# D4.1 SECURITY MODELLING NOTATION FOR EVOLVING SYSTEMS

Siv Houmb (OU/Telenor), Shareeful Islam (TU Munich), Jan Jurjens (TUD), Martin Ochoa (TUD), Michael Hafner (UIB), Frank Innerhofer-Oberperfler (UIB), Manuela Weitlaner (UIB), Benjamin Fontan (THA), Edith Felix (THA), Federica Paci (UNITN), Frédéric Dadeau (INR), Boutheina Chetali (GTO)

## Document information

| | |
|---|---|
| | D4.1 |
| **Document Title** | Security Modelling Notation for Evolving Systems |
| **Version** | 1.9 |
| **Status** | Draft |
| **Work Package** | WP 4 |
| **Deliverable Type** | Report |
| **Contractual Date of Delivery** | 31 January 2010 |
| **Actual Date of Delivery** | 22 January 2010 |
| **Responsible Unit** | TUD |
| **Contributors** | UNITN, GTO, INR, SMA, OU, THA, UIB, TUD |
| **Keyword List** | |
| **Dissemination level** | PU |

## Document change record

| Version | Date | Status | Author (Unit) | Description |
|---|---|---|---|---|
| 0.1 | 21.09.2009 | Draft | Jan Jürjens (TUD) | Outline of the deliverable |
| 0.2 | 10.11.2009 | Draft | Martin Ochoa (TUD) | Integration into First draft |
| 0.3 | 10.11. 2009 | Draft | Siv Houmb (OU/Telenor), Shareeful Islam (TU Munich), Jan Jurjens (TUD), Martin Ochoa (TUD) | Sections 2 to 4 and 6 to 9 |
| 0.4 | 10.11. 2009 | Draft | Boutheina Chetali (GTO) | Section 5: Background on the GP |
| 0.5 | 10.11.2009 | Draft | Michael Hafner, Frank Innerhofer-Oberperfler, Manuela Weitlaner (UIB) | Section 10: SecureChange Process |
| 0.6 | 10.11.2009 | Draft | Benjamin Fontan,Edith Felix (THA), | Section 12: Thales Security vs UMLseCh |
| 0.7 | 10.11. 2009 | Draft | Federica Paci (UNITN) | Section 11: Security Requirements |
| 0.8 | 10.11.2009 | Draft | Frédéric Dadeau (INR) | Section 13: Testing |
| 0.9 | 3 .12.2009 | Draft | Martin Ochoa | Minor fixes |
| 1.0 | 11.12.2009 | Draft | Martin Ochoa | Draft for internal reviewing |
| 1.1 | 11.12.2009 |  | All authors | Updates on their sections |
| 1.2 | 11.12.2009 | Draft | Martin Ochoa | Minor fixes |
| 1.3 | 08.01.2010 | Draft | Elisa Chiarani (UNITN) | Quality check completed; minor remarks |
| 1.4 | 09.01.2010 | Draft | Federica Paci(UNITN) | Review; comments, Update on Security Requirements |
| 1.5 | 09.01.2010 | Draft | Benjamin Fontan (THA) | Review, comments, Update on Thales Security vs UMLseCh |
| 1.6 | 09.01.2010 | Draft | Bruno Legeard (SMA) | Review, comments |

|  |  |  |  |  |
|---|---|---|---|---|
| 1.7 | 09.01.2010 | Draft | Fabrice Bouquet (INR) | Review, comments |
| 1.8 | 18.01.2010 | Draft | Siv Houmb (OU/Telenor), Shareeful Islam (TU Munich), Jan Jurjens (TUD), Martin Ochoa (TUD) | Update on sections 2-4, 6-9 |
| 1.9 | 21.01.2010 | Final | Elisa Chiarini (UNITN) | Final Quality Check |

# Executive summary

This document describes a Security Modelling notation for evolving systems (UMLseCh). This notation extends the UMLsec modelling language by means of stereotypes (a UML lightweight extension mechanism). This notation can be used to model evolution on general distributed systems, but also extensions are proposed to deal with evolving secure software on Smartcards (Section 6).

Connections with WP2 (SecureChange Process), WP7 (SecureChange Testing) and WP3 (Security Engineering Requirements) are also shown, as well as links to the Thales DSML modelling notation (Sections 10 to 13).

After introducing the Global Platform (Section 5) and its design using UMLseCh (Section 7), an example of modelling the e-Purse application on the Global Platform for smartcards is given, in the context of the POPS Case Study (Section 8).

Finally, an overview of the work in progress towards T4.2 which deals with the formal foundations of security preservation under evolution is also presented (Section 9).

The design notation and verification techniques for secure evolving systems presented in this deliverable aim to be usable in the context of the various change dimensions and change perspectives mentioned in [HD09].

# Index

# 1 Introduction

As stated in the Description of Work of SecureChange, the objective of Work Package 4 is to develop a model-based design approach that is tailored to the needs of the secure development of evolving systems. The approach will glue requirements techniques built in WP3 with the final code and configuration and will provide tools that allow the user to automatically analyze the models against these its a-priori requirements, also taking into account the configuration scenarios that are likely according to the assessment analysis. It will provide as output models that will be used for model-based testing in WP6 and a traceable link to the implementations that are statically analyzed in WP7.

On the one hand, the design models themselves have to be designed in a way that will make future system evolution feasible. On the other hand, the model analysis techniques to be developed need to be able to analyse the models not only against the a-priori given security requirements, environment assumptions, and threat scenarios, but it should be possible to analyse models against variations and changed in these requirements, assumptions, and scenarios.

A first step in this direction, as described in Task 4.1 is to extend a security design modelling notation with evolutive security properties. The goal of this document is to present such a Security Modelling notation for evolving systems that extends an existing modelling notation. The chosen notation is UMLsec [Jür05a], which is itself an extension of UML. It allows modelling systems at different levels of abstraction and to formally verify security properties on the model. The extension, called UMLseCh, operates basically on three levels:

a) An abstract notation, used to communicate with clients and/or to annotate changes in the system

b) A concrete notation, applicable to all diagrams in the system and all levels of refinement, that allows to precisely state the evolution on the modelling element(s).

c) A notation for secure evolving systems in smartcards

The usage of the notation is illustrated in various examples, ranging from basic applications to a more comprehensive case study: the smartcard-based e-Purse application, in the context of the POPS Case Study.

We also give an overview of the work being carried out for task T4.2 “Formal foundations”. Although still work in progress, we present some formal results in security preservation under evolution. In particular, the preservation of secrecy under component composition is treated.

The document also presents various connections with security related methodologies specific to other Secure Change Work Packages, such as security requirements engineering and test generation from models. Links with industry specific modelling notations such DSML (used by the Secure Change partner Thales) are also highlighted.

## Sections Walk-through

**Section 2** introduces the UMLsec notation for model-based security engineering, which is the selected Security Modelling notation to be extended.

**Section 3** describes the Change Taxonomy considered in this deliverable based in the Taxonomy provided by WP 5 [HD09].

**Section 4** introduces the UMLseCh notation, an extension of UMLsec to model evolving systems, which is the main task of this deliverable.

**Section 5** introduces the Global Platform for smartcards, as an introduction for the following sections.

**Section 6** presents further extensions to UMLsec specific for smartcards.

**Section 7** shows the application of UMLseCh to the Global Platform Architecture.

**Section 8** discusses the e-Purse application model in the context of UMLseCh.

**Section 9** gives an overview of methods to ensure secrecy preservation after composition of System Components. This is helpful to ensure secrecy preservation after changes in components, a common form of evolution.

Sections 10 to 13 present connections with other Work Packages in SecureChange:

**Section 10** shows the links between UMLseCh and the general Secure Change process.

**Section 11** describes the link between security requirements engineering in the context of WP3 and UMLsec models.

**Section 12** describes the Thales modelling environment for security and possible ways to relate it to UMLsec.

**Section 13** gives an overview of the methodology used to generate tests from models in the context of WP7.

# 2 Background: UMLsec

In Model-based Security Engineering [Jür05a], [Jür05b], [Jür06] universal security requirements (such as secrecy, integrity, authenticity and others) and security assumptions on the system environment, can be specified either within a UML specification, or within the source code (Java or C) as annotations.

One can use MBSE within model-based development (see Figure 2.1). Here one first constructs a model of the system. Then, the implementation is derived from the model: either automatically using code generation, or manually, in which case one can generate test sequences from the model to establish conformance of the code regarding the model. The goal is to increase the quality of the software while keeping the implementation cost and the time-to-market bounded.

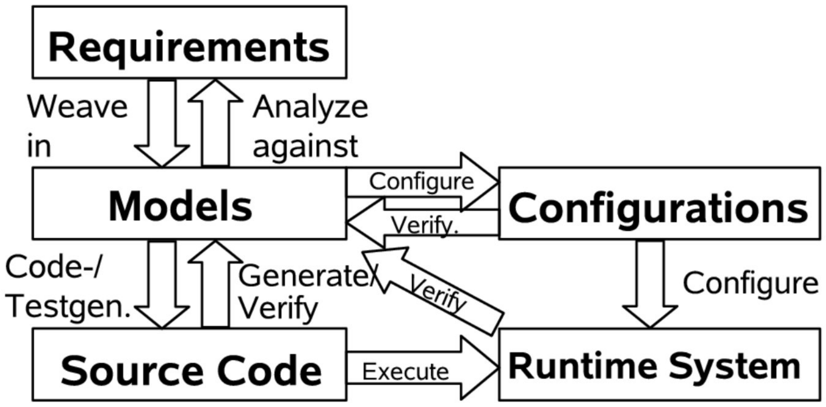


Figure 2.1: Model Based Security Engineering

In UMLsec, recurring security requirements, such as secrecy, integrity, and authenticity are offered as specification elements by the UMLsec extension. These properties and its associated semantics are used to evaluate UML diagrams of various kinds and indicate possible security vulnerabilities. One can thus verify that the desired security requirements, if fulfilled, enforce a given security policy. One can also ensure that the requirements are actually met by the given UML specification of the system. UMLsec encapsulates knowledge on prudent security engineering and thereby makes it available to developers who may not be experts in security. The extension is given in form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate security requirements and assumptions on the system environment. *Constraints* give criteria that determine whether the requirements are met by the system design, by referring to a precise semantics mentioned below.

The tags defined in UMLsec represent a set of desired properties. For instance, “freshness” of a value means that an attacker cannot guess what its value was. Moreover, to represent a profile of rules that formalise the security requirements, the following are some of the stereotypes that are used: «critical», «high», «integrity», «internet», «encrypted», «LAN», «secrecy», and «secure links». If relevant, their profile also contains the possible attackers associated to them as shown in Figure 2.2.

| Stereotype | **Threats** *default*() | **Threats** *insider*() |
|---|---|---|
| Internet | {delete, read, insert} | {delete, read, insert} |
| Encrypted | {delete} | {delete, read, insert} |
| LAN | Ø | {delete, read, insert} |

Figure 2.2: Attackers and threats per stereotype in the UMLsec

Figure 2.2 gives the default attacker, which represents an outsider adversary with modest capability. This kind of attacker is able to read, delete, and insert messages on an Internet link. On an encrypted Internet link, such as a virtual private network, the attacker might still be able to delete messages, without knowing their encrypted content, by bringing down a network server. However, an average adversary would not be able to read the plaintext messages or insert messages encrypted with the right key. Of course, this assumes that the encryption is set up in a way such that the adversary does not get hold of the secret key. The default attacker is assumed not to have direct access to the local area network (LAN) and therefore not to be able to eavesdrop on those connections.

The definition of the stereotypes allows for model checking and tool support. As an example consider «secure links». This stereotype is used to ensure that security requirements on the communication are met by the physical layer. More precisely, when attached to a UML subsystem, the constraint enforces that for each dependency d with stereotype s in the set ({«secrecy»,«integrity»,«high»}) between subsystems or objects on different nodes, according to each of the above stereotypes, there shall be no possibilities of an attacker reading, or having any kind of access to the communication, respectively. A detailed explanation of the tags and stereotypes defined in UMLsec can be found in [Jür05a].

In Figure 2.3 we show the UMLsec metamodel.

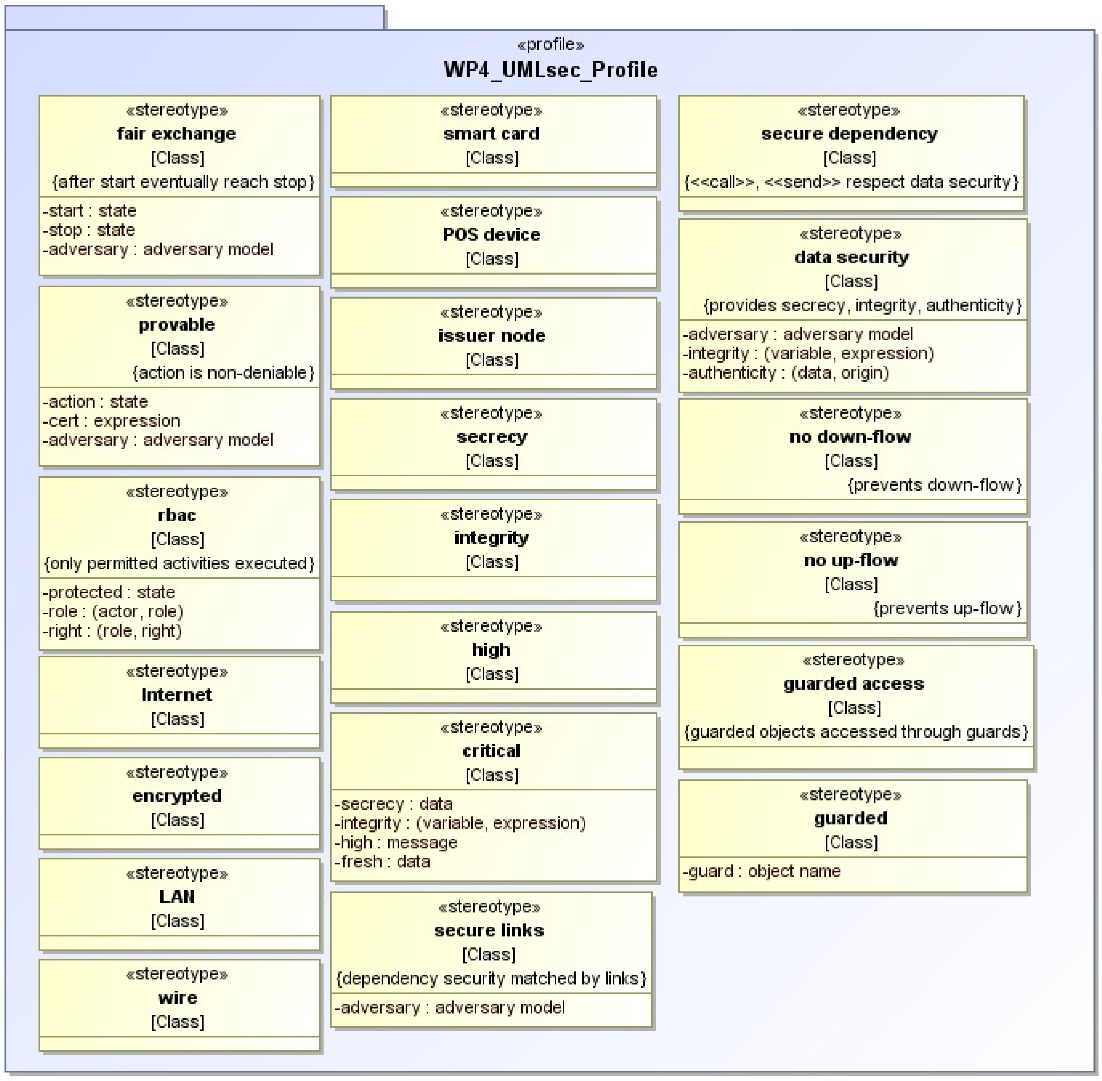

«profile»
WP4_UMLsec_Profile
«stereotype» fair exchange [Class] {after start eventually reach stop}
-start : state
-stop : state
-adversary : adversary model
«stereotype» provable [Class] {action is non-deniable}
-action : state
-cert : expression
-adversary : adversary model
«stereotype» rbac [Class] {only permitted activities executed}
-protected : state
-role : (actor, role)
-right : (role, right)
«stereotype» Internet [Class]
«stereotype» encrypted [Class]
«stereotype» LAN [Class]
«stereotype» wire [Class]
«stereotype» smart card [Class]
«stereotype» POS device [Class]
«stereotype» issuer node [Class]
«stereotype» secrecy [Class]
«stereotype» integrity [Class]
«stereotype» high [Class]
«stereotype» critical [Class]
-secrecy : data
-integrity : (variable, expression)
-high : message
-fresh : data
«stereotype» secure links [Class] {dependency security matched by links}
-adversary : adversary model
«stereotype» secure dependency [Class] {<<call>>, <<send>> respect data security}
«stereotype» data security [Class] {provides secrecy, integrity, authenticity}
-adversary : adversary model
-integrity : (variable, expression)
-authenticity : (data, origin)
«stereotype» no down-flow [Class] {prevents down-flow}
«stereotype» no up-flow [Class] {prevents up-flow}
«stereotype» guarded access [Class] {guarded objects accessed through guards}
«stereotype» guarded [Class]
-guard : object name

Figure 2.3: The UMLsec metamodel.

We give now a brief description of each stereotype in the UMLsec profile.

**fair exchange** (for use case diagrams)

Intuitively, this stereotype represents the security requirement that any transaction should be performed in a way that prevents both parties from cheating. When applied to a subsystem containing a use case diagram, it requires that this subsystem can be refined by another subsystem only if that is also stereotyped «fair exchange». Note that this usage of the «fair exchange», stereotype has only an informal meaning, as opposed to the stereotypes below. In particular, “refinement" is meant here in an informal sense. It just serves as an example how the security requirements included as stereotypes in the other kinds of diagrams below can also conveniently be included in use case diagrams.

**fair exchange** (for activity diagrams)

This stereotype, when applied to subsystems containing an activity diagram, has associated tags {start}, {stop}, and {adversary}. The tags {start} and {stop} take pairs (good; state) as values, where good is the name of a good to be sold and state is the name of a state. If there is only one good to be sold in a given system specification, the value good can be omitted. The tag {adversary} specifies an adversary type relative to which the security requirement should hold. The associated constraint requires that, for every good to be sold, whenever a {start} state in the contained activity diagram is reached, then eventually a {stop} state will be reached, when the system is executed in presence of an adversary of the type A specified in the tag {adversary}.

**provable**

A subsystem S may be labelled «provable», with associated tags {action}, {cert}, and {adversary}. The tag {cert} contains an expression which serves as proof that the action at the state given in the tag {action} was performed. The tag {adversary} specifies an adversary type relative to which the security requirement should hold. The stereotype «provable» then specifies that S may output the expression E given in {cert} only after the state with name in {action} is reached, when executed in presence of an adversary of the type A that is specified in the tag {adversary}.

**rbac**

This stereotype of subsystems containing an activity diagram enforces role-based access control in the business process specified in the activity diagram. It has associated tags {protected}, {role}, and {right}. The tag {protected} has as its values the states in the activity diagram the access to whose activities should be controlled. The {role} tag may have as its value a list of pairs (actor ; role) where actor is an actor in the activity diagram, and role is a role. The tag {right} has as its value a list of pairs (role; right) where role is a role and right represents the right to access a protected resource. The associated constraint requires that the actors in the activity diagram only perform activities for which they have the appropriate rights.

**Internet, encrypted, LAN, wire, smart card, POS device, issuer node**

These stereotypes on links (resp. nodes) in deployment diagrams denote the respective kinds of communication links (resp. system nodes). We require that each link or node carries at most one of these stereotypes. An adversary A is associated to each link as in figure Figure 2.2. (and similarly for the nodes).

**secrecy, integrity, high**

These stereotypes, which may label dependencies in static structure or component diagrams, denote dependencies that are supposed to provide the respective security requirement for the data that is sent along them as arguments or return values of operations or signals. These stereotypes are used in the constraint for the stereotype «secure links».

**critical**

This stereotype labels objects or subsystem instances containing data that is critical in some way, which is specified in more detail using the corresponding tags (for example: {secrecy}, {integrity}, {authenticity}).

**secure links**

This stereotype, which may label subsystems, is used to ensure that security requirements on the communication are met by the physical layer, given the adversary type A that is specified in the tag {adversary} associated with this stereotype.

**secure dependency**

This stereotype, used to label subsystems containing static structure diagrams, ensures that the «call» and «send» dependencies between objects or subsystems respect the security requirements on the data that may be communicated across them, as given by the tags {secrecy}, {integrity}, and {high} of the stereotype «critical».

**data security**

This stereotype labelling subsystems has the following constraint. The behaviour of any subsystem S stereotyped «data security» respects the data security requirements given by the stereotypes «critical» and the associated tags contained in the subsystem, with respect to the threat scenario arising from the deployment diagram and given the adversary type A that is specified in the tag {adversary} associated with this stereotype.

**no down-flow, no up-flow**

These stereotypes of subsystems prevent the indirect leakage or corruption of sensitive data: It enforces secure information flow by making use of the tag {high} associated with the stereotype «critical».

**guarded access**

This stereotype of subsystems is supposed to mean that each object in the subsystem that is stereotyped «guarded» can only be accessed through the objects specified by the tag {guard} attached to the «guarded» object.

# 3 Change Taxonomy

This section outlines the type of changes that we consider in the Secure Change project and why and how change comes about, based on [HD09]. The focus is on reflecting change on the model level to ease system evolution by ensuring effective control and tracking of changes. The goal is to tackle possible challenges and problems arising from change up-front. Change are considered from three aspects: (i) Change dimension, (ii) Change perspectives, and (iii) Change schedule.

Change dimension allows us to trace the kind (cause) of change and up-front tackle known problems as experience is gained. Change perspectives support the change dimension by making explicit the circumstances under which the change happen, while change schedule capture the timeliness at which change happens.

Working on the model level has several advantages, most notable the abstraction level and easy to understand notation of design models which makes them particularly suitable for tracing change as we will see demonstrated later in the document. Design models represent the early exploration of the solution space and are the intermediate (negotiation level) between requirements and implementation. Design models can be used for what-if analysis in requirements negotiation and to abstract away the details of an implementation, making it understandable and suitable to support decision processes. Design models can also be explicit and formal and therefore be used to specify, analyse and trace changes directly.

Changes in software systems that we consider are either functionality-driven or security-driven. Changes introduced to fix a newly discovered security threat or to patch the system due to a security attack are referred to as security-driven changes. All non-security related changes are called functionality-driven changes. More details are given in the following.

## 3.1 Change Requests

We use stereotypes, tagged values and constraints to model change in the UML models. We also advocate to use the design models for change exploration and decision support when considering how to integrate new or additional (existing) security functions and to explore the security implications of planned system evolution. To maintain the security properties of a system through change, the change must be explicitly expressed such that its implications can be analysed a priory. In addition, it is important to check whether the foreseen change can be explored using the current design models, before a change exploration analysis can be such analyses to be effective, changes should be formalized as change requests.

The purpose of the pre-analysis is to ease into the change, i.e. moving from the current system state to the new and changed system state, whether in the past, present or future. As we anticipate multiple rounds of change, and also changes to earlier changes, it is important to formalize each change in a manner that it can be separated and traced in the design models, but also to allow change-driven security verification and re-verification. The latter is important, although difficult. However, note that expressing change as change requests according to our formalism is not a strict requirement as such.

The pre-analysis involves the following activities, which should be undertaken before allowing any change in the UMLsec design models:

- Check if the model permits the change outlined in the change request
- Check which parts of the model permit changes
- Check whether existing security properties are affected by the change and to what extend they will or may be affected
- Check which artefact will or may be affected by the change
- Check what values under the artefact that need to be updated
- Check if there are additional/new values that need to be included
- Check if and how early security verification results can be reused

## 3.2 Change Dimension - Model Level

There are two main kinds of change: (1) Functionality (system)–driven and (2) Security–driven. Functionality-driven changes covers normal system evolution, i.e. additional requirements, new customer demands, refinements or extensions of existing functionality and similar. Security-driven changes are introduced to a system because of e.g. newly identified security threats and as to repair after being subject to a successful security attack or attack attempt. Security-driven changes can also stem from requirements, law and regulations and changes in a company's security policy. This means that both functionality– and security–driven changes are forms of adaptations to the changing system environment and usage environments. For security, new ways of using the system often refer to the discovery of a back-door or an irregular manner of exploring system functionality, which in principle can be positive, but most often is negative, i.e. misuse.

In an industrial application context (such as the risk assessment scenarios considered in WP5), changes are tackled differently depending on their implications and the size and kind of system change. For the model level change analysis, we consider two change dimensions (as for risk analysis, however, note the differences in the definitions) to distinguish between small and larger changes: (a) Evolution and (b) Revolution.

*Evolution* - Smaller changes to one or more components and/or services (functional or security; can also be both) of the system that accumulates over time. For the GlobalPlatform, examples are: changes to application data, applications and card data.

*Revolution* - Significant changes to several components and/or services (functional or security; can also be both) introduced at a specific point in time or during a limited time-frame. For the GlobalPlatform, examples are: changes to platform code and changes to hardware and software interfaces

## 3.3 Change Perspectives - Model Level

There may be multiple reasons triggering change and these triggers are used to decide on the security analysis strategy, i.e. the perspective and goal of the analysis, as we will see demonstrated in Section 7.9. We separate between three change perspectives: (i) Maintenance perspective, (ii) Continuous perspective and (iii) Unplanned perspective.

