MASTER IN COMPUTER SCIENCE

Modelling future evolution in a security oriented context

Marchal, Loïc

Award date:
2010

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2009-2010

Modelling Future Evolution in a Security Oriented Context

Loïc Marchal

Mémoire présenté en vue de l'obtention du grade de master en informatique.

I

Abstract

English

Software evolution and security remain to be challenges faced by developers of software systems. Due to the predictability of certain evolution and security issues, the inclusion of techniques to address such anticipated factors is recommended at the early stages of software development. This thesis examines the possibility to extend UMLsec in order to address the representation of future evolution and an approach to execute the modelled evolution. Since UMLseCh is an extension of the original UMLsec, the resulting model can be verified using the method defined by the UMLsec approach.

An extension of the UMLsec profile was achieved by adding new stereotypes and tagged values in order to include the evolution concepts. Formal semantics was proposed to specify the appilcation of an evolution. An abstract syntax was then formally defined to allow represention of the UMLseCh constructs and diagrams. These concepts could form a preliminary foundation for future research in software evolution and security.

Français

L'évolution des systèmes ainsi que leur sécurité demeurent aujourd'hui encore un défi auquel les développeurs doivent faire face. Vu la prédictabilité de certaines évolutions et la présence de plus en plus importante des exigences de sécurité, la prise en compte des ces facteurs à un stade précoce du cycle de développement est plus que souhaitable. Le travail présenté dans ce mémoire examine la possibilité d'étendre UMLsec dans le but d'inclure la représentation des possibles futures évolutions. Une approche pour exécuter ces évolutions est également présentée.

Une extension du profile UMLsec a été réalisée en ajoutant des stereotypes et tagged values d'UML afin d'inclure les concepts de l'évolution des modèles dans le language. Une sémantique formelle a été définie afin de représenter l'application d'un changement et une syntaxe abstraite a permis de formellement représenter les constructions et diagrammes d'UMLseCh.

Ces concepts ont rendu possible l'élaboration d'un travail préliminaire ouvrant la voie à une recherche future dans le domaine de l'évolution des modéles et de la préservation de la sécurité.

Acknowledgements

I would like to sincerely thank my supervisors, Pr. Pierre-Yves Schobbens and Pr. Jan Jürjens. This thesis would not have been possible without them. I was able to explore and develop an understanding of the subject with their guidance and assistance throughout my research.

In addition, I would like to offer my gratitude and respect to my colleagues from the Univeristy of Dortmund, and those who supported me in any manner to complete this project. In particular, Martìn Ochoa Ronderos, my working partner for the research conducted during my internship.

Lastly, I would like to thank family and friends for their encouragement. A very special thanks to Chooi Peng Yap for proofreading my thesis.

Loïc Marchal

Contents

List of Figures

Chapter 1

Introduction

Software evolution has been stated as a major concern since the seventies [Leh74] when it appeared that software could not be implemented in a definite and conclusive manner. Instead, they constantly evolved and underwent modifications to adapt to users and environment [LP76]. Such modifications could be caused by a variety of reasons. For example, a change in the company's strategy would result in the need of change in requirements. New needs could also arise within the users and therefore functionalities would have to be added or obsolete ones removed. In other instances, the system could simply have an error that needed correction. Eventually, it was acknowledged that a need for evolution had to be considered at the early stages of software development so that the systems would be designed in a way that facilitates changes [Par79]. The idea for the need of software maintainability was hence included into the software development process.

Although the idea of software evolution was taken into consideration after [Leh74], it continues to be a challenge for researchers of software evolution [MWD+05]. Throughout the years, research has been conducted to provide techniques and methods to support certain aspects of evolution. However, it remains an active field of research today.

While certain changes are unpredictable, others are expected to happen. Expected evolutions that may happen can be caused by an implementation of functionalities that is postponed. Reasons for postponement include budget restrictions, awaiting the availability of an upcoming technology, or anticipation of the users needs or future requirements. Several application domains could have a short design time and a long life span, causing a need for the system to be constantly redesigned. Therefore, it appears advisable to model those predicted evolution at the early stages of development. This is achieved by providing techniques to apply modifications automatically when the decision to apply expected changes finally materializes.

In addition to software evolution, it appeared that security was a growing requirement for software systems. While software systems in the past were isolated in a company, many were later connected to the world through the Internet. This allowed the offer of access to services, but it also provided a means for potential adversary to damage the system. Such attacks include identity fraud, and unauthorised retrieval of private information. It appeared that security should be taken into account at the early stages of software development, as opposed to the final stages before release. UMLsec [Jür10] is an extension of UML that provides a notation and a methodology to design secured software at such an early stage.

The systems that require security still need to evolve. Therefore, a challenge for such systems is to ensure that they will remain secured after an evolution. It thus appears important to bring together the two concepts described above, i.e. the modelling of evolution and the modelling of security requirements, both at the early stage of development. In doing so, it would then be possible to model systems together with their possible future evolution and their security requirements.

In this thesis, we present an approach that aims to address the modelling of evolution of security critical software. More precisely, we take the UMLsec extension as a base for security modelling and extend it to include the concept of possible future evolutions. We define a notation with a UML profile that can be applied on UMLsec diagrams to express model evolutions. We then describe formally the semantics of the notation in order to specify how a change is applied to the model. We finally conclude and introduce the future work.

Chapter 2 introduces background and related work for each of the related domain, namely software evolution, model transformation and security modelling. It also presents the UML Profile mechanism used to define our notation. Chapter 3 then briefly describes UMLsec.

In Chapter 4 we introduce the extension of UMLsec, called UMLseCh. We present the profile, describe the notation and illustrate how it can be used by means of examples. In Chapter 5, we define the formal foundation of the language in order to provide techniques to apply the changes modelled by the notation. Finally, we conclude in Chapter 6 and present recommended future work.

Chapter 2

Background and Related Work

The work described in this thesis concerns several domains among the modelling principles, including concepts such as security modelling, model checking, model tranformation and software evolution. This chapter briefly describes the related concepts and presents a state of the art of these different fields. Because the material presented in Chapters 3 and 4 is directly based on UML, and more precisely defines extensions of UML, a review of the UML extension mechanism based on profiles is also given.

2.1 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 softwares and show the difference that exist between those techniques and the approach developped in this thesis.

The importance of evolution in the process of developing large softwares was first discussed in the early seventies [Leh74], after a study of the software process in 1968 [Leh69]. 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 [LP76]. These modificactions 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 [Leh74] to the last two laws, first published in [Leh96]. The first six laws were also revisited in [Leh96].

In addition, it was stipulated in [LRW+97] that softwares keep growing and changing while [Par94] argued that softwares 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 [MWD+05]. 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 [MWD+05]. Other challenges exist around the concept of change and evolution management, refer to [MWD+05] for the complete description. Nonetheless, we can present several achievements in the context of software evolution. One of the first and best known good practise 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 practise 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 [Par79]. However, [RLL98] 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 [MWD+05]. However, to be able to interpret the collected data, [MWD+05] mentions the necessity of defining a *theory of evolution*, which follows the idea presented in [LRK00] and [LR01]. On a smaller scale, a classification of 12 different types of software evolution and software maintenance was given in [CHFRT01]. This taxonomy, focused on the purpose of a change, is refined in [BMZ+05] where other points of view, such as the *how*, the *when* or the *where*, are considered. In [BLO03], a list of possible changes that could occure on UML diagrams is given, together with an approach to define impact analysis. Another idea is also presented in [GD06], 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 to analyse 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, [Läm04] presents the evolution of rule-based programs and provides an operator suite for the trans-

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

Figure 2.1: Transformation of a rule-based program [Läm04]

formation 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 2.1 illustrates an example of such a transformation. This approach is thus only focused on the code. The self-adaptive softwares [ST09] 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 thesis, this solution treat changes when they occur but do not anticipate them. Both the solution presented in [Läm04] and [ST09] focus on the level of source code and hence differs from the technique defined in this thesis, which aim to represent evolution on models expressed by UML diagrams. Representing evolution by model transformations will be discussedd in the next section.

An approach to model transformation and evolution at a high-level of abstraction is given in [Sto]. 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 developped around UML, the models are generally considered as typed trees where 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 2.2. 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 2.3. This approach is very similar to the one presented in this thesis. 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 4 represents the evolutions more explicitly. Indeed, Figure 2.2 does not clearly indicate how the elements are updated and this information cannot be found on the class diagram of Figure 2.3 either. In addition, UMLseCh does not include graphical representations as the ones of Figure 2.2 and remains compliant to the UML specifications [OMG09]. The solution

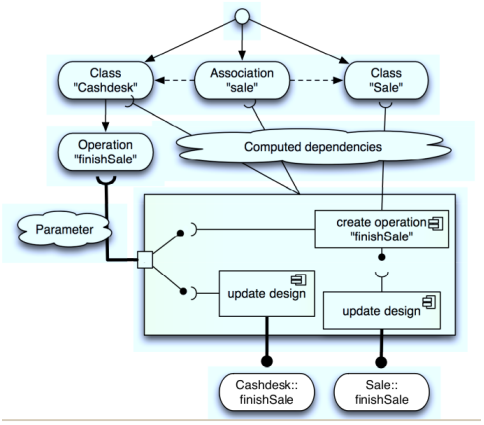

Figure 2.2: Representation of a concrete evolution illustrated in [Sto]

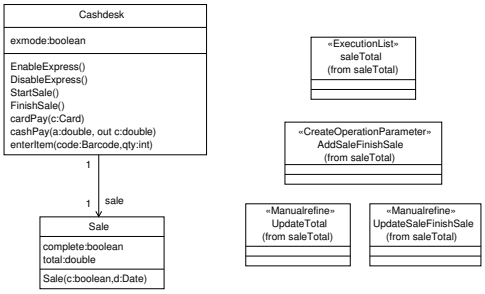

Figure 2.3: Representation of a concrete evolution on the class diagram, as shown in [Sto]

of [Sto] and the one described in this thesis thus target a challenge specifying that "modelling 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" [MWD+05].

2.2 Model Transformation

Applying evolution and changes on models evidently requires to execute model transformations. 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 exists 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.

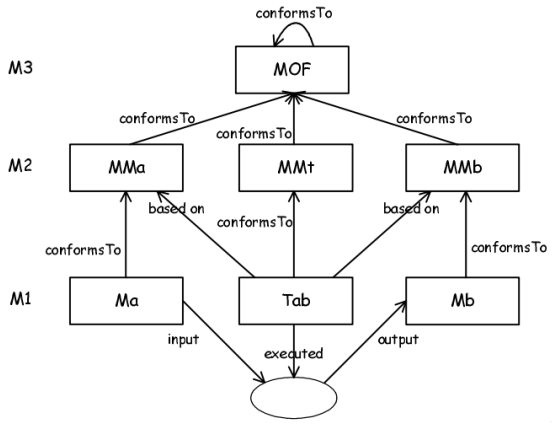

Figure 2.4: Context of ATL Transformation as shown in [JK06]

As mentiomed in [Sun09], the evolution of a model is often supported by model transformation rules written in specific languages. The OMG [Gro] defined a standard for expressing model transformations, called QVT (Query/View/Transformation) [OMG05]. As mentioned in [JK06], the growing importance of model transformations led the OMG to publish a QVT request for proposal (RFP) [OMG02]. Several propositions answered that RFP, which then evolved toward a single proposal [OMG02]. Others however continued to develop independently, as for example the ATLAS Transformation Language (ATL) [Lana]. 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 [JK06]. The concept of model transformation with ATL is shown in Figure 2.4. The idea of sofware 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 2.5, where 'newOp()' is simply an abstract representation of the added operation to facilitate the example.

Tefkat [LS] is another example model of a transformation language, which in this case is based on F-Logic [KLW95]. An example of a Tefkat rule, taken from [LS] where the complete transformation can be found, is shown in Figure 2.6.

