // List Vectorentries
81  textOutput.writeLn ("");
82  textOutput.writeLn ("
        ################################################################");
83  textOutput.writeLn ("### Add Vector:");
84
85  // Add Vector
86  for(Iterator iter_add = add.iterator(); iter_add.hasNext();){
87    StructUMLseChDelta delta_add = (StructUMLseChDelta) iter_add.next();
88    textOutput.writeLn ("--- Value: " + delta_add.getValue());
89    textOutput.writeLn ("ID    : " + delta_add.getID());
90    textOutput.writeLn ("Path  : " + delta_add.getPath());
91    textOutput.writeLn ("Parent: [" +  umlident.modelElementType(delta_add.
          getParent().getClass().getName()) + "] " + delta_add.getParent().getName
          ());
92  }
93
94  textOutput.writeLn ("");
95  textOutput.writeLn ("### Delete Vector:");
96
```

```
97    // Delete Vector
98    for(Iterator iter_del = del.iterator(); iter_del.hasNext();){
99      StructUMLseChDelta delta_del = (StructUMLseChDelta) iter_del.next();
100     textOutput.writeLn ("--- Value: " + delta_del.getValue());
101     textOutput.writeLn ("ID    : " + delta_del.getID());
102     textOutput.writeLn ("Path  : " + delta_del.getPath());
103     textOutput.writeLn ("Parent: [" +  umlident.modelElementType(delta_del.
            getParent().getClass().getName()) + "] " + delta_del.getParent().getName
            ());
104   }
105
106   textOutput.writeLn ("");
107   textOutput.writeLn ("### Substitute Vector:");
108
109   // Substitution Vector
110   for(Iterator iter_subs = subs.iterator(); iter_subs.hasNext();){
111     StructUMLseChDelta delta_subs = (StructUMLseChDelta) iter_subs.next();
112     textOutput.writeLn ("--- Value: " + delta_subs.getValue());
113     textOutput.writeLn ("ID    : " + delta_subs.getID());
114     textOutput.writeLn ("Path  : " + delta_subs.getPath());
115     textOutput.writeLn ("Parent: [" +  umlident.modelElementType(delta_subs.
            getParent().getClass().getName()) + "] " + delta_subs.getParent().getName
            ());
116   }
117
118   return true;
119 }
```

Listing B.12: Check-Methode des Commands `DumpAllUMLseChElements`

The check method needs same parameters as the one in the previous chapter. In line 3 to 8 of listings B.12three counter variables are created. They will contain the amount of found tagged values, stereotypes and dependency stereotypes and how many of them belong to UMLseCh. It is easy to determine those values as stereotypes and tagged values that belong to UMLseCh start with “add“,“delete“ or “substitute“. You can see that changes by these values are divided in three groups. In line 11 to 13 a vector of `StructUMLseChDelta` type is created for each group. The related class can be found in listing B.5. Those three vectors should contain all made changes to an UML-diagram by `UMLseCh`. After creating objects of type corepackage (line 19,20) and `UMLTypeScanner` (line 22) the analysis of the diagram can begin. First the algorithm analysis the tagged values therefore an object of taggedValueClass type is created which is iterated in line 27. In line 28 within the for-loop the current element is assigned to the variable `tagValue`. The program is now in the for-loop that means a tagged value has been found and counter variable `tag_exist` has to be incremented. With help of the variable tagValue it is possible to examine tagged values more closely. The if-statement in line 30 to 35 checks if the found element is UMLseCh relevant and not null. Line 32 for instance checks whether the name of the found element starts with “add“, “delete“ or “substitute“. In an UML-diagram you can find the name before equals signs. When all if-conditions are fulfilled the type of the element is determined and added to correct vector (line 38-40). Then an output (line 42 to 44) informs the user about found elements. Line 45 increments the counter variable for UMLseCh tagged values. In line 50 to 59 basically the same happens with stereotypes in a diagram. An object of `StereotypeClass` type is created and iterated. When a stereotype element is found a message is printed and the related counter is incremented. This block does not store anything in vectors, for changes only tagged values are imported. The third loop (line 65-76) runs through the dependencies. As we

need stereotypes of dependencies another inner loop is needed to iterate them. This is shown in line 63 to74. Principally these nested loops work like the two before as well as the counter variables for found elements. This counter variables are now important. In line 77, 78 and 79 the check method pints out how many elements of the three different types (TaggedValues, Stereotypes and Stereotypes of Dependencies) has been found and how many of these belong to UMLseCh. Now we come to the last part of the check method. To check if all three vectors contain the correct information they are iterated and their formatted content is displayed. The Iteration of the add vector is extended over line 85 to 92. The for-loop in line 86 to 92 runs through all objects in the vector and make them available by casting with the variable `delta_add` (line 87). In line 88 to 91 an output is generated that needs in places the `UMLTypeScanner` form chapter 7 (for instance in line 91).