The maintenance perspective covers all changes done to maintain the core services, functional and security features and of a system. These changes are made in a controlled environment (e.g., by taking a GlobalPlatform card to the “Card Locked” state, performing the specific update, execute additional integrity and security analysis and tests, and then actively taking the card back to the “SECURED” state of yet again normal operation) and at a planned and announced point in time. Maintenance may involve small (evolution) or large (revolution) changes, but are always carried out in a controlled manner, meaning that there are most likely less undesired and unforeseen after-the-fact implications of these changes.

The continuous perspective covers all small changes that are made to a system that contributes to a consistent system service level. (For GlobalPlatform, continuous perspective concerns changes to application data and applications, but not security domains and general platform applications, as these are only changed under maintenance perspective or under unplanned circumstances). Only small (evolution) changes are permitted under this perspective.

The unplanned (patch) perspective covers all changes, small (evolution) and large (revolution), necessary to recover from a security attack, card failures and similar or to protect the card against new and evolving security threats. For example, for a GlobalPlatform card the card issuer can decide to block a particular security domain and all its applications upon discovery a security attack or serious threats against a specific security domain.

The design notation and verification techniques for secure evolving systems presented in this deliverable aim to be usable in the context of the various change dimensions and change perspectives mentioned above.

## 3.4 Change Schedule - Model Level

Changes might occur at any time during a system's life-cycle. On the model level we define change and time in terms of change schedule as following: (i) Past change, (ii) Current change, and (iii) Future change.

Past change denotes changes already committed and employed in the system, i.e. past system evolution. Such changes happens for many reasons, most often because of circumstances resulting in a number of patches to a system, often also applied within a short time. These changes are harder to backtrack and more permanent than the current and future changes (for GP, applying several changes during pre-issuance without explicitly checking each change before applying the next, where the previous change becomes past change when immediately applying an additional change). For security analysis purposes it is important to differentiate between the status of changes according to a change schedule. For past changes, it only makes sense to execute security analysis to discover security flaws or vulnerabilities arising as a consequence to the already employed changes. Any countermeasures to such changes will be in form of a new changes such as either a patch to deal with the security problems caused by past changes. Past changes occurs at time $t_0 - x$, where $t_0$ denotes the current time and $x \in P$.

Current change refers to changes currently being committed to the system, during the actual addition process. Such changes can be abandoned and altered while being committed as a consequence of e.g. a security-driven change analysis. The main purpose of such an analysis is to merge the change into the system on an abstract and a prior manner, such that potential security flaws or functionality problems can be discovered before actually committing the change. In other words, current changes occurs at time $t_0$.

Future change covers all planned changes and are used to express the kind, perspective and other change specific details (see Section 7.9 for examples of change specific details of interest) of future permitted changes. Such can be used to prevent known change problems, e.g. change types that already have proven to introduce security or functionality problems in the system. Future changes happens at time $t_0 + y$, where $y \in P$.

# 4 UMLsec + Change: Meta-Model

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

This profile, UMLseCh, is a further extension of the UML profile UMLsec in order to support system evolution in the context of model-based secure software development with UML. It is a "light-weight" extension of the UML in the sense that it is defined based on the UML notation using the extension mechanisms stereotypes, tags, and constraints, that are provided by the UML standard. For the purposes of this document, by "UML" we mean the core of the UML 2.0 which was conservatively included from UML 1.5[1].

As such, one can define the meta-model for UMLsec and also for UMLseCh by referring to the meta-model for UML and by defining the relevant list of stereotypes and associated tags and constraints. The meta-model of the UMLsec notation was defined in this way in [Jür05a]. In this section, we define the meta-model of UMLseCh in an analogous way.

In its current version, the UMLseCh notation is divided in two parts: one part intended to be used during abstract design, which tends to be more informal and less complete in its use and is thus particularly suitable for abstract documentation and discussion with customers (cf. Section 4.1), and one part intended to be used during detailed design, which is assumed to be more detailed and also more formal, such that it will lend itself towards automated security analysis (cf. Section 4.2).

## 4.1 UMLseCh: Abstract Design

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

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

We distinguish between past, current and future change as described in Section 3.

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

The abstract stereotypes makes up three refinement levels, where the upper level stereotype are « change ». « change » is the parent stereotype, can be attached to subsystems and are used across all UML diagrams. The meaning of the stereotype is the annotated modelling element and all its sub-elements has or is ready to undergo change.

« change » is refined into the three change schedule stereotypes: (i) « past_change » representing changes already made to the system (typically between two system versions), (ii) « current_change » representing changes currently being made to a system, and (iii) « future_change » specifying the future allowed changes, as described in Section 3.

To track and ensure controlled change actions one needs to be explicit about which model elements that permits change and what kind of change that is permitted on a particular model element. For example, it should not be allowed to introduce audit on data elements that are private or otherwise sensitive, which is annotated using the UMLsec stereotype « secrecy ». To avoid such conflict, security analysis must be undertaken. This can partly bee addressed by the UMLsec tool-set and are partly work in progress.

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

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

**Past and current change** The « new » stereotype is attached to a new system part that is added to the system as a result of a functionality-driven or a security-driven change. For security-driven changes, we use the UMLsec stereotypes secrecy, integrity and authenticity to specify the cause of security-driven change; e.g. that a component has been added to ensure the secrecy of information being transmitted. This piece of information allows us to keep track of the reasons behind a change. Such information is of particular importance for security analysis; e.g. to determine whether or which parts of a system (according to the associated dependencies tag) that must be analysed or added to the target of evaluation (ToE) in case of a security assurance evaluation.

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

The tagged value: $\{$dependencies=yes/no$\}$ is used to document whether there is a dependency between other system parts and the new/modified system part. At this point in the work, we envision changes to this tag, maybe even a new stereotype to keep track of exactly which system parts that depends on each other. However, there is a need to gain more experience and to run through more examples to make a decision on this issue, as new stereotypes should only be added if necessary for the security analysis or for the security assurance evaluation. Note that the term dependencies are adopted from ISO 14508 Part 2 (Common Criteria) [Com07].

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

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

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

**Summary** Figure 4.1 lists the abstract UMLseCh change stereotypes and Figure 4.2 lists the abstract UMLseCh tagged values.

A summary of the abstract notation metamodel can be found in Figure 4.3.

| Stereotype | Base Class | Associated stereotypes | Tags | Constraints | Description |
|---|---|---|---|---|---|
| change | subsystem | | | | specifies that changes are allowed or changes that have been made to the subsystem |
| past_change | subsystem | change | | | specifies an already committed change (past) |
| current_change | subsystem | change | | | specifies an ongoing change |
| future_change | subsystem | change | | | specifies that future changes are permitted |
| new | all | past_change current_change | dependencies, version_number | | specifies addition of a new element (past or current) |
| modified | all | past_change current_change | dependencies, version_number | | specifies modification to existing element (past or current) |
| deleted | all | past_change current_change | | | specifies removal of element (past or present) |
| allowed_add | all | future_change | dependencies, version_number | | specifies that adding new elements are permitted (future) |
| allowed_modify | all | future_change | dependencies, version_number | | specifies that modifying existing elements are permitted (future) |
| allowed_delete | all | future_change | | | specifies that deleting existing elements are permitted (future) |

Figure 4.1: UMLsecCh abstract design stereotypes

| Tag | Stereotype | Type | Multip. | Description |
|---|---|---|---|---|
| dependencies | new, modified, allowed_add, allowed_modify | all | * | specifies whether there are dependencies with other elements |
| version_number | new, modified | all | * | specifies the change number |

Figure 4.2: UMLsecCh abstract design tags

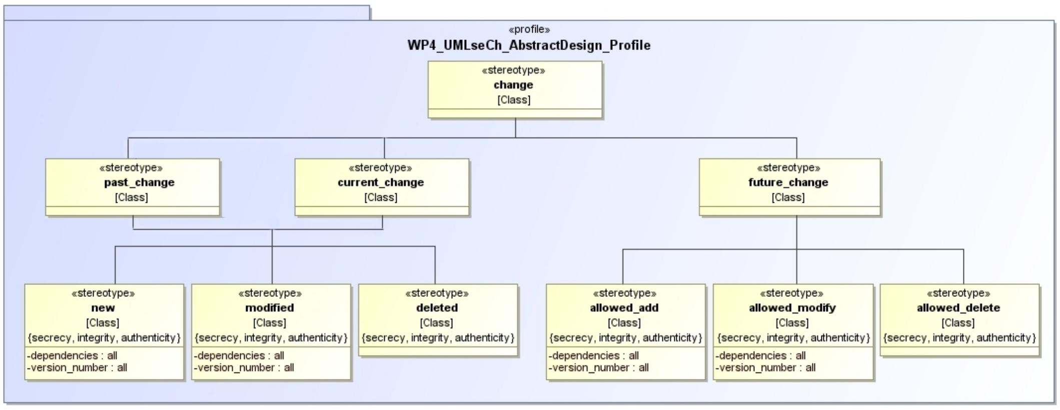

«profile»
WP4_UMLseCh_AbstractDesign_Profile
«stereotype» change [Class]
«stereotype» past_change [Class]
«stereotype» current_change [Class]
«stereotype» future_change [Class]
«stereotype» new [Class] {secrecy, integrity, authenticity}
-dependencies : all
-version_number : all
«stereotype» modified [Class] {secrecy, integrity, authenticity}
-dependencies : all
-version_number : all
«stereotype» deleted [Class] {secrecy, integrity, authenticity}
«stereotype» allowed_add [Class] {secrecy, integrity, authenticity}
-dependencies : all
-version_number : all
«stereotype» allowed_modify [Class] {secrecy, integrity, authenticity}
-dependencies : all
-version_number : all
«stereotype» allowed_delete [Class] {secrecy, integrity, authenticity}

Figure 4.3: UMLseCh metamodel for the abstract notation

### 4.1.1 Examples - Abstract Change

This section demonstrates the use of the abstract change notation to annotate past, current and future changes. The goal of the annotation is to provide information as input to conflict detection and resolution analysis (for security-driven changes), security analysis (for fulfilment of required security properties) and security assurance evaluation.

**Future Change** Future change notation is used to specify allowed changes. Often, during development and design the permitted future changes would be added to the model e.g. to avoid repeating mistakes made in past and current changes.

Figure 4.4 shows a simple subsystem consisting of a sender node and a receiver node with associated components. The only allowed action in the subsystem is for the sender to send data to the receiver. This can be an example of the simplicity that one can have very early in the design phase and which gets extended as the design phase moves forwards and the design matures.

We use the UMLseCh stereotype « change » together with the UMLseCh « future_change » to specify that it concerns future changes. The change types per-

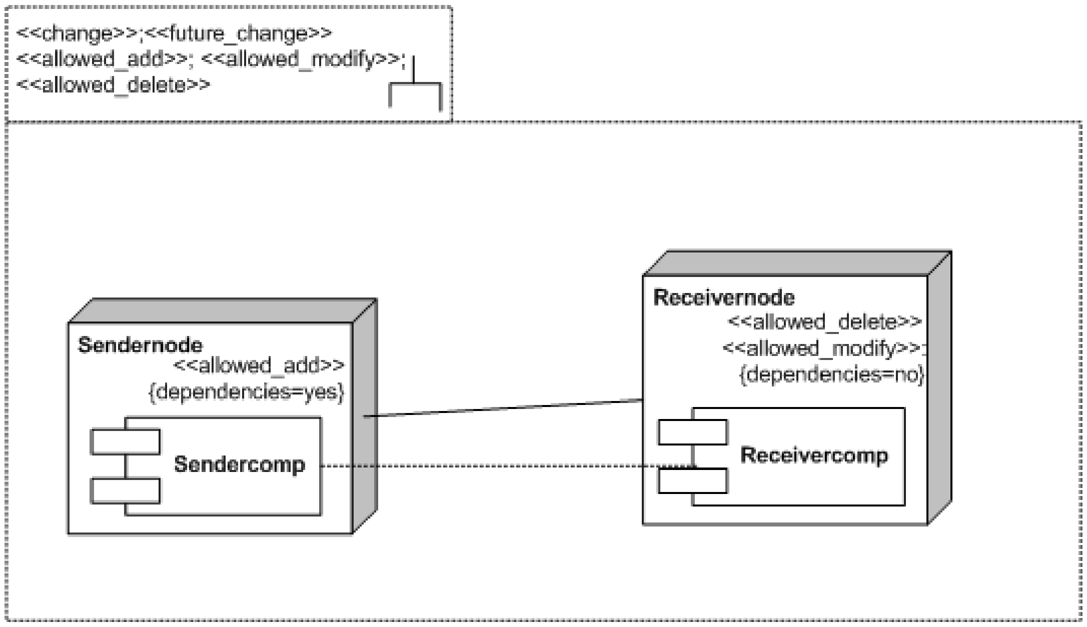


Figure 4.4: Annotating permitted future change using abstract change notation

mitted on the subsystem are all of adding a new element, modifying an existing element and deleting elements. Furthermore, it is only allowed to add and modify elements associated with the sender node. Elements in the receiver node can also be deleted, as can be seen in the figure.

Somewhat later in the design, the subsystem has matured and the associated classes have been identified and specified. Figure 4.5 shows the updated design, with the sender and receiver classes and interfaces included. As can be seen in the figure, the allowed future changes have been revised and deleting elements is no longer an allowed future change.

**Past Change** Past changes are those that are already committed to the system. Note that we distinguish between past and current change, which is of particular importance for tool support, as we will see demonstrated later. In cases where future changes are specified prior to change, the future changes are used to restrict and control past changes.

Figure 4.6 gives a simple example of using the abstract « past_change » notation to specify that a new element has already been employed in the subsystem. The sub-

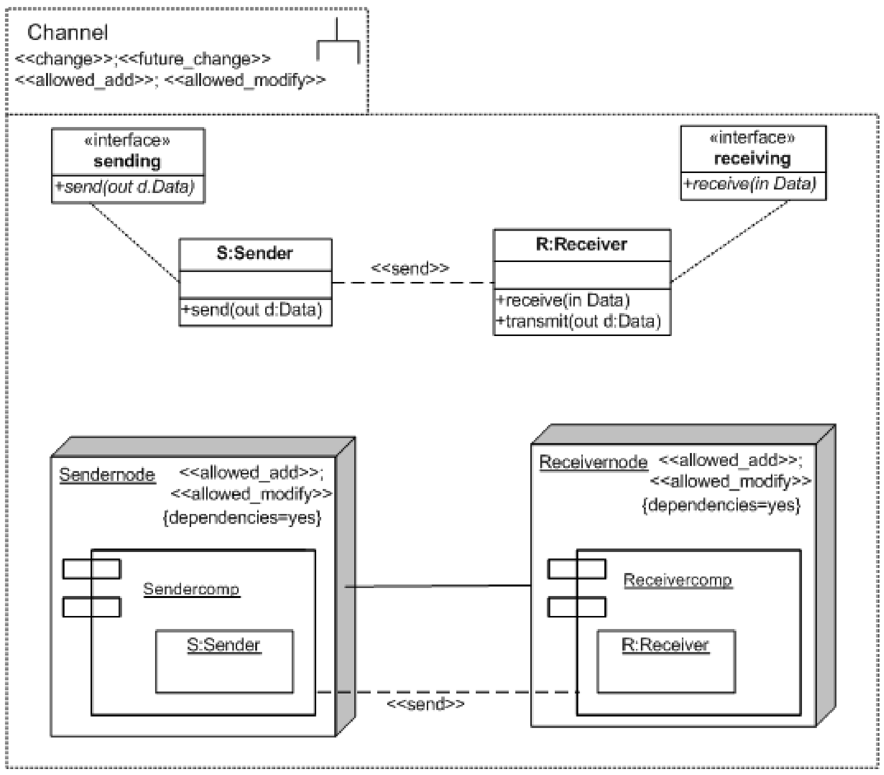


Figure 4.5: Annotating permitted future change using abstract change notation

system is the same simple system comprised of a sender and receiver node and components, as used for annotating future change. As can be seen in the figure, the subsystem is annotated with the stereotypes « change » and « past_change » This should be interpreted in the following way. A change has already been deployed in the subsystem. The change is a past introduction of a new element to the subsystem. We then examine the elements in the subsystem and see that the added element is the “encryption service”, which has no dependencies on the other elements in the subsystem. We can also deduce that the added element is new, as the version number given is 0 (0 is given as the version number when the element is first introduced). As also can been seen in the figure, the encryption service has three associated operations: encrypt, decrypt and sign.

**Current Change** Current change represents ongoing modifications to a subsystem and differs from past changes in that the change has not yet been incorporated into the system design or code, meaning that security analysis can be applied to the change to discover e.g. any undesired consequences of the change a priori. Current changes can most often easily be rolled-back, while this is more difficult in the case of past and already committed changes. In cases where future changes are

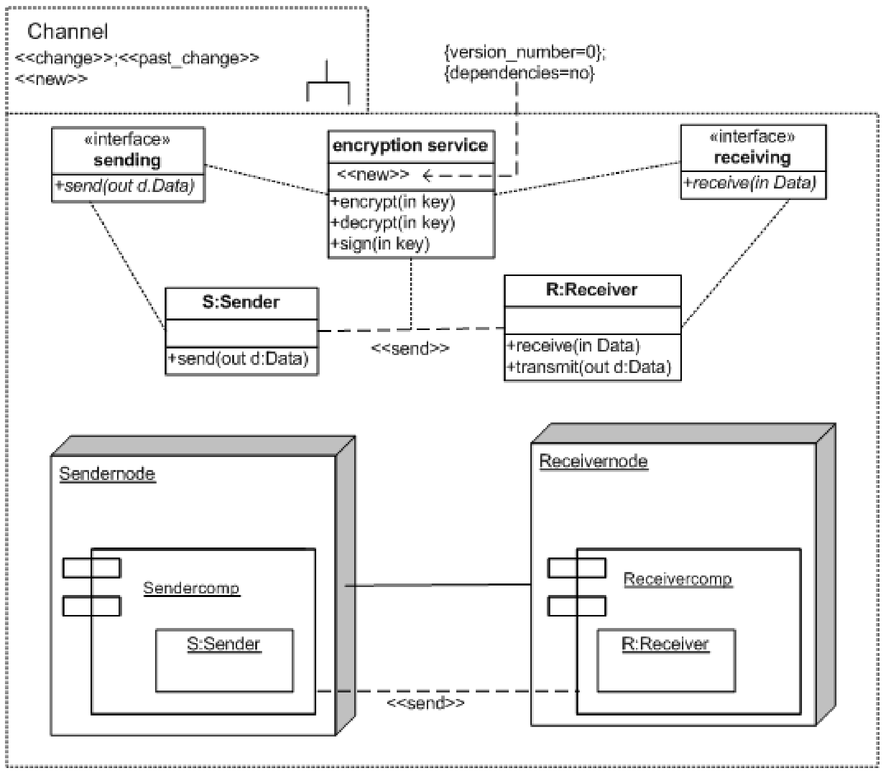


Figure 4.6: Using abstract change notation to annotate the introduction of a new element

specified prior to change, the future changes are used to restrict and control current changes.

Figure 4.7 shows the same subsystem, but at a later stage. It has been discovered that there is a need to add functionality to verify signatures and this change is about to be introduced into the subsystem. At this stage, security analysis can be applied and any undesired consequences of the change can be discovered and fixed a priori. The version number is now set to 1, as the semantic meaning of '1' is the first modification to an already existing element.

### 4.1.2 Abstract change stereotypes - Semantics

This subsection elaborates on the ascribed meanings of the abstract change stereotypes as defined and described in Section 4.1. The abstract changes stereotypes deals with higher levels change artefacts and allow one to address change from both a general and specific levels. The main purpose of these stereotypes is to document, trace and analyse implications of evolution and revolution kinds of changes.

The abstract stereotypes are categorized into three groups and structured into three

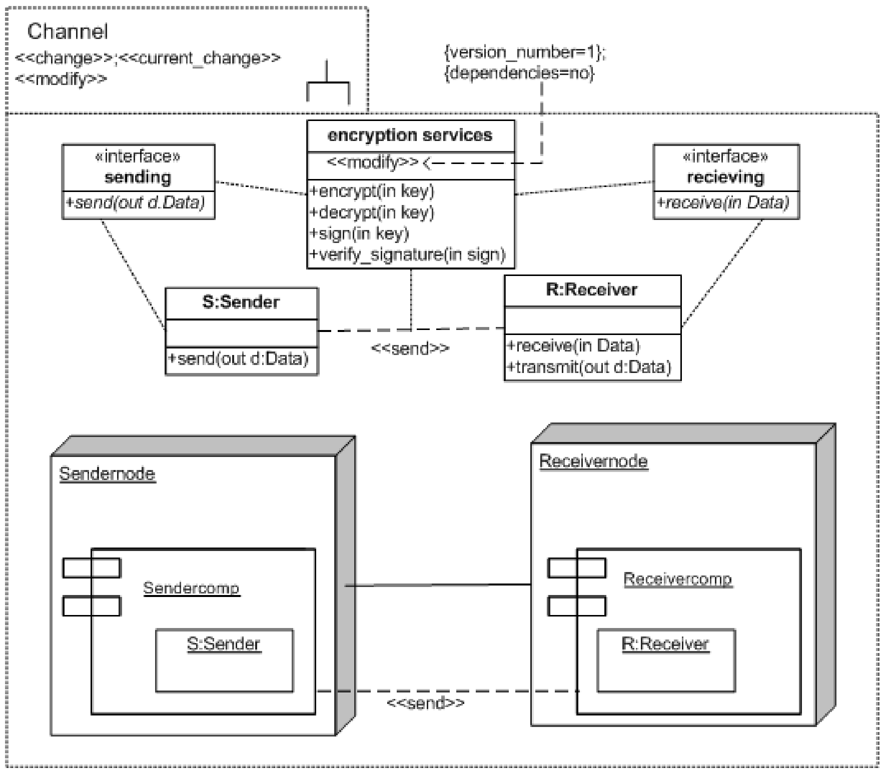


Figure 4.7: Using abstract change notation to annotate modification of an exisiting element

levels, where lower levels represent refinements of the upper levels. For example, new is a refinement of current_change, which again is an refinement of change. The three categories of abstract change stereotypes are: (i) past change, (ii) current change, and (iii) future change.

Past changes document events already happened and are important for documentation purposes and to enable a structured after-the-fact analysis of the implications that these changes has had on the core system functionality and the desired security properties of the system. Such analysis may result in not acceptable results, meaning that the change or parts of the change should be undone and that roll-back actions are required.

Current changes are used to evaluate required or desired modifications to the system, such as its application and application data. The stereotype annotates design models with the proposed change in a way that the meaning of the change on the affected system parts can be analysed. The meaning can be implications (indirect consequences) or direct undesired modification of system or security behaviours of the system. As a minimum, the security properties should be preserved through any current type of change, if not the change must be re-considered.

Future changes denotes modifications to system parts or the security behaviour of the system that are permitted, along with documenting up front which system parts that specific changes are permitted for. Acceptable future changes can be derived from the result of past and current changes (it is wise to deny changes that already have showed to introduce undesired consequences) or document changes that the various parts will be able to handle in a manner that avoids, as a minimum, breach of the security properties of the system.

**change** is the most abstract change stereotype and is used to mark a past, current or future change. The stereotype allows for tracing change throughout the system life-cycle. The change refines into. past_change; current_change; future_change.

**past_change** is a refinement of stereotype change used to document changes already applied to the system. This stereotype documents changes already introduced to the system and are used by tools to check for preservation of system security properties after the fact (after the change has already been applied). The stereotype past_change refines into: new; modify; delete.

**current_change** is a refinement of stereotype change used to denote present change. This stereotype should preferable be used to "test" the consequences of proposed changes a priori and represent a change control tool. The stereotype past_change refines into: new; modify; delete.

**future_change** is a refinement of stereotype change used to model future permitted change. The stereotype can be used to prevent future undesired changes (already known as a result of unsuccessful past or current change analysis) and to specify up-front how a system preferable should evolve. This stereotype is part of the change control tool. The stereotype future_change refines into: allowed_add; allowed_modify; allowed_delete.

### 4.1.3 Abstract change stereotypes - Tool support

The abstract change stereotypes in the UMLseCh profile enable an easy and effective way to identify and analyse the specific parts of a system that has or is about the undergo change. These stereotypes also makes it possible to learn from previous change cases and learn from experience and thus avoid introducing known change problems in the future. Tool support for the abstract stereotypes serves three purposes: (i) detecting and repairing past changes, (ii) assisting current change processes and (iii) prepare the system for future change processes, ensuring a continuous and controlled system change.

The most abstract change stereotype change is attached to subsystems and used to specify that change, not specifying the type, is permitted in that particular subsystem. This stereotype does not imply any kind of change and is merely used to enable the tool to easily identify the parts of a system that have been, are undergoing or permits change. Consequently, the change stereotype enables the tool to narrow the analysis scope (systems parts not concerned with change does not need to be analysed).