For Tefkat as for ATL, they both represent the evolutions with rules using

```
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 2.5: 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 2.6: Example of a transformation using Tefkat

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 [Wil03b] is a graphical model transformation language that was also developed to answer the QVP RFP [OMG02] 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 2.7. 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 2.4. No transformation operator exist in UMLX for an addition and thus, adding an element is not explicitly modelled. Instead, transformation shows the result with the new element added on the model. Finally, a main difference with the work described in this thesis 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 [Wil03a], where UMLX is used to model transformations from UML class diagrams to relational data bases (RDBMS).

MOLA (MOdeling transformation LAnguage) [KBC04] 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 2.8 shows an example of a MOLA program, described in [KBC04], 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

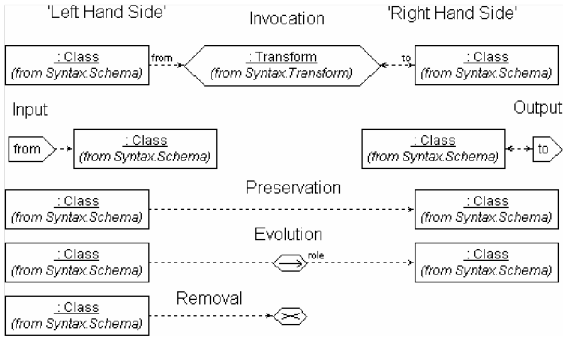

Figure 2.7: UMLX Transformation Syntax [Wil03b]

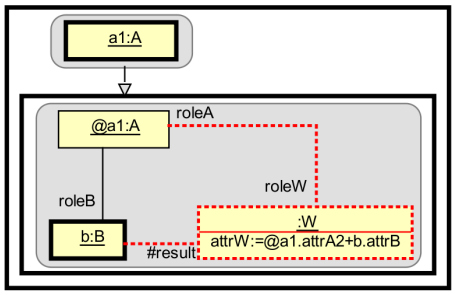

Figure 2.8: Example of a transformation using a MOLA program [KBC04]

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 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 [KBC]. The same comparison as the one between the approach defined in this thesis and UMLX can be made for Mola.

2.3 Security Modelling

The security context required in the approach defined in this thesis is modelled with UMLsec [Jür10]. It is an extension of UML that formulates security requirements as non-functional requirements. Because UMLsec will fully belong to the notation defined further in this document, it will be described in more details in Chapter 3. Other works and research have been achieved to use UML for security systems development. However, they differ from

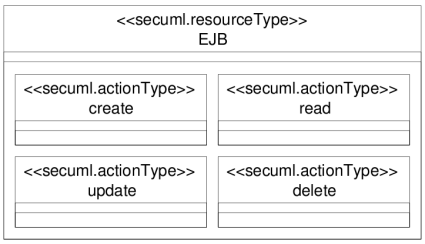

Figure 2.9: SecureUML used for EJB [LBD02]

UMLsec in the sense that "they cover a less comprehensive set of security requirements, mostly focussing on role-based access control requirements" [Jür10]. In the following, we introduce two approaches for including security concepts with UML. The first is SecureUML [LBD02], focused on the role-based access control, and the second is SecurityAssessmentUML [HH03], which provides a representation for model-based security estimation.

SecureUML [LBD02] is an extension of UML that provides and approach based on role-based access control with additional support for specifying authorization constraints. A metamodel defines the abstract syntax and the notation used with class models in UML is defined as a UML Profile. The vocabulary defined for SecureUML provides a means to express different aspects of access control, such roles, role permissions and user-role assignments. An example illustrating the use of SecureUML in the context of Enterprise Java Beans (EJB), taken from [LBD02], is shown in Figure 2.9. As mentioned above, this approach differs notably from the one defined with UMLsec since SecureUML concerns only role-based access control.

SecurityAssessmentUML [HH03] is another extension of UML which provides support for UML sequence and activity diagrams for risk identification and UML activity diagrams for risk analysis. It is also defined with a profile that provides a specific notation, including associated icons. This notation allows one to formulate the documentation of output from risk identification and risk analysis in a security assessment on the UML models. An example illustrating the SecurityAssessmentUML notation on a UML sequence diagram is given in Figure 2.10. The diagram of this example represents the result from a risk analysis showing that the system could possibly be infected by a virus since the email gateway does not have any antivirus. As for SecureUML, SecurityAssessmentUML differs from UMLsec in the sense that it targets a different type of security concept. In this case, it only focuses on security estimation by modelling results from risk analysis.

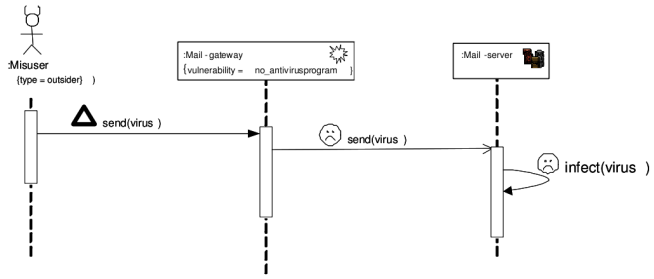

Figure 2.10: SecurityAssessmentUML on a sequence diagram [HH03]

2.4 UML Profile Mechanism

UML [Lanb], the standard defined and maintained by the Object Management Group (OMG) [Gro], has been widely adopted by both the industry and academic world and is now generally accepted as the de-facto modelling language for software engineering. In recent years, following this success, the need to use UML in specific contexts has fundamentally increased, as it is shown in several publications, such as [Küh] or [MFV06]. It is the case, for example, with Web Engineering [MFV06], Aspect-Oriented Programming (AOP) [KKKS], Architecture Description, security or Quality of Services (QoS) [CP04]. However, as it was initially designed for general purposes, UML "out of the box" might not be suitable for these specific domains. Thus, a mechanism to mold UML appears crucial in order to define a Domain Specific Language (DSL). The OMG offers two extension mechanisms for UML, described in [OMG09]; a light-weight mechanism based on profiles and a heavy-weight mechanism based on the Meta-Object Facility (MOF).

The MOF is an OMG's standard that provides a framework for defining and manipulating metadata and metamodels and providing interoperability between them. UML is a MOF-compliant modelling language; in other words, the UML metamodel is a MOF model. UML can therefore be directly extended using the MOF standard by adding new "meta-elements" (meta-classes, meta-attributes, meta-associations, etc). This approach has the benefit of offering the possibility to completely tailor the language. From an additive point of view (note that adapting the language could be done, for example, by deleting concepts or redefining properties, which is also possible through the MOF approach), it allows one to define new concepts independently from the others, as opposed to the lightweight mechanism which only adds element additively so that it satisfies the standard semantics of the UML metamodel. It therefore offers a higher level of expressiveness and gives more aleternatives to integrate the new concepts in the language. For

this reason, several authors, as for example in [HKC05] or [PM03], have argued that certain concepts should be first-class elements in UML. However, extending UML by modifying its meta-model presents a major disadvantage: the existing tools only support the standard UML metamodel, as it is defined in [OMG09], and will thus need to be refactored in order to support the extended language. The "heavyweight" extension mechanism will not be considered further in this work and the extensions presented in chapters 3 and 4 will be achieved using the "lightweight" UML profile mechanism.

The UML Profile mechanism, as it is described in [OMG09], provides a lightweight extension mechanism that allows one to customise UML. It uses three extensibility mechanisms, namely stereotypes, tagged values and constraints. Stereotypes allow one to define new modelling elements from existing ones by refining the semantics of these existing elements. Their notation consists of the name of the stereotype written between guillemets; i.e. a stereotype named *stereotype* will be written « *stereotype* ». If a model element is extended by more than one stereotype, the stereotypes can be represented in a comma-separated list written between guillemets, as described in [OMG09]; i.e. n stereotypes named $stereotype_1, ..., stereotype_n$ will be written « $stereotype_1, ..., stereotype_n$ ». However, if stereotype properties are defined, the selected notational form for these properties, as it is explained in the next paragraph, will not allow to easily connect, graphically, the properties to the right stereotype. This "list notation" will thus not be adopted afterwards. An icon can also be defined to graphically represent an extended model element, but this notation will not be used either. Once defined, the stereotypes are then attached to the model element to extend. As mentioned above, this method is strictly additive. It cannot create new first-class elements but only provides refinement of the existing UML modelling elements. Stereotypes hence do not exist by themselves and are always attached to at least one base element. An example of a refined semantics of a UML model element using a stereotype is shown in Figure 2.11. The stereotype « LAN » is attached to a link of a deployment diagram in order to create a new type of link. This new model element inherits the characteristics of the extended model element, i.e. the link of a deployment diagram, and in addition, refines its semantics to define a specific type of link. In this case, it creates a model element that represents a local area network (LAN) link. The stereotype « LAN » is part of the UMLsec extension [Jür10] that will be presented in chapter 3. In [OMG09], as opposed to [OMG00], there is no restriction on the number of stereotypes that can be assigned to a model element.

Tag definitions allow one to define properties by adding attributes to a model element. The name given to such an attribute is called *tag*. Until [OMG00], tag definitions could be directly associated to any model element.

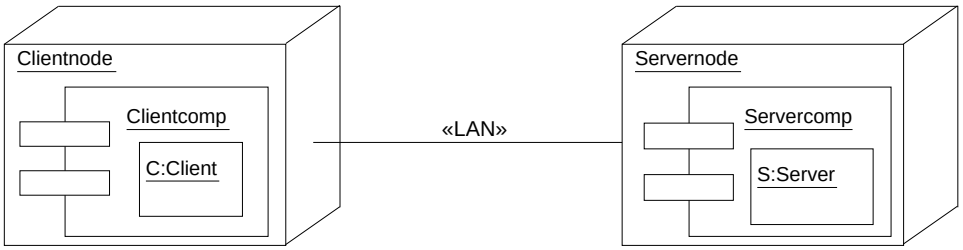

Figure 2.11: Example of the use of a stereotype « LAN »

In [OMG01], it was deprecated but still supported. Since [OMG09], such attributes can only be defined on a stereotype and thus can only extend model elements that are extended themselves by at least one stereotype. Tags can be either a DataTag, which represents data values, or a ReferenceTag, which represents a reference to another model element. When a stereotype is applied to a model element, its associated attributes receive values called *tagged values*. A tagged value consists of a name-value pair and three alternative notations are described in [OMG09]. The first consists of attaching a comment symbol to the extended model element in which the tagged values are shown, each of them being listed under the name of the associated stereotype. The second notation is represented by a box (similar to the notational form of a class) in which the tagged values are listed under the name of the concerned stereotype. Finally, the third notation consists of writing the tagged value between curly brackets next to the associated stereotype; i.e. for a name *tag* and a value *value*, the tagged value will appear as { *tag* = *value* }. For a boolean attribute, the value is not written on the diagram and instead, the name of the attribute appears if the value is *true* and does not appear if the value is *false*. The third notational form has been prefered and will thus be used in the rest of this work. If the property is multi-valued, the values are placed in a comma-separated list; i.e for a tag *tag* and values *value1*, *value2* and *value3*, and the tagged value will appear as {*tag* = *value1*, *value2*, *value3*}.

Finally, constraints can be added to a stereotype in order to define well-formedness rules that the stereotyped model element will have to fulfill.

To create an extension of UML, one has to define a UML *profile*. A *profile*, described in [OMG09] as a package stereotyped « profile », collects definitions of stereotypes, tagged values, constraints and notational representations. It can extend the UML metamodel or any other profile and gives a specific semantics to the extended model elements. Divers UML profile has been defined, for example the UML Profile for Software Development Processes [Küh], the UML Profile for Modelling Knowledge-Based Systems

[AEBK04], the UML Profile for Role-Based Access Control [CB09] or the UML Profile for AspectJ [Eve07]. As mentioned above and discussed as an example in [MFV06], the UML Profile extension mechanism presents the main advantage of being compliant with the UML metamodel and therefore can be supported by the existing tools. In Chapter 3, the UMLsec profile, that allows one to model security requirements, will be presented in more details. The UMLseCh profile, which is an extension of the UMLsec profile for evolutive systems, will be described in Chapter 4.

2.5 Discussion

The previous sections introduced several approaches from different domains related to the work presented in this thesis. While each of the existing solutions described above addresses one of the domains independently from the others, the approach defined in this document aims to bring these different fields together.

This chapter referenced [MWD+05], which argues that software evolution is still considered as a challenge and that languages should include such concepts. One approach addressing this challenge [Sto] was presented in Section 2.1. Different model transformation languages were then described, some being graphical, like UMLX [Wil03b] or MOLA [KBC04], while others are text-based, such as ATL [Lana] or Tefkat [LS]. The graphical notation would be defined by creating specific notational constructs that are not compliant to the UML metamodel. It was mentioned that some simplified parts of these languages could be used to represent the evolution of a model as we do in this thesis. This could be achieved by restricting the techniques to transformations where the target model is compliant to the same metamodel as the source model. More complexe evolutions can of course also be represented with such languages. After a relatively important effort of research, no approach for modelling possible future evolution with a UML profile could be found.

The techniques presented in this thesis also aim to place the evolution in a secure context. In particular, because the security of the evolved model should be preserved, several concepts modelling security requirements were also introduced in this chapter. However, these techniques addresses different types of requirements and security concepts from the ones used in this thesis. Again, to the extent of the author's knowledge, no approach considering the evolution in the context of a model-based development approach for security-critical software was found.

Chapter 3

Modelling Security with UMLsec

In this chapter, we present the UMLsec approach for modelling secure softwares. We first briefly describe the concept used to formally verify the security of the system. We then introduce the notation and the abstract syntax that will be extended in Chapter 4 in order to define UMLseCh to include evolution concepts.

3.1 UMLsec Overview

UMLsec is an extension of UML aimed to model security requirements and verify 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. A system is divided in components given by subsystems and communicating together. A subsystem may also have $C_1, \ldots, C_n$ subcomponents, which may also communicate through the links of the deployment diagram modelled in that subsystem. A behovioral semantics is given in the form of UML Machines which specifies the execution of subsystems and their communications. UML Machines, defined in [Jür10], are a special type of Abstract State Machines (ASM) [Gur95]. They are statemachines communicating with their environment by exchanging messages, using input queues and output queues to store these messages. The behavioral interpretation $[\![\mathcal{S}]\!]$ of a subsystem $\mathcal{S}$ is thus given by a UML Machine defined in [Jür10]. A scheduler is also assumed to take a message from an output queue of a UML Machine and place it in the input queue of the intended UML Machine.

UML Machines are also defined to represent the behaviour of adversaries.