```
umlident.modelElementType(
      delta_add.getParent().getClass().getName()
)
```

The lines 97 to 104 and 109 to 116 do the same to the delete and substitute vector. At the end the method returns **true** because the check was successful.

B.6 The class UmlTypeScanner

At this point we discuss methods from the `UMLTypeScanner`. We start with the method `modelElementType`, which can be found in listing B.13.

```
1   // Identifies the modelelementtype
2   public String modelElementType(String _classname){
3     try{
4       Pattern pattern = Pattern.compile(".*\\.(.*)\\$Impl");
5       Matcher matcher = pattern.matcher(_classname);
6       matcher.find();
7       return matcher.group(1);
8     }
9     catch(java.lang.IllegalStateException e){
10      System.out.println(e.getMessage());
11      return "ERROR";
12    }
13  }
```

Listing B.13: Methode `modelElementType`

The method parses an input string and returns the relevant part. The name of an assigned string contains the type of the model element. When the method getClass() is applied to an object the class type of the object is returned. Therefore this method is used to find out of which type an element is. For this the class path and the last part of the string has to be cut off. The method uses pattern matching to achieve this goal. When the method was successful the result is returned. Otherwise the exception is caught and the method returns an “error“ string. The following method `diagramType` identifies with the help of in the model contained elements which type of diagram it is. The method can be found in listing B.14.

```
1   // Identifies the type of the diagram
2   public String diagramType(CorePackage _core) {
3
4     boolean classifierRole = false;
5     boolean message = false;
6     SystemVerificationLoader.logger.trace("Iterating over all diagramelements");
7     for(Iterator it = _core.getModelElement().refAllOfType().iterator();it.
          hasNext();){
8       ModelElement me = (ModelElement) it.next();
9       if(me != null){
10        String typ = this.modelElementType(me.getClass().getName());
11        if(!typ.equals("") && !typ.equals("null")){
12          System.out.print(typ+" #");
13
14          if(typ.equals("ComponentInstance") || typ.equals("NodeInstance"))
              return "Deployment Diagram";
15          if(typ.equals("ActionState") || typ.equals("PseudoState")) return "
              Activity Diagram";
16          if(typ.equals("Class") || typ.equals("UmlClass")) return "Class Diagram
              ";
17          if(typ.equals("UseCase")) return "Use Case Diagram";
18          if(typ.equals("Initial")) return "Statechart Diagram";
19
20          if(typ.equals("Message")) message = true;
21          if(typ.equals("ClassifierRole")) classifierRole = true;
22        }
23      }
24    }
25    if(classifierRole){
26      if(message) return "Sequence Diagram";
27      else return "Collaboration Diagram";
28    }
29
30    //Diagramtype couldn't be identified by its components
31    System.err.println("Diagramtype couldn't be identified by its components");
32
33    return null;
34
35  }
```

Listing B.14: Methode `diagramType`

At the beginning two boolean variables `classifierRole` and message are initialized with **false**. These variables are set to **true** when in line 20 and 21 an element message or `classifierRole` in a diagram is found. These elements have to be stored to make it possible to distinguish between a sequence diagram or collaboration diagram. Like in the `DumpAllModellElements` check method all diagram elements are iterated. The type of the diagram can be detected through the elements which are specific for a diagram. In line 14 and the following it is shown how the different types can be detected. Our example is the deployment diagram. When a diagram contains an element “ComponentInstance“ or “NodeInstance“ it can be clearly identified. So the method can be terminated without any further examinations. Line 20 and 21 are interesting as they show that a method can’t be terminated instantly because a sequence diagram contains messages as well as `classifierRoles`. The collaboration diagram contains only `classifierRoles`. That means if only a `classifierRole` is found the algorithm can not decide which diagram type it is. This decision is made in line 25 ff. after execution of the for-loop. Please note that a sequence diagram without messages will be detected as a collaboration diagram. If there is no diagram, because no elements are detected, an error message is written to

the console and returns the `string null`.

Next is the method

public boolean identifiable(CorePackage _core){...},

which contains only the following line:

return this.diagramType(_core)!=**null**;.

So it just calls the before explained method. It just identifies whether a diagram could be detected or if the method returns **null** . Furthermore a method will be explained which returns **true** when the diagram is of the type which is described by the second parameter. The according code is in listing B.15.