### 4.1.4 Security Checking Methodology after the addition of elements

If new elements have already been introduced to the system, the impact of this action should be investigated. The tool does this in the following way:

- Start with the before model of the system (the model before the change was introduced into the system)
- Introduce the new element into the before model
- Identify affected elements in the relevant subsystem
- Identify security properties of the affected elements
- Create the after model
- Analyse the after model against the before model (delta analysis):
  - Security analysis of new element locally in the relevant subsystem – check for preservation of security properties after composition with the existing system
  - Compare the before and the after models – check if security properties of before model is preserved in after model for the relevant subsystem

### 4.1.5 Security Checking Methodology after the modification of elements

Modifying existing elements follows much the same analysis process as for introducing new modelling elements into the system. The only difference is that existing elements may have dependencies across subsystems and these must be identified and analysed, in addition to checking the security implications locally in the relevant subsystem.

The process that the tool follows for analysing modification type of change is the following:

- Start with the before model of the system (the model before the modification was introduced into the system)
- Introduce the modification into the before model
- Identify affected elements in the relevant subsystem
- Identify any dependencies (both in the local subsystem and across the system specified by the tag {dependencies})
- Identify relevant security properties of the affected elements
- Create the after model
- Analyse the after model against the before model (delta analysis):
  - Security analysis of modified element locally in the relevant subsystem – check for preservation of security properties after modification
  - Compare the before and the after models – check that the security properties of before model is preserved in after model for the relevant subsystem
  - Security analysis of dependencies – check for preservation of security properties of dependent model elements

### 4.1.6 Security Checking Methodology after the deletion of elements

Security analysis for delete type of change is carried out in the following way by the tool:

- Start with the before model of the system
- Identify affected elements locally in the relevant subsystem
- Identify relevant security properties of the affected elements
- Identify any dependencies (both in the local subsystem and across the system specified by the tag {dependencies})
- Remove the element from the before model to create the after model
- Analyse the after model against the before model (delta analysis):
  - Analyse for preservation of security properties locally in the subsystem
  - Compare the before and the after models – check that security properties of before model is still preserved in the after model locally in the subsystem

– Security analysis of dependencies – check for preservation of security properties after removing modelling element of dependent model elements

Deleting elements from a system should be done with care as such changes may result in undesirable and also hidden security weaknesses if not done in a controlled manner. Of particular importance are to check any dependencies on other system elements. E.g., if the key generation element is removed without updating the dependent cryptographic operations the operation will fail or worse end in an introduction of static encryption keys.

#### 4.1.7 Security Checking Methodology for Future Kinds of Change

Future changes should learn from unsuccessful past and current changes. This means that tool support for future kind of changes should include a learning capability that ensures that future permitted changes do not conflict with prior change experience. I.e., the model should not permit changes that have previously proven to introduce security weaknesses or problems with the system functionality. Allowed addition, modification and deletion of elements in a system will therefore be updated as change experience is gathered. The security analysis process for each follows that of above.

## 4.2 UMLseCh: Concrete Design

We further extend UMLsec by adding so called "concrete" stereotypes: these stereotypes allow to precisely define substitutive (sub) model elements and are equipped with constraints that help ensuring their correct application.

Figure 4.8 shows the stereotypes defining table. The tags table is shown in Figure 4.9.

The UMLseCh metamodel is summarized in Figure 4.10.

### 4.2.1 Description of the notation

In the following, we describe informally the semantics of each stereotype.

*substitute*

The stereotype substitute attached to a model element denotes the possibility for that model element to evoluate over the time and what are the possible changes. It has three associated tags, namely {ref}, {substitute} and {pattern}.

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

Figure 4.8: UMLsecCh concrete design stereotypes

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

Figure 4.9: UMLsecCh concrete design tags

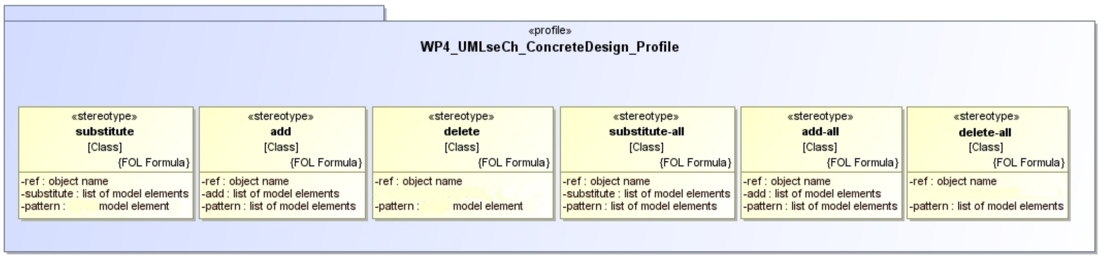


Figure 4.10: UMLseCh metamodel for the concrete notation

These tags are of the form { ref = CHANGE-REFERENCE }, { substitute = MODEL-ELEMENT } and { pattern = CONDITION }. The tag {ref} takes a string as value, which is simply used as a reference of the change. The value of this tag can also be considered as a predicate and take a truth value to evaluate conditions on the changes, as we explain further in this section. The tag {substitute} has a list of model element as value, which represents the several different new model elements that can substitute the actual one if a change occurs. An element of the list contained in the tagged value is a model element itself if it can fit in the tag notation (e.g. a stereotype, "{substitute = « stereotype »}"). If the model element cannot fit in the tag notation (e.g. a diagram), it is placed in a package and this package name is the element of the list contained in the tagged value. The last tag, {pattern}, is optional. If the model element to change is clearly identified by the syntactic notation, i.e. if there is no possible ambiguity to state which model element is concerned by the stereotype « substitute », the tag pattern can be omitted. On the other hand, if the model element concerned by the stereotype « substitute » is not clearly identifiable, the tag pattern must be used. This tag has a model element as value, which represents the model element to subsitute if a change occurs. In order to identify the model element precisely, we can use, if necessary, the abstract syntaxe of UMLsec, defined in [Jür05a].

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

« substitute »

{ ref = encrypt-link }

{ substitute = encrypted }

{ pattern = Internet }

to the link concerned by the change.

The stereotype « substitute » also has a constraint formulated in first order logic. This constraint is of the form [CONDITION]. As mentioned earlier, the value of the tag {ref} of a stereotype « substitute » can be used as the atomic predicate for the constraint of another stereotype « substitute ». The truth value of that atomic predicate is true if the change represented by the stereotype « substitute » for which the tag {ref} is associated occured, false otherwise. The truth value of the condition of a stereotype « substitute » then represents whether or not the change is allowed to happen (i.e. if the condition is evaluated to true, the change is allowed, otherwise the change is not allowed).

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

« substitute »

{ ref = encrypt-link }

{ substitute = encrypted }

{ pattern = Internet }

[change]

*add*

The stereotype « add » is similar to the stereotype « substitute » but, as its name indicates, denotes the addition of a new model element. It has three associated tags, namely {ref}, {add} and {pattern}. The tag {ref} has the same meaning as in the case of the stereotype « substitute », as well as the tag {add} (which here is the equivalent of the tag {substitute}, i.e. a list of model elements that we wish to add). The tag {pattern} has a slightly different meaning in this case. Indeed, while with stereotype « substitute », the tag {pattern} represents the model element to substitute, with the stereotype « add » it does not represent the model element to add, but the model element concerned by the addition, as for evident reasons, we cannot associate the stereotype « add » to a model element that does not exist yet.

The stereotype « add » is a syntactic sugar of the stereotype « substitute », as a stereotype « add » could always be represented with a stereotype « substitute ». Nevertheless, the semantic of « add » will change slighly from one situation to another. Indeed, in the case of, for example, a class diagram, adding a new method to a class will consist of substituing the set of methods of that class with a new set that is the copy of the former set in which we add the new method. Formally, if $s$ is the set of methods and $m$ the new method, the new set of methods is:

$$s' = s_0 \cup \{m\}$$

where $s_0$ is a copy of $s$. However, in the case of an activity diagram, adding a new node on the flow will consist on substituing an activity edge (the one placed were we want to add the new node) with an activity subdiagram. In the simple case of adding a new node, this subdiagram will simply be the new node together with the two activity edges connecting this node with the previous ones and the next ones on the flow. In the case of adding a more complex subdiagram, the activity edge will be substituted with this subdiagram in which we replace the entry point by the previous node of the substituted activity edge and the exit point by the next node of the substituted activity edge. Because of this inconsistency of the semantics, the use of the stereotype « add » will be restricted in certain diagrams, namely the state diagrams and the activity diagrams. For these particular diagrams, « add » will be limited to the addition of model elements that do not imply any substitution of activity edge, as stereotypes for example. For the addition of model elements that imply a substitution of an activity edge, the stereotype « substitute » will be used.

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

*delete*

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

The stereotype « delete » is a syntactic sugar of the stereotype « substitute », as a stereotype « delete » could always be represented with a stereotype « substitute ». Nevertheless, as for the stereotype « add », the semantic of « delete » will change slighly from one situation to another. In the case of a class diagram, deleting a method will consist of substituing the set of methods of that class with a new set that is the copy of the former set in which we remove the method. Formally, if $s$ is the set of methods and $m$ the method to delete, the new set of methods is:

$$s' = s_0 \setminus m$$

where $s_0$ is a copy of $s$. In the case of an activity diagram, deleting a node will consist of substituing the node and the activity edges connecting the node with the previous ones and the next ones on the flow by activity edges connecting the previous nodes of the deleted node with the next nodes of the deleted node. Again, as for the stereotype « add » this inconsistency of the semantics will lead to a restricted use of the stereotype « delete » in the state diagrams and the activity diagrams. For these particular diagrams, « delete » will be limited to the deletion of model elements that do not imply any substitution of activity edge, as stereotypes for example. For the deletion of model elements that imply a substitution of an activity edge, the stereotype « substitute » will be used.

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

*substitute-all*

The stereotype « substitute-all » is an extention of the stereotype « substitute » that can be associated to a (sub)model element or to a whole subsystem. It denotes the possibility for **a set of (sub)model elements** to evoluate over the time and what are the possible changes. The elements of the set are sub elements of the element to which this stereotype is attached (i.e. a set of methods of a class, a set of links of a Deployment diagram, etc). As the stereotype « substitute », it has the three associated tags {ref}, {substitute} and {pattern}, of the form { ref = CHANGE-REFERENCE }, { substitute = MODEL-ELEMENT } and { pattern = CONDITION }. The tags {ref} and {substitute} have the exact same meaning as in the case of the stereotype « substitute ». The tag {pattern}, here, does not represent one (sub)model element but **a set of (sub)model elements** to substitute if a change occur. Again, in order to identify the list model elements precisely, we can use, if necessary, the abstract syntaxe of UMLsec, defined in [Jür05a].

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

« substitute-all »

{ ref = encrypt-all-links }

{ substitute = « encrypted » }

{ pattern = « Internet » }

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

To illustrate the use of the parametrisation in the stereotype « substitute-all », we have the following example. Assume that we would like to substitute all the secrecy tags in the stereotype « critical » by the integrity tag, we can attach:

« substitute-all »

{ ref = secrecy-to-integrity }

{ substitute = { integrity = X } }

{ pattern = { secrecy = X } }

to the model element to which the stereotype « critical » is attached.

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

*add-all*

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

*delete-all*

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

### 4.2.2 Semantics

In the following, we give the semantics of the UMLseCh stereotypes. This semantics is based on the fact that every model element of a system can be expressed precisely using the abstract syntax of UMLsec, defined in [Jür05a].

*substitute*

Assume that { ref = $R_1$ } and there exists other UMLseCh stereotypes with references $R_2, \cdots, R_n$. The application of « substitute » on a UMLsec (sub)model element $M$ of a subsystem $S$ (specified in { pattern = $M_1$ }) yields:

In case [Constraint $R_2, \cdots, R_n$] evaluates to true and the type of $M$ is the same as the type of $M'_1, \cdots, M'_n$, as specified in { substitute = $M'_1, \cdots, M'_n$ }, then it returns the list $M'_1, \cdots, M'_n$. Otherwise it returns $M$.

*add*

The « add » semantics is similar, but here, the tag { pattern = $M$ } does not refers to a (sub)model element, but to a list of (sub)model elements that belongs to the model element concerned by the addition (for example, the list of attributes of a class, the list of classes of a class diagram etc.)

Its semantics is then:

In case [Constraint $R_2, \cdots, R_n$] evaluates to true and the type of the elements of the list $M$ is the same as the type of $M'_1, \cdots, M'_n$, as specified in { add = $M'_1, \cdots, M'_n$ }, then it returns the list $M :: M'_1, \cdots, M :: M'_n$. Otherwise it returns $M$.

*delete*

For the stereotype « delete », the tag { pattern = $M$ } refers to a (sub)model element.

Its semantics is:

In case [Constraint $R_2, \cdots, R_n$] evaluates to true, it returns the empty set $\emptyset$. Otherwise it returns $M$.

*substitute-all*

The semantics of « substitute-all » is similar to the semantics of « substitute », but here, the tag { pattern = $M_1, \cdots, M_n$ } does not refer to a (sub)model element but to a list of (sub)model elements.

Its semantics is then:

In case [Constraint $R_2, \cdots, R_n$] evaluates to true and the type of the elements of the list $M_1, \cdots, M_n$ is the same as the type of $M'_1, \cdots, M'_m$, as specified in { substitute = $M'_1, \cdots, M'_m$ }, then it returns the list of lists $\{M''_1, \cdots, M''_n\}$

where $M''_i = \{n$ copies of $M'_i\}$

$\forall i : 1 \leq i \leq m$. Otherwise it returns $M_1, \cdots, M_n$.

*add-all*

The semantics of « add-all » is similar to the semantics of « add », but here, the tag { pattern = $M_1, \cdots, M_n$ } does not refer to a (sub)model element but to a list of (sub)model elements.

Its semantics is:

In case [Constraint $R_2, \cdots, R_n$] evaluates to true and the type of the elements of the list $M_1, \cdots, M_n$ is the same as the type of $M'_1, \cdots, M'_m$, as specified in { add = $M'_1, \cdots, M'_m$ }, then it returns the list of list:

$$\{\{M_1 :: M'_1, \cdots, M_n :: M'_1\}, \cdots, \{M_1 :: M'_m, \cdots, M_n :: M'_m\}\}.$$

Otherwise, it returns $\{M_1, \cdots, M_n\}$.

*delete-all*

The semantics of « delete-all » is similar to the semantics of « delete », but here, the tag { pattern = $M_1, \cdots, M_n$ } does not refer to a (sub)model element but to a list of (sub)model element.

Its semantics is then:

In case [Constraint $R_2, \cdots, R_n$] evaluates to true, it returns the empty set $\emptyset$. Otherwise it returns $\{M_1, \cdots, M_n\}$.

### 4.2.3 Examples

**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, one could change the model to ensure sender's data secrecy by adding a « critical » stereotype with the tagged value { secrecy = d } where d represents the sender's data. This possible change can be modelled by adding the stereotype « add » with the tagged values { ref = make-data-secret } and { add = « critical », { secrecy = d } } on the current diagram. Here, the tag { pattern = MODEL-ELEMENT } can be omitted. Indeed, the stereotype « critical » can only be added to a class and the stereotype « substitute » is located inside the class. It is therefore clearly indicated where « critical » should be added.

To ensure data secrecy, 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 » with the stereotype « encrypted ». Again, this change can be modelled by adding the stereotype « substitute » with the tagged values { ref = make-link-secure }, { substitute = « encrypted » } and { pattern = « Internet » } on the link stereotyped « Internet ». The diagram of this example is shown in Figure 4.11.

To ensure that the link is encrypted when the data secrecy is required, we add the following condition to the stereotype with reference make-critical:

[make-link-secure].

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 different changes modelled by the stereotypes. One could indeed change the model by adding the stereotype « critical » on the sender class, i.e. apply the stereotype « add » with reference { ref = make-data-secret }. Another transi-

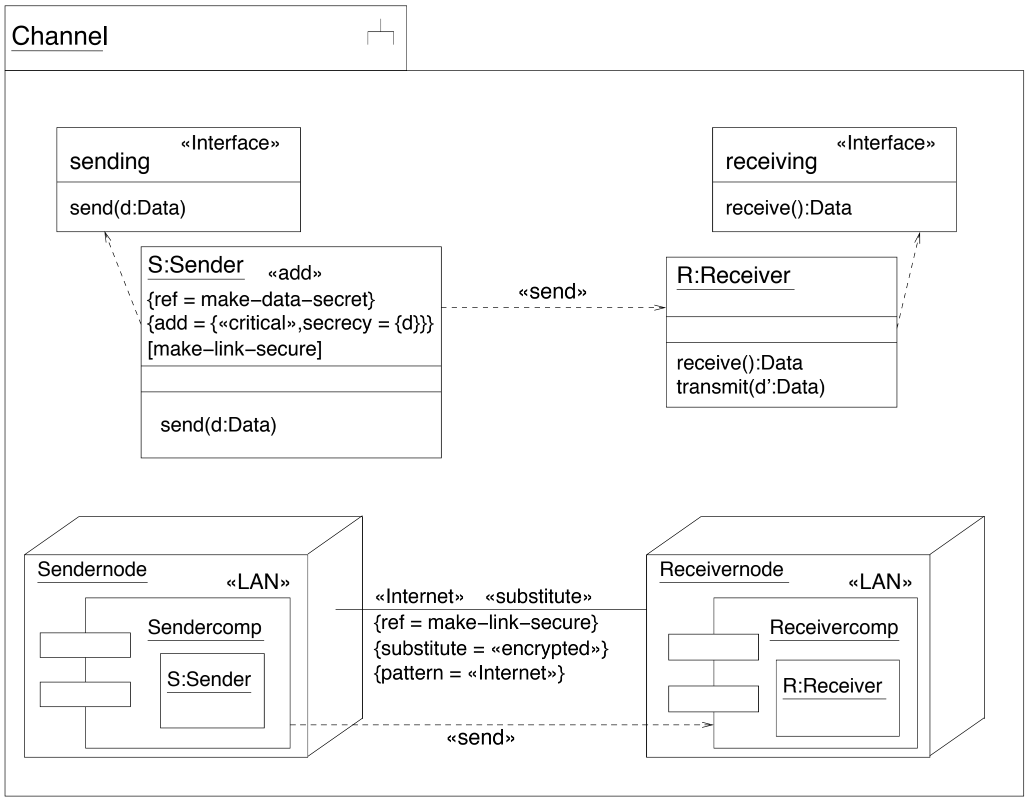


Figure 4.11: First example of evolution

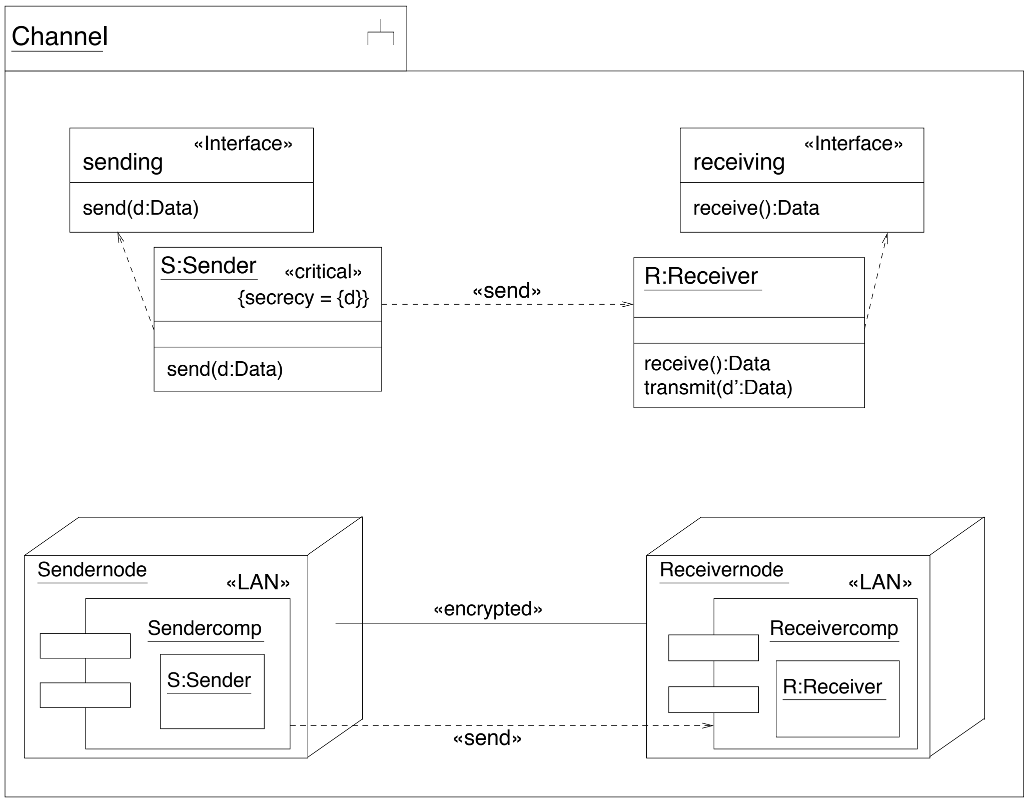


Figure 4.12: One application of the modelled possible changes

tion could be to substitute the stereotype « Internet » with the stereotype « encrypted ». Or finally, one could apply both changes on the model. In this case, the first transition is not allowed. Indeed, if one simply applies the change modelled by the stereotype « add » with reference { ref = make-data-secret }, this transition violates the constraint

[make-link-secure].

On the other hand, the second transition is correct. One could certainly decide to encrypt the link although the secrecy of the data is not requested. Applying both the changes, represented by the stereotypes with reference { ref = make-critical } and { ref = make-link-secure }, is, of course, also allowed. The model resulting from this particular transition is shown on Figure 4.12.

**Booking a flight** For the second example, we use the following scenario. Bob had booked a flight for a business trip. Because of bad weather conditions, the flight is cancelled. In consequence, Bob decides to promptly book a new flight on his mobile device. However, some changes could have happened between the cancelled

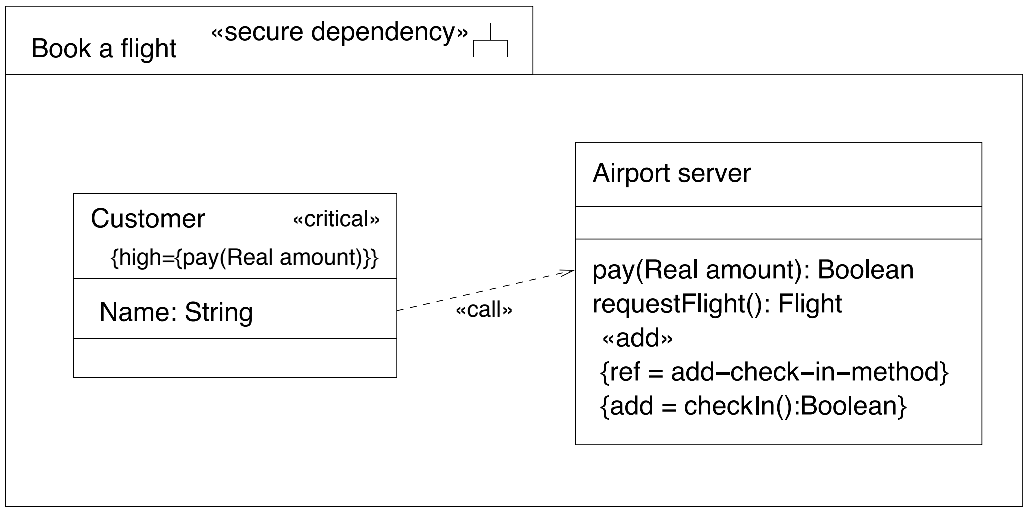


Figure 4.13: Class diagram for the flight booking example

flight and the moment when Bob wants to book a new one. A possible change for the airport system would be to offer a new online check-in service. To add this functionality, one needs to add a check-in method in the class diagram. To model this possible change, we add the stereotype « add » with the tagged values { ref = add-check-in-method } and { add = checkIn():Boolean } on the class diagram. This result is shown in Figure 4.13. Note that we use the syntactic notation defining methods in 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 located inside the class and the model element to add being a method, the place where the model element should be added is implicit and therefore the tag { pattern = MODEL-ELEMENT } can be omitted.