Different kind of threats arising from the presence of an adversary exist on links and components. These threats are the possibility for the attacker to read, insert or delete information on the links as well as the possibility to access a node connected to a link. Adversaries can also be of different types, such as the insider or the default adversary. UMLsec represents the security concepts on the model by using stereotypes and tagged values. Certain stereotypes, such as « Internet » or « encrypted », thus have associated threats while others, such as « secure links » or « secrecy », have associated constraints. The function $\mathsf{Threats}_A(s)$ is defined as a function that takes an adversary of type A and a stereotype s associated to a link or a component and return a a subset of $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$. $\mathsf{Threats}_A(s)$ represent the set of abstract threats that exist on a link or node stereotyped s given the presence of an adversary of type A. From this set of abstract threats, the set $\mathsf{threats}^{\mathcal{S}}_{\mathsf{A}}(x)$ of concrete threats on a link or node x in a subsystem $\mathcal{S}$ in the presence of an adversary of type A is derived and defined as "the smallest set satisfying the following conditions. If each node n containing x carries a stereotype s_n with $\mathsf{access} \in \mathsf{Threats}_A(s_n)$ then for every stereotype s attached to x, we have $\mathsf{Threats}_A(s) \subseteq \mathsf{threats}^{\mathcal{S}}_A(x)$ and if x is a link connected to a node that carries a stereotype t with $\mathsf{access} \in \mathsf{Threats}_A(t)$ then $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\} \subseteq \mathsf{threats}^{\mathcal{S}}_A(x)$"[Jür10]. To illustrate how these concepts are used in UMLsec, we can consider the following simple example which takes the case of the preservation of the secrecy over a link stereotyped « Internet » with the presence of the $default$ adversary. The stereotype « secure links » attached to the subsystem requires that the security requirements defined on dependencies of the deployment diagram are respected. Assume a dependency stereotyped « secrecy » between two objects, each of them being located in a nodeand the nodes being connected by a link l, defines the following constraint: $\mathsf{read} \notin \mathsf{threats}^{\mathcal{S}}_{\mathsf{A}}(l)$. However, UMLsec defines $\mathsf{Threats}_{default}(\text{« Internet »}) = \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$ and thus $\mathsf{read} \in \mathsf{Threats}_{default}(\text{« Internet »})$. The secrecy is therefore not ensured on a link stereotyped « Internet » and the constraint « secure links » is violated. Other important security properties, such as integrity, authenticity or freshness are defined in a similar way. Formal definitions of these properties are also given in a form adapted to the context of the UML Machines and thus to the execution of the system and to the behaviour of the adversary. UMLsec defines the knowledge of an adversary $\mathcal{A}$ defined as $\mathcal{K}_{\mathcal{A}}$. This knowledge is the initial knowledge $\mathcal{K}^0_{\mathcal{A}}$ at the startup of the system and can be iteratively extended during the execution of the system. The execution of a subsystem $\mathcal{S}$ in the presence of an adversary $\mathcal{A}$ is defined as $[\![\mathcal{S}]\!]_{\mathcal{A}}$. We do not describe the execution of UMLsec subsystems and the verification of the security in any more detail because such a level of detail is beyond the scope of this work. The reader should refer to [Jür10] for a complete definition of the behvioural semantics and the formal verification of the security concepts.

UMLsec also defines a notation to represent the security concepts and requirements on the UML diagrams. It uses the UML Profile mechanism, presented in Chapter 2. The stereotypes and tagged values of the UMLsec profile define a notation that gives a means to generally represent different type of security principles and concepts among different types of diagrams. More precisely, although the profile concerns all of UML, the notation is based and used on a subset of UML which includes simplified versions of class and object diagrams, deployment diagrams, activity diagrams, statechart diagrams, sequence diagrams and subsystems. The base classes of UMLsec stereotypes are shown in the next section in Figure 3.1. The UMLsec profile, as well as the notation, the abstract syntax, the behavioural semantics and the tool support are defined in [Jür10]. In the next sections, we present some part of [Jür10] that will be the background of our further work. In particular, we present the UMLsec notation, that consists in stereotypes and tagged values, we give examples, and we describe the UMLsec abstract syntax that will be extended in Chapter 5 in order to include the UMLseCh stereotypes and thus include a means to model possible evolutions of models.

3.2 UMLsec Notation

In this section, we briefly introduce the UMLsec notation. We show the set of stereotypes used to represent the security concepts and requirements of a model and their associated tag definitions. We also show how to use this notation by means of examples. However, we only illustrate how to model the representation of security oriented system with UMLsec but we do not formally verify the security requirements with the techniques mentioned in the previous section.

The complete list of UMLsec stereotypes together with their base class, their associated tags, their constraints and a descriptions is given in the table of the Figure 3.1. These stereotypes can be used on the model elements of the simplified diagrams of UMLsec in order to model the security concepts needed in a critical system. Figure 3.2 shows the list of UMLsec tags together with their associated stereotype, their type, their multiplicity and a description. The complete description of the stereotypes and their associated rules, the well-formedness rules and the security threats and constraints associated to those stereotypes can be found in [Jür10]. Such a detailed description is beyond the scope of this work. In the following, we simply show an example of a critical system modelled with UMLsec.

As an example of use of the UMLsec notation, we can illustrate the two following situations, taken from [Jür10]. The first example, shown in Figure 3.3, models a secure channel at a high level of abstraction. We assume a

Stereotype	Base Class	Tags	Constraints	Description
fair exchange	subsystem	start, stop, adversary	after start eventually reach stop	enforce fair exchange
provable	subsystem	action, cert, adversary	action is non-deniable	non-repudation requirement
rbac	subsystem	protected, role, right	only permitted activities executed	enforces role-based access control
Internet	link			Internet connection
encrypted	link			encrypted connection
LAN	link, node			LAN connection
wire	link			wire
smart card	node			smart card node
POS device	node			POS device
issuer node	node			issuer node
secrecy	dependency			assumes secrecy
integrity	dependency			assumes integrity
high	dependency			high sensitivity
critical	object, subsystem	secrecy, integrity, authenticity, high, fresh		critical object
secure links	subsystem	adversary	dependency security matched by links	enforces secure communication links
secure dependency	subsystem		«call», «send» respect data security	structural interaction data security
data security	subsystem	adversary, integ., auth.	provides secrecy, integrity, authenticity, freshness	basic data security requirements
no down-flow	subsystem		prevents down-flow	information flow condition
no up-flow	subsystem		prevents up-flow	information flow condition
guarded access	subsystem		guarded objects accessed through guards	access control using guard objects
guarded	object	guard		guarded object

Figure 3.1: The UMLsec stereotypes

Tag	Stereotype	Type	Multip.	Description
start	fair exchange	state	*	start states
stop	fair exchange	state	*	stop states
adversary	fair exchange	adversary model	1	adversary type
action	provable	state	*	provable action
cert	provable	expression	*	certificate
adversary	provable	adversary model	*	adversary type
protected	rbac	state	*	protected resources
role	rbac	(actor, role)	*	assign role to actor
right	rbac	(role, right)	*	assign right to role
secrecy	critical	data	*	secrecy of data
integrity	critical	(variable, expression)	*	integrity of data
authenticity	critical	(data, origin)	*	authenticity of data
high	critical	message	*	high-level message
fresh	critical	data	*	fresh data
adversary	secure links	adversary model	1	adversary type
adversary	data security	adversary model	1	adversary type
integrity	data security	(variable, expression)	*	integrity of data
authenticity	data security	(data, origin)	*	authenticity of data
guard	guarded	object name	1	guard object

Figure 3.2: The UMLsec tags

subsystem $\mathcal{C}$ composed of a class diagram, a deployment diagram, an activity diagram, a statemachine for each of the activities in the activity diagram and a set of offered operations and signals. The object Sender sends the data d to the object Receiver. Since this data should remain secret, a stereotype « critical » is added to the object Sender with the associated tagged value { secrecy = d }. To ensure the secrecy on the physical layer, the link between the node containing the object Sender and the node containing the object Receiver is stereotyped « encrypted ». In general, since the data should travel in a secure way, the stereotype « data security » is attached to the subsystem. This example is very similar, but slightly different from the case of the stereotype « secure links » described in Section 3.1 since here, the constraint is defined by the stereotype « data security ». Note again that this example aims to illustrate how the notation can be used to model security on UML diagrams, but do not show how the security is formally verified.

The example of Figure 3.3 shows the secure channel at a high level of abstraction, since the security is modelled by a link stereotyped « encrypted ». UMLsec allows however to model the encryption more concretely. This is shown in Figure 3.4. In this case, the link is stereotyped « Internet », but the data is explicitly encrypted on the UML diagrams. Verifying the security of this subsystem goes slightly further the simple case presented in Section 3.1. In particular, the preservation of the secrecy is formally defined with the previous knowlegde of the adversary A (of type $default$) $\mathcal{K}_A^{\mathsf{p}}$ and the encryption elements. The proof that the security requirements (in this case the secrecy of the data) are fulfilled is given in [Jür10].

The previous examples illustrated how to use the UMLsec notation to represent security requirements on a subsystem. All the possibilities offered by UMLsec have not be shown in this example, but such a completeness is beyond the scope of this thesis. We do not aim to illustrate the use of all of the stereotypes defined in Figure 3.1. More example of how to use the UMLsec notation can be found in [Jür10].

3.3 UMLsec Abstract Syntax

The semantics of UML, defined in [OMG09], is only described in natural language, which can be ambiguous and thus insufficient when the elements of UML needs to be used formally. It is especially the case when trying to define the execution of the behavioural units or when providing tool support for the UML models. To overcome this limitation and verify security of UML models, UMLsec provides a behavioural semantics given by UML Machine rules as well as formal definitions of security principles that have to be verified, as we mentioned it in Section 3.1. In order to apply such

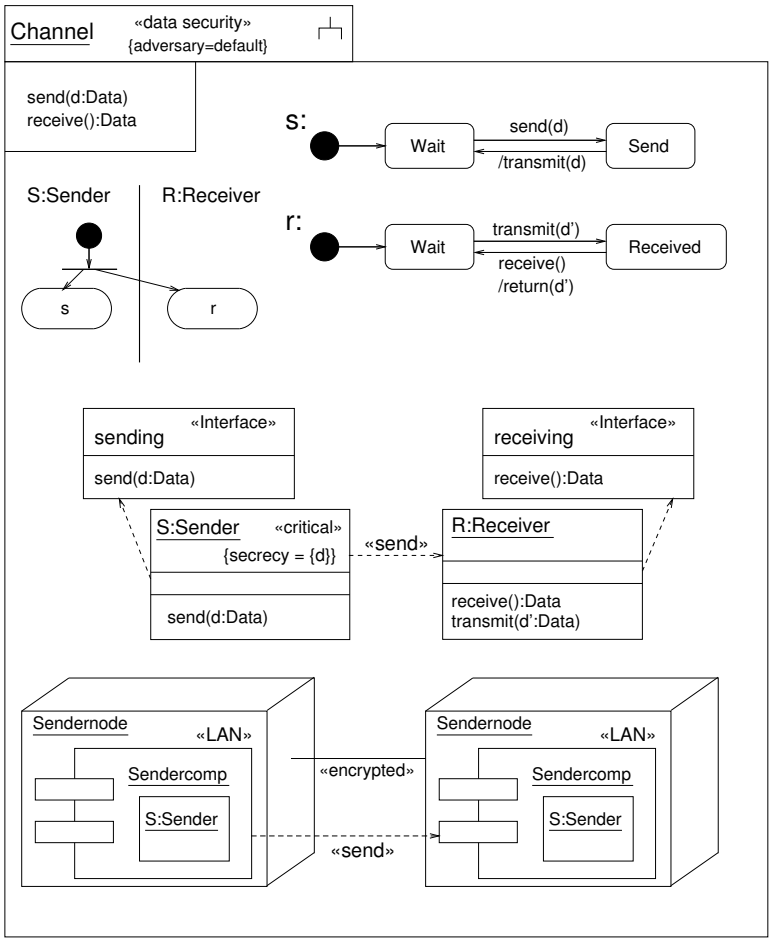

Figure 3.3: Example of a channel

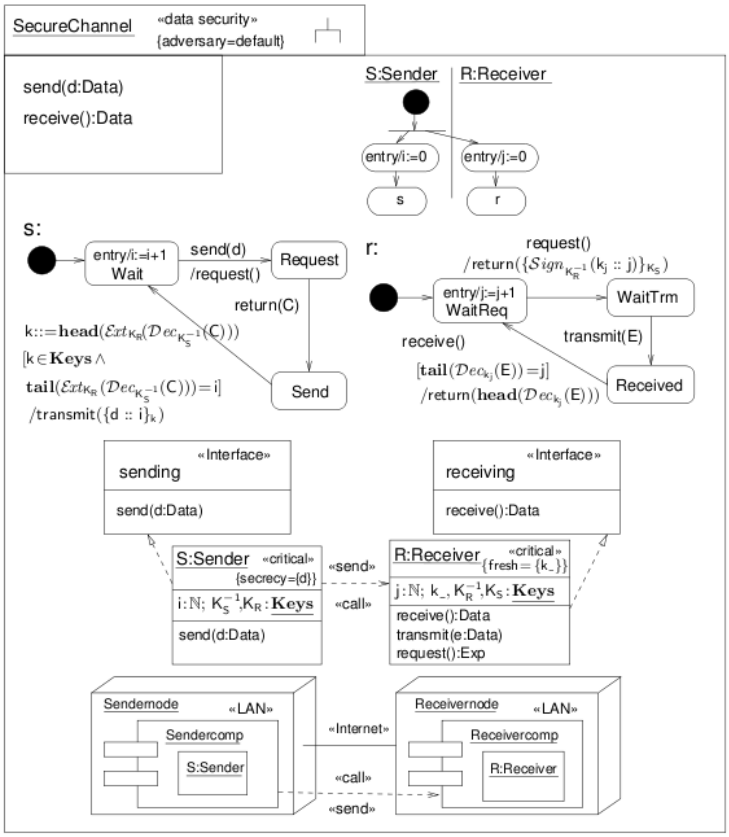

Figure 3.4: Example of a secure channel

a formal behaviour on the UML diagrams, it is also necessary to define an abstract syntax, which gives a means to represent the diagrams formally. In the following, we present the abstract syntax of UMLsec as it is defined in [Jür10]. Again, we do not consider here the complete formal foundation and semantics of UMLsec. In particular, we ignore the formal definitions for the preservation of the security as well as the behavioural semantics given by UML Machines rules to only focus on the UMLsec abstract syntax. This means that certain concepts used in the abstract syntax of UMLsec for the communication of the UML Machines, such as the sets of events or signals, will be ignored. This simplification will not affect our objective to extend UMLsec. The reader should however refer to [Jür10] for a complete description of the UMLsec formal semantics. The abstract syntax described further will be extended in Section 5.4 in order to take the UMLseCh principles and elements into account. It will also provide a means to define the behaviour of the application of changes and the rules that ensure the preservation of consistency. This will also be presented in Section 5.4.

3.3.1 General Concepts

In the abstract syntax defined below, certain elements will be defined as elements belonging to one of the following sets: the classic set of sequences of characters **String**, the set of stereotype definition **StereoNm**, the set of stereotype instances **Steretypes**, and the set of boolean expression **BoolExp**. These sets are simply mentioned here, but they will described in more details in Section 5.1.

The special event $\mathsf{ComplEv}$ is also assumed to be the *completion event*, as defined in [Jür10] and [BCR00]. This concerns the transition of statechart diagrams. More precisely, a transition t has a triggering event, represented by $\mathsf{trigger}(t)$. A transition t with $\mathsf{trigger}(t) = \mathsf{ComplEv}$ is called a *completion transition*. The *completion event* "indicates the completion of the source state" [BCR00], such that no additional event is necessary to trigger the *completion transition*.

3.3.2 Object Diagrams

UMLsec defines the operations of an object as a 3-tuples $O = (oname, args, otype)$, given by:

- an operation name $oname \in$ **String**;
- a set $args$ of arguments of the form $A = (argname, argtype)$ where $argname$ is the argument name and $argtype$ its type; and
- the type $otype$ of the return value.

If the operation has no return value, the type *otype* is the empty type $\emptyset$.

An Object is represented by a 6-tuple $O = (oname, cname, stereo, aspec, ospec, int)$ where:

- $oname \in$ **String** is an object name;
- $cname \in$ **String** is a class name;
- $stereo \subseteq$ **StereoNm** is a set of stereotypes;
- *aspec* is a set of attribute specifications of the form $A = (aname, gtype)$ where $aname \in$ **String** is the attribute name and *gtype* the attribute type;
- *ospec* is a set of operations; and
- *int* is a set of interfaces of the form $I = (iname, ospec)$ where $iname \in$ **String** is the interface name and *ospec* a set of operation specifications.

With the constraint that operations with the same name in different interfaces have the same type.

A dependency is defined as a 5-tuple $d = (dname, dep, indep, int, stereo)$ given by:

- a dependency name *dname*;
- the names *dep* of the dependent and *indep* of the independent class, signifying that *dep* depends on *indep*;
- an interface name *int* (the interface of the class *indep* through which instances of *dep* accesses instances of *indep*; this field contains the name of the independent class if the access is direct); and
- a stereotype $stereo \in \{\text{«call»}, \text{«send»}\}$.

An object diagram is a pair $O = (\mathsf{Objects}(D), \mathsf{Dep}(D))$ given by a set $\mathsf{Objects}(D)$ of objects and a set $\mathsf{Dep}(D)$ of dependencies. Object specifications of a same class has the same class name. In addition, we require that the names of different objects are mutually distinct.

3.3.3 Class Diagrams

Class diagrams are very similar to Object diagrams, with one difference in the definition of a class. Concretely, a class is simply defined as an object, as described above, $C = (oname, cname, stereo, aspec, ospec, int)$ where *oname*

is the empty string.

A class diagram is then 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.

3.3.4 Statechart Diagrams

A state S is given by $s = (\mathsf{name}(S), \mathsf{entry}(S), \mathsf{state}(S), \mathsf{internal}(S), \mathsf{exit}(S))$, where:

- $\mathsf{name}(\mathsf{S}) \in \mathbf{String}$ is the name of the state;
- $\mathsf{entry}(S)$ is the entry action;