```
1  // Identifies if the type of the diagram is equal to the given type
2    public boolean diagramType(CorePackage _core, DiagramType aType) {
3      switch (aType) {
4          case DeploymentDiagram:
5              return this.diagramType(_core).equals("Deployment Diagram");
6          case ActivityDiagram:
7              return this.diagramType(_core).equals("Activity Diagram");
8          case ClassDiagram:
9              return this.diagramType(_core).equals("Class Diagram");
10         case UseCaseDiagram:
11             return this.diagramType(_core).equals("Use Case Diagram");
12         case StatechartDiagram:
13             return this.diagramType(_core).equals("Statechart Diagram");
14         case SequenceDiagram:
15             return this.diagramType(_core).equals("Sequence Diagram");
16         case CollaborationDiagram:
17             return this.diagramType(_core).equals("Collaboration Diagram");
18         default:
19             return false;
20     }
21
22   }
23
24   public static enum DiagramType {
25       DeploymentDiagram, ActivityDiagram, ClassDiagram,
26       UseCaseDiagram, StatechartDiagram, SequenceDiagram,
27       CollaborationDiagram
28   }
```

Listing B.15: zweite Methode `diagramType`

The second parameter in line 2 is an enumeration type which is classified in line 24. A switch statement allows us to distinguish between the values and returns **true** when the diagram in `CorePackage` is the expected diagram by the value. Otherwise it returns **false**.

At least a method has been implemented which identifies and returns the ID of the passed model element. The implementation is in listing B.16. The ID is a hexa-decimal number therefore it is returned as a `String` and not as an `int`. The ID is filtered by a pattern matcher (line 4-6) which gets as input string the output of the `toString()` method applied on a model element. In order to prevent a system crash an exception is caught when no ID can be found. In this case an “error“ string is returned instead of an ID.

```
1  //Identifies ID
```

```
2  public String modelElementID(ModelElement _me){
3    try{
4      Pattern pattern = Pattern.compile(".*ID: (.*)  MID:.*");
5      Matcher matcher = pattern.matcher(_me.toString());
6      matcher.find();
7      return matcher.group(1);
8    }
9    catch(java.lang.IllegalStateException e){
10     System.out.println(e.getMessage());
11     return "ERROR";
12   }
13 }
```

Listing B.16: Methode `modelElementID()`

C Downloads

The UMLsec tool, including the UMLseCh plugins can be downloaded in:

`http://www-jj.cs.tu-dortmund.de/jj/umlsectool/Plugins/index.htm`

The screencast and examples of the plugin presented in this Deliverable can be downloaded in:

`http://inky.cs.tu-dortmund.de/main2/jj/umlsectool/manuals_new/UMLseCh_Static_Check_SecureDependency/index.htm`