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 booking a flight. 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. This possible change can be modelled with our notation. At first, one needs to model the check-in functionality which will give us a new activity diagram. This new activity diagram can be considered as a sub-diagram of the main flight booking activity diagram. To model the possible change on the main diagram, one then has to add the stereotype « add » with the tagged values { ref = add-online-check-in } and { add = Check-in }. The value of the tag { add = VALUE } represents the package VALUE containing the sub-diagram to add. In this example, the sub-diagram is contained in the package Check-in. Note that the

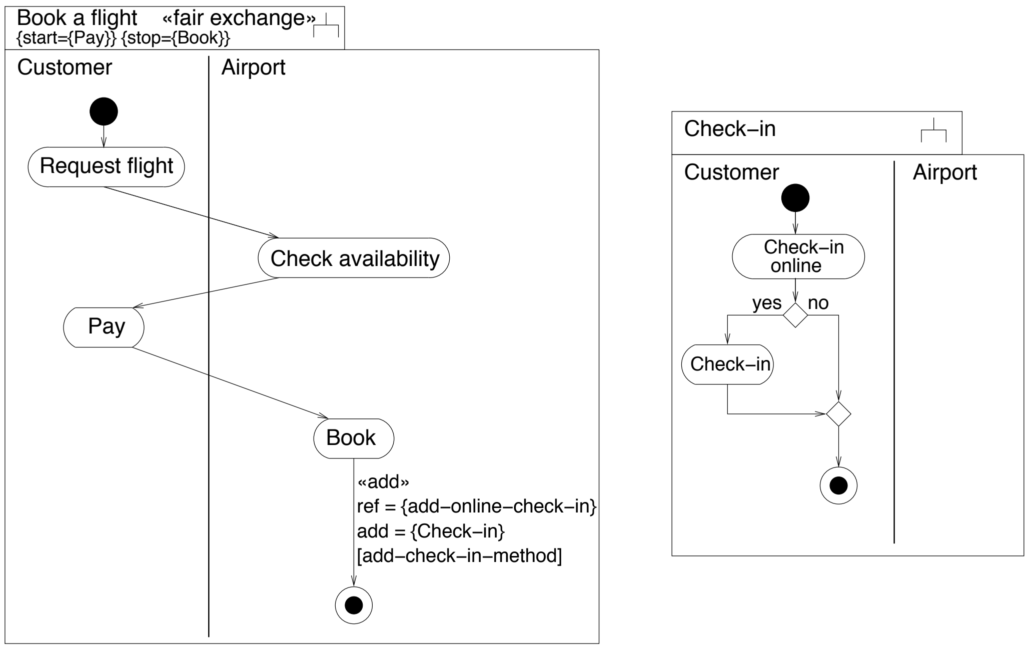


Figure 4.14: Activity diagram for the flight booking example

stereotype « add » is associated with the activity edge linking the action Book and the exit point of the activity diagram. This means that the sub-diagram contained in the package Check-in will be added on this activity edge. In other words, the workflow will go from the action Book to the entry point of the sub-diagram and then from the exit point of the sub-diagram to the exit point of the main diagram. We can fairly assume that the model element concerned by the stereotype « add » is clearly identified and therefore the tag pattern is not requested. The activity diagram of this example is shown in Figure 4.14.

Note that to have the Check-in node on the activity diagram, which means a check-in action in the workflow, it is necessary to have a check-in method in the class diagram. Therefore, we add the following condition to the stereotype with reference add-online-check-in in the UML model:

[add-check-in-method].

In this example, we only have one possible transition per diagram. Figure 4.15 shows the model resulting from the application of the changes on both diagrams. Note that although only one change can happen per diagram, both the changes must happen together, otherwise the constraint:

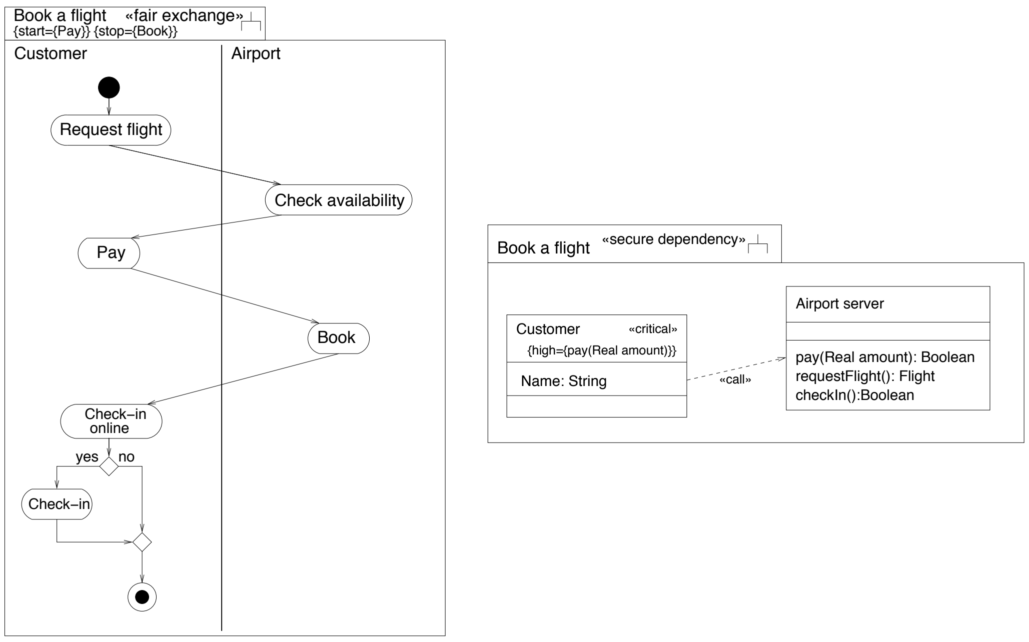


Figure 4.15: Application of the modelled possible change

[add-check-in-method]

will be violated.

**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 ("Simple example"): 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 attended 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, for security reasons, one could require the data to also remain secret. To model this new security requirement, one thus can add a tag { secrecy = X } on each stereotype « critical » having a tag { integrity = X }, where X is the meta-variable representing the data. To model this possible change, instead of adding a stereotype « add » with the related tags on each stereotype « critical », we add a stereotype « add-all » on the subsystem with the tagged values { ref = make-data-secret }, { add-all = { secrecy = X } }, which represents the model element to add, and { pattern = { integrity = X } }, which allows us to represent which model elements are concerned.

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

[make-link-secure].

The diagram of this example and the result of applying the changes modelled by both stereotypes are shown respectively in Figure 4.16 and Figure 4.17.

**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 give the following scenario. A client can exchange data with a server and another client can exchange secret data with a secured server. Both server can also communicate to exchange data. On our actual model, the links are stereotyped « Internet » and therefore do not provide any secrecy of the data. To make the system secure, one has to encrypt the link. This can be done by changing the stereotype « Internet » with the stereotype « encrypted » on the link between the secured server and the client and on the link between the servers. However, the link between the client and the normal (i.e. not secure) server does not need to be encrypted. This change can be modelled with our notation by adding a stereotype « substitute-all » with the relevant tagged values. We add the tagged values { ref = make-link-secure } and { substitute-all = « encrypted » } and for the condition of the tag pattern, we use the abstract syntax of deployment diagrams defined in [Jür05a]. Therefore, to express precisely which links should be affected by the changes, we have:

{ pattern = {*l1* = (nds(*l1*), ster(*l1*)), *l2* = (nds(*l2*), ster(*l2*))} }

where:

nds(*l1*) = (SClientnode, SServernode),
nds(*l2*) = (Servernode, SServernode)
and ster(*l1*) = ster(*l2*) = {« Internet »}.

The diagram of this example is shown in Figure 4.18 and the transition resulting from applying the changes is shown in Figure 4.19.

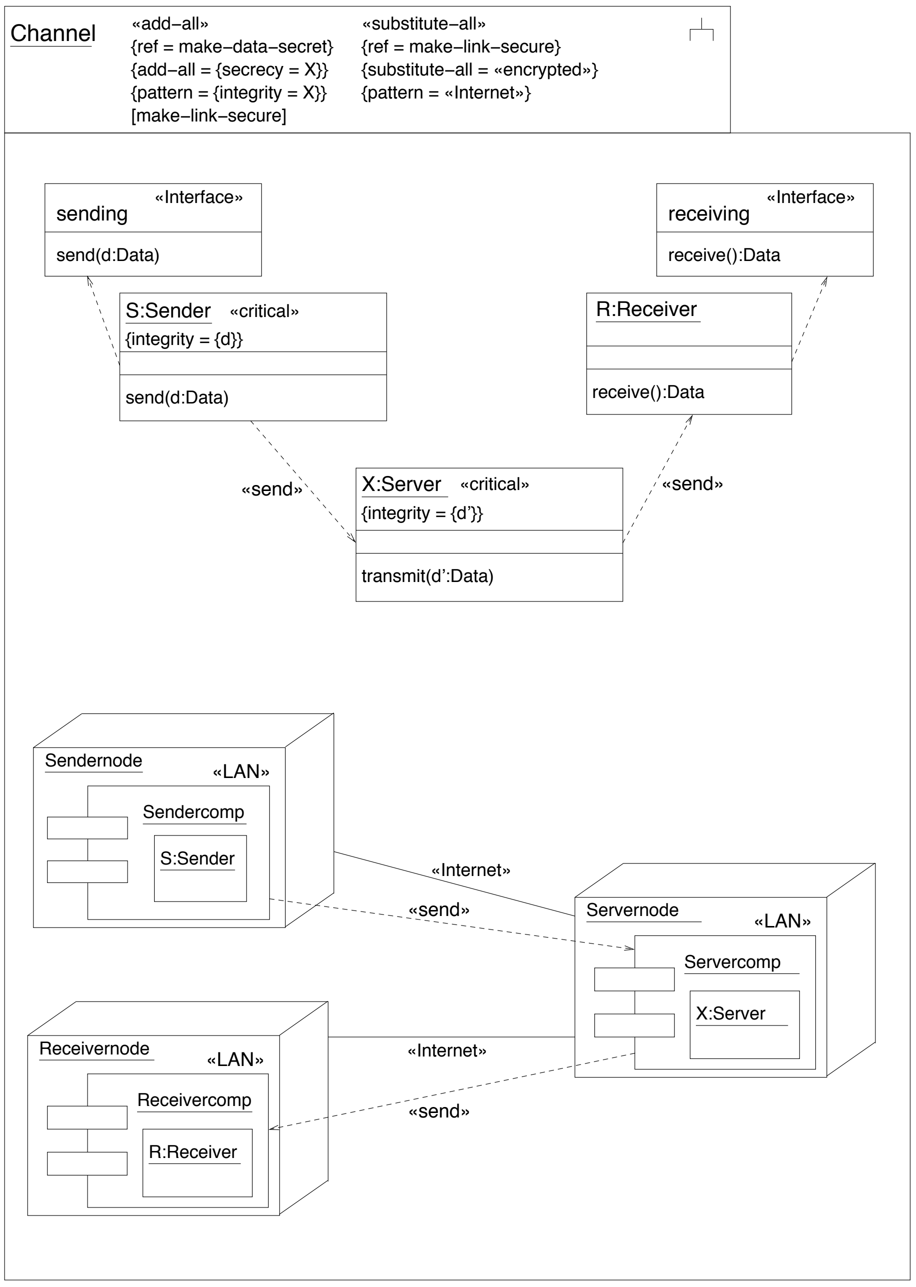


Figure 4.16: Example of use of add-all and substitute-all

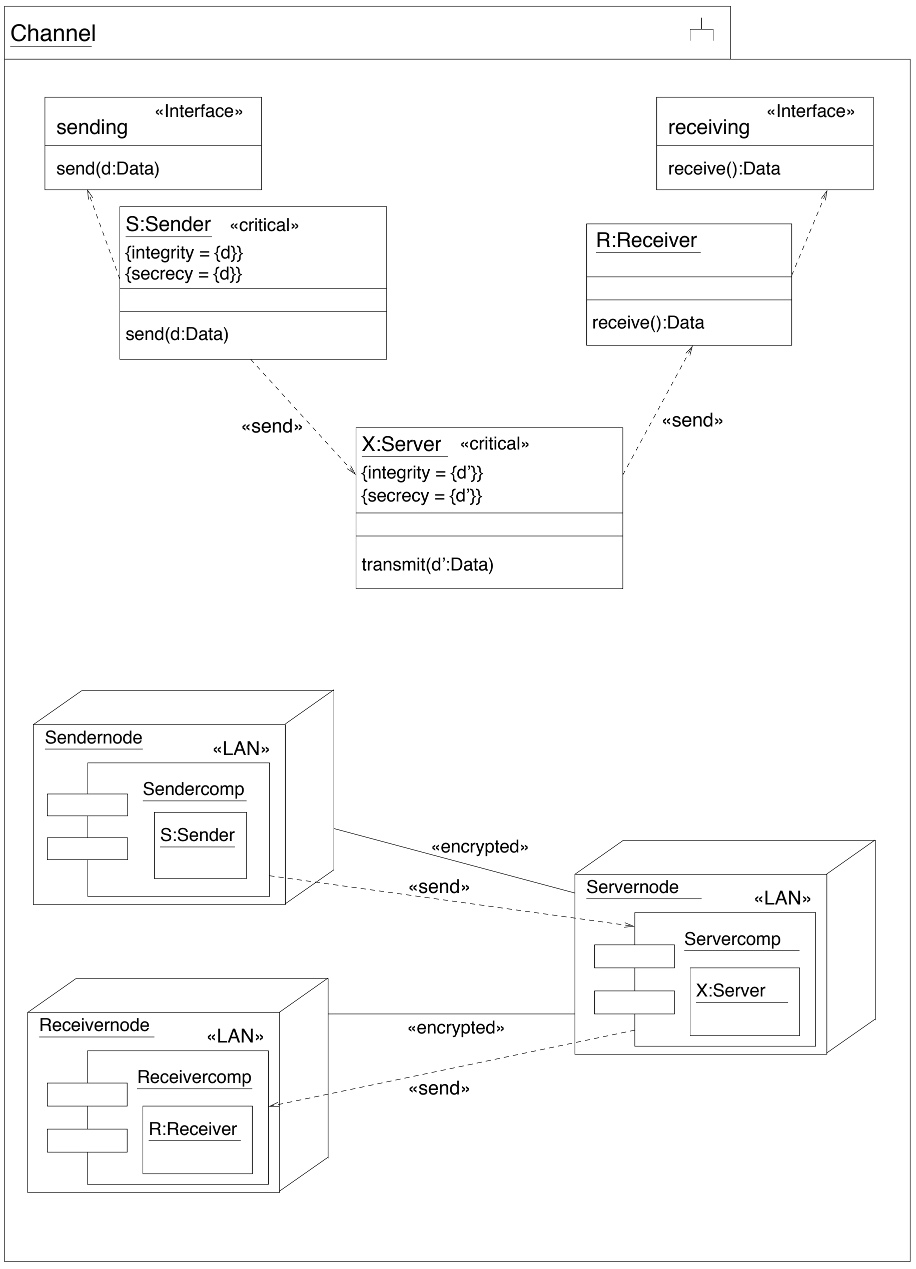


Figure 4.17: Application of the changes

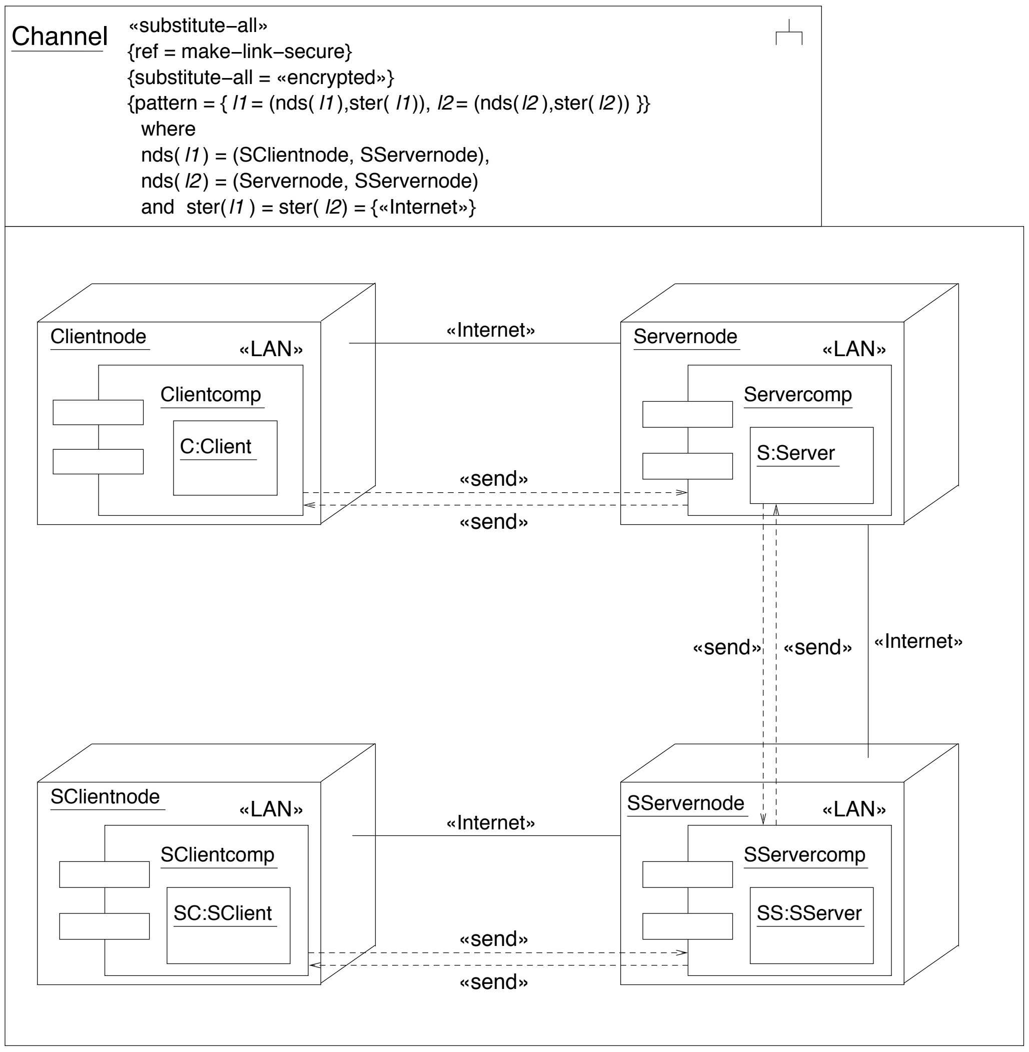


Figure 4.18: Selection of the links for the substitute-all

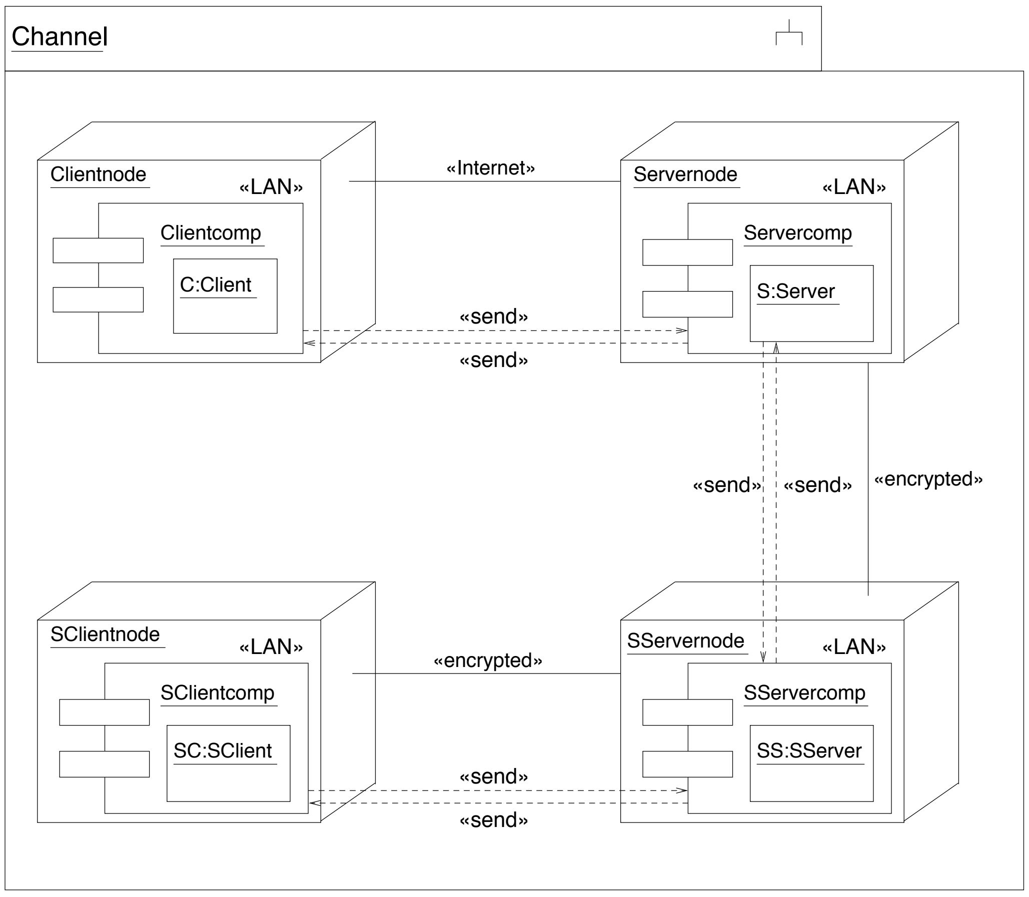


Figure 4.19: Selection of the links for the substitute-all

# 5 Background: Global Platform

We describe in the following the background for the smart cards use case used in the project. Note that with respect to the modelling only some specific aspects should be modelled, in particular to be able to represent the changes and their impact vs the properties. For example, the security modelling notation must allow at least the modelling of an aspect of the use case (control, data, etc) and the modelling of the changes. Then a security property must be ensured before and after the changes, for example see Section 5.4.1 property 5 .

That means that the modelling notation developed in this Work Package 4 will be used by the Work Package WP6 for verification and Work Package 7 for testing, that are the two means used in Gemalto to assess security changes.

## 5.1 Introduction

The POPS use case involves a smart card (an UICC card) intended to be plugged in a mobile phone or other mobile devices to provide services to an end user (card holder). We will consider a multi-application and a cross-sector card because this USIM card will be used for mobile payment.

This smart portable object is composed of:

- An embedded Integrated Circuit (IC),
- An embedded Operating System (OS) which provides classical OS features (memory access, etc.) using OS functionalities,
- A Java Card System (JCS) according to [Inc03a], built on top of the OS which manages and executes applications called applets. It also provides APIs to develop applets on top of it, in accordance with Java Card specifications.
- A GlobalPlatform (GP) including the application loader and the APIs, which provides a common and widely used interface to communicate in a secure way with the external world, in accordance with [GPS06] specifications. This environment is primarily used to manage the contents of the card.
- (U)SIM APIs, which provide a means to specifically interact with (U)SIM applications, according to [ETSI-TS131.130] specifications.

The architecture of this object is presented in Figure5.1. For the purpose of the project, we will focus on the GP and JCS part of this architecture. We assume that in post issuance, the card is managed over the air (OTA) e.g., using SMS.

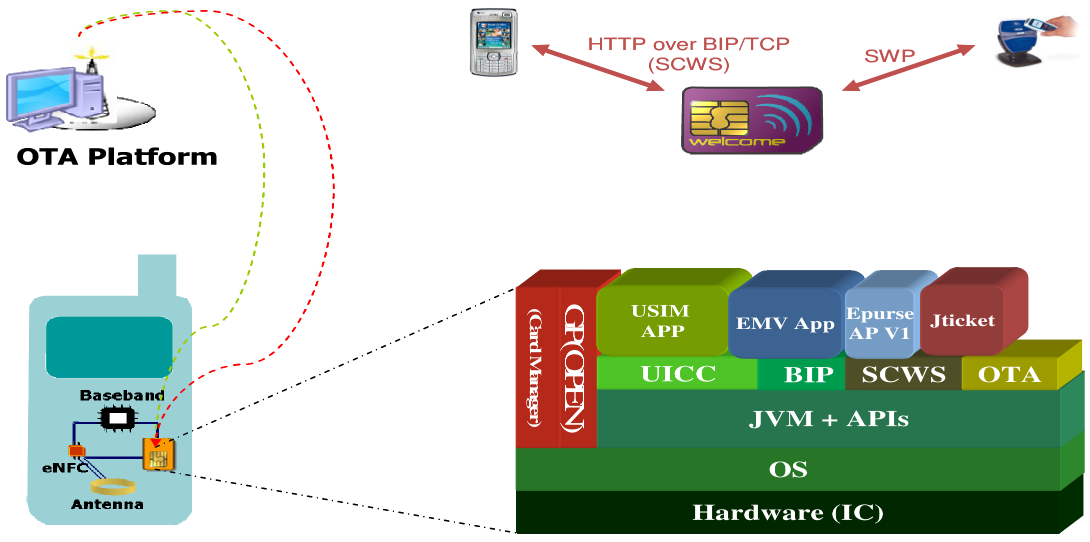


Figure 5.1: Architecture of POPS

### 5.1.1 Java Card System

The Java Card System is defined by a runtime environment (JCRE) specified in [Inc03b], that includes:

- A virtual machine (JCVM) specified in [Inc03c]: the JCVM is responsible for interpreting the byte-codes of the applets.
- A set of APIs specified in [Inc03a]: that provides commonly used services to the applets. Some support services for managing securely and efficiently the applets.

In contrast to a normal Java runtime environment, the JCRE is always in a running state on the card.

**Java Card Firewall** Since smart cards are mainly used in fields where security is very much an issue, a special security concept was designed for Java Card. This additional security is ensured in smart cards via a context firewall system. The basic concept is that only one applet can be selected at one time. Each applet belongs to a specific context. One or more applets may belong to the same context. In current Java Card technology, all applets sharing the same package are in the same context (package context). Only the objects belonging to the context of the selected applet can be accessed. Whenever an applet is deselected and an applet belonging to another context is selected, the context is also deselected and the other context becomes active (selected). The JCRE ensures that references to objects do not cross over context borders. The only objects that may be referenced over a context border are special objects owned by the JCRE (JCRE Entry Point Objects and global arrays). For example, these are instances of exceptions that the applet might want to return. Consequently, applets in a context run in isolation to those in another context and to the JCRE, and they only have access to objects and methods within this context.

**Shareable interfaces** Sometimes, it is necessary to communicate between appets (i.e. getting through the firewall). In Java Card, the interactions between two applets are done using the shareable interfaces. The server applet (which provides some services) defines a shareable interface including these services. The client applet connects to the interface to get the services. The JCRE ensures that only the services defined in the interface are available to the client. The server applet may also implement an access control on the client applets asking for its services. This ensures that only the allowed clients can use its services. The access control is mainly based on the identifier of the applets (AID – Applet identifier).