- $\mathsf{state}(\mathsf{S})$, is the set of substates of S;
- $\mathsf{internal}(S)$ is the internal activity of the state;
- $\mathsf{exit}(S)$ is the exit action; and
- $\mathsf{stereo} \subseteq \mathbf{Stereotypes}$ is a set of stereotypes.

The set of states of a diagram D is $called\mathsf{State}_D$. It is disjointly partitioned into the sets $\mathsf{Initial}_D$ of initial states in D, Final_D of final states, Simple_D of simple states, Conc_D of concurrent states, and Sequ_D of sequential states. Therefore, we have, for a state S, that $\mathsf{state}(S) \subset \mathsf{State}_D$. A transition is defined as $t = (\mathsf{source}(t), \mathsf{trigger}(t), \mathsf{guard}(t), \mathsf{effect}(t), \mathsf{target}(t))$, given by:

- a state $\mathsf{source}(t) \in \mathsf{States}_D$, the source state of t;
- an event $\mathsf{trigger}(t)$, the triggering event of t;
- a Boolean expression $\mathsf{guard}(t) \in \mathbf{BoolExp}$, the guard of t;
- an action $\mathsf{effect}(t)$, which is performed when firing t; and
- a state $\mathsf{target}(t) \in StatesD$, the target state of t.

The set of transitions of a diagram D is called $\mathsf{Transition}_D$. It has a subset of internal transitions, called $\mathsf{Internal}_D$, such that $\mathsf{Internal}_D \subseteq \mathsf{Transition}_D$. For every state $S \in \mathsf{State}_D$, we have the following conditions:

- We have $\mathsf{Top}_D \in \mathsf{Conc}_D \cup \mathsf{Sequ}_D$;
- For every $S \in \mathsf{Sequ}_D$ there exists exactly one $T \in \mathsf{state}(S) \cap \mathsf{Initial}_D$ (which we write as $\mathsf{init}(S)$);

- $S \in \mathsf{Simple}_D \cup \mathsf{Final}_D \cup \mathsf{Initial}_D$ implies $\mathsf{state}(S) = \emptyset$ and $S \in \mathsf{Conc}_D$ implies that $\mathsf{state}(S)$ has at least cardinality 2;
- $T \in \mathsf{Conc}_D$ and $S \in \mathsf{state}(T)$ implies $S \in \mathsf{Conc}_D \cup \mathsf{Sequ}_D$;
- For all $S, T \in \mathsf{States}_D$, $\mathsf{state}(S) \cap \mathsf{state}(T) = \emptyset$ implies $S = T$;
- For $S \in \mathsf{Initial}_D \cup \mathsf{Final}_D \cup \mathsf{Top_D}$, we have $\mathsf{entry}(S) = nil$, $\mathsf{internal}(S) = Nil$, and $\mathsf{exit}(S) = nil$;
- Let the relation $\prec$ on states $S \in \mathsf{States}_D$ be defined by $S \prec T$ if there exist states $S_1, \cdots, S_n$ with $n \geq 1$ such that $S_1 = S$, $S_n = T$, and $S_i \in \mathsf{state}(S_{i+1})$ for $i < n$. Then $\prec$ is acyclic (in particular irreflexive), and fulfills the condition that for all $S, T, U \in \mathsf{States}_D$ with $S \prec T$ and $S \prec U$ we have $T \prec U$ or $U \prec T$. Top_D is the largest element in States_D with respect to $\prec$;

and for every $t \in \mathsf{Transition}_D$, we have:

- $\mathsf{source}(t) \notin \mathsf{Final}_D \cup \mathsf{Top}_D$ (final states and the top state have no outgoing transitions);
- $\mathsf{target}(t) \notin \mathsf{Initial}_D \cup \mathsf{Top}_D$ (initial states and the top state have no incoming transitions);
- $\mathsf{source}(s) = \mathsf{source}(t) \in \mathsf{Initial}_D$ implies $s = t$ for any $s, t \in \mathsf{Transitions}_D$;
- $\mathsf{source}(t) \in \mathsf{Initial}_D$ implies $\mathsf{trigger}(t) = \mathsf{ComplEv}$ and $\mathsf{guard}(t) \equiv true$ (where $\equiv$ denotes syntactic equality);
- For any $S \in \mathsf{States}_D$, $\mathsf{source}(t) \in \mathsf{state}(S)$ implies $S \in \mathsf{Sequ}_D$ and $\mathsf{target}(t) \in \mathsf{state}(S)$;
- $\mathsf{trigger}(t)$ must be of the form $op(exp_1, \ldots, exp_n)$ where $exp_1, \ldots, exp_n$, called parameters, must be mutually distinct;
- If $t \in \mathsf{Internal}_D$ then $\mathsf{source}(t) = \mathsf{target}(t)$; and
- Multiple completion transitions leaving the same state must have mutually exclusive guard conditions. For $s, t \in \mathsf{Transitions}_D$ such that $\mathsf{source}(s) = \mathsf{source}(t)$ and $\mathsf{trigger}(s) = \mathsf{trigger}(t) = \mathsf{ComplEv}$, the condition $\mathsf{guard}(s) \wedge \mathsf{guard}(t)$ evaluates to false for any variable valuations.

A statechart diagram is defined by $D = (\mathsf{Object}_D, \mathsf{States}_D, \mathsf{Top}_D\ , \mathsf{Transitions}_D)$, 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, and a set $\mathsf{Transitions}_D$ of transitions.

3.3.5 Sequence Diagrams

A lifeline is defined as a pair (O, C), given by an object O of class C. The set of lifelines of a sequence diagram D is called $\mathsf{Obj}(D)$. Lifelines can communicate together with connections. A connection is defined as 4-tuple $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$, given by:

- the source object $\mathsf{source}(l) \in \mathsf{Obj}(D)$ of the connection;
- the guard $\mathsf{guard}(l) \in \mathbf{BoolExp}$ of the connection;
- the message $\mathsf{msg}(l)$ of the connection; and
- the target object $\mathsf{target}(l) \in \mathsf{Obj}(D)$ of the connection.

The set of connections of a sequence diagram D is called $\mathsf{Cncts}(D)$. For each $l \in \mathsf{Cncts}(D)$, we have that $\mathsf{source}(l) \in \mathsf{Obj}(D)$ or $\mathsf{target}(l) \in \mathsf{Obj}(D)$, or both. A sequence diagram is then simply defined as a pair $D = (\mathsf{Obj}(D), \mathsf{Cncts}(D))$.

3.3.6 Activity Diagrams

In UMLsec, activity diagrams are presented as a special type of statechart diagrams. In particular, any construct of the simplified version of activity diagrams can be expressed using the concepts of statechart diagrams. 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 $S \in \mathsf{State}_D$ is given by:

- $\mathsf{name}(S) \in \mathbf{String}$ the name of the state;
- the entry action $\mathsf{entry}(S)$;
- the set $\mathsf{state}(S) \subseteq \mathsf{States}_D$ of substates of S;
- the internal activity $\mathsf{internal}(S)$, also called do-activity of the state;
- the exit action $\mathsf{exit}(S)$; and
- the name $\mathsf{swim}(S)$ of the swimlane containing S.

The set of transitions an activity diagram D is also called $\mathsf{Transitions}_D$, with $\mathsf{Internal}_D \subseteq \mathsf{Transitions}_D$. A transition $t \in \mathsf{Transitions}_D$ is given by:

- the source state $\mathsf{source}(t) \in \mathsf{States}_D$ of t;
- the guard $\mathsf{guard}(t) \in \mathbf{BoolExp}$ of t; and
- the target state $\mathsf{target}(t) \in \mathsf{States}_D$ of t.

The conditions for statechart diagrams, presented in Subsection 3.3.4, also apply for the states and the transitions of activity diagrams.

3.3.7 Deployment Diagrams

A component is defined as a 3-tuple $C = (name, int, cont)$ where *name* is the component name, *int* is a set of interfaces that can possibly be empty and *cont* is the set of subsystem instance and object names contained in the component. A node $N = (loc, comp)$ is then given by:

- the name *loc* of its location, and
- a set of contained components *comp*.

Nodes are connected by links. 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 $\mathsf{ster}(l) \subseteq$ **Stereotypes** is a set of stereotypes. Components can also be "connected" by dependencies. A dependency is formally a 4-tuple $d = (clt, spl, int, stereo)$ where:

- *clt* is the name of the component being the client of the dependency;
- *spl* is the name of the component being the supplier of the dependency;
- *int* is the interface of *spl* accessed by the dependency, if the access is direct, $int = spl$); and
- $stereo \subseteq$ **Stereotypes** is a set of stereotypes.

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.

3.3.8 Subsystems

In UMLsec, "subsystem" means "subsystem instance". A subsystem is defined as a 8-tuple $\mathcal{S} = (\mathsf{name}(\mathcal{S}), \mathsf{Msgs}(\mathcal{S}), \mathsf{Ints}(\mathcal{S}), \mathsf{Ssd}(\mathcal{S}), \mathsf{Dd}(\mathcal{S}), \mathsf{Ad}(\mathcal{S}), \mathsf{Sc}(\mathcal{S}), \mathsf{Sd}(\mathcal{S}))$, given by:

- the name $\mathsf{name}(\mathcal{S})$ of the subsystem;
- a set $\mathsf{Msgs}(\mathcal{S})$ 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 $\mathsf{Ad}(\mathcal{S})$; and

- 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.

Note that UMLsec also offer the possibility to have UML Machine rules to specify the behavior of objects appearing in $\mathsf{Ssd}(\mathcal{S})$. We do not present this possibility here since we do not take the UML Machines into account. A static structure diagram $D = (\mathsf{SuSys}(D), \mathsf{Dep}(D))$ is given by a set $\mathsf{SuSys}(D)$ consisting of objects or subsystem instances, and a set $\mathsf{Dep}(D)$ of dependencies $(dep, indep, int, stereo)$ where the difference from the dependencies defined for object diagrams is that *dep* and *indep* can also be subsystems. The names of the subsystems or objects also have to be mutually distinct.

The following constraints also have to be fulfilled to ensure consistency of subsystems. For each activity a of an activity diagram, such that a is in the swimlane O, there exists either a subsystem $\mathcal{S} \in \mathsf{SuSys}(\mathsf{Ssd}(\mathcal{S}))$ such that $\mathsf{name}(\mathcal{S}) = a$, or a statemachine $D \in \mathsf{Sc}(D)$ such that $O = \mathsf{Object}_D$ and $\mathsf{context}(D) = a$, or a sequence diagram $D \in \mathsf{Sd}(\mathcal{S})$ with $O \in \mathsf{Obj}(D)$ and $\mathsf{contex}(D) = a$ and such that $a = D.O$.

Chapter 4

Modelling Evolution with UMLseCh

4.1 Preliminary remarks

4.1.1 Applicable Subset of UML

UML is a language that covers a large amount of domains and concepts, modelling both static and dynamic contexts. Such a coverage requires a lot of constructs and diagrams. Among them, we can mention the activity diagrams, the class diagrams, the communication diagrams, the component diagrams, the composite Structure diagrams, the interaction overview diagrams, the package diagrams, the profile diagrams, the statemachine diagrams, the sequence diagrams, the timing diagrams or the use case diagrams. UMLseCh will follow the same principle as UMLsec: the profile concerns all of UML but the behavioural semantics will be limited to a subset of UML. This means that theoretically, the language defined here can be used with any diagram of UML. However, for the formal foundations that will describe the behaviour adopted when applying a change, we will use a restricted part of UML (i.e. the same subset as the one used for the UMLsec behavioural semantics). This subset includes simplified versions of Class and Object diagrams, Statechart diagrams, Sequence diagrams, Activity diagrams, deployment diagrams and Subsystems.

Theoretically, it could be possible to define a formal semantics specifying the behaviour of a change for any diagram of UML. However, the full UML specifications having an important quantity of constructs and diagrams, such a completeness is beyond the scope of this work. In addition, UMLseCh has not been developed uniquely as a language for model transformation, but also as a language that verifies the preservation of the security over the evolution. For this, UMLseCh includes the stereotypes, tag definitions and

constraints of UMLsec. As mentioned in Chapter 3, UMLsec uses a subset of UML for its formal semantics and tool support. Defining a formal semantics and behaviour for diagrams outside of this subset would thus only allow one to run the application of a change modelled by the UMLseCh notation, but not to verify, using the UMLsec tool, whether the security of the system modelled on that diagram is preserved when the change occurs. Hence the UMLseCh formal semantics and abstract syntax will focus on this reduced part of UML. Note again that this is especially important for the tool support. The notation can be used outside of the subset to model changes in general but no formal behaviour will be given for applying these changes. It leaves room for users to apply UMLseCh on any diagram and to define their own semantics for the parts of UML not covered in the following. One could also extend the tool so that it covers a broader user-defined semantics. Alternatively, UMLseCh could also be used intuitively for the parts not covered by the formal foundations. Again, this would be limited to a modelling scope since no tool support could be provided for this part. The choice to either use the notation intuitively or to extend the formal semantics is left to the reader. In the following, the descriptions and examples will remain within the simplified part of UML defined in this section.

4.1.2 UML Namespaces

As mentioned above, UMLseCh is based on the lightweight extension mechanism of UML, which uses string-based notation. However, such sequences of characters are insufficient to graphically represent model transformation in a complete way. Indeed, most of the UML elements commonly used in diagrams, such as Classes, Lifelines or Components, are represented with a graphical notation, which cannot be shown in a tagged value. Alternatively, they could be represented by defining a syntax only based on strings, but this solution would conisderably reduce the readibility of the notation. Hence to define a language that allows complete graphical transformation using only stereotypes, tagged values and UML-compliant notation, we will use *namespaces*.

A *namespace* is "an element in a model that contains a set of *named elements* that can be identified by name" [OMG09]. It is represented in [OMG09] by the construct Namespace and is itself a *named element*, which provides the possibility to define hierarchies. The notation of a *namespace* is the name itself and the notation for hierarchies is defined by a double colon. For example, a *named element* n contained in a *namespace* m_2, itself contained in a *namespace* m_1, will be written $m_1 :: m_2 :: n$. Using *namespaces* to store UML elements evidently requires to follow the UML specifications for *namespaces*. One of the consequences is that only certain model elements can be contained in a *namespace*, while some others are not

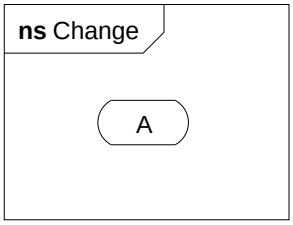

Figure 4.1: Example of namespace

allowed. More specifically, only *named elements*, defined in [OMG09] by the construct NamedElement, can be owned by a *namespace*. It actually represents a very large part of the UML elements and includes all of the elements of our simplified version, described in the previous section. The restriction for *namespaces* to own only *named elements* thus will not affect our notation.

UML does not provide a specific notation for *namespaces*, since concrete subclasses of the abstract metaclass Namespace will define their own specific notation. Theses subclasses can be commonly used model elements, such as Classes, Nodes, Statemachines or Packages, which indeed have their own graphical notation. To fulfill our need of a UML elements container, we will define an abstract graphical representation for *namespaces*. A *namespace* is represented by a rectangle with a hexagon at the top left, which contains the name of the *namespace* preceded by the word "**ns**". Hierarchical representation can be used in the name of a *namespace*. Note that this representation is abstract, since *namespaces* can have their own notation at the concrete level. This means that a classical UML tool case does not provide such a notation "out of the box" and the concrete representation has to be used. Depending on the context, the concrete representation could be a *package*, a *statemachine* or a *interaction* for example. Figure 4.1 shows an example of a *namespace* called "*Change*" and containing a *state* called "A". Again, this *state* will be contained in a statemachine at the concrete level and in a tool[1]. Following the UML specifications, the text notation that can be used to represent the *state* A is: *change* :: A. This *namespace* can be used in a change stereotype to reference the container of a complex substitutive or additive model element. This will be detailed in Sections 4.2.3 and 4.2.4.

Elements that are not *named elements* will not be covered by the UMLseCh notation. The reader should consult the UML specifications to find out whether a model element is a *named element* and thus can be stored in a