### 5.1.2 GlobalPlatform

The Global Platform [GPS06] consists in:

- A **runtime Environment**, running on top of the Java Card runtime environment, responsible for providing a secure storage and execution space for applications to ensure that each application's code and data can remain separate and secure from other applications on the card. Fixed memory addresses can be allocated to each application on the card, preventing each of them from accessing the memory space assigned to another application.
- **Global Platform API**. While the Runtime Environment provides generic services needed by a basic smart card application, the Global Platform environment, being primarily the Card Manager, provides additional services relating to card and application management and a mechanism for securing communication between a card and an off-card entity.
- The **Card Manager**. The Card Manager represents the Issuer's interest on the card by preventing unauthorized use of the card. The Card Manager is what enables the Card Issuer to maintain ultimate control of the card and its contents. The Card Manager supports the following four functions; Command Dispatch, Content Management, Security Management and Security Domain.
- **Card Content Management**. The applications and data on the card represent the differentiated and customisable services that can be offered to cardholders. There will be one or more applications on the card, and each of these applications will need to connect with a  terminal, containing the complementary terminal component of the application before it can be used. These applications and data can be loaded or removed during Pre-issuance and Post-Issuance (e.g. a load file is a CAP file for Java Card™).
- **Security Domains**. In addition to the Issuer Security Domain, separate Security Domains can be established on the card to protect application providers or groups of applications. Security Domains enable the applications of various providers to share space on a card without compromising the security of any particular provider or application. Security Domains also allow the application owner to control its applications without the Issuer having to share its keys with the provider. The use of Security Domains is ideal when the Issuer is dealing with a trusted provider who is capable of maintaining its own applications. This prevents the Issuer from incurring the administrative overhead associated with

monitoring and controlling applications, which are not part of its core business. Multiple Security Domains can coexist on the card.

Using the extradition operations, the Security Domains may be structured in one or several hierarchies in the card. In a hierarchy, an Security Domain may use the services provided by the Security Domain that is extradited to it.

#### 5.1.2.1 GlobalPlatform API

The Java card Runtime API provides the basic services of a smart card application. The Global Platform environment provides another layer of service. These services are accessed through a separate Global Platform API that handles such things as secure off-card communication, card or applet lockdown during security threats, and enables secure applet personalization such as key loading. The Global Platform API interacts between the applications and the Card Manager or the Security Domains. It is important to know that the Global Platform API acts as the link to the services offered by Card Manager and security domains, while the Runtime API acts as a link to the services offered by the underlying operating system.

The Global Platform API provides another level of interoperability for application developers. The application providers can create a single version of the application that works through the Global Platform API to leverage the unique services of the Card Manager and Security Domains. This avoids costly application development costs across multiple Issuer systems.

**Secure Channel protocol** : Global Platform API provides a secure service for communication with the devices or server through Secure Channel Protocol. The secure communication through the Global Platform API is available for use by the applications through services supported by the Card Manager and the Security Domains. The services are authentication, confidentiality, and integrity of the messages. This secure communication is critical to off-card communication with devices. The Global Platform API provides the method for this secure communication by opening the channel and securing messages exchanged between the on-card application and the off-card application of the device/server. Depending on the security needs of the business model, different Secure Chanel Protocols are available. The Secure Channel Protocol SCP01 and SCP02 offer secure communication based on symmetric keys while SCP10 uses the assymmetric keys.

**Global Platform Personalization Support :** The Global Platform API provides services for personalization on the card. This is another service available to the applications. The Global Platform API provides the services to accomplish this task using standardized tools that application providers can tap into during the development of their particular application.

**Global Platform Card Global Services :** The Global Platform API provides mechanisms that allow applications to offer on-card services to other applications. An application who decides to propose a Global Service can register this service. .A second application can requests this service without having to know who is actually providing the service. The client applet uses an authentication method defined by the server applet. This authentication method restricts access to the service and manages a trusted list of services. An example of this method is the Java Card shareable interfaces.

**Cardholder Verification Methods:** Another use of the Global Platform Global Services is to manage access to the cardholder verification methods (CVM) present on the card. For example, a cardholder may have a PIN or password that is held on the card in a CVM space. Global Platform API provides applications access to this cardholder verification service. This allows the applications to verify the cardholder's authenticity through a centralized, shared space. Cardholders can then change their PIN access once and have all applications on the card use the same revised access code. Although applications need not use the underlying Global Platform functionality in order to be fully functional on a Global Platform card, there are many significant advantages to be achieved when an application does use this functionality.

Using a Global Platform Secure Channel (e.g. during personalization) allows an application to minimize its own code size by leveraging the card's platform security mechanisms and minimizes the impact on systems using the standardized secure communication protocols.

Using the Global Platform Life Cycle State management gives an application the advantage of making its Life Cycle State available to off-card management systems in a standardized manner through Issuer Security Domain commands. It also provides useful functionality for supporting application specific risk management policies and implementing associated enforcement mechanisms.

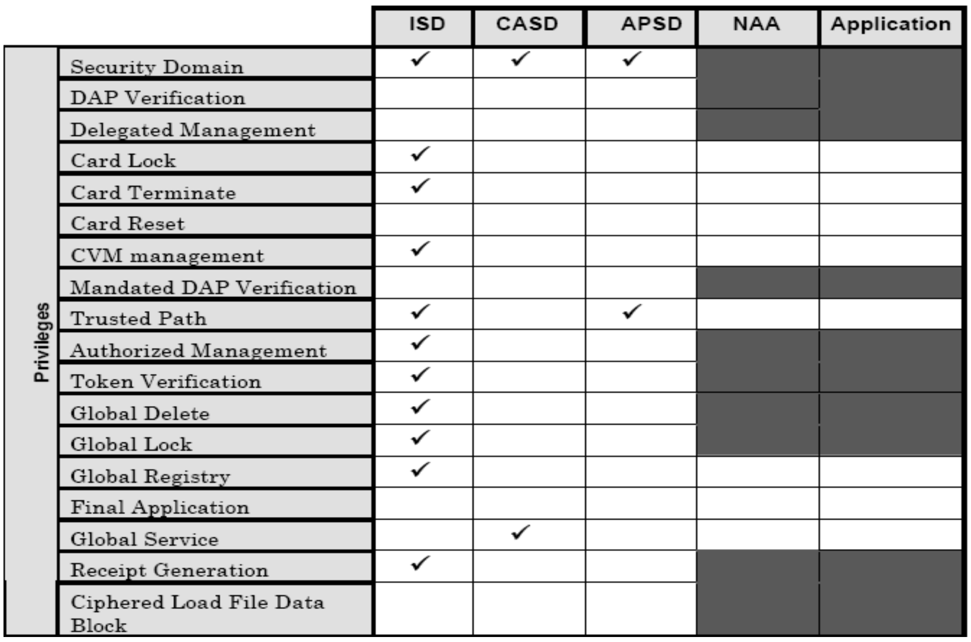
Using the Global Platform CVM (a.k.a Global PIN) in applications that employ a user PIN can simplify the application itself, and can increase card usability by having a common PIN for multiple applications.

### 5.1.3 UICC configuration

The UICC configuration is a specific configuration of GP that is dedicated to USIM cards [GPS08]. These cards implement ETSI-related specifications. The UICC configuration requires an issuer security domain (ISD), AP (application provider) security domains, and optionally a trusted CA (controlling authority) security domain.

Figure 5.2 describes the privileges of the security domains in the UICC configuration. This means for example that the AP security domain have at least the *trusted path* privilege while the CA has at least the global service privilege.

The UICC configuration requires that the ISD supports the SCP80 secure channel protocol specified in [ETSI-TS-102.225]. It is also required that in any Security Domain hierarchy, there is at least one "Authorized Management" security domain. Moreover, there is not more than one "Authorized Management" security domain in each path from the root to the leaves.



| Privileges | ISD | CASD | APSD | NAA | Application |
|---|---|---|---|---|---|
| Security Domain | ✓ | ✓ | ✓ | ▨ | ▨ |
| DAP Verification | | | | ▨ | ▨ |
| Delegated Management | | | | ▨ | ▨ |
| Card Lock | ✓ | | | | |
| Card Terminate | ✓ | | | | |
| Card Reset | | | | | |
| CVM management | ✓ | | | | |
| Mandated DAP Verification | | | | ▨ | ▨ |
| Trusted Path | ✓ | | ✓ | | |
| Authorized Management | ✓ | | | ▨ | ▨ |
| Token Verification | ✓ | | | ▨ | ▨ |
| Global Delete | ✓ | | | ▨ | ▨ |
| Global Lock | ✓ | | | ▨ | ▨ |
| Global Registry | ✓ | | | | |
| Final Application | | | | | |
| Global Service | | ✓ | | | |
| Receipt Generation | ✓ | | | ▨ | ▨ |
| Ciphered Load File Data Block | | | | ▨ | ▨ |

Figure 5.2: Privileges in the UICC configuration

### 5.1.4 Card life cycle

The main actors are:

- The IC manufacturer who is responsible for designing and manufacturing the IC
- The smart card manufacturer who is responsible for designing the OS, JCS, GP and for manufacturing the card
- The application providers who are responsible for developing the applications
- The card issuer who is the owner of the card
- The verification authority who is responsible for verifying the applications before allowing them to be loaded.

The card has the following phases:

**Pre- issuance**: The IC is developed and manufactured by the IC manufacturer in parallel with the smart card software (OS+JC+GP) by the card manufacturer. Then in a production and personalization phase, the software is "embedded", the JC and GP environment are installed and initialized on the card. It is a **pre-personalization phase**. In a **personalization phase**, the card being compliant to GP and JC, the application can be installed and initialized. and the whole is personalized by the card manufacturer.

**Post-issuance:** The normal usage of the card when it is in the hand of the final user.

The specificities of the "open" card implies that software (applications) could be "loaded" on the card that is already on the field. This loading is done contactless over the air (OTA) or with a reader over the Internet (OTI).

### 5.1.5 Application development life cycle

A Java Card application is developed and compiled on a host using a standard compiler. Those standard Java class files are then *converted* into a Java Card Converted Applet file (CAP File). Export files representing the imported tokens are input for the converter too. The converter resolves in fact external references and adapts bytecode accordingly; An off-card byte-code verifier may be used to statically check the CAP Files. After thorough testing on a software-simulation environment, such as the JCWDE environment, and a hardware emulation (optional), the CAP File can be loaded by a loader and installed on the card. The installation could be performed within a secure environment, in a pre-issuance phase or in a post issuance for open cards. The life-cycle of an application is described in Figure 5.3.

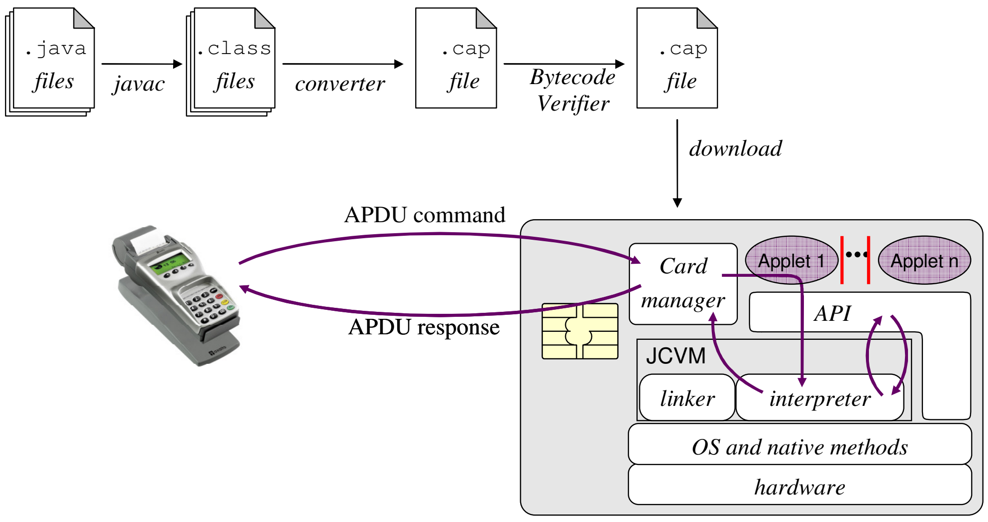


Figure 5.3: Application life-cycle

## 5.2 The scenario

In this section, we describe a scenario of using an UICC card that includes the development of applications, their loading on the card and contents management using the global platform commands, their execution and some changes on the card.

An overview of the scenario is given in Figure 5.4. A mobile network operator, proposes a SIM card to its customers that will also be used for payment as a contactless credit/debit card and for mass-transport as a contactless ticket card.

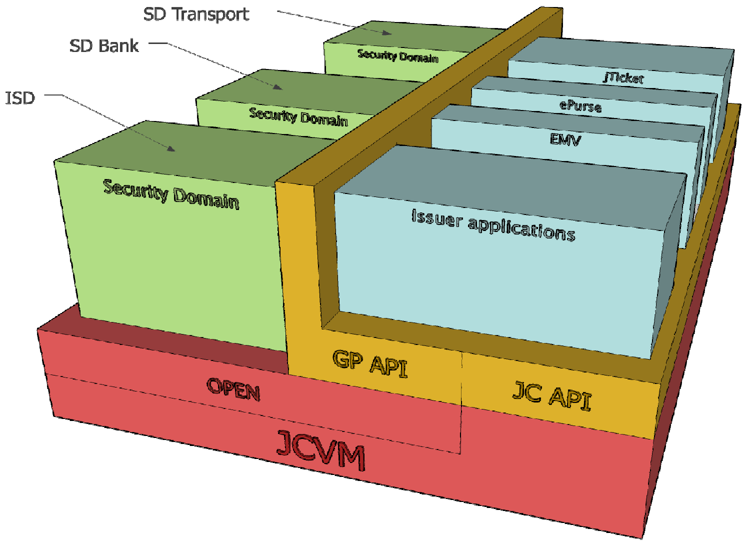


Figure 5.4: Overview of the scenario

### 5.2.1 Actors

The scenario involves several off-card actors that will be represented by on-card entities:

- The mobile operator is the card issuer (He bought the card from the card manufacturer)
- A bank that develops and provides some banking applications
- A transport operator that develops and provides some transport applications

### 5.2.2 Samples of applications

The application involved in the scenario are from different markets, banking and transport field. Those applications will be the on-card representative of the actors and of the services they provide.

- An e-ticketing application (JTicket) that will be used as a set of tickets :
    - Each access decrement the counter (N :number of tickets) by 1
    - When the counter is 0, the access is denied (JTicket needs to be refilled)
    - Jticket refill :
        - Use the epurse to buy tickets
        - Two cases :
            - Transaction accepted : N is reinitialized

    - Transaction refused : the requested amount is higher than JTicket_limit (the limit amount that epurse grants to JTicket)

- EMV: a classical credit-debit application
- E-purse (an electronic purse): for any e-purse transaction, the purchasing amount must be lower than a pre-defined limit (epurse_limit)

### 5.2.3 Flow

We describe here the main steps to load the application on the card, illustrating the use of GP commands (blue typesetting in the figures).

First in the pre-personalisation phase that is proprietary because the GP is not yer available on the card:

- we create the issuer security domain, ISD
- we initialize the issuer security domain by loading its keys

Then in the personalization phase that follows the GP specification

- we create the bank security domain (SD_Bank): see Figure 5.5
- we personalize the keys of the bank: see Figure 5.6
- we load the EMV application, that has been provided by the bank to the issuer, into the SD_Bank: see Figure 5.7

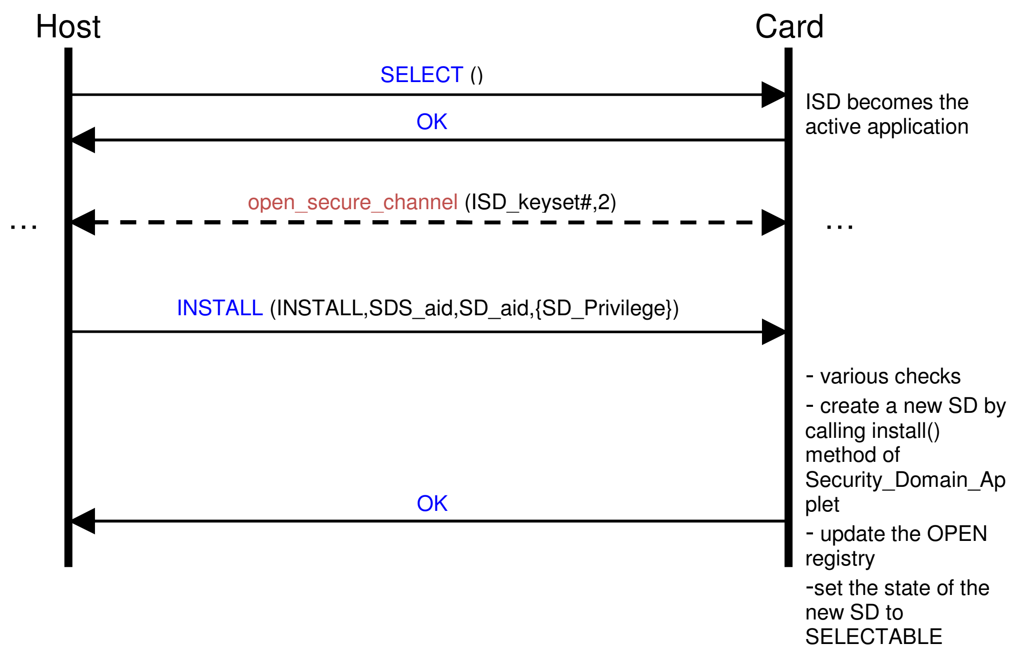

Host
Card
SELECT ()
ISD becomes the active application
OK
open_secure_channel (ISD_keyset#,2)
…
…
INSTALL (INSTALL,SDS_aid,SD_aid,{SD_Privilege})
- various checks
- create a new SD by calling install() method of Security_Domain_Applet
- update the OPEN registry
-set the state of the new SD to SELECTABLE
OK

Figure 5.5: Create a new security domain

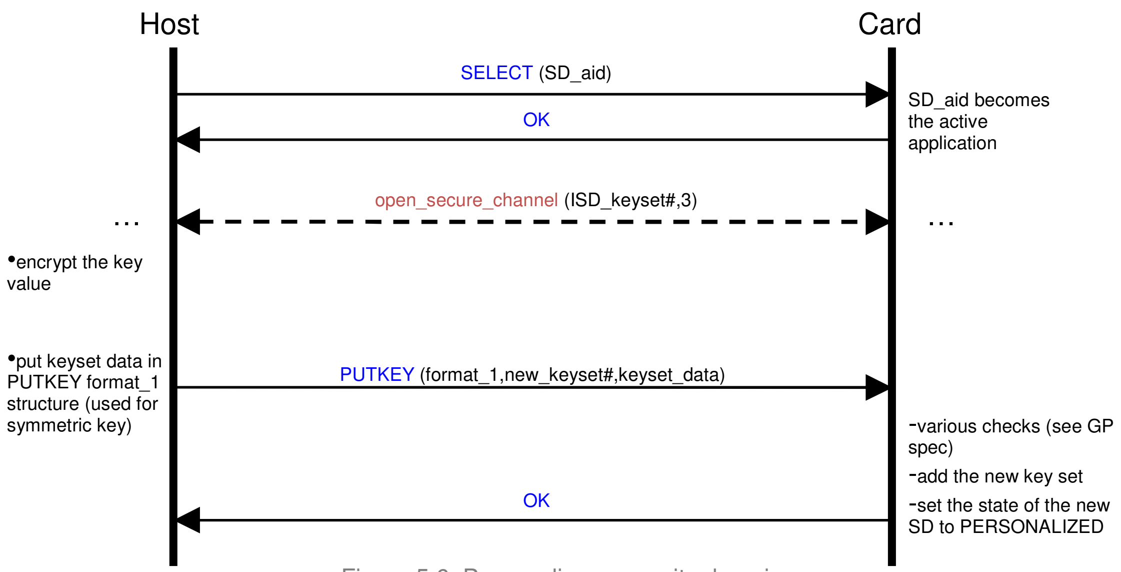

Host
Card
SELECT (SD_aid)
OK
SD_aid becomes the active application
open_secure_channel (ISD_keyset#,3)
...
...
•encrypt the key value
•put keyset data in PUTKEY format_1 structure (used for symmetric key)
PUTKEY (format_1,new_keyset#,keyset_data)
-various checks (see GP spec)
-add the new key set
-set the state of the new SD to PERSONALIZED
OK

Figure 5.6: Personalize a security domain

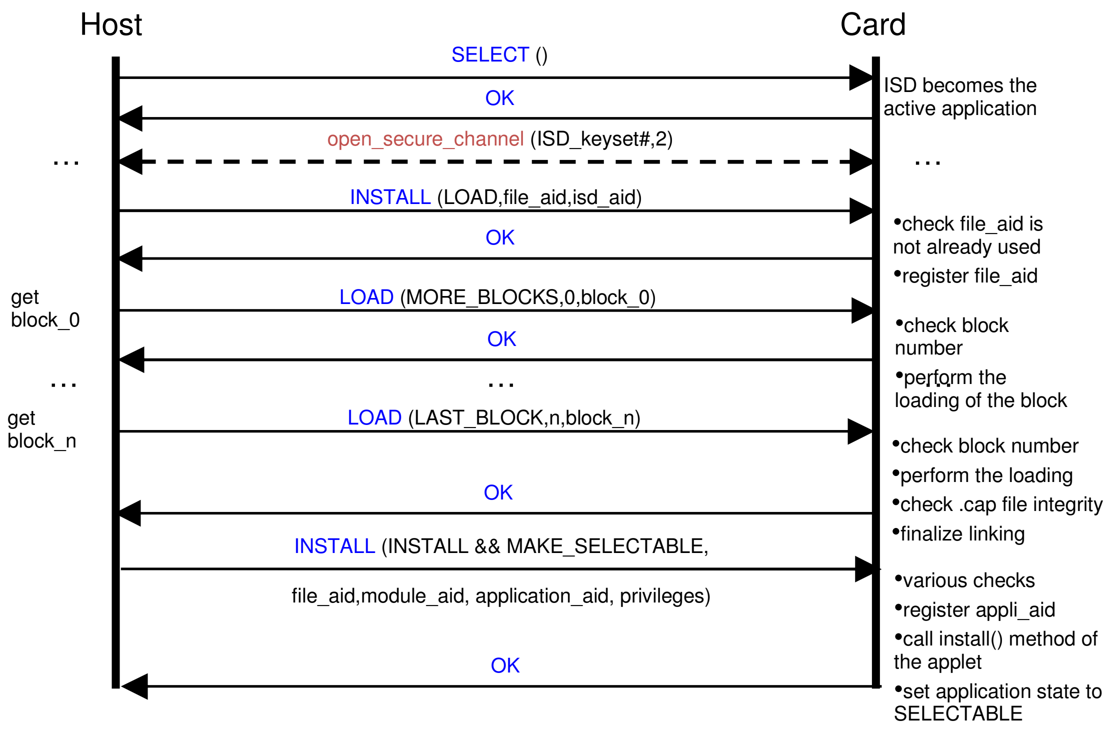

Host
Card
SELECT ()
OK
ISD becomes the active application
open_secure_channel (ISD_keyset#,2)
...
...
INSTALL (LOAD,file_aid,isd_aid)
•check file_aid is not already used
OK
•register file_aid
get block_0
LOAD (MORE_BLOCKS,0,block_0)
•check block number
OK
•perform the loading of the block
...
...
get block_n
LOAD (LAST_BLOCK,n,block_n)
•check block number
•perform the loading
OK
•check .cap file integrity
•finalize linking
INSTALL (INSTALL && MAKE_SELECTABLE,
file_aid,module_aid, application_aid, privileges)
•various checks
•register appli_aid
•call install() method of the applet
OK
•set application state to SELECTABLE

Figure 5.7: Load a new application (application_aid)

The process `open_secure_channel(keyset#,security_level)` opens a secure channel between the card and an external entity using a specific keyset and providing a specific security level (integrity, integrity and confidentiality, or none). Figure 5.8 shows how it is done in SCP02 using the GP commands.

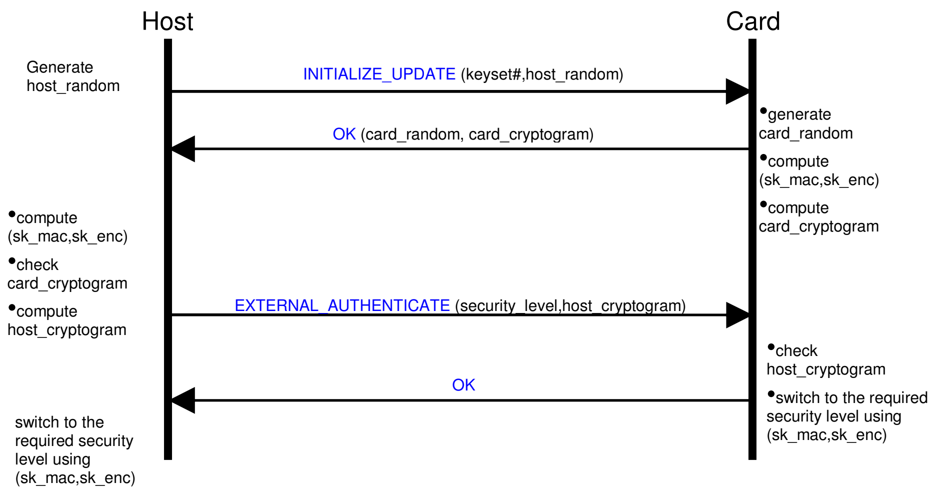

Host
Card
Generate host_random
INITIALIZE_UPDATE (keyset#,host_random)
•generate card_random
OK (card_random, card_cryptogram)
•compute (sk_mac,sk_enc)
•compute card_cryptogram
•compute (sk_mac,sk_enc)
•check card_cryptogram
•compute host_cryptogram
EXTERNAL_AUTHENTICATE (security_level,host_cryptogram)
•check host_cryptogram
OK
•switch to the required security level using (sk_mac,sk_enc)
switch to the required security level using (sk_mac,sk_enc)

Figure 5.8: Open a secure channel following SCP02

### 5.2.4 Implementation details

The EMV, Jticket and e-purse applications are implemented as Java Card applets. Each of them has an unique identifier (AID).

The security domains are implemented by a proprietary applet.

- each security domain is an instance of this applet (created with "SD_Privilege")
- each security domain has a unique identifier (AID)

## 5.3 Changes and Evolution

We focus on the changes during the usage phase of the UICC card, because in the other phases, the changes are managed in a secure environment (in card manufacturer or card issuer highly-protected sites). In order to illustrate the security properties, two categories are considered: administrative and usage changes.

### 5.3.1 Administrative changes

- The first administrative change is the creation of the security domain associated to the transport operator (SD_transport). This step is necessary in order to download the ticketing application (JTicket). Then, JTicket is loaded into SD_transport.
- The bank offers an e-purse service: the e-purse application is loaded into SD_bank

### 5.3.2 Application data changes

The changes on the application data includes:

- Change of Jticket_limit
- Change of epurse_limit

### 5.3.3 Security policy changes

The change of the policy will be illustrated by allowing the refill of the epurse using EMV.

## 5.4 Requirements

### 5.4.1 GP requirements

1. All entities on the card must be uniquely identified.
2. The loading of applications in post issuance must respect the policy of the card.
3. The application installation must be safe, e.g. to ensure that all external references of the application are valid.
4. The application deletion must be secure, i.e., it shall not leak previous information to the new application to be allocated in the same memory space.
5. Card content management must be done by authorized actors and ensure:
    - The consistency of card and application life-cycle
    - The enforcement of the card issuer policy

### 5.4.2 Application development requirements

1. All the applications loaded on the card must follow the guidelines to develop secure Java Card applications
2. All the applications must be bytecode verified (off-card or on-card). If the verification is done off-card then:
    - the adequacy between the export files used in the verification and those used for installing the verified file must be ensured
    - no modification of the file is performed between the verification and the signing operation (by the verification authority)

### 5.4.3 Change-related requirements

1. If an actor is added on the card, the corresponding security domain must not be able to access the applications from other security domain.
2. If an actor is added on the card, the consistency of the existing security policy must be preserved, e.g. the privileges policy.
3. Adding an application or updating an application must not generate

    - Illegal interaction with existing applications
    - illegal modification of GP system data (e.g. card or application's life-cycles, privileges, AID)
    - illegal modification or leak of GP user data (e.g. ISD/SD keys, global PIN)
4. Updating an application code or data must preserve its consistency/correctness (with respect to its specification).

### 5.4.4 UICC specific requirements

1. The security domain keys of the application providers are generated and stored in a secure way.
2. The transmission of keys to the application provider must be trusted and performed in a secure way.
3. Isolation between the security domain hierarchies: a security domain or application in a hierarchy is not allowed to access to the data/services of the other hierarchies.
4. Isolation between branches (a branch is a path from the root to a leaf) in a hierarchy: the data and services of a security domain are only accessible to the security domains and applications that are under it in the hierarchy.

# 6 Smartcard specific extensions of UMLseCh

## 6.1 UMLseCh stereotypes for Smartcards

This section attempts to employ the foundations (i.e. abstract and concrete modelling concepts) for expressing change in smartcard-type platforms. We follow the GlobalPlatform specification to identify the relevant security properties and behaviour for smartcards [GPS06]. We start with identifying the security goals. The aim of this exercise is to define the security constraints that must be analysed for fulfilment to attain the security goals. These goals not only support rationale about identifying security requirement but also support security analysis in particular to perform risk assessment when new change arise [Isl09]. Furthermore tracing of the initial system objective throughout the system evolution can also be facilitated by the goals. We identify UMLsec stereotypes to support these requirements so that system design can support the requirements [Jür05a].

The primary security objective (goal) of a Smartcard is to ensure the security and integrity of the card's components throughout the life cycle of the card. Of particular importance are the following components: The runtime environment; Security Domains; and all general smart-card and security domain associated Applications and Application Data. The security measures identified to fulfil this goal are the following: Data integrity, Resource availability, Confidentiality, Identification, Authentication and Authorization Control. For the Runtime Environment this implies a number of security requirements:

- Security Requirements RE1: The Runtime Environment shall provide a secure interface to all Applications of the smart-card, including general card applications and its associated application data, Security Domains and its associated applications and application data.
- Security Requirements RE2: The Runtime Environment Secure Interface shall protect the runtime environment security mechanisms from be bypassed, deactivated, corrupted or otherwise circumvented.

To address these two security requirements (RE1 and RE2) we need to define a new UMLseCh stereotype. The syntax of this new stereotype is: « secure interface », associated to the UML base class interface or link end-points. Semantically, this stereotype specifies that the security mechanisms of the associated entity cannot be bypassed, deactivated, corrupted or otherwise circumvented. The entity having «secure interface» associated to it must adhere to the semantic meaning of the stereotype and this must be verified as being fulfilled by the design solution and implementation. Secure Change will develop security analysis methodology and verification tools for such analysis.

- Security Requirements RE3: The Runtime Environment shall provide secure memory management, where secure memory management is defined as consisting of:
  - All application code and data (including transient session data), the runtime environment itself and its associated data (including transient session data) shall be protected from unauthorised access from on-card components;
  - In cases where more than one logical channel is supported by a smartcard, all concurrently selected Application code and data (including transient session data) and the runtime environment itself and its associated data (including transient session data) shall be protected from unauthorised access from on-card components;
  - All previous contents of the memory shall not be accessible when the memory is reused;
  - The memory recovery process shall be secure and consistent in case of a loss of power or withdrawal of the card from the card reader while an operation is in progress.

For this security requirement (RE3) we also need to define a new UMLseCh stereotype. The syntax of the new stereotype is: « secure runtime env ». The base class of this stereotype is sub-system, node and components and the stereotype can be used in deployment diagrams to specify the sub-system of the runtime environment. Semantically this stereotype defines the sub-system and its contained nodes and components involved in secure memory management, as specified above. The associated constraint are: « memory management ». The memory management constraint is used to specify how memory should be managed. The semantics of this constraint is specified by RE3.

- Security Requirements RE4: The Runtime Environment shall provide communication services with off-card entities that ensures that data and command transmission are according to the specific communication protocol rules.
- Security Requirements RE5: All communication between the Runtime Environment and off-card entities shall be protected from unauthorized disclosure.
- Security Requirements RE6: All communication between the Runtime Environment and off-card entities shall be protected from unauthorized modification.

The security requirements RE4, RE5 and RE6 are addressed by the existing UMLseCh stereotypes « secure links », « secrecy » and « integrity ».

## 6.2 Modelling change in Smartcards

As stated, UMLseCh stereotypes are employed to model change within the secure design diagrams. Therefore we investigated the detailed about the existence stereotypes to support both architectural and detailed design of the smart card system. This is because initially our focus is to use the existing stereotypes to enforce security in design in particular integrity due to any change during the system life cycle. However, note that all changes can not be supported by the existing stereotypes. Change may arise from functional or security perspective and classified through change to application data, change to application, change to platform data, change to platform code and change to hardware and software interfaces. Therefore any type of change may arise at any time and adequate security analysis is required before supporting the change. Several existent stereotypes such as: « fair exchange », « smart card », « authenticity », « integrity », « secrecy » and « secure links » are already proven to be relevant for the smart card system by based on the GP specification. To support the change we prefer to follow the below given strategy:

- Initial consideration would be whether existing model element such as stereotypes, tags and constraints support the relevant change. If yes then we retain the detailed of the stereotype.
- Otherwise, we try to identify the minimum permissible change through the tags and constraints of the stereotype through adding new tags or constraints, modifying the existence tag values to support the change.
- Otherwise, new stereotype would be introduced along with tags and constraints to support the change.

We propose to introduce new stereotypes along with tags values and constraints by investigating the specification to enhance the stereotype repository of UMLseCh as well as to support the smart card system. These stereotypes if require include the elements of the meta model for change. E.g. associate tag value of version=<version_ number> {dependencies=<yes/no>}. This facilitates to perform proper security analysis on any part of the system before approving the change. Figure 6.1 outlines the new defined stereotypes, together with their tags and constraints to support any future change of the system environment.

Detailed about the tags values within the stereotype are also given in Figure 6.2. The concept describes for the tag is instance level. Therefore the stereotypes along with

| Stereotype | Base Class | Associate stereotype | Tags | Constraints | Description |
|---|---|---|---|---|---|
| secure runtime env. | subsystem, node | « integrity », « authenticity » | data, code, version_number, dependencies | Separate storage for data, code and secure memory deallocation | To ensure proper memory allocation and deallocation by the runtime environment |
| secure interface | node, link | « integrity », « authenticity » | data, version_number, dependencies | Security mechanism cannot bypassed deactivated corrupted circumvented | To ensure proper implementation of security mechanism within the associated entity |
| legitimate process | subsystem, node | « provable », « integrity », « authenticity », « secure links », « adversary » | action, right, ressources, version_number, dependencies | Only permitted legal activities | To ensure legitimate processing by the user application provider as per the specification as well as introducing any new policies and regulations |
| data filter | node, link | « guarded » | action, ressources, adversary | Monitor data and filter (if required) | To ensure availability of data and code during the life cycle of system components |

Figure 6.1: UMLseCh stereotypes

| Tag | Associate stereotype | Type | Description |
|---|---|---|---|
| data | « secure runtime env. » « secure interface » | variable | specific information about the memory by the application |
| code | « secure runtime env. » | variable | specific information about the memory allocation by the application code |
| version_number | all possible stereotypes that handle change | state | version state by number due to any change |
| dependencies | all possible stereotype that handle change | boolean | dependencies due to change by a specific stereotype in system |
| action | « legitimate process » « data filter » | value | possible actions supported by node |
| right | « legitimate process » « data filter » | value | right to perform the action |
| resource | « legitimate process » « data filter » | value | resource require to perform the action |

Figure 6.2: UMLsec tags

associate tags and constraints assume to support any change mainly from security perspective of the system.

A summary of the UMLseCh metamodel for smart-cards can be found in Figure 6.3.

**Example: smart card** Now we want to use this notation to design a smart card system based on the GlobalPlatform specification. We design the physical architecture of the components through deployment diagram and architectural design of the components through state chart, activity and class diagram. Details of the design diagrams based on the GlobalPlatform specification can be found in Section 7.1.

## 6.3 Semantics of the Smart-cards stereotypes

The newly defined stereotypes for smart-card is now elaborated to provide common understanding how they can be used to ensure security under GP environment. Note

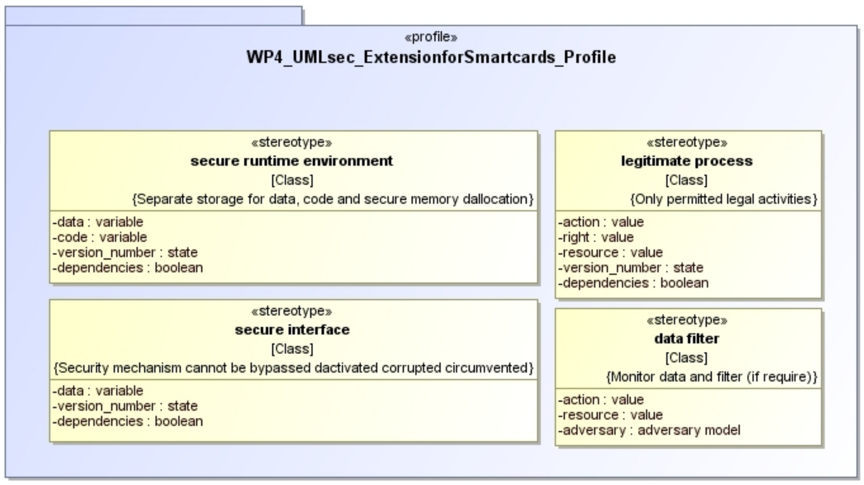


Figure 6.3: UMLseCh metamodel for the smart-card notation

that as we follow the GlobalPlatform specification to defined these stereotypes therefore we assume the stereotypes compliance with the requirements from the specification. The main purpose of these stereotypes is to ensure security in the Runtime Environment. The stereotype « secure runtime env. » mainly concerns with memory management and « secure interface » concerns with the security mechanism supported under the Runtime Environment. Both of them link associate with the existence « integrity » and « authenticity » stereotypes.This is because whenever participants under the platform access to the memory for any purpose then their authentication is required as well as any modification by the participant shall preserve integrity. For instance there is an authentication mechanism for the users under the Global platform. We assume it as an instance of the « secure interface ». When user under the participating entities attempt to log in then by any means authentication mechanism must not by pass, deactivated, corrupted, and circumvented. It preserve the « authenticity » of the system. Thus model should allow the preservation of authentication mechanism for node or established communication link. This way all existing security mechanism should preserve its goal by employing the « secure interface » stereotype.

The stereotype « legitimate process » is rather high level and more concerns with the relevant legislation that should compliance within the system environment. For instance, implementing of smart-card within the Federal Republic of Germany must compliance with the German Federal Data Protection Act (FDPA).Therefore business partners under the GlobalPlatform offer smart-card based service to the customers within Germany must preserve privacy of the customer data. This stereotype is applicable within this context to integrate legal constraints into the system design. Several existent UMLseCh stereotypes are also related with « legitimate process » such as

« integrity »,« authenticity », « provable », and « secure links ». It allows the permissible actions from the relevant legislation for the system environment. For instance, consider according the Federal Data Protection Act, data can not distribute to any third party with out data owner consent. Therefore application provider who is responsible to store customer private data must now distributed it to another party even though the party is under Global Platform. Thus permissible action through data owner consent is an instance under the right of the « legitimate process » stereotype.Note that the stereotype also includes possible right and obligation from the related security and privacy policies of the system.

# 7 Applying UMLseCh to the Global Platform Specification

## 7.1 Introduction

This section applies UMLseCh as part of model based security engineering for secure design of the GlobalPlatform architecture. We follow the GlobalPlatform specification to understand the basic concept of the system architecture. In particular, we examined the specification document for the system, card, and security architecture and the life cycle models. Several components under the GlobalPlatform card architecture such as security domain, runtime environment, trusted framework, GlobalPlatform environment, GlobalPlatform API, and card content management and security goals and requirements are analyzed to model node, functions, subsystem through UMLseCh deployment,activity, and class diagram. The life cycle of the card is modeled through UMLseCh state chart diagram. Furthermore,we try to model the change through the diagrams along with the stereotypes and tags. Therefore, it facilitates different view of the system through several level of abstraction and able to capture and trace any relevant change of the System model. Main focus lies to trade-off between permissible changes with minimum cost-effective security analysis so that existence security property would not be destroyed due to the change. The deployment diagram and class diagram is constructed based on the abstract specification from the global platform. UMLseCh provides concrete detail specification of the system. Therefore we start from the abstract level and continue to the detail architecture from the security requirements. Thus the scope of this section is based on the input from the GlobalPlatform specification. Note that well-designed security architecture is crucial to protecting the structure and functions of the card with the Global Platform system. UMLseCh attempt to serve this purpose and our contribution is within this context. Main participants of the GP platform within any business environment are as follows:

- The card issuer/ mobile network operator (MOB)
- The application provider/ business partners of card issuer/transport
- The controlling authority
- The cardholder

## 7.2 Goal-driven security risk management model (GSRM)

We employ the goal-driven modelling approach to perform the security risk management activities for the GlobalPlatform. The model initially identifies the goals from the system components that require attaining to satisfy the security properties of the system and then system vulnerabilities and threats that directly or indirectly obstruct these goals. These risks are then analysed to identify its consequence within the system under environment. And finally mitigation actions identify and suitable one selects to countermeasure the risks in order to attain the goals. GSRM focuses on holistic view to perform risk management for the GlobalPlatform components considering both technical and non-technical perspective. This is because only the system technical issues can not ensure total security of the overall infrastructure. Security require supports from the proper user action involve with the Global Platform environments, well defined policies and procedures, appropriate administrative control, and so on. Note that non-technical issues are mainly focus on security management and policy in the overall system infrastructure. Any change occurs throughout the states of the life-cycle should analysis through GSRM to support the evolution. There are several activities under the risk management method of GSRM such as *identify and model goals*, *identify and model the risk-obstacle*, *assess risk*, and *treat & monitor risks*. These activities are refined into tasks to perform specific action for the security risk management. There are several artefacts produce through GSRM in particular risk treatment action concern more with the refinement of the initial higher level goals into to more concrete requirements so that Design can support the requirements through UMLseCh stereotypes, tag values and constraints.

#### Goals

Goals are the objective, constraints, problems, and expectation from the system environment that satisfy the main security properties including confidentiality, integrity, availability, authenticity, and non-reputation. Goal satisfaction requires cooperation of the system agent. For instance, user as system agent use password in login the system. In this context user is the agent with support the authenticity by performing an action. To identify and model the goals, initially our focus is to consider the main system components and elements and factors that comprise these components. The identified goals can be initially higher level therefore it requires proper refinement from higher level to lower lever finer goals. The more the goals refined the easier to identify the threat obstacle that obstruct the goals. Main components and the relevant elements and factors under the components of the Global Platform system are given below:

- **Security domain**
  - **Service**
    - Key handling
    - Cryptographic operation
    - Digital signature
    - Provider verification
  - Security policies
  - Secure channel protocol
- **Runtime environment**
  - **Storage management**
    - Execution space
    - Data space
  - Communication services
- **Trusted Framework**
  - Security assessment
  - Interapplication communication services
- **GlobalPlatform Environment(OPEN)**
  - Logical channel management
  - Card content management
  - Executable load file under card content management
- **GlobalPlatform API**

Therefore, we identify several goals based on these components and their elements and factors. Before elaborating these goals, we have identified several main high level goals from the overall system environment. They are:

- Attain[ApplicationProvidersBusinessNeeds]
- Attain[SecurityOfTheOverallEnvironment]
- Maintain[SecureCommunication]
- Adapt functional and security driven change

However these high level goals are refined through several sub-goals by the system components, elements and factors

- Maintain[ProperOperationsbySecurityDomain]
  - Maintain[SecureSerivces]
  - Ensure data security
  - Share common card space by preserving seucirty
    - Maintain[ProperCryptographicOperation]
    - Proper cryptographic keys generation and distribution
  - Attain[CardContentIntegrity]
  - Proper implementation of secure channel
    - Ensure identification and authentication for every participating entities
    - Message and data integrity
    - Confidentiality and authentication of message
    - Secure message exchange between on card and off-card entities
  - Enforce security policies in Pre-Issuance and Post-issuance phase
- Secure runtime environment
  - Maintain[SecureStorageManagement]
  - Separate storage space for application code and data
  - Authorized access in single or multiple logical channels
  - Secure memory recovery process
- Proper operation of GlobalPlatform API
  - Secure off-card communication
  - lock down card or applet during unresolve security threat
  - Secure link among card manager and security domains
  - Proper on card services offer to the application
- Trusted framework

## 7.3 Security requirements

A set of security requirements are specified to support these goals. These requirements are categorized based on the components of the global platform including runtime environment, memory management, communication services, etc. Some requirements are:

- Runtime environment shall provide an interface to ensure that the security mechanisms cannot be bypassed, deactivated, corrupted or otherwise circumvented.
- Each application code, data and data relating to the runtime environment shall protect from any unauthorized access.
- All participating entities shall uniquely identified.
- The system shall add any new application to support global business service by the card, but addition shall not generate illegal interaction with existing application
- The transmitted data through the communication channel should ensure integrity.
- System shall verify integrity while updating the software running on the card.
- OPEN shall ensure that card content changes are authorized by the card issuers.
- The key generation and distribution shall confirm be done in secure manner

## 7.4 Secure architecture of GP specification by UMLseCh

Security requirements are traced within the UMLseCh diagrams by incorporating UMLseCh stereotypes. Therefore, we employ existent UMLsec stereotypes to address the identified security requirements and if required introduce new stereotypes to support the Global platform specification. To cope with the evolution from the card issuer, application providers, and card under the GlobalPlatform we integrate UMLseCh stereotypes. Our initial focus is to integrate the UMLseCh abstract stereotype to model the change by using UMLsec along with the extending UMLseCh stereotypes. The secure architecture cover both architecture and detailed design of the system. For simplicity reasons, all confidential information including key, customer financial information, and code are viewed as data. Therefore, UMLseCh requires providing adequate protection of existence as well as updating data. We start with a UMLseCh deployment diagram to specify the physical layer of the specification and continue a class diagram for detailed elaboration of the individual components.

**Deployment diagram**

Deployment diagram focuses on the interaction among the main components within the physical layer of the GlobalPlatform. It includes physical nodes and their associated or contained logical components. A node in deployment diagram may contains components and connects with other nodes through solid line. Components within the different nodes may also connect through broken arrows. Therefore the deployment view also shows the links between the nodes and the security requirement posed upon these links. Several nodes by following GolbalPlatform specification are: GP smart card, off card entity card issuer, controlling authority and application providers. Security domain is the main component to manage security issues relating to the services for these components. Therefore nodes such as card, card issuer, controlling authority and application providers have their individual security domain to support security services like key handling, cryptographic operations,identification and authentication, and many more. Furthermore, the card also contains instance for every application, Executable Load File(ELF),GP register,OPEN, Trusted Framework, and so on through mutable persistent memory. Figure 7.1 shows three nodes instance such as card holder, card issuer and application provider. The off card entity mainly concerns service update, card content management and application provider with application management along with the primary operations by the nodes. Secure memory management, access control, secure communication are the main security requirements for the subsystem. Therefore to ensure confidentiality and integrity of the data communicate within the established channel the link should be secures through « secure links » stereotype. Furthermore for proper memory management and implementation of security mechanism we include « secure runtime env. » and « secure mechanism ».

**Modelling change** The change arises at any time during the system life cycle from the components of the nodes within the subsystem. For instance, change to the application data is very common and frequent for this context. This is because security domain is a kind of application that contains the data for controlling the application and once application change any thing then it must update to its individual domain. Thus change from the security domain is more important compare to the change of other application data due to the application privilege hierarchy. Change can also arise from the general platform application such as secure runtime environment and OPEN. Therefore this subsystem supports change by following the UMLseCh stereotypes for both abstract and concrete design. We include the stereotypes « change », « current_change », and « future_change » along with associate stereotypes for the subsystem. The tag values version=<version_ number>; {dependencies=<yes/no>} is also associate for the UMLseCh stereotypes to specify the change version and dependencies among the components within the subsystem. The abstract stereotypes relating to change also associate with the UMLsec stereotypes to explicitly specify the security properties which relate with the change. Such as change in the security run-time environment is handled by the UMLseCh stereotypes within the « secure runtime env. ».