[1]Note that a state machine should contain at least one initial state to be executable. The UML specifications and the model of statemachines given in those specifications, however, allow statemachines to have no initial state [OMG09].

namespace. If it is not the case, the element will not be usable with the UMLseCh notation. Again, the possibility to extend the formal semantics and the tool support to cover other diagrams and complex model elements, and thus to define a behaviour for model elements that are not NamedElement, is left to the reader.

4.2 The UMLseCh Extension

4.2.1 The Profile

As it is specified in the Catalog of UML Profile Specifications [Pro], 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 4.2.2, 4.2.5 and 5.4, define the UMLseCh profile, following the structure described above.

The UMLseCh profile concerns all of UML. Figure 4.2 shows the list of stereotypes, together with their tags and constraints. These stereotypes do not have parents. Figure 4.3 shows the corresponding tags. The tag ref is a DataTag and the tags substitute, add and delete are all ReferenceTags.
It is actually difficult to determine if theses three tags are DataTags or ReferenceTags. Indeed, as it will be described in the following sections, the UMLseCh tagged values associated to these three tags are model elements, but those model elements do not exist on 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 inside a change stereotype[2]. 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

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

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

Figure 4.2: UMLseCh stereotypes

containing model elements are considered as a 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. Figure 4.2 and Figure 4.3 both follow the notation used in [Jür10] for the UMLsec profile definition[3]. 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. The formal meaning of the stereotypes as well as the formal behaviour of the changes that they model are given in Chapter 5. Although UMLseCh could be used alone as an evolution modelling 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. The diagram representing the UMLseCh profile is shown in Figure 4.4.

[3]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 [OMG09].

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

Figure 4.3: UMLseCh tags

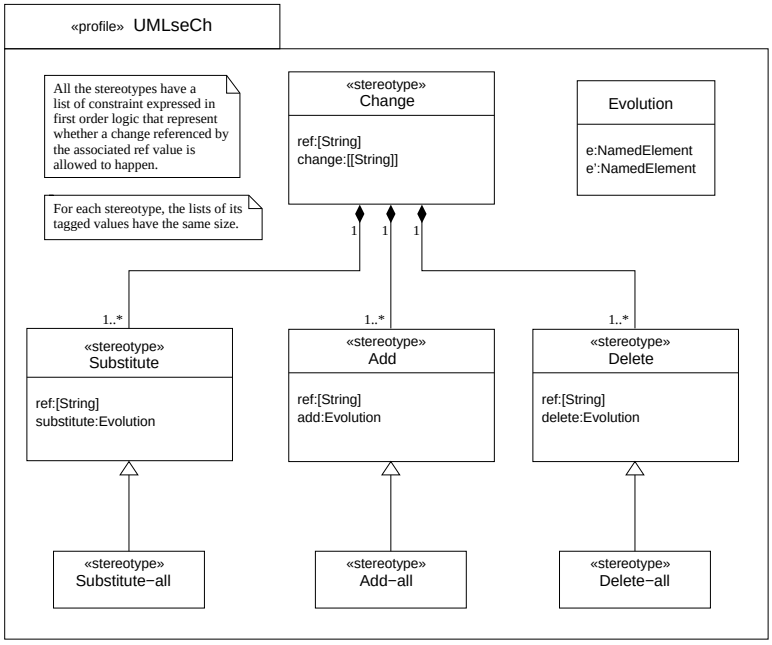

Figure 4.4: UMLseCh Profile

4.2.2 Description of the Notation

« substitute »

The stereotype « substitute » attached to a model element denotes the possibility for that model element to evolve over time and defines what are the possible changes. 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 modelled 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 takes 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 occurence of the same e in the list is allowed[4]. However, two occurences of the same pair (e, e') cannot exist in the list, since it would modelled 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 behaviour that will be described in Section 4.2.3. 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 modelled 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

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

the stereotype is attached, the element e is necessary[5]. 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 of UMLseCh, defined in Section 5.4. 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 a inconsistent syntactic state. However, it does not stop one from modelling 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 further in Chapter 5. 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 4.5.

The stereotype « substitute » also has a list of optional constraints formulated in first order logic. This list of constraints is written between square brackets and is of the form $[(\mathsf{ref}_1, \mathsf{CONDITION}_1), \ldots, (\mathsf{ref}_n, \mathsf{CONDITION}_n)]$, $\forall n \in \mathbb{N}$, where, $\forall i : 1 \leq i \leq n$, ref_i is a value of the list of a tag ref and $\mathsf{CONDITION}_n$ can be any type of first order logic expression, such as $A \wedge B$,

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

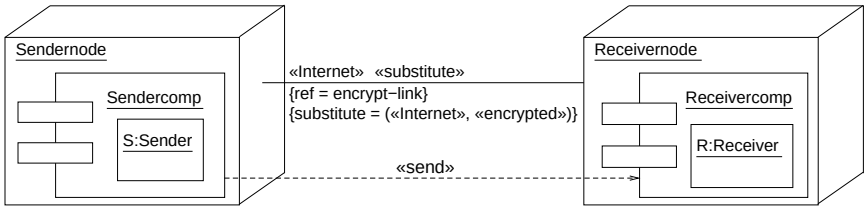

Figure 4.5: Example of stereotype « substitute »

$A \vee B$, $A \wedge (B \vee \neg C)$, $(A \wedge B) \Rightarrow C$, $\forall x \in N.P(x)$, 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 occured, false otherwise. Formally, the predicate should be *"x occured"* or $P(x)$, assuming that $P(x)$ is the predicate *"x occured"* and where x 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 occured"*[6]. 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 4.6. To express that two changes, referenced respectively by change_1 and change_2, have to occur first in order to allow the

[6]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 *occured"* will be defined.

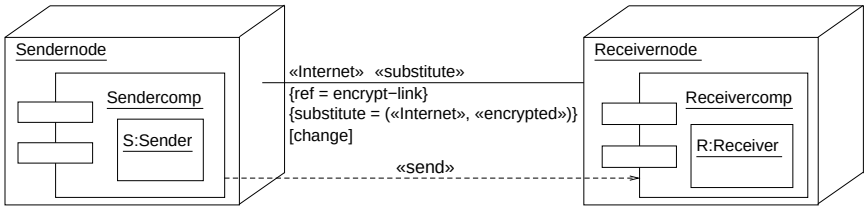

Figure 4.6: Example of a constraint of a stereotype

change referenced encrypt-link to happen, the constraint:

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

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

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

is added to the stereotype. Complete examples will be given in Section 4.3.

« 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 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 behaviour that will be described in Section 4.2.4. 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 consists 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 Chapter 5.

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 ».

« 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 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 ».

« substitute-all »

The stereotype « substitute-all » is an extention 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 associ-

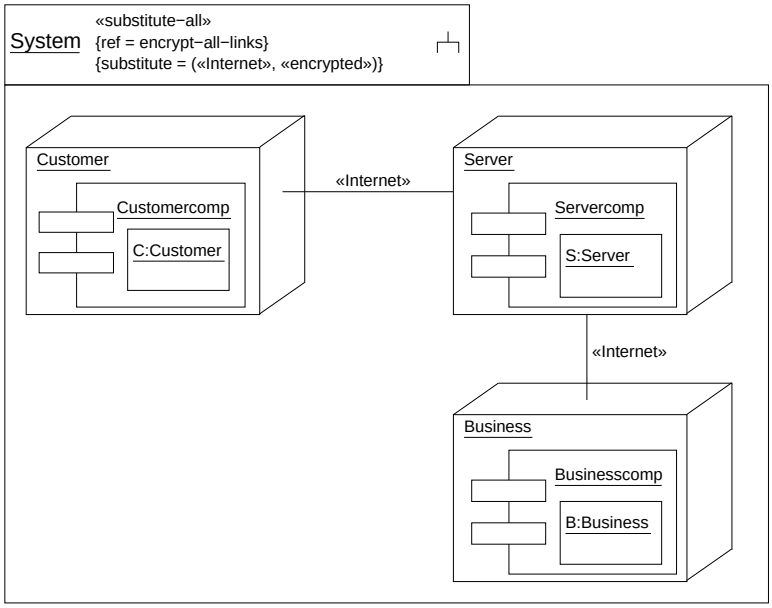

Figure 4.7: Example of stereotype « substitute-all »

ated 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 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 caracteristics. 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 4.7.

A pair (e, e') of the list of values of a tag substitute here allow a parametrisation 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 use in the value of both the elements of a pair. The following example illustrates the use of the parametrisation 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 ».

« 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 ».

« delete-all »

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

« 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[7]. Each list thus represents the list of *sub-changes* of a *composite change* modelled by the stereotype « change ». Applying a change modelled by « change » hence consists in applying all of the concerned *sub-changes* **in parallel**.

[7]By type change, we mean the type that includes « substitute », « add », « delete » and « change ». Not only « change ».

Any change being a *sub-change* of a change modelled by « change » **must** have the value of the tag ref of that change in its condition. Therefore, any change modelled by a *sub-change* can only happen if the change modelled 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 ensure that *sub-changes* cannot be applied by themselves, independently from their *super-stereotype* « change » modelling the *composite change*.

An example of the use of a stereotype « change » will be given in Section 4.2.4 where the use of complex additive elements will be described.

4.2.3 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 character (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 4.1.2. 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, that will be detailed by means of examples further in this section.

To define the behaviour of a complex substitution, we need to differentiate two types of model elements. The first type includes the model elements that *connects* 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 modelled 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. Modelling 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 indentified 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.

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 modelled 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 modelling of this change as well as its application are shown in Figure 4.8. 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 [OMG09]. 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 names-

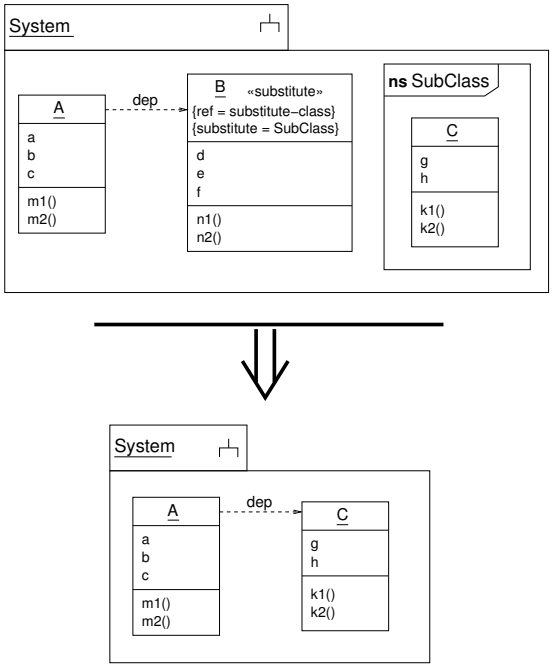

Figure 4.8: Model the possible substitution of a Class

paces 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 4.9. 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[8]. Only a restricted amount of situation will allow this way of modelling the change of a *connector*. 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. It might also be impossible for statemachines or activity diagram.

[8]With ArgoUML, this example can be modelled 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

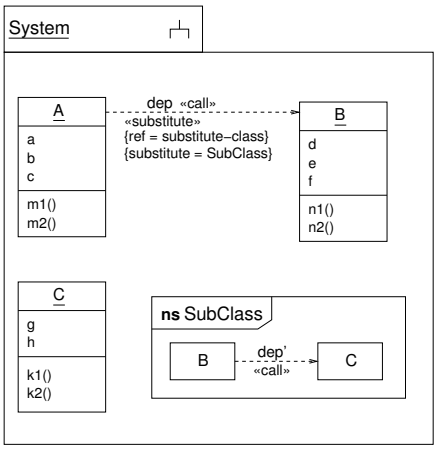

Figure 4.9: Model the possible substitution of a dependency

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*[9]. An example of this is shown in Figure 4.12 of Section 4.2.4, where states are placed in a package, although, for different reasons, this example models a wrong addition. Because of the requirements defined above, modelling possible changes with connectors as substitutive model elements will thus be avoided, unless the situation together with the tool that is used allows it in an easy way. 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 4.2.2. Addition and deletion will be discussed in the following sections. Alternatively, one can also use an expression based on the abstract syntax of transitions, presented in Chapter 3 and extended in Chapter 5, 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 readibility will be slightly reduced but no complex element and thus namespace will be required. Considering the example of Figure 4.9, the value of e' could be replaced by the following expression:

$$d = (\text{"}dep'\text{"}, A, B, B, \langle\langle \mathsf{call} \rangle\rangle) \quad d \in \mathsf{Dep}(CD),$$

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

[9] 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.

4.2.4 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 behaviour 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 for *connectors* and dynamic behaviour diagrams. As mentioned in Section 4.2.2, 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 a inconsistent state. This category of model elements includes *connectors* of all types of diagrams and any model element from dynamic behaviour diagrams, with one exception described below. In certain cases, we can ensure the integration of the new added model elements with a special behaviour 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 4.2.5.

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, modelling 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* behaviour. The *merge*, as its name indicates, adds the new model elements and *merge* the parts of the additive model elements that are already present on the model. This behaviour 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 modelled by placing two states, B

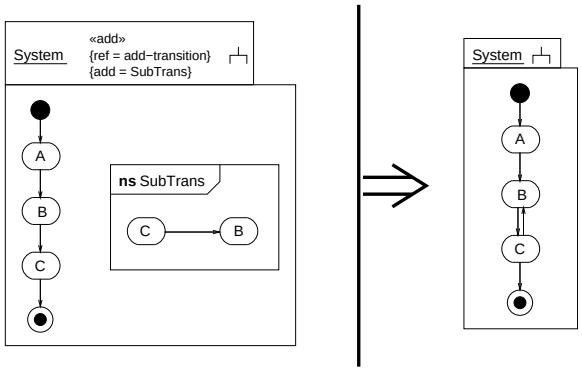

Figure 4.10: Possible addition of a new transition using the *merge*

and C, and a transition from C to B in a namespace[10]. The name of this namespace is then used as the element e' of the pair (e, e') representing the relevant addition in the tag `add`[11]. This model as well as the result of applying the change is shown in Figure 4.10. Note that if final states are placed in the namespace containing the additive elements, they will not be merge 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 behaviour 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 consists in adding the *lifeline* contained in the *namespace* to the model. An example of such a change is shown in Figure 4.11, where a *lifeline*, named C, could possibly be added, if the change occurs, to a Sequence diagram containing two *lifelines*, A and B.

4.2.5 Problems with Stereotypes «add» and «delete»

As mentioned in Section 4.2.2, adding or deleting a model element might generate problems or difficulties that do not exist with a substitution. This

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

[11] This principle could also be used with subsitutions. However, even if it could be useful in certain cases, it is not indispensable. For simplicity reasons, it will not be defined in this version of UMLseCh.

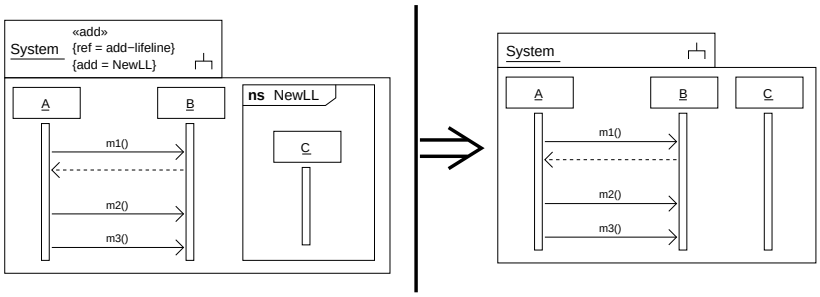

Figure 4.11: Example of an addition of a *lifeline* in a Sequence diagram

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 illustrate 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 the formal foundations of the notation, in Chapter 5.

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 modelled by this stereotype would lead to the situation shown in Figure 4.12, 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 model 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

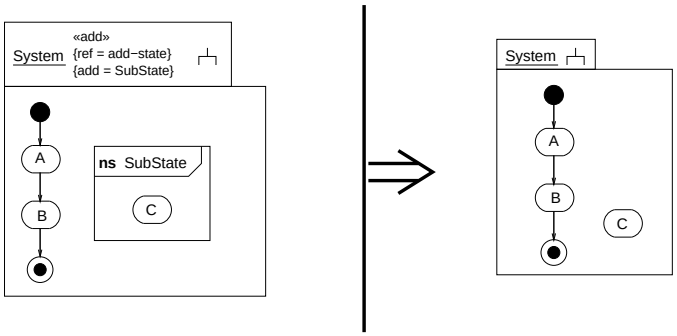

Figure 4.12: Inconsistent diagram resulting from an inappropriate addition

wrong modelling 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 4.13. Following the semantics of the stereotype « substitute » in the case of complex substitutive elements, presented in Section 4.2.3, 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 represents a sub-diagram. The result obtained after the substitution and the result initially intended are equivalent, since when the composite 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[12]. Other possibilities exist.

Another solution would be to use the *merge* operation. However, as explained in Section 4.2.4, 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 modelled by a stereotype « change »

[12]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 solutin is however not allowed by the actual version of UMLseCh.

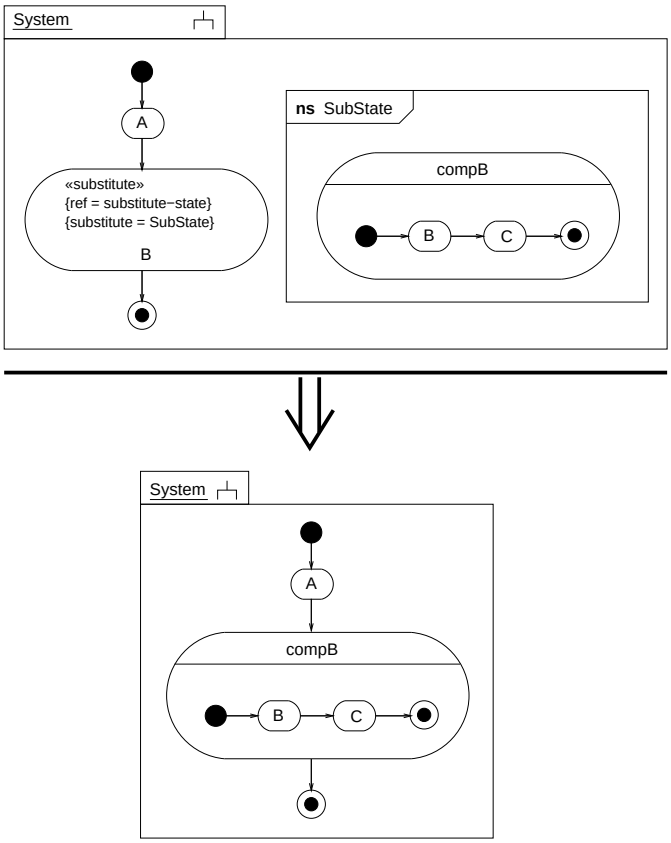

Figure 4.13: A solution to the problem of Figure 4.12

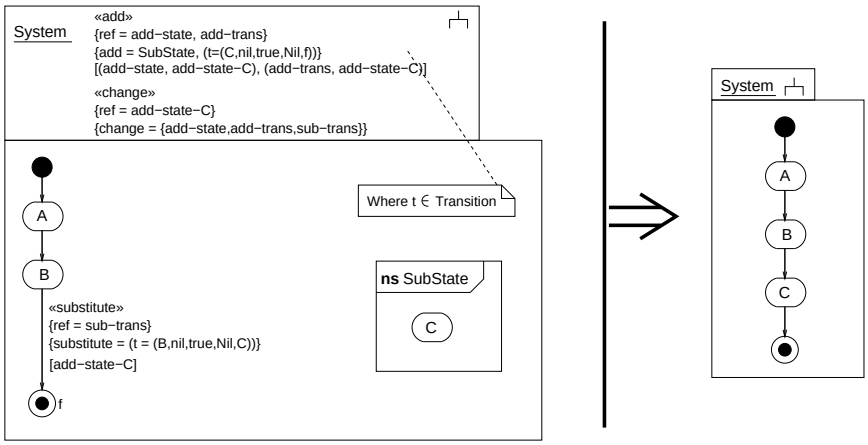

Figure 4.14: Another solution to the problem of Figure 4.12

representing composite changes.

Finally, another solution would be to model the change with three stereotypes, each of them modelling one of the three changes necesseray 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 modelled by using a composite change, as shown in Figure 4.14. 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 modelled 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 5.4.

The case of « delete »

Applying a change modelled 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 4.15. No extra semantics have been defined for the behaviour of the application of a change modelled by a stereotype « delete ». Applying a deletion modelled by a stereotype « delete » on a model element will thus not be allowed if it does not fulfill the constraint defined in Section 5.4, which is the case of the stereotype « delete » of Figure 4.15.

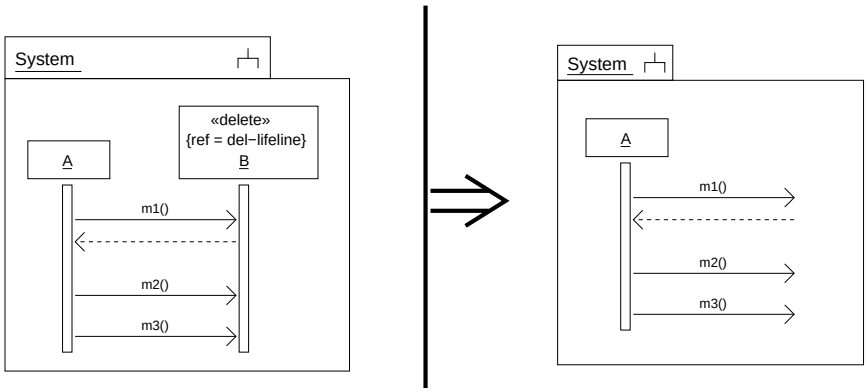

Figure 4.15: Incorrect diagram resulting from an unallowed « delete »

4.3 Examples of Use of the Notation

4.3.1 First Simple Example

As a first example, we give the following simple scenario. A sender sends data to a receiver. The link between the sender and the receiver has the stereotype « Internet ». Therefore, the current design does not provide security, since the stereotype « Internet » is not sufficient to ensure any of the main security requirements (i.e. secrecy, authenticity, integrity and freshness). However, at the modelling stage of the development, the possibility to evolve towards a secure environment is already considered[13]. The security requirement that could be added is the preservation of the sender's data secrecy. This can be modelled by having a « critical » stereotype with the tagged value { secrecy = d } on the class diagram, where d represents the sender's data, and by encrypting the link between the sender and the receiver. This possible evolution can be modelled by adding a stereotype « add » with the tagged values { ref = make-data-secret } and { add = « critical » { secrecy = d } } on the current diagram. Note that although the value of the tag add might look like two model elements, it actually represents only one element, i.e. the stereotype with its associated tagged values. The element in the tag add can be shown alone and not in the form of a pair, since the stereotype « substitute » is attached to the class to which the stereotype « critical » should be added.