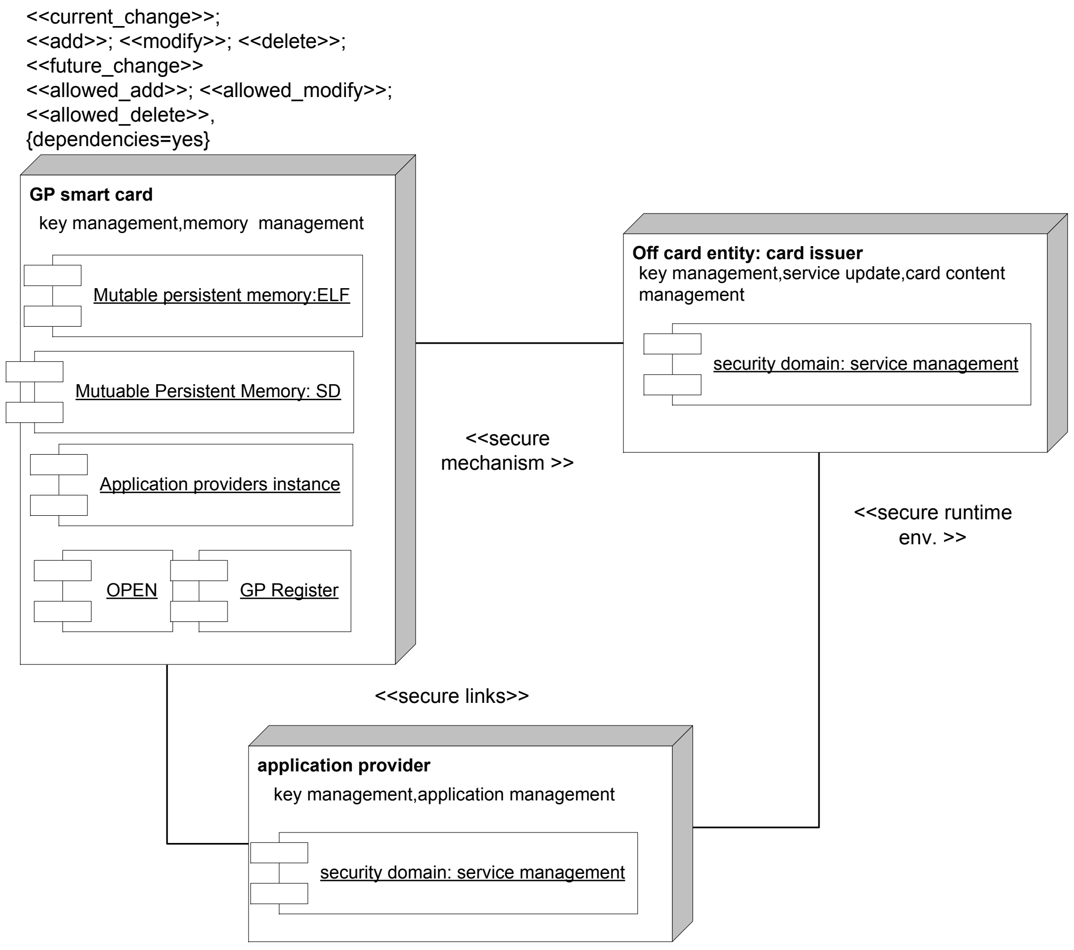

<<current_change>>;
<<add>>; <<modify>>; <<delete>>;
<<future_change>>
<<allowed_add>>; <<allowed_modify>>;
<<allowed_delete>>,
{dependencies=yes}
GP smart card
key management,memory management
Mutable persistent memory:ELF
Mutuable Persistent Memory: SD
Application providers instance
OPEN
GP Register
Off card entity: card issuer
key management,service update,card content management
security domain: service management
<<secure mechanism >>
<<secure runtime env. >>
<<secure links>>
application provider
key management,application management
security domain: service management

Figure 7.1: Deployment diagram

## Statechart diagram

We model the life cycle of the global platform components through the state chart diagram. Every component in particular the card itself, application, and security domain has several states through out its life cycle. UMLseCh models the behavior of these states by analysing the input requires for each state, relevant security issues to support proper operations within the state and output produced by the state. Thus statechart diagram specifies the sequence of state that an entity, object or component goes through in response to events along with the responding actions.

**Card life cycle** By following the GlobalPlatform specification, the card contains five main states throughout its life cycle. Initially OP_READY and INITIALIZED states are performed during Pre-Issuance phases of the card before supporting other components of the platform. At the start of the card life cycle, OP_READY state enters to initiate the card with several parameters such as its security domain, keys, application providers information, etc. Run time environment and OPEN is ready for execution by the runtime_env() and OPEN_ready() messages. Both of these messages transform data **d** to initialize the execution state for run time and OPEN. Action key_exch () performs to exchange the initialized key with the selected application and the issuer security domain. By the action select_app (), issuer selects applications explicitly. By following these actions the card enters into the INITIALIZED state. The state INITIALIZED is an administrative card production state and irreversible from OP_READY() to INITIALIZED() state. It may be used to indicate some initial data and support any change originates from the entities so that it can capture and integrate before card is ready to issue for the card holder.

After setting the initial data, the card is in the Post-Issuance state to use the card for specific purposes by the card holder. At the beginning, the card enters in the SECURE state. The state intends to operate the card and if require action such as enforce_policy(p1, p2,....) execute to implement certain policies among card issuers, applications providers and user. The card can be disabled under certain circumstances. Therefore, state CARD_LOCKED provides capability to disable the selection of the security domain and applications. Nevertheless, state from SECURE to CARD_LOCKED is reversible so that reuse of the same card is possible after controlling certain unavoidable circumstances. Finally TERMINATE state enter to end the card life cycle. All states can directly enter into the TERMINATE state. Main purpose of this state is to disable functionality belongs to the card and logical destroy through message destroy_card. This ends the stated of the card life cycle.

**Modelling change** Except TERMINATE, remaining states are capable to perform any change of the card life cycle. However in TERMINATE state, we need to deallocate the occupied memory by the card. Our focus is mainly with the changes at Post-Issuance phase because the card is used by the card holder within states of this phase. LOCKED state may performs change from any unplanned perspective in particular when change arise due to security attacks and overall system failure. Note that LOCKED state is not able to perform any type of card data management change such as domain key, data, etc. SECURED state mainly supports change due to maintenance and continuous operation of the card and the application providers, for instance modification of key values, addition of new features, and so on. Several function such as update_key() and update_data() support change for maintenance at SECURE state and disable_domain(), disable_app() are mainly focus at LOCKED state to handle any change from unplanned perspective. Figure 7.2 below shows the underlying states of the card life cycle. UMLsec stereotype « legitimate process » is relevant to support any change of existing policies and legal constraints from the environment. Therefore we add UMLseCh abstract stereotype « future_change » for « legitimate process » to allow addition, modification, and deletion of policies and legal constraints. Furthermore at the end of life cycle « secure runtime env. » also require to proper deallocation of memory by the card. Therefore « delete » added with the « secure runtime env. »

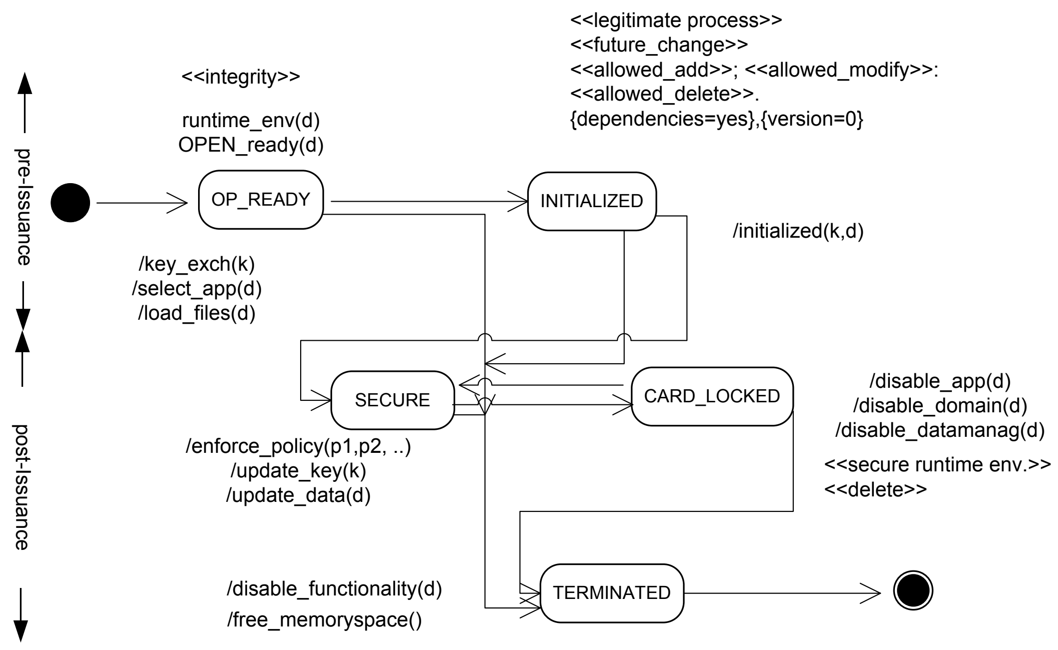


Figure 7.2: Statechart diagram of card life cycle

**Application and Security domain life cycle** The life cycle of the application and domain instantiate from an executable module i.e executable load file. Executable load file has only one state which loads the file to the card and updates the card register with the relevant information. OPEN sets the application life cycle state to the initial state during the application installation process. At any state of the life cycle, the OPEN as well as the application itself control the security protection by setting the life cycle state to LOCKED. OPEN also controls the deletion of an application from the card.

**Application Life Cycle** The application life cycle has three different states. Application may also define its own dependent states which would be controlled by the application itself. The first is the INSTALLED state that links the application executable code by link_code() and allocate required memory by allocate_memory(). Once these actions are executed then certain information from the application enters to the Global Platform registry by message send(entryGPR). After that, the application is properly installed and functional by entering to the SELECTABLE state. This state receives commands from the off-card entities by message receive(command). The application can be LOCKED in the LOCKED stated by OPEN, and associate security domains. The state LOCK, as security management control, prevents selection, and execution of application with action or prevent_selection(d) or prevent_execution(d). Once the application is in LOCKED state then certain components such as associate security domain with both global and local lock privilege is allowed to unlock the application through unlock_application(d) action. At any point OPEN, if require, delete application as a part of change. However delete must confirm the consistency of an application with other application and associate security features.

**Modelling change** The change arise due to maintenance, continuous and unplanned operation within the application life cycle to support the change type like change to the application data and the application itself. In particular, from the LOCKED to the other states, the data of a specific application may change by the OPEN. Therefore stereotype « modify » and « allowed_modify » is relevant for this state. The change can initiate from OPEN, security domain of the application, off-card entities, and global lock privilege. Any change in terms of add, modify, and delete is mainly performed in the SELECTABLE state and « modify » is mainly applicable to the LOCKED state. The DELETE state mainly considers UMLseCh stereotypes « delete » and « allowed_delete ». Furthermore UMLsec stereotypes « secure runtime env. » and « integrity » are essential for the application life cycle. Figure 7.3 below shows the state chart diagram of the application life cycle.

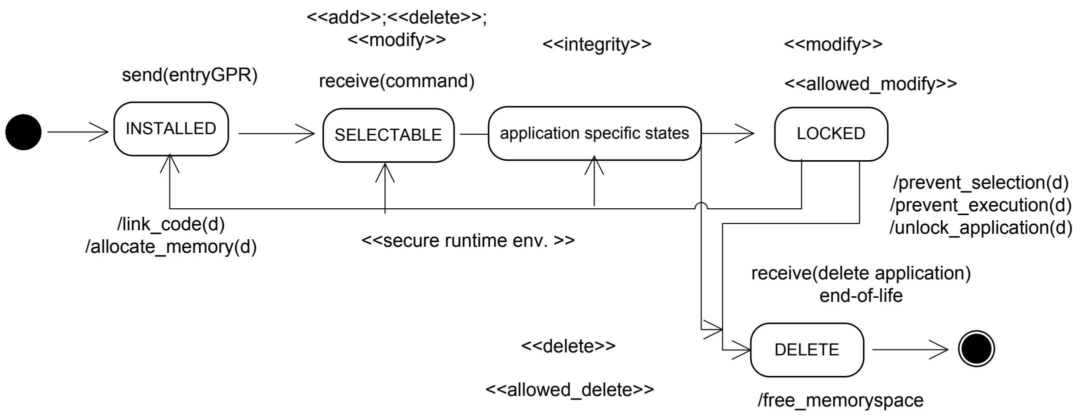


Figure 7.3: Statechart diagram of application life cycle

**Security Domain Life Cycle** There are four different states by the security domain life cycle. Initially INSTALLED state specifies that the security domain is an entry to the GP register by entry_GPR(d). However the state neither offer to select the domain by application nor associate with the executable load files. SELECTABLE state after the INSTALLED also cannot be associated with executable load files and application. Therefore the services are not available at this stage. The security domain services are available to the associate application through PERSONALIZED state. The personalization data and keys for full runtime functionality are available at this stage. Next state is to LOCKED the security domain, if require, as a part of security management, so that application cannot select the security domain, no executable load file is associate with the security domain. Finally security domain can be removed through DELETE state. Therefore space occupied to memory as well as entries to the global platform register by the domain need to be deleted. Figure 7.4 depicts the state chart diagram for the security domain.

**Modelling change** The changes for the security domain are mainly considered at PERSONALIZED, LOCKED, and DELETE state. The changes in security domain is critical for ensuring security service for the overall infrastructure. Therefore « integrity » is included to the life cycle for ensuring accurate modification of data and application. Similar like other component's life cycle LOCKED state mainly handles the unplanned changes by preventing the application to select certain services from the security domain, or lock the entire security domain. We include UMLseCh stereotypes like « add », « allowed_add »,« modify », « allowed_modify », « delete », and « allowed_delete » to support both current and future change.

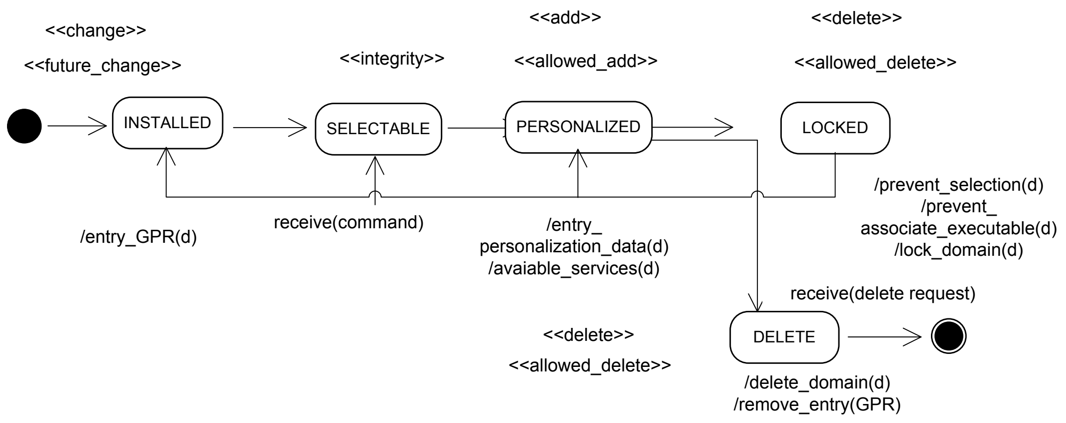


Figure 7.4: Statechart diagram of security domain life cycle

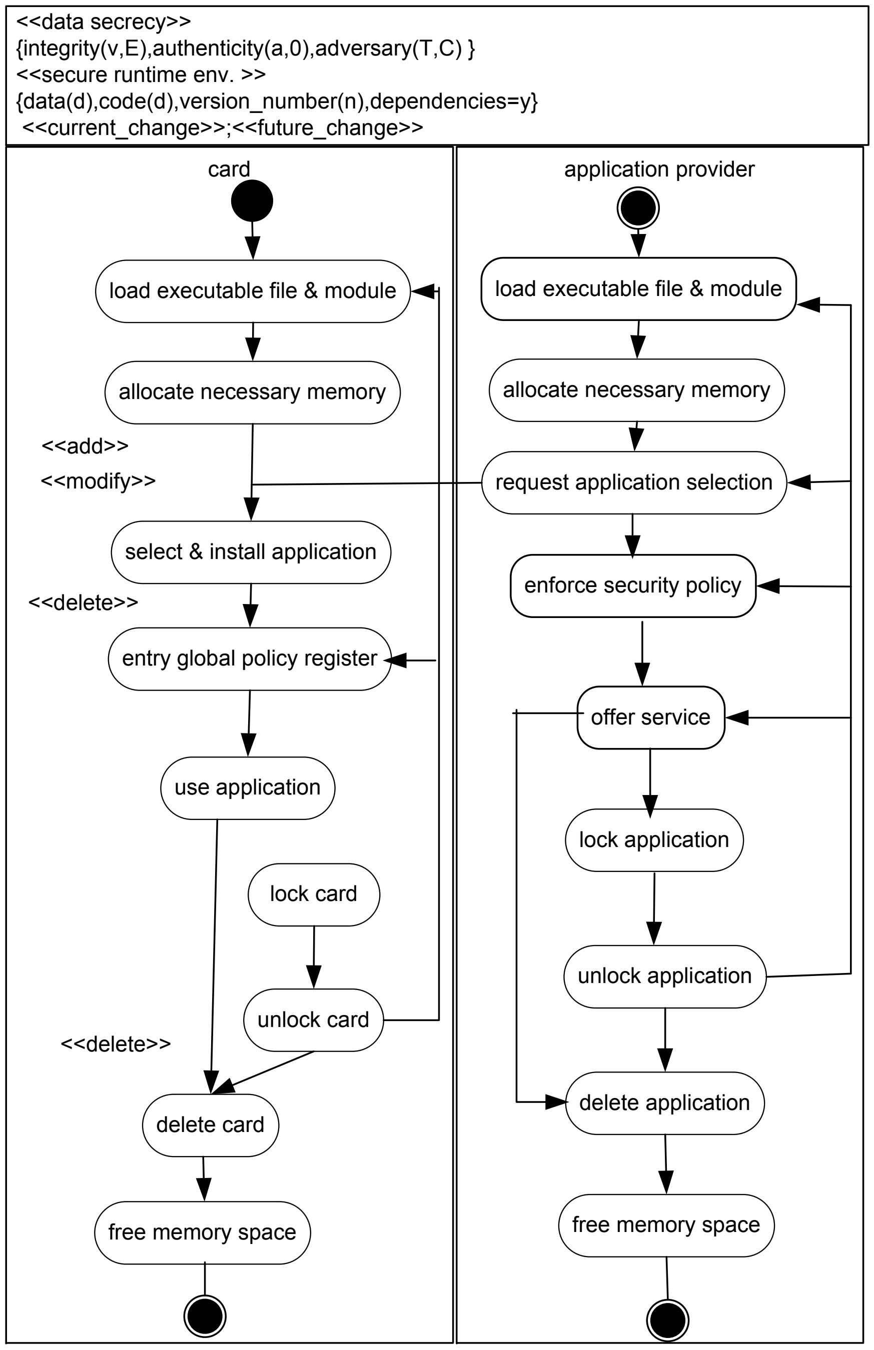
## 7.5 Activity diagram

We attempt to identify the activities within the several states of the card, application throughout their life cycle by following the interactions among them. The activity diagram models these interactions of interaction. We follow the card and application life cycle along with the possible interactions between them based on the GolbalPlatform specification. Initially, OP_READY and INITIALIZED state, executable files and relevant modules are loaded by the card. Similarly, during INSTALLED state executable file and modules are loaded by the application. Both of them require to allocate memory to install the code and entry into the GP register. Once the application is selected by the card then necessary policies for the application are updated and ready to use through the card. At any time, both components can be in LOCK state. In this state, no application is available to the card issuer and application itself. Card and application provider may delete through delete operation. Figure 7.5 shows the course if interactions and individual activities of card and application provider.

**Modelling change** During the course of the activities, several actions are performed such as entry into Global Platform register, allocate memory, prevent selection and execution. These actions support certain change due to maintenance, continuous, and unplanned based on environment, target system functionalities, legal context, and business process. This require to cope all types of change such as add, modify, and delete to the overall GP infrastructure. We assume UMLsec stereotype « secure runtime env. », « data secrecy » are essential for the interactions. Furthermore UMLseCh stereotypes are also applicable for the activity diagram.

## 7.6 Class diagram

The GP register (GPR), OPEN, Security domain, application are the main components that store relevant information about the application, card, and environment under the GlobalPlatform.We treat these components as main classes for the overall infrastructure.Every class contains attributes to represent the characteristics of the class and operations of the classless are supported by the functions. For instance by using Put_add(), Get_data() the GPR allow to input at its registray, by Delete_data() it can delete any entry from the registray. It stores several information such as application_provider_ID, security_domain_AID, card_management, app_management, card_lifecycle_state, app_lifecycle_state and counter.GPR update the GP API so that other classes can communicate with the GPR such as OPEN. OPEN also supports the Executable Load File and Executable Load Module to entry the application into the card and the GPR API. Security domain control the security policy instance and application is also relate with GPR. Figure 7.6 represents the class diagram.



<<data secrecy>>
{integrity(v,E),authenticity(a,0),adversary(T,C) }
<<secure runtime env. >>
{data(d),code(d),version_number(n),dependencies=y}
<<current_change>>;<<future_change>>
card
application provider
load executable file & module
allocate necessary memory
<<add>>
<<modify>>
select & install application
<<delete>>
entry global policy register
use application
lock card
unlock card
<<delete>>
delete card
free memory space
load executable file & module
allocate necessary memory
request application selection
enforce security policy
offer service
lock application
unlock application
delete application
free memory space

Figure 7.5: Activity diagram

**Modelling change** The changes are frequent for the class diagram because it store all information relating to the every individual classes. The UMLseCh stereotypes « integrity » and « authenticity » are essential for the sub system. Furthermore all UMLseCh stereotypes are applicable for the class diagram.

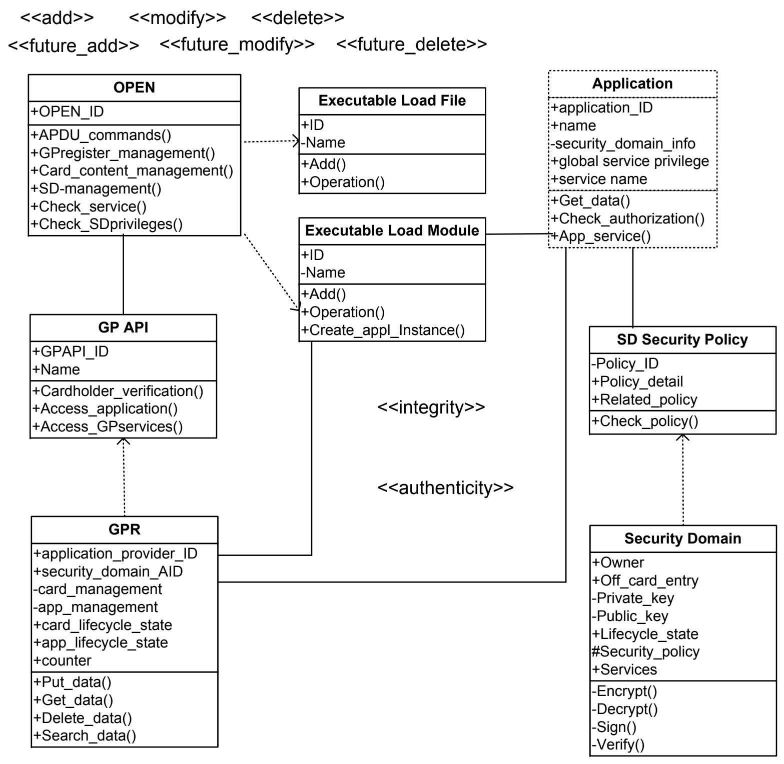


Figure 7.6: Class diagram

## 7.7 GP Security Domains - Clarifications Relevant for Change

The paradigm change introduced by the Global Platform specification is the ability to support several off-card entities on the same card (smartcard). This means that several off-card entities shall be able to own and manage data and applications on the same card. Earlier, the card issuer (being it the card manufacturer or the card distributor or card owner, which is what a mobile operator are for SIM cards for GSM and 3G) have been in charge of evolving and maintaining the card after it has been issued to the card user. Hence, application providers relied on the card issuer for maintaining their applications. E.g., for banking applications, the card issuer both installed and managed the banking service on the card on behalf of the banking provider. This hierarchy is based on a business model that trust on the card can only be maintained by a single installer, manager and card control entity, and this entity has traditionally been the mobile operator. This business model has now changed, opening up for multi-provider card management and service providing environment and thus multiple provider, of various kinds, can now make direct revenue of their on-card services.

To support the multi-provider business model, a GP card is virtually separated separate into a number of sections on the card, each associated with one off-card entity. This separation is virtual as it cannot be done hardware-based, which would have provided complete separation of sections on the card, because the number of off-card entities is not know at the time of card pre-issuance. Therefore, software-based card separation is necessary, also referred to as a software sandbox. This software-based card separation is called Security Domain for GP cards. Also, not only is the number of off-card entities unknown, the number may also dynamically change throughout the GP card life-cycle, and hence real-time installation, modification and deletion of security domains must be supported.

Software separation can only be achieved by means of (software) applications and therefore the multi-provider separation on the card is merely virtual. This complicates change management and normal maintenance operations on the card as we must handle possible application dependencies and interrelations and potential breach of the virtual separation. We will discuss these issues in depth in the following sections, but first we aim at clarifying the relationship between security domains and applications, as this relationship is essential for understanding change.

A security domain is considered to be a specific-type" of application and is also treated as such on the card. Applications in GP are loaded to the card in executable load files (ELF). During card pre-issuance phase, ELFs can be loaded into both immutable persistent memory and mutable persistent memory. In cases where an ELF is loaded into the immutable persistent memory it cannot be changed, as this means that the EFL is coded into the hardware and then shield. This also means that no changes can be made to EFLs located in the immutable persistent memory. Typically, the immutable persistent memory is used to store cryptographic keys, crypto seed procedures, random number generators and other data relevant to security credentials or the making of these and permanent card identities. The mutable persistent memory is accessible in all card phases and therefore support change. If, for example, the number of security domains were known at card pre-issuance phase, the security information used to separate the security domains could have been loaded to immutable persistent memory and the change of compromising the separation between security domains would have been largely reduced. However, this is often not the case and in our change examples we assume installation of security domains both in the card pre-issuance and post-issuance phase.

Furthermore, in the previous GP Specification an EFL was associated with a Security Domain and all Applications instantiated from an EFL, when installed, were also associated with the same Security Domain. This method was restrictive and to closely related to the Java card implementation of GP card. The new scheme provides application extradition. Application extradition allows an application that is already associated with a security domain to be extradited and associated with another security domain. Another benefit provided by this enhancement is that in addition to EFL and applications, now applications within EFLs become visible in the GP Registry at the time that the EFL is registered.