To ensure data secrecy, as mentioned above, the stereotype « critical » with the tagged value { secrecy = d } is not sufficient. The link has to be encrypted. Therefore, one also has to change the stereotype « Internet » of the Deployment diagram so that it is now stereotyped « encrypted ». Again,

[13]The reason why the secured design is not applied directly could be, for example, because of budget restrictions, or because the technology is awaited, but not ready yet.

this change can be modelled by adding the stereotype « substitute » with the tagged values { ref = make-link-secure } and { substitute = (« encrypted », « Internet ») } on the link stereotyped « Internet ». Because the encryption of the link is compulsory to ensure the data secrecy, the change make-link-secure should happen first to allow the change make-data-secret to take place. Indeed, applying the addition of the stereotype critical without having an encrypted link would result to a model that is correct syntactically, but incorrect according to the UMLsecsemantics. To model this constraint, the following condition can be attached to the stereotype labelled make-data-secret:

[make-link-secure].

With this condition, the change make-date-secret can be applied only if the link is already encrypted, which ensures the data secrecy requirements. Note that the change make-link-secure can happen although the addtition of « critical » does not take place. Indeed, encrypting a link in an environment where no security requirements are defined does not affect the security[14]. The diagram of this example is shown in Figure 4.16. Note that we ensured the preservation of the security manually in this example by forbidding the change make-date-secret to occur before encrypting the links. UMLseCh aimed to verify such requirements. Thus if the condition was not [make-link-secure] was not added to the stereotype « add », a tool implementing UMLseCh should warn that the security will not be preserved over that evolution.

In this example, there are two stereotypes representing possible changes that could occur on our model. Therefore, there are several possible transitions, each resulting from the application of one of the changes modelled by the stereotypes. One could change the model by adding the stereotype « critical » on the sender class, i.e. apply the stereotype « add » with reference { ref = make-data-secret }. However, this transition is not allowed since it violates the constraint

[make-link-secure].

Another transition could be to substitute the stereotype « Internet » with the stereotype « encrypted ». This transition is correct. Indeed, one could certainly decide to encrypt the link although the secrecy of the data is not requested. Finally, one could apply both changes on the model. This transistion is, of course, also allowed. The model resulting from the application of this particular evolution is shown on Figure 4.17.

[14]But might require unecessary ressources!

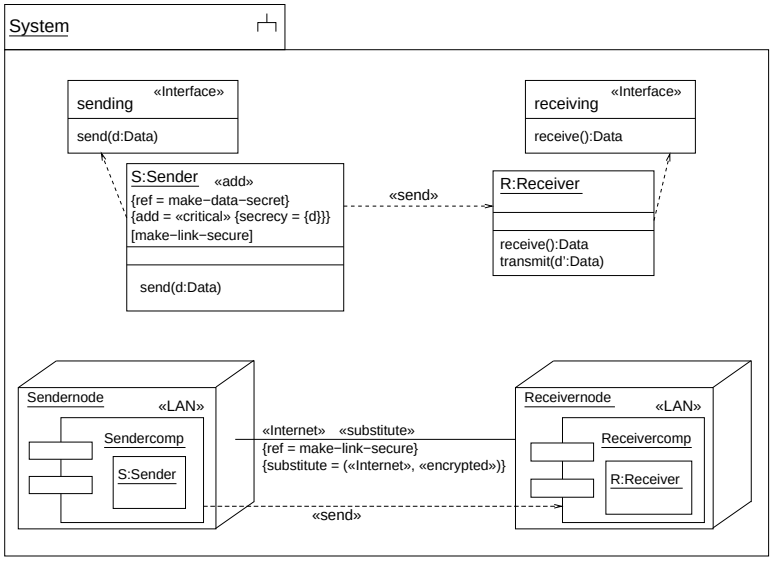

Figure 4.16: First example of evolution

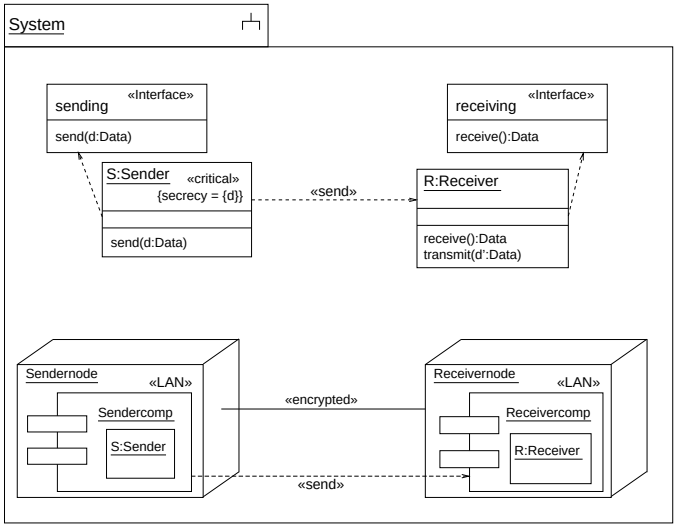

Figure 4.17: One application of the modelled possible changes

4.3.2 Booking a Flight!

For the second example, we use the following scenario. An airline decides to offer the possibility for its customers to book a flight online. Online check-in is also considered, but because of budget restrictions, this feature has to be postponed to future evolution. To model such a system, we define a Subsystem with an object diagram and an activity diagram. The object diagram has two objects, one representing the ticketing system and one representing the customer. To book a flight, the customer's system will call some of the operations offered by the ticketing system. There is thus a dependency with the stereotypes « call » between the customer and the airline system. The process of booking a ticket requires the customer's personal information to travel through the system. This personal information must remain secret, therefore, a stereotype « critical » with an associated tag secrecy is attached to the object representing the client. The other attributes and methods necessary to model the environment will also be present on the objects of the diagram. The activity diagram describes the workflow of a flight reservation. Since the future addition of an online check-in is already considered, this possible evolution can be modelled with UMLseCh stereotypes on both diagrams.

To allow check-in, the ticketing system would need an extra operation that we call checkIn. To model this possible addition, we thus add the stereotype « add » with the tagged values { ref = add-checkIn-operation } and { add = checkIn (p:PersonalData, b:BookingData):Boolean } on the object T:Ticketing. Note that we use the syntactic notation defining operations in the value of the tag add, so that it cannot be confused with other model elements that could also be added or changed, such as attributes or stereotypes. Note also that the stereotype « add » being attached to the object T:Ticketing and the model element to add being an operation, the place where the model element should be added is implicit and therefore the the element in the tag add can be shown alone instead of in the form of a pair. Since the object C:Client accesses the ticketing system through its interface, we also add the operation in the interface. Of course, we ensure that the operation is added to the class first.

This new check-in functionality also changes the workflow of booking a flight. Therefore, it is necessary to update the activity diagram to model the possibility of using the check-in functionality during the process of a flight reservation. In this particular situation, we need to add an optional online check-in on the workflow, which means new actions, decision nodes and activity edges. These new model elements can be put together in order to form a partial activity diagram that represents the check-in action. By partial, we mean that it is not a full activity diagram, but just a restricted

part that contains the relevant model elements. Concretely, this partial diagram contains the additive model elements requested to add the check-in functionality. This represent a complex additive element, thus following the behaviour described in Section 4.2.4, these model elements are stored in a namespace, called CheckIn. To easily integrate the additive model elements to the rest of the activity diagram, some elements of the initial model will be included to the namespace CheckIn. If the addition happens, these elements will hence be merged with the existing ones, as described in Section 4.2.4. However, the final node, following the node Book on the workflow, as well as the transition between the node Book and the final node, will remain after the application of change, although they should disappear. A stereotype « delete » is thus attached to both of them. Applying the deletion first is not allowed, because it would leave the diagram in an inconsistent state and applying the addition first will not ensure that the two stereotypes « delete » will be applied after. The three changes thus have to take place simultaneously. This is modelled by a composite change. The following stereotypes will thus be applied on the activity diagram. A stereotype « add » with the tagged values { ref = add-checkIn } and { add = CheckIn } will be attached to the subsystem, a stereotype « delete » with the tagged value { ref = delete-final-transition } will be attached to the transition between the activity node Book and the final activity node, and a stereotype « delete » with the tagged value { ref = delete-final-node } will be attached to the final activity node. Since the changes modelled by these stereotypes must happen in parallel, a composite stereotype « change » with the tagged values { ref = add-online-checkIn } and { change = add-checkIn, delete-final-transition, delete-final-node } will be attached to the subsystem.

To have Check-in node on the activity diagram, which means a check-in action in the workflow, it is necessary to have a check-in operation in the class diagram. Therefore, the change add-online-checkIn can happen only if the operation added by the change has happened. To model this constraint, the condition [add-check-in-method] is added to the stereotype with reference add-online-checkIn. The subsystem modelling the ticketing system is shown in Figure 4.18. Note that in this example, the condition does not ensure a security concern.

In this example, a few transitions are possible for the subsystem. We consider the one applying all the changes. This transition consists in adding the operation on the class diagram, then on adding the check-in action on the activity diagram. Indeed, the other order is not allowed by the constraint [add-check-in-method]. Figure 4.19 shows the model resulting from the application of the changes modelled by the UMLseCh stereotypes of the subsystem.

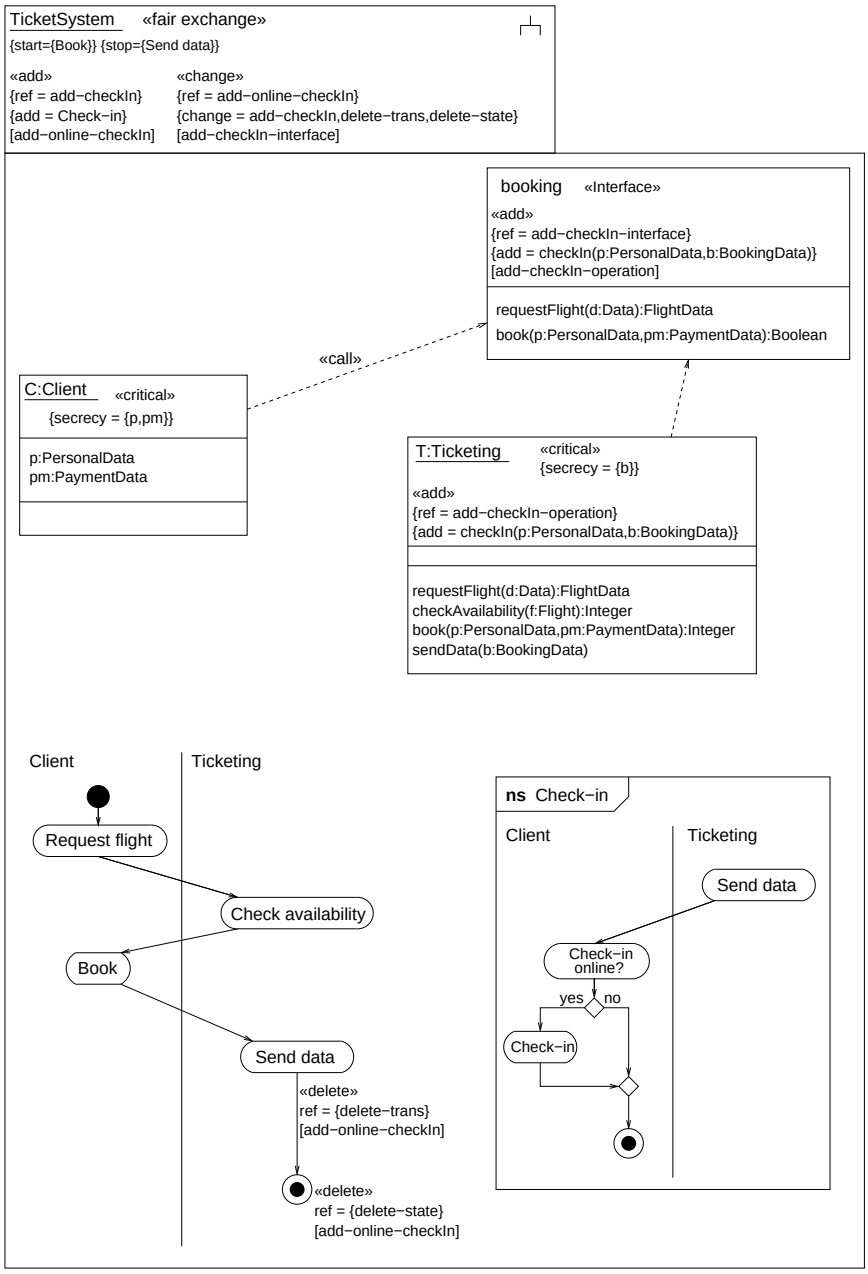

Figure 4.18: Subsystem for the online ticket reservation service

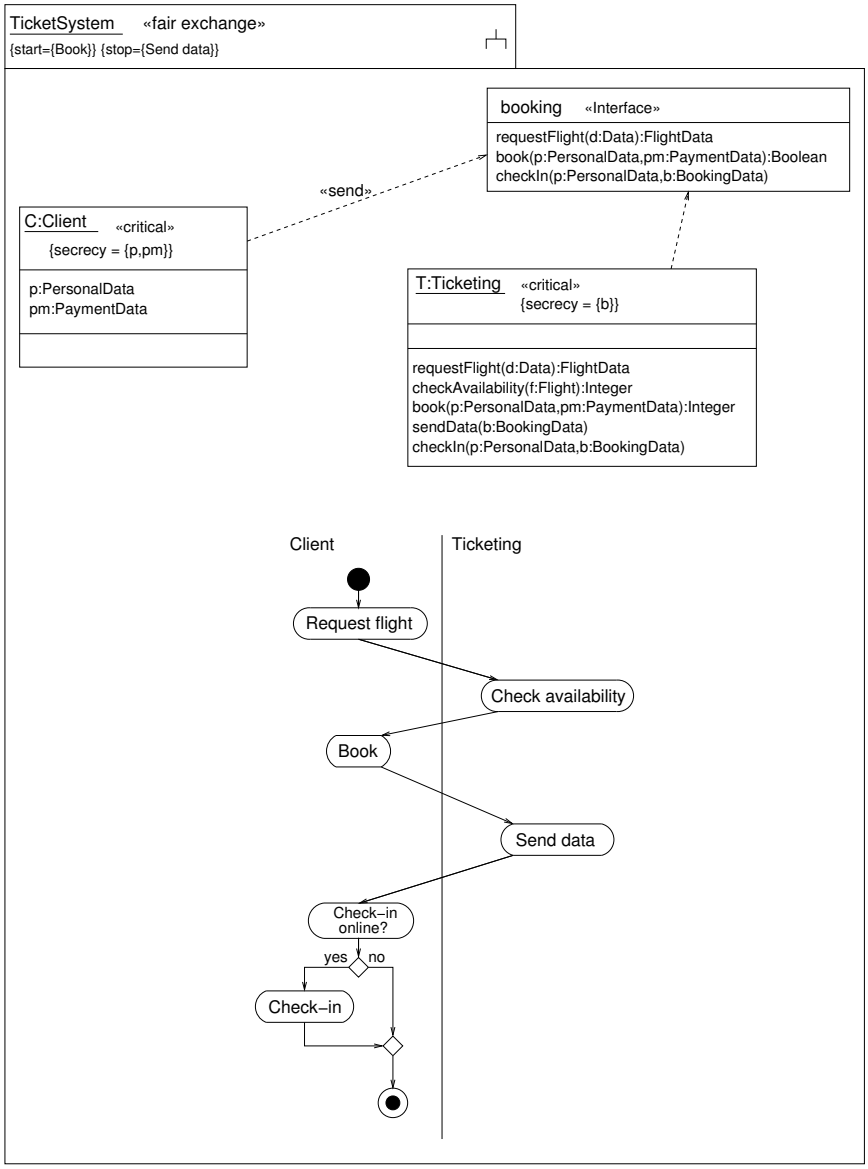

Figure 4.19: Application of the change modelled on the online ticket system

4.3.3 Generalisation - add-all, substitute-all

In this example, we show an application of the stereotypes « add-all » and « substitute-all ». The scenario is similar to the one of the first example, described in Section 4.3.1: a sender sends data to a receiver. However, here, the sender does not know the location of the receiver and therefore sends the data to a server. The server then sends the data to the intended receiver. The server may also provide extra services (e.g. translate the data, add additional information, etc). In consequence, the data sent by the server may be different from the data sent by the sender. Our system already provides integrity for the data. Nevertheless, as for the first example, the possibility to include additional security requirements is already considered at the modelling stage of the development. In this case, the possible evolution is the following: in addition to preservation of the integrity, the system should also ensure data secrecy. To model this possible addition, one thus can add a tag { secrecy = X } on each class having a stereotype « critical » with associated tagged value { integrity = X }, where X is the meta-variable representing the data. However, instead of adding a stereotype « add » with the related tagged values on each class having a stereotype « critical » and a tagged value { integity = X }, we add a stereotype « add-all » on the subsystem with the tagged values { ref = make-data-secret } and { add = ({ secrecy = X }, { integrity = X }) }, which represents the model elements to add and model elements are concerned respectively.

As explained in the example of Section 4.3.1, the stereotype « critical » with the tag { secrecy = X } is not sufficient to ensure data secrecy. The link has to be encrypted. Again, the possibility to encrypt all the links of the model in the future can be modelled by using the stereotype « substitute-all » with the tagged values { ref = make-link-secure }, and { substitute = (« Internet », « encrypted ») }. For the same reason as the one in the first example, the following constraint, is attached to the stereotype with reference { ref = make-critical }:

[make-link-secure].

On the other hand, encrypting all the links although the change make-critical has not happened is allowed, since it does not affect the security of the system. The diagram of this example and the result of applying the changes modelled by both stereotypes are shown respectively in Figure 4.20 and Figure 4.21.

4.3.4 Selection of links - substitute-all

This example aims to show that the selection of the model elements on which the changes should apply can be define precisely. To show this, we

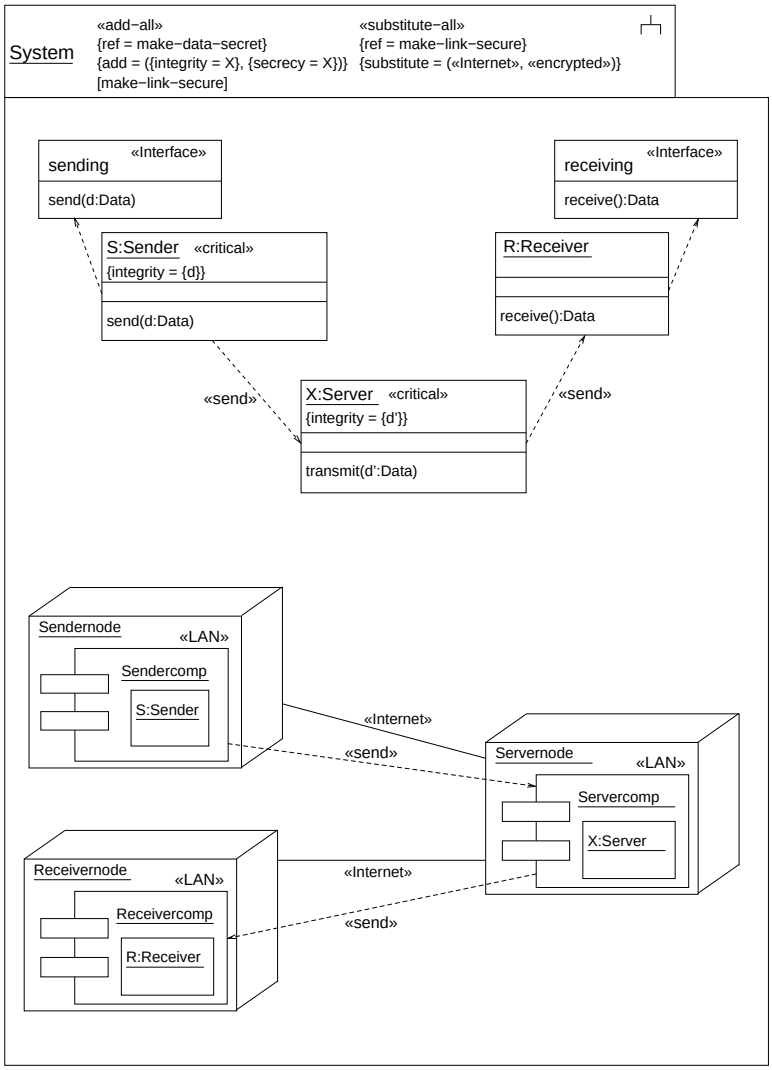

Figure 4.20: Example of use of « add-all » and « substitute-all »

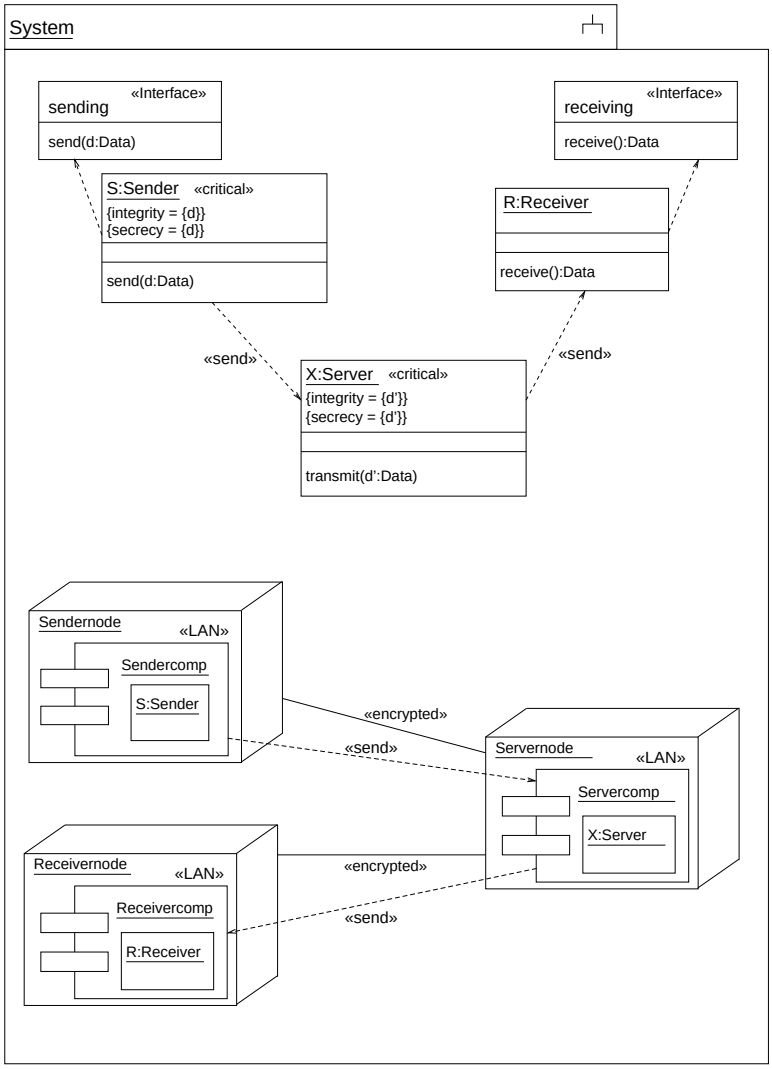

Figure 4.21: Application of the changes

give the following scenario. A client can exchange data with a server and another client, called *secured client* can exchange secret data with a secured server. Both servers can also communicate to exchange data. At the modelling stage of the development, the links of the deployment diagram are stereotyped « Internet » and therefore do not provide any secrecy of the data. However, the possibility to add this security requirement is considered as a future change, postponed to the moment when the budget will be available. This evolution would consists in encrypting some of the links of the deployment diagram, i.e. in changing some of the links stereotyped « Internet » so that they would be stereotyped « encrypted ». More precisely, when the change occurs, the links concerned by this substitution will be the link between the secured server and the secured client and the link between the two servers. On the other hand, the link between the normal (i.e. not secure) client and the normal server will not need to be encrypted. To model this, a stereotype « substitute-all » with the tagged values { ref = make-links-secure } and { substitute = (l_1, l_2, « encrypted ») } is attached to the subsystem. Here, since only certain specific links are concerned, the first value of the pair representing the substitution needs to be defined. Since the links of the diagram do not have any specified unique name, we cannot use the classical UML composite namespace notation. Instead, we use the abstract syntax of deployment diagrams defined in Section 5.4. In consequence, to express precisely which links should be affected by the changes, the following expression is placed in a comment note attached to the stereotype labelled make-links-secure:

$$\{\, \mathsf{pattern} = \{ l1 = (\mathsf{nds}(l1), \mathsf{ster}(l1)),\ l2 = (\mathsf{nds}(l2), \mathsf{ster}(l2))\} \,\}$$

where $\mathsf{nds}(l1) = (\mathsf{SClientnode}, \mathsf{SServernode})$, $\mathsf{nds}(l2) = (\mathsf{Servernode}, \mathsf{SServernode})$ and $\mathsf{ster}(l1) = \mathsf{ster}(l2) = \{ \text{« Internet »} \}$.

The diagram of this example is shown in Figure 4.22 and the transition resulting from applying the changes is shown in Figure 4.23.

4.3.5 Unsecured evolution

Figure 4.24 shows an obvious example of an evolution that does not preserve the security.

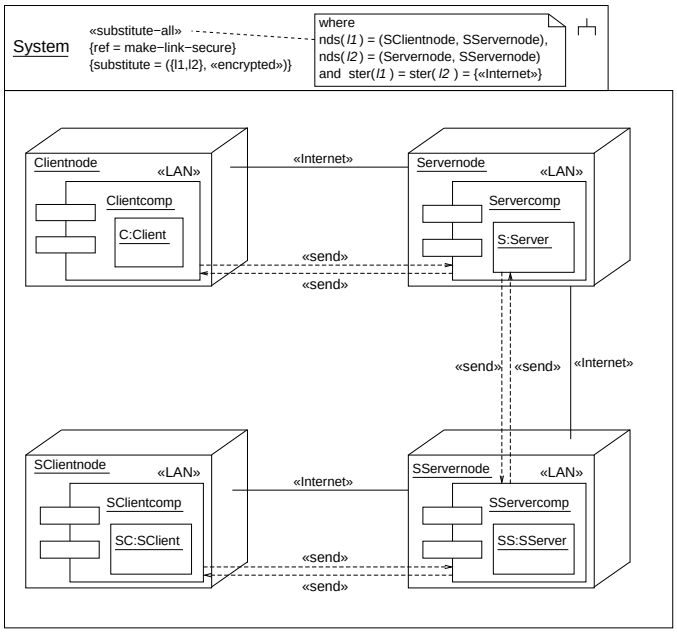

Figure 4.22: Example of use of « substitute-all » and pattern

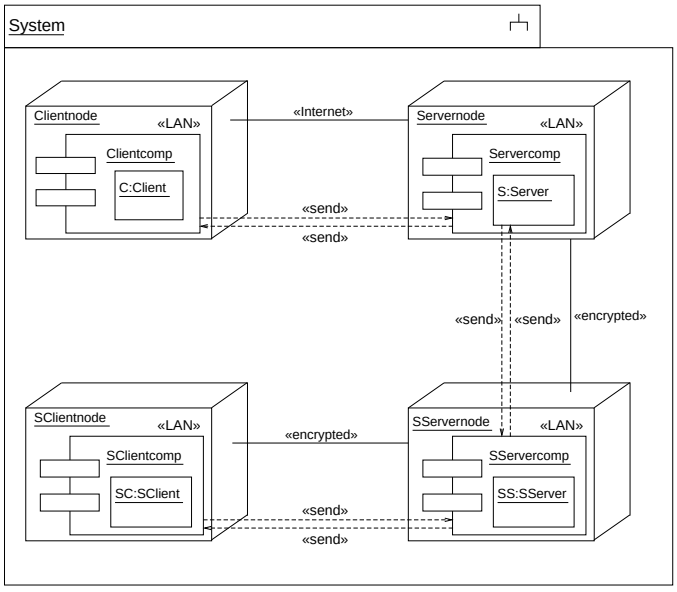

Figure 4.23: Result from applying of the modelled change

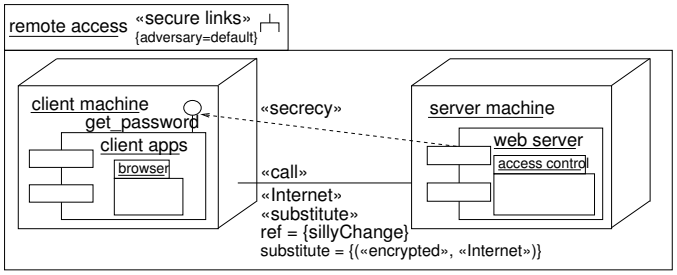

Figure 4.24: Unsecure evolution

Chapter 5

Formal Foundation of UMLseCh

5.1 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" [OMG09]. 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 5.1. *A UML named element is a tuple* $e \in$ **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.*

In addition to *named elements*, we can define *namespaces.* A *namespace* is defined in [OMG09] as an abstract container of *named elements*, the *namespace* being itself a *named element.* We assume a set **Namespaces** that contains all the namespaces of the instance of a model and give the following definition.

Definition 5.2. *A UML namespace is a pair* $(n, elts)$*, with* $n \in$ **String** *the name of the namespace and* $elts \subseteq$ **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 5.3. *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 $s \in$ **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 5.4. *An instance of a stereotype is defined as* $s^\tau \in$ **Stereotypes**, *where* $s \in$ **StereoNm** *is the stereotype definition of* s^τ.

This simply ensure that any stereotype instance is an instance of a stereotype defined in the set **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 5.5. *Let* s *be a stereotype definition, such that* $s \in$ **StereoNm** *and* s^τ *be a stereotype instance, such that* $s^\tau \in$ **Stereotypes** *and such that* s^τ *is an instance of the stereotype definition* s*, as defined in the definitoin 5.4,* τ *is defined as:*

τ : **Stereotypes** $\rightarrow$ **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 modelling 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 4.2 are of the type change. Formally, the set **ChangeNm** represents the set of stereotype definitions of type change. It is defined as follow.

Definition 5.6. *The set* **ChangeNm** $\subset$ **StereoNm** *is the set of definitions of stereotypes of type* change*, such that:*

$$\textbf{ChangeNm} \equiv \{\text{«}\textit{change}\text{»}, \text{«}\textit{substitute}\text{»}, \ldots, \text{«}\textit{delete-all}\text{»}\}^1$$

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 **Change** $\subset$ **Stereotypes**. Formally, a change stereotype can hence be precisely specified by the following definition.

Definition 5.7. *A stereotype of type* change *is a stereotype instance* $s^{\tau} \in$ **Change** *such that* $s \in$ **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 **TagNm** of tag definitions. It includes all the tag definitions of the stereotype definitions. Formally, we have **TagNm** $\equiv \cup_s tagnm(s)$, $\forall s \in$ **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 4, 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 instances of tag change, substitute, add and delete. We also define the set **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 5.8. *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.

[1] The complete list of stereotypes can be found in Figure 4.2 where the UMLseCh profile is defined

Definition 5.9. *Let $tags(s)$ be a function that returns the tags of a stereotype s. If $s \in$* **Change***, we have $tags(s) \subseteq$* **TagChange**.

The values associated to a tag ref of a change stereotype represents a list of labels such that each any 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 5.10 (Unicity of the ref values)**.** *Let* **TagRef** *be defined as* **TagRef** $\equiv \{t_1, t_2, \ldots, t_k, \ldots\}$ *and $\upsilon(n)$ be a function that returns the value of a tag n. Assume $T \equiv (\upsilon(t_1) \uplus \upsilon(t_2) \uplus \cdots \uplus \upsilon(t_k) \uplus \cdots)$ 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 υ 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 **Boolean** $\equiv \{true, false\}$. We also assume the set of boolean expressions **BoolExp**. The set **String** is the common set of sequences of symbols and digits.

5.2 New Elements for the UMLseCh Abstract Syntax

To provide formal semantics of the changes modelled 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 $s \in$ **Stereotypes** is a tuple given by $s = (name, tag, constraint)$ where:

- $name \in$ **StereoNm** is the name of the stereotype;
- the set $tag \subseteq$ **Tag** is the set of associated tags; and

- *constraint* $\in$ **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 5.11. *Let s be a stereotype such that $s \in$* **Stereotypes***, its set of tags is given by the function:*

$tags$: **Stereotypes** $\rightarrow \mathcal{P}$(**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 5.12. *Let s be a stereotype such that $s \in$* **Stereotypes***, its constraint is given by the function:*

$cons$: **Stereotypes** $\rightarrow$ **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 4.2 in Chapter 4, a stereotype of type change must have two associated tags and an associated constraint. Formally, we have the following.

Definition 5.13. $\forall s \in$ **Change** *such that $s = (name, tag, constraint)$, we have:*

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

where $ref \in$ **TagRef** *and $change \in$* **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 $change$, that return respectively the tag of type ref and the tag of type change of a stereotype s, such that $s \in$ **Change**.

Definition 5.14. $\forall s \in$ **Change***, its tag of type* ref *is given by the function:*

ref : **Change** $\rightarrow$ **TagRef**
$n \mapsto e$
where $e \in tags(n) \cap$ **TagRef**.

Definition 5.15. $\forall s \in$ **Change**, *its tag of type* change *is given by the function:*

$change$: **Change** $\rightarrow$ **TagTr**
$n \mapsto e$
where $e \in tags(n) \cap$ **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 **TagRef** and one **TagTr**, and **TagRef** $\cap$ **TagTr** $= \emptyset$.

In [OMG09], 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 element. Precisely, the abstract representation of elements that cannot 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 of the model element. 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 $t \in$ **Tag** given by $t = (tag, value)$, where:

- $tag \in$ **TagNm**, is the name of the tag; and
- $value \subseteq$ **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 $n \in$ **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 5.16. *Let* t *be a tag such that* $t \in$ **Tag**, *its name is given by the function:*

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

Definition 5.17. *Let t be a tag such that $t \in \mathbf{Tag}$, its set of values is given by the function:*

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

As defined in the UML specification [OMG09], 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, mentioned in the definition **??** and written $\emptyset$, such that $\emptyset \in \mathbf{Tag}$ and all its 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 4.2.2, 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 5.10. 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 modelled by the stereotype. Before we can define this precisely, we need to refine the definition of the empty tag $\emptyset$.

Definition 5.18. *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 5.19. *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 return the position of an element in the set of values of a tag of type* change. *These funtions will be define in the next section.*

Finally, *namespaces* need to be included in the UMLseCh abstract syntax. As mentioned in Section 4.1.2, a *namespace* does not exist by itself. It is modelled 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 5.2, a *namespace* can simply be define as an ordered pair $n = (sname, elts) \in$ **Namespaces** given by:

- a *namespace* name $sname \in$ **String**; and
- a set $elts \subset$ **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 **StereoNm**. It is thus correct to consider the name *sname* of a namespace as an element of **String**.

Definition 5.20. *Let* ns *be a namespace such that* $ns \in$ **Namespaces**, *its set of named elements is given by the function:*

$elts$: **Namespaces** $\rightarrow \mathcal{P}$(**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 5.21. *Let e be a namespace in its model element representation, such that $e = (e_1, \ldots, e_n)$, as defined in the definition 5.1, with $n \in \mathbb{N}$, $e \in$ **Elements** and $e_k \in$ **Elements** $\cup\, \mathcal{P}($**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 5.1, 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*[2].

5.3 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 5.1 and 5.2. Assume a function *space* that returns the namespace $n \in$ **Namespace** of a model element e, with $e \in$ **Elements**, or the model element e itself if e is not contained in any namespace[3]. Each change, namely a substitution, an addition or a deletion, can easily be defined as follow.

Definition 5.22 (Local substitution)**.** *Let e be a UML model element and e' the substitutive model element of e, such that $e, e' \in$ **Elements**, and let $N \in$ **Namespaces**, such that $N = (n, S)$, with n a name of e and $S \subset$ **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 5.23 (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 in e, such that $e \in$ **Elements** and $e' \subset$ **Elements**, and assume $ns(e)$ the namespace general representation of e, such that $ns(e) = (n, S)$, with n a name of e and $S \subset$ **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* behaviour described in Section 4.2.4. Indeed, assume that e' is a set of instances of model elements, such that $e' = \{e'_1, e'_2, \ldots, e'_{k-1}, e'_k, e'_{k+1}, \ldots, e'_n\}$,

[2]This condition is necessary to apply ns on subsystems. See Section **??** for the abstract syntax of subsystems

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

with $e'_i \in$ **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 remove the duplicate elements of the set.

Definition 5.24 (Local deletion)**.** *Let e be a UML model element to delete from the model, such that $e \in$* **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 $S \subset$* **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 expressable 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 thesis 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 5.22, 5.23 and 5.24 to the model, remove the tagged values associated to that change and update the function that evaluate the predicate *"occured"*. To update the tagged values of a change stereotype, such that the information relative to the change are removed after the change occurs, the following functions will be necessary.

Definition 5.25. *Let $t = (tag, values)$, with $t \in$* **TagChange***, be a tag of type* change *such that the ordered set values $\equiv \{v_1, \ldots, v_k, \ldots, v_n\}$, $\forall n \in \mathbb{N}$ 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 5.26. *Let* $t = (tag, values)$*, with* $t \in$ **TagChange***, be a tag of type* change *such that the ordered set values* $\equiv \{v_1, \ldots, v_k, \ldots, v_n\}$, $\forall n \in \mathbb{N}$ *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 return the right condition of a change, based on the value of the tag ref. To update the predicate *"occured"*, we need to define a function mapping a boolean value to a value of a tag ref. The set **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 belong to the set **TagRef**. It is hence possible to define a function ψ representing the predicate "occured", which is true if the change labelled by the tag ref given as argument of the predicate occured.

Definition 5.27. *The function* ψ *representing the predicate "occured" is defined as:*

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

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 5.22. 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 5.28 (Substitution)**.** *Let e be a UML model element and* e' *the substitutive model element of e, such that* $e, e' \in$ **Elements***, and let* $N \in$ **Namespaces***, such that* $N = (n, S)$*, with n a name of e and* $S \subset$ **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:*

$$substitute : (\textbf{Elements} \times \textbf{Elements}) \rightarrow \textbf{Elements}$$

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

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

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 5.3 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 4.2.2.

Definition 5.29 (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* $e \in$ **Elements** *and* $e' \subset$ **Elements***, and assume* $ns(e)$ *the namespace general representation of* e*, such that* $ns(e) = (n, S)$*, with* n *a name of* e *and* $S \subset$ **Elements**. *An addition of* e' *in* e *is defined by a function* $add(e, e')$ *such that:*

add : (**Elements** $\times$ **Elements**) $\rightarrow$ **Elements**
$(e, e') \mapsto substitute(e, e'')$

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

Again, as for the definition 5.23, the *merge* behavior is automatically supported by this definition.

Definition 5.30 (Deletion)**.** *Let* e *be a UML model element to delete from the model, such that* $e \in$ **Elements**, *and let* $N \in$ **Namespaces**, *such that* $N = (n, S)$*, with* n *a name of* N *and* $S \subset$ **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:*

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

where $\emptyset \in$ **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 modelled 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 occured. Let s bet the stereotype modelling 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 $\{\,\mathsf{ref} = V'\,\}$ and $\{\,\mathsf{change} = C'\,\}$, 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 modelling 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 modelled the changes, a roll-back function can be defined easily.

5.4 UMLseCh Formal Semantics

In this section, we describe the UMLseCh abstract syntax, which is an extention 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 concept defined for the behavioural 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 behvioural 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 behavioural semantics and UML Machine rules, defined in [Jür10], 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 **StereoNm** in Section 5.1. For UMLseCh, the stereotypes will be stereotypes instances that belong to the set **Stereotypes**[4]. 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 mention explicitly.

Note that the abstract syntax described in the following define 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 $e, e' \in$ **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

[4]Note that the set **StereoNm** is called **Stereotypes** in [Jür10], but is not to be confounded with the set **Stereotypes** defined here!

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.

5.4.1 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 a UMLseCh model should preserve those rules and definitions so that the resulting model is a 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 modelled evolution, all of the conditions must be fulfilled.

All of the elements used in UMLseCh diagrams are UML named element, 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 represention. The sets cannot contain duplicate elements and thus, it is not allowed to add an element that already exists 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 evolution 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 5.5. The second condition cannot be formally verified since base class are not defined in our abstract syntax. One should thus always ensure that the modification respects the base class definiton of the stereotype placed on the model by the application of the modelled evolution.

5.4.2 Object Diagrams

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 contains stereotypes of type change. However, a stereotype definition $stereo \in \{\text{«}\,\mathsf{call}\,\text{»}, \text{«}\,\mathsf{send}\,\text{»}\}$ is defined as in the case of UMLsec. This allows to verify certain constraint concerning call and send operations defined in the UMLsec formal semantics. Refer to [Jür10] 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 $\mathsf{Dep}(D)$ of dependencies. The same conditions as for UMLsec holds here.

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 modelling 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 modelling 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.

Preservation of the Consistency

When modifying a Object diagram, several concistency 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 of an interface requires that this operation also exists in the object o of which the interface belong. 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 $i = (iname_i, ospec_i, stereo_i)$, $\forall i \in int$.

5.4.3 Class Diagrams

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.

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.

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.

5.4.4 Statechart Diagrams

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 $\mathsf{stereo} \subseteq$ **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 $\mathsf{stereo} \subseteq$ **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.

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.

Preservation of the Consistency

On statemachine, the specific elements concerned by changes that could affect the consistency are states and transitions. When a transition t is subsituted 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 occured. 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 ingoing 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 if the transition connects the initial state directly to the final state, which is pretty unlikely!

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:

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

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 ingoing 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 4.2.5.

5.4.5 Sequence Diagrams

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 $\mathsf{stereo} \subseteq$ **Stereotypes** of stereotypes

The set of lifelines of a sequence diagram D is called $\mathsf{Obj}(D)$. Connections are extended in the same way with a set of stereotypes. A connection is thus defined as a 5-tuple $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l), \mathsf{stereo})$ where $\mathsf{stereo} \subseteq$ **Stereotypes** is a set of stereotypes and $\mathsf{source}(l)$, $\mathsf{guard}(l)$, $\mathsf{msg}(l)$ and $\mathsf{target}(l)$ 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 [OMG09] 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 $D = (\mathsf{Obj}(D), \mathsf{Cncts}(D))$.

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.

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 on of the two objects, one representing the source object and one target object, must be an oject of the sequence diagram. Before applying the substitution or the addition, we thus verify the following rule:

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

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:

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

5.4.6 Activity Diagrams

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.

Application of a Change

The constructs of activity diagrams are the same as the ones of statechart diagrams, with an additional concept of swimlane. Changes of swimlanes cannot be modelled 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.

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:

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

5.4.7 Deployment Diagrams

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 instance and object names contained in the component, and $stereo \subseteq$ **Stereotypes** is a set of stereotypes. A node is a 3-tuple $N = (loc, comp, stereo)$ where $stereo \subseteq$ **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 $\mathsf{ster}(l) \subseteq$ **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 $stereo \subseteq$ **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.

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.

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:

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

Similarily, 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:

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

5.4.8 Subsystem

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 state-chart 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 modelled 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:

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

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 theoratically 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 modelled 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 increase the readability. The following evolution can, on the other hand, be modelled 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 modelled by attaching UMLseCh stereotypes to the subsystem, but other modifications, such as modifying an operation of a class, can also be modelled in such a way by giving an expression that precisely identifies the sub-element to modify. These evolutions can thus either be modelled 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.

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 behaviour, being in the form of a statemachine or a sequence diagram, the substitution of this behaviour should be substituted by another behaviour 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 behaviour 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 behviour and that the evolution consists in adding that behaviour, the following constraint is defined to ensure that an activity has only one associated behaviour. 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 decribe 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 event and in messages respectively. Again, those operation must be defined within the subsystem $\mathcal{S}$ and in particular in the static structure diagram $\mathsf{Ssd}(\mathcal{S})$.

5.4.9 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 [Gra81]. 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 implicitely 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 occuring 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 concerved 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 verify 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 occure, 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 forbiden 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 5.1. Deleting the final state would be forbiden for a single change, as defined in the consistency rules of Section 5.4.4. Deleting the transition between the state A and the final state is also forbiden by a condition of Section 5.4.4. 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

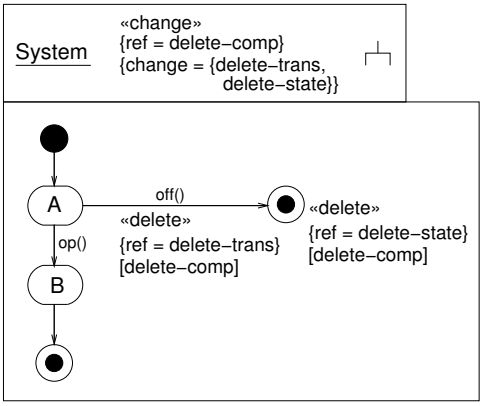

Figure 5.1: Example of an allowed composite change

that consists in deleting the connected transition is applied. This can be formalised by assuming that the parallel change delete-trans 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 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 ressources 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* behaviour defined in Section 4.2.4 will follow the same principle as the composite changes.

Chapter 6

Conclusion

Using UML to model softwares at the early stages of the development cycle presents incontestable advantages. It allows important design decisons to be taken into account at an early stage and is widely used, thus known by many developers and supported by many tools. However, as it was initially design for general purposes, it lacks means to express specific important concepts such as security or evolution. Nevertheless, UML offers extensions mechanisms that allow one to augment the language by adding new constructs. Using the lightweight extension mechanism allows to keep the advantages of the tool support for the graphical modelling and makes it easy for the developers to learn and use.

It was argued in Chapter 2 that security and evolution were two key concepts that should be envisaged at the early stages of the development cycle. Therefore, defining those concepts as included in the language would address this issue. The UMLsec extension is a formally-defined approach to consider the security requirements at the stage of modelling the system. It provides a notation applicable on the UML diagrams and methods to verify the fulfillment of the security requirements.

We define UMLseCh to address the evolution issue. UMLseCh extends UMLsec in order to add the evolution approach while keeping the security concepts. It provides language constructs to express the substitution, the addition or the suppression of simple or complex model elements. As for UMLsec, it defines the notation with a UML profile that includes the UMLsec profile and add stereotypes and tagged values to represent future evolutions. Using the lightweight extension mechanism of UML based on profiles, the notation can be used with any tool compliant to the UML metamodel.

We also define the abstract syntax of UMLseCh by extending the abstract syntax of UMLsec to formally include the concepts of UMLseCh. By giving

a formal semantics to the UML constructs and to the UMLseCh notation, we show that the modelled evolutions can be executed in order to obtain a new model on which the changes were applied, and where the UMLsec notation has remained. The evolved model is thus a UMLsec model which still expresses the security requirements and can be verified again, using the method and tool offered by UMLsec. The formal semantics also allows to define consistency rules ensuring that the result model of the transformation is still compliant to the UMLseCh abstract syntax.

The UMLseCh approach, although still incomplete, seems very promising in modeling possible future evolution of model together with security requirements. However, there is still a lot of work to be done. In particular, this thesis only represents a (hopefully useful) beginning towards the elaboration of a more complete solution to model future evolution and verify the preservation of the security over the evolution. There are a few interesting directions that we would like to recommend.

The implementation of a tool support that applies the changes modelled by UMLseCh and transforms the model into the evolved model appears as a main step forward for the support of the notation. Such a tool would provide a means for the developers to benefit from the notation so that they do not need to manually apply the evolution.

Another main achievement would be to define a more elaborated method to verify the preservation of the security. So far, the preservation is verified by simply applying a verification of the evolved model using the UMLsec method. However, the composition of processes [Jür] seems to offer a formal approach to verify the preservation of the security without having to recheck the model completely.

Other improvements could be given to the notation. For example, the semantics of the delete could be refined so that it offers more options. It could define a more advanced behaviour when the element to delete is connected to other model elements. The possible options could be similar to the ones in SQL, in the sense that they would either disallow the suppression, as it is defined now, or propagate it. The propagation would, for instance, delete all the connected model elements. Further reasearch could also be conducted so that the modification applied on the UML models are directly and automatically applied to the corresponding code.

Finally, a real scale case study could be applied on an existing system in order to place the approach in a real context.

Bibliography

[AEBK04] Mohd Syazwan Abdullah, Andy Evans, Ian Benest, and Chris Kimble. Developing a uml profile for modelling knowledge based systems, 2004.

[BCR00] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the dynamics of uml state machines. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 223–241, London, UK, 2000. Springer-Verlag.

[BLO03] 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.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September/October 2005.

[CB09] Çağdaş Cirit and Feza Buzluca. A uml profile for role-based access control. In *SIN '09: Proceedings of the 2nd international conference on Security of information and networks*, pages 83–92, New York, NY, USA, 2009. ACM.

[CHFRT01] Ned Chapin, Joanne E. Hale, Juan 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.

[CP04] Vittorio Cortellessa and Antonio Pompei. Towards a uml profile for qos: a contribution in the reliability domain. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 197–206, New York, NY, USA, 2004. ACM.

[Eve07] Joerg Evermann. A meta-level specification and profile for aspectj in uml. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 21–27, New York, NY, USA, 2007. ACM.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *INTERNATIONAL JOURNAL ON SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE (JSME)*, 18:207–236, 2006.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations, 1981.

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

[Gur95] Yuri Gurevich. *Specification and validation methods*. Oxford University Press, Inc., New York, NY, USA, 1995.

[HH03] Siv Hilde Houmb and Kine Kvernstad Hansen. Towards a uml profile for security assessment. In *In UML'2003, Workshop on Critical Systems Development with UML*, 2003.

[HKC05] Yan Han, Günter Kniesel, and Armin B. Cremers. Towards visual aspectj by a meta model and modeling notation. In *In 6th AOM, AOSD*, 2005.

[JK06] 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.

[Jür] Jan Jürjens. A domain-specific language for cryptographic protocols based on streams.

[Jür10] Jan Jürjens. *Secure Systems Development with UML*. Springer-Verlag, Berlin, Heidelberg, 2010.

[KBC] Audris Kalnins, Janis Barzdins, and Edgars Celms. Basics of model transformation language mola.

[KBC04] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language mola. In *in: Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004*, pages 14–28, 2004.

[KKKS] Mohamed M. Kande, Mohamed M. K, Jörg Kienzle, and Alfred Strohmeier. From aop to uml: Towards an aspect-oriented architectural modeling approach.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, 1995.

[Küh] Thomas Kühne. Making modeling languages fit for model-driven development.

[Läm04] Ralf Lämmel. Evolution of rule-based programs, 2004.

[Lana] The ATLAS Transformation Language. http://www.eclipse.org/atl/.

[Lanb] The Unified Modeling Language. http://www.uml.org/.

[LBD02] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, London, UK, 2002. Springer-Verlag.

[Leh69] M. M. Lehman. The programming process. In *IBM Res. Rep. RC 2722*, IBM Res. Centre, Yorktown Heights, September 1969.

[Leh74] 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.

[Leh96] 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.

[LP76] 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.

[LR01] 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.

[LRK00] M M Lehman, J F Ramil, and G Kahen. Evolution as a noun and evolution as a verb, 2000.

[LRW+97] 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.

[LS] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat.

[MFV06] Nathalie Moreno, Piero Fraternalli, and Antonio Vallecillo. A uml 2.0 profile for webml modeling. In *ICWE '06: Workshop proceedings of the sixth international conference on Web engineering*, page 4, New York, NY, USA, 2006. ACM.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi 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.

[OMG00] OMG. *Unified Modeling Language: Superstructure, version 1.3.* Object Modeling Group, Mars 2000.

[OMG01] OMG. *Unified Modeling Language: Superstructure, version 1.4.* Object Modeling Group, Septembre 2001.

[OMG02] OMG. Mof 2.0 query/views/transformations rfp, 2002.

[OMG05] OMG. *MOF QVT Transformation Specification.* Object Modeling Group, 2005.

[OMG09] OMG. *Unified Modeling Language: Superstructure, version 2.2.* Object Modeling Group, February 2009.

[Par79] D. L. Parnas. On the criteria to be used in decomposing systems into modules. pages 139–150, 1979.

[Par94] David 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.

[PM03] Jorge Enrique Pérez-Martínez. Heavyweight extensions to the uml metamodel to describe the c3 architectural style. *SIGSOFT Softw. Eng. Notes*, 28(3):5–5, 2003.

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

[RLL98] David Rowe, John Leaney, and David Lowe. Defining systems evolvability - a taxonomy of change. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:0045, 1998.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[Sto] Volker Stolz. An integrated multi-view model evolution framework.

[Sun09] 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.

[Wil03a] Edward D. Willink. A concrete uml-based graphical transformation syntax :the uml to rdbms example in umlx, 2003.

[Wil03b] Edward D. Willink. Umlx: A graphical transformation language for mda, 2003.

Index