In order to avoid confusion between selectable applications and the applications within an EFL a new term has been introduced i.e. Executable Module. The term Executable Module is intended to identify the one or more applications present within an EFL. This is essential to take into considerations when tackling change, as it means that applications can move between security domains and that there now are no strict and permanent association between security domain and its associated applications. For this to be effective, applications are loaded and managed by off-card entities via security domains, but live their separate lives on the card ones installed and made selectable. Applications may have their own security credentials and functions, that are not controlled by the security domain. Application may also use the security domain security credentials for communication, but in such cases the communication is routed via the security domain that performs e.g. the encryption or signing on behalf of the application. The application does not get direct access to the security credentials of a security domain. Please note the difference between secure runtime management and secure runtime environment.

The card user is often referred to as the card owner and has traditionally been a customer of the card issuer, i.e., the mobile operator. For GP, customers can be directly associated with an application provider and do not need to have any relation with the card issuer. The customer-provider relationship is now service and application oriented rather than card issuer oriented. This is part of the GP paradigm shift (from one entity doing all the management and card update to several entities performing these activities).

The stakeholder model of the GP card significantly differ from that of today's smartcards. In addition to the traditionally card issuer and card user stakeholder model, the GP card may have one or more application providers associated to it as discussed above. There may also be other off-card entities involved, such as controlling authorities of various kinds depending on the GP card application domain and usage environment (banking has different laws and regulators than telecommunication, which differs from health care).

To summarize for change management purposes: There are no stable walls" between the security domains and applications belonging to a particular security domain may be extradited from this security domain and re-associated with another security domain. Hence, there are no guarantees that security domains remains truly separate through change and that changes within one security domain does not affect any other security domains or applications running on the card. Separation between security domains are done by means of encryption and signature (keying). Each security domain is encrypted on the card with a separate key associated with and only known to the relevant off-card entity.

Applications are installed using the card management modules of the relevant off-card entity. The card management modules are associated with an on-card security domain and card management is done via this security domain using the cryptographic key of the security domain (by encrypting and signing packets). Ones installed, the application operates separately from the security domain during normal use. Applications may have their own security functions and may decide to encrypt the communication using their own set of keys. Thus, there is a clear separation between card content, including application, management and normal card operation. This means that there is no real and persistent software segmentation on the card, so changes may have implications outside of the relevant security domain and application.

The card issuer security domain is the on-card representative of the card issuer, which issues the card. The card issuer maintains the general data and applications on the card. There are general GP applications. The card issuer may also have other application running on the card, which are not GP general applications (for security management). Then, each application provider is represented on the card by a security domain. They use this domain to manage their applications. Whether an application runs on its own or controlled by a particular security domain depends on the application preferences and security policy.

## 7.8 Notes about change in GP

One challenge when it comes to change is the problem of shareable services (across applications on a GP card) such as banking services (credit, debit) and the issue of ownership and how these applications are managed and controlled. In particular, when there are some policy update to a banking application, such as those needed to reflect changes in bank business processes, these will have affect on all applications using these applications.

In general, there can be shareable general GP applications that are managed by the card issuer (these are usually related to security management and APIs). Then the bank can be one off-card entity represented by a security domain on the card and that provides shared card services. This works fine because only the bank can manage the applications associated with its security domain. However, the application runs on the card together with all the other applications and these application can provide services to each other (most applications will run side by side in a secure runtime environment; that is, they will be prevented from changing each others code and interfering with each others processes during run-time). The challenge is to look into how to manage change during run-time (when the card is in use) as there actually are possibilities for applications to influence each other during run-time (remember that once an application is updated, only this application is locked during the update. All other applications may still run). We assume an API-type of model, where an application offers some services and requests some services in a secure manner (within the secure runtime environment). If one look at the layered network communication model, one layer provides services to other layers and as long as the service/info provided does not change the integrity of the protocol is maintained throughout changes within the layers. However, the integrity of the protocol is broken as soon as the info/service exchange type, content and pattern is altered such that what is provided differs from what is expected across the layers. For example, if the banking application offers one type of transferring money and then changes it such that the information produced from other applications (e-purse or similar) does not match with what the banking application expects.

## 7.9 Change examples Walkthrough

This section do a general walkthrough (to generalize over change also outside of the case study; the GlobalPlatform) and gives concrete examples for the five kinds of changes that we considered for a GP card.

**Kinds of Change Considered for a GlobalPlatform (GP) Card** For the purpose of the following change analysis, we consider five kinds of change for a GP card.

**Change type 1** Change to Application Data

**Change type 2** Change to Application

**Change type 3** Change to Platform Data

**Change type 4** Change to Platform Code

**Change type 5** Change to Hardware and Software Interfaces

These five types of change are categorized into evolution and revolution kinds of change (ref. change taxonomy). Change type 1 covers all changes made on data belonging or used by particular applications on the GP card. Recall that a security domain is a kind of application, which host the security domain specific applications. Thus, changes to data belonging to a security domain is considered to be changes to application data, but such changes will have a larger effect than changes to data belonging to applications within a security domain. This is due to the application privilege hierarchy, in which a security domain most often have more extensive privileges than a “normal” application. In any case, application data represent minor changes to a GP card compared to the other change types.

Changes to applications can be on several levels (change type 2). We can have changes to general platform applications, changes to security domains and changes to applications managed via particular security domains. Changes to general platform applications can have serious implications on other applications on the card, but also the card management routines. Examples of general platform applications are OPEN, Runtime Environment, Access Control applications, APIs, etc.

Changes to platform data covers all changes made to the data used and operated by the general platform applications (change type 3). Platform data are information belonging to or used by general platform applications or the off-card card content management modules. Changes to security policies associated with content management modules is an example of platform data change.

Changes to platform code covers all changes made to the core and supportive services and components on the card (change type 4). Examples are the trusted framework, runtime environment, OPEN and the APIs on a GP card. Platform code also covers changes to the part of the bootstrap procedure stored in the mutable persistent memory.

Changes to hardware and software interfaces (change type 5) are significant changes to the card that might affect more or less all components and services of the card. On the model level and taking the three security requirements into consideration, this type of change might affect mechanisms and services associated with the «secure interface» and «secure runtime environment» stereotypes.

**Change Type 1 – Change to Application Data** To demonstrate the magnitude of impact that changes to application data can how the operation and security of a GP card, we first run through an example addressing the maintenance change perspective under the evolution change dimension.

For our purposes we consider a Java Card implementation of the GP card. This means that an application is in principle a java applet. Application data is stored in the GP registry which is controlled by the OPEN. No other entities on the card have direct access to the GP registry and thus all requests to access the GP registry is routed via OPEN. Application data can only be changed by the owning Application (i.e., the application associated with the particular application data in the GP registry). There is one exception though and that are for changes to cardholder information (PIN, keys, certificates and profile data). Such changes is restricted in the way it can be accessed and changed and only the off-card entities Card Issuer and Application

Provider are permitted to perform these changes and only via the CI Security and Key management module and the AP Application management module respectively.

There are several categories of application data change:

- Load new application data
- Replace existing application data
- Modify existing application data
- Delete application data
- Load new cardholder information
- Replace existing cardholder information
- Delete existing cardholder information

In addition, these categories of changes address both past and current change and possible future changes. For the future changes, we need to annotate the model to specify what changes that are permitted on the model and to which elements. We give examples and provide a general discussion later.

First, we look into how to represent and manage change when loading new application data to the card after the card has been issued and is in use by the customer. This will help us understand the challenges we face when identifying and specifying change privileges and allowed change implications for future changes; change as a first-class citizen.

The Runtime Environment, in which all security domains, the OPEN and all applications run within, is a Java applet running in Flash. The GP Registry is also running as an instance in Flash (has an operation representation or a handle into the actual registry) and within the Java applet of OPEN there is an access controller applet that manages all access requests to the GP registry. The Runtime Environment Java Applet must run under the requirements associated with the stereotype «secure runtime environment», while all access to the OPEN from a security domain, applications or general platform applications to OPEN must preserve and respect the rules associated with the stereotype «secure interface» .

We consider first change to Signature Applet Data. In particular, we study how to add a new security rule to signing operation of the signature applet. Security rules are specified as attributes to signing operation of the signature applet. For this example we consider the Card Issuer signature applet. This implies that the signature applet is owned and operated by the off-card entity Card Issuer, which means that it can only be managed using the Card Issuer Security Domain Applet and that only the Card

Issuer can make the change. Furthermore, all changes must be executed over the Card Issuer Application management module.

**Application Data CR Steps – Overview and Example** To control the change we have specified the need of formulating change as a change request and we have defined a change process that should be followed. For the change example that we study, these steps can be formulated as follows:

1. Step 1: Check authorization privileges for the change request and the authenticity of the CR
2. Step 2: Put Signature Applet (application SDX1_AP1) in state Locked
3. Step 3: Identify relevant Executable Load Files and associated Executable Modules for the signature applet
4. Step 4: Update security policy by adding the new security rule to signature applet
5. Step 5: Check and verify successful addition of new rule to security policy
6. Step 6: Re-load relevant EFLs and ELFMs (and thereby the signature applet(application instance))
7. Step 7: Put Signature Applet in state SELECTABLE

There are specifications on which entities that can request and execute the various changes on a GP card. It is therefore necessary to check that the requesting entity has the necessary privileges to execute the change. When this is done, the authenticity of the CR must be checked (to avoid replay and fabrication attacks). The latter includes an integrity check. This covers step 1 of the change procedure.

For our example the application Data CR Step 1, including checking change request authorization Checks can be specified as:

- IF Request_ID=Card_Issuer THEN
  - Check_ReqModule(module_ID, Result) – This checks if ReqModule_ID == Valid Card Issuer Application management module and returns True/False in variable Result
  - Add_securityRule(security_rule, Result) – This adds and controls the successful (correct) addition of new security rule to signing applet

- ELSE
  - Silent reject – you do not want to send an error message in this case as that can be used by an attacker to deduce privilege rules for particular operations
  - Return to last known good card state

Application Data CR Step 2: Signing Applet runs within the state SDX1 Personalized, where SDX1 is the Card Issuer Security Domain. SDX1_AP represent Signature Applet before change. Before the change can be executed the Signature Applet must be put in the Locked application state. Note though that all security domains and not affected applications maintains normal operation while the CR is processed.

Application Data CR Step 3. In step 3 the relevant Executable Load Files and Executable Modules for the CR are identified by means of searching the GP registry. In cases where a relevant ELF serves more than the signature applet, the associate applications must also be put in the locked state and possible implications on the other applications must be investigated before executing the CR.

Application Data CR Step 4. Step 4 updates the security policy for signature applet with a new security rule. Important to note is that the Card Issuer Security Domain has an associated Security Policy (as all security domains have) and that a new security rule is added using the 'add_security_rule' operation of the Security Policy class. This operation operates on the SD Security Rule class, which holds the list of security rules for the associated security domain, application and load files. Security rules are associated with each security domain, application and load file(s) and stored in the GP registry. This means that updating the security policy with a new security rules in practise is to add the new security rule to the SD Security Rule table for the Card Issuer Security Domain in the GP Registry. All changes to application data results in changes to information in the GP Registry. We can look at the GP Registry as a set of database tables with associations. Making small changes to application data usually means adding information in one or several of the existing tables. Making larger changes to application data usually means to add new fields in one or several tables and maybe even adding a new table.

**Change Type 2 – Change to Application** Applications runs separately from their associated security domains on a GP card. This means that application may interact, at least in theory. As for the security domains, which are themselves applications, applications are installed, maintained and removed from a GP card using the card content management module. Furthermore, the card content management module ensures that only authorized off-card entities can perform the specific actions.

From the perspective of change, being it functionality-driven or security-driven evolution or revolution kind of change, we are concerned with the interrelation and dependencies between applications on the GP. In particular, we are concerned about cases where changes made to one application affects another application, within one security domain but also across security domains. We use the tagged values of the current and future change stereotypes to enable this checking, as is described in the following.

Application change process is similar to that for application data change process and we consider the following changes to applications:

- Load new application
- Replace existing application
- Modify existing application
- Delete application

**Change Type 3 – Change to Platform Data** Platform data are information such as security domain and application priviliges and other general platform information (such as version of GP, identity of card issuer, etc). From GP specification version 2, security domain and application privileges can be dynamically modified during active use of the card. Such modifications are done using the card content management modules. From the perspective of SecureChange it is important to check that only authorized changes can be performed and that the changes made does not open up for unauthorized changes.

Other platform data changes of interest are changes to the management functionality of the card. From GP specification version 2, it is possible to dynamically restrict the card content management functionality for security domains and OPEN. This means that in theory, a malicious user could change the management functionality in such a way that it gets access to insall malware on the GP card.

**Change Type 4 – Change to Platform Code** The GlobalPlatform is an evolving standard and there are already several expected extensions to the standard. Some of these extensions may result in a need to change smaller or larger parts of the platform code to either offer new functionality on the card or to change existing functionality. One envisioned change are changes to the way security domains are separated.

Changes to platform code includes changes to the actual implementation of OPEN, to the APIs, to the trusted environment and the other general platform functionalities. Such changes differ significantly from changes to platform data and applications, as these changes can alter the way application run on the platform, the way applications are maintained and the way that the internal parts of the card communicates.

**Change Type 5 – Change to HW/SW Interfaces** Hardware and software interfaces on a GP card are used for off-card entities to manage the card and for daily support and for the components on the card to exchange information or consume services across applications. Hardware interfaces are immutable and only introduced and modified during the pre-issuance phase. Software interfaces on the contrary can be modified after the card has been issued to the card user. Software interfaces are implemented as APIs and used by off-card entities and applications to request and offer information and services.

Changes to hardware interfaces would have to be tested before issuing the card and from the point of view of SecureChange we are only interessted in the security implications that the proposed change (we only consider future change for hardware interfaces) will have on the application data, applications, paltform data and platform code of the GP card.

Changes to software interfaces can be both current and future changes and may lead to a circumvent of security mechanisms. It is therefor important to check the security implication of such changes and if and how the security properties can be maintaned also after the change (actually, what it takes to preserve the security properties through change).

# 8 Applying UMLseCh to the e-Purse

In this section, we consider a specific application example in the context of the Global Platform specification, namely a smart-card based electronic purse system.

## 8.1 Overview of the application

E-purse is a small portable device that contains "electronic money" that can be used for any for low-value transactions. It is integrated with the extension capability of the smart card so that card includes additional value added services from business partner of the card issuer besides primary service. This indeed facilitates to save cost in terms of these secondary services offer by the card issuers such as mobile network operator as well as support multiple uses of a single card. As recent smart cards are based on the Global Platform specification, therefore the additional applications such as e-purse also require to compliance with the specification to support its services under the Global Platform environment.

When smart card allows to support additional services from the application providers then there should have specific area for the application within the card called application security domain. Security Domains act as the on-card representatives of off-card authorities and enforce the security policies defined by the owner. It determines the scope and responsibility of each actor participate within the environment. Every participating entities such as card issuer, application provider has its own individual security domain. To access the secure area, card issuer uses a secure connection called a Secure Channel. Once a connection is established, the Secure Channel provides an end-to-end secure communication path between an on-card security domain and an off-card entity. Subscripted user of the card issuer when agreed with the e-Purse application then he/she can use e-purse to buy any specific item electronically. One of this item is electronic-ticketing (e-ticket). E-ticketing is an electronic system for issuing, checking and paying for tickets predominantly for public transport.Therefore e-Purse is used to buy e-Ticket only when both the application providers of e-Purse and e-Ticket agreed to interact their services under GP through a card issuer.

## 8.2 Challenges on the overall infrastructure

There are some challenges exist within the existence infrastructure. The card issuer security domain simply have a set of security functionalities which the application can use. Therefore all communication have to go via the security domain because it is the security domain responsible to sign or encrypt packets on behalf of the application. However card issuer security domain as such does not control the application, only the changes to maintain such as update or delete the application and application data. Therefore application is responsible to ensure its security as well as maintain any change within its data and code. But it requires to synchronize any change from the application with other participating entities such as card issuer, other application providers, GP environment within the context. Because most of the applications are running side by side in a secure runtime environment; that is, they will be prevented from changing each others code and interfering with each others processes.And the applications also provide service to each other. Therefore how to manage change during run time in particular when the card is in use is non-trivial task. We need to preserve integrity as well as other security properties due to any change from any application to the overall system infrastructure.

## 8.3 Case study scope

The main scope of the case study in general is to ensure security to the overall system architecture where several actors such as mobile network operator, customer, bank, and transport as application providers are working concurrently to serve some purposes. In particular, our focus is to model the secure design of the system that support any change during the system evolution and revolution. The context is imperative because the card is not working alone, due to the Global Platform several other application providers integrated their services within the card to support value added service for the customers. Therefore ensuring security in particular integrity is one of the main goals of the overall environment. The objectives of the case study are to:

- (**O1**): Secure design in terms of architectural and detailed design of the e-Purse application under Global Platform environment
- (**O2**): Trace requirement to the UMLseCh design diagrams and vice versa
- (**O3**):Model change throughout the design diagrams by using UMLseCh modelling elements so that system would preserve the security properties due to change

## 8.4 Detailed design

### Use case diagram

We consider Mobile Network Operator (MOB) as card issuer who is responsible for the overall infrastructure. Initially MOB proposes SIM as smart card to the customer which includes additional service such as e-purse, e-ticketing from the business partners of the MOB within the card. Customer once accept the card becomes a subscripted user of the MOB and able to make use of the service that offer through the card. We develop an UMLseCh use case diagram to sketch the scenario among the participating actors. We integrate stereotype «fair exchange» to represent security requirements that any transaction should be performed in a way that the actors involve within the transactions prevent from any cheating. Figure 8.1 depicts the use case diagram of the actor interacting within the application.

**Modelling change** This subsystem can support rather high level change. For instance due to change of the business process, new service may be added to the card. This implies that the card content should adapt with the future changed base on the newly defined business case of the MOB. It may also possible the MOB accept new application provider within the card. Therefore a new application would also be installed into the card and user would be able to use of the new application. Therefore we consider possible future changed for this subsystem. Figure 8.1 includes the « change » stereotype to support the abstract change for the subsystem. The stereotype includes dependencies and version_number tag to specific the dependencies with other stereotypes that are relevant for the change within the subsystem.

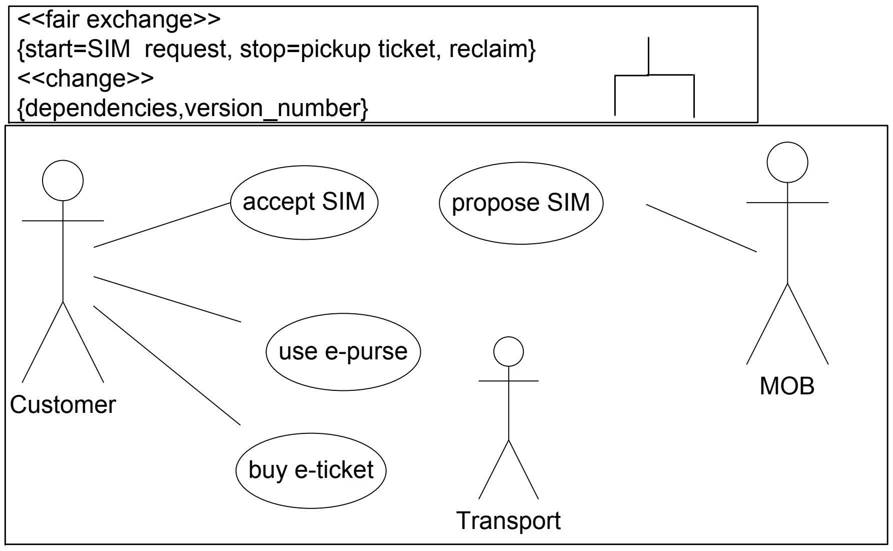


Figure 8.1: Use case diagram of detailed design

**Sequence diagram**

Subscripted user is able to make use of the e-purse to purchase e- ticket. User initially requests to select the e-purse application service to purchase the ticket through the e-purse. Once the e-purse service is activated by the MOB, then user has to send the ticket_id and amount to the card manager for continue the payment through e-purse. Note that, only the subscripted users provide consent to used the e-purse service through their existence SIM are allowed to purchase e-ticket through the e-purse. User then proceeds further to use_ticket() to the transport. Transport verifies the user information through the MOB and Jticket_limit before approve the purchase. If verification fails, then user denies to continue with the purchase. Otherwise user is able to purchase the ticket once adequate balance is available to the user e-purse account through the EMV debit and credit service of the bank. Transport can debit the amount from the e-purse as a part of shareable service among transport and bank. Therefore several SHAREABLE_SERVICE such as debit called among the administrative entities during the course of the transactions. Finally if sufficient balance is not available in e-purse then transaction would be refused by the transport. Figure 8.2 shows the sequence of actions performed among the participating entities.

**Modelling change** There may have several possible changes during these sequences of actions. However for these sequences most of the changes are from the maintenance perspective. For instance, change of user data such as e-purse limit and account balance, application data such as Jticket limit are common for this scenario. These changes are rather trivial to manage. And we allow to integrate « integrity » tag for this context. Additionally the change for this context is concrete, therefor we further include UMLseCh stereotype « substitute » for the concrete change through ref tag. However when participants like bank, transport, and MOB responsible to manage their own applications then change may affect the SHARABLE_SERVICE in particular change of policies. Therefore adapt the change is non-trivial for this scenario.

## 8.5 Architectural design

**Statechart diagram**

The card goes through several states throughout the life cycle. The states accommodate into Pre-Issuance, Issuance, and Post-Issuance phases of the card life cycle. Manufacture of the card mainly supported by Pre-Issuance and partial Issuance phase. While use of the card mainly supported by Issuance and Post-Issuance phase. Initially during manufacture, enable, and personalize states of the card, card hardware is assembled and application data as well as key are loaded into the card. GlobalPlat-

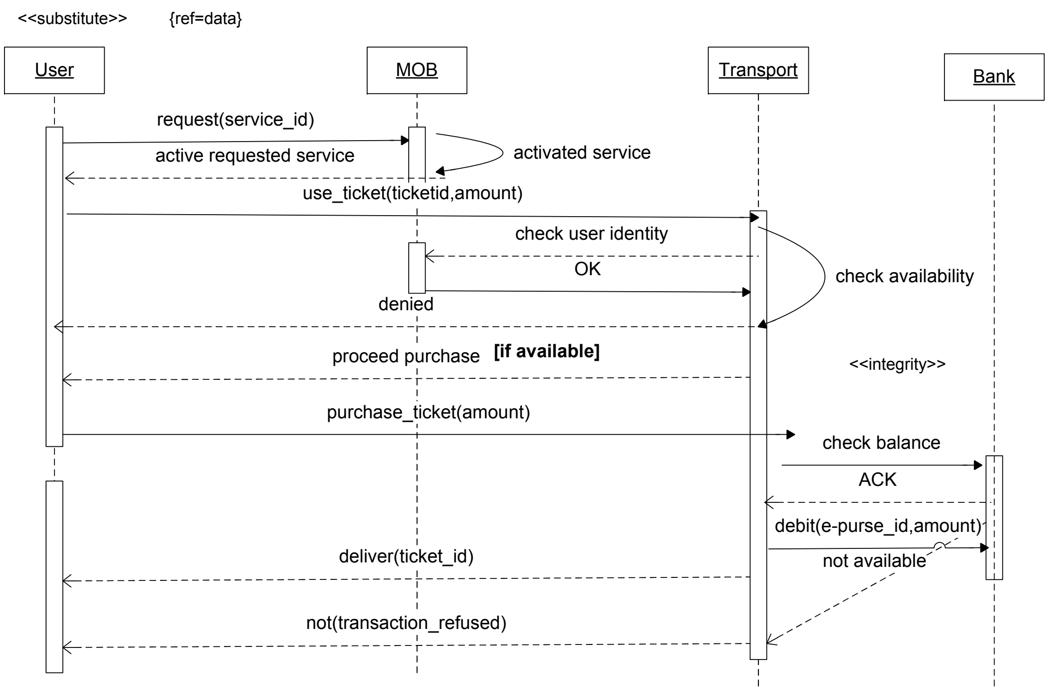


Figure 8.2: Sequence diagram of detailed design

form specification defines duties that require to perform by the all participating entities within the card throughout the life cycle. Once personalization has taken place, then separate activation process is required in order to prevent the card from misuse by the applications before card holder start to use the card. Prior start to use the card, card holder must need to be authenticated to gain access the data and service that belongs to the card. Finally the card is terminated due to expiry, lost or any unplanned problems such as security attack to the card manager or applications security domain. The state must deactivate the card so that the card refuses to authenticate credentials in off-line situations. Figure 8.3 outlines the sequence of state of the card life cycle.

**Modelling change** Main focus is to consider any change relating to Post-issuance phase of the card. There may have several changes at post-Issuance sates of the card life cycle where some of them are foreseen and other is unforeseen. Main challenge lies for the future change of the card life cycle in particular when any application provider change its existence policy and the partners should adapt the amendment of the policy. For instance, the bank has changed its policy so that transport can now not only debit money when adequate amount is available to the e-Purse account but also the amount can be credited from bank to support the customer purchase order, as the bank value added service. Integrity is mainly required to all states of the life cycle to adapt any change from present and future. Authenticity is also through out the life cycle. Therefore « authenticity »,« integrity » stereotypes are included into the state chart diagram. To adapt change, in particular supporting addition and modification three different UMLseCh stereotypes are included « change » in general and in particular « allowed_add », « allowed_modify ». Further dependencies and version_number tags are also included to support the UMLseCh stereotypes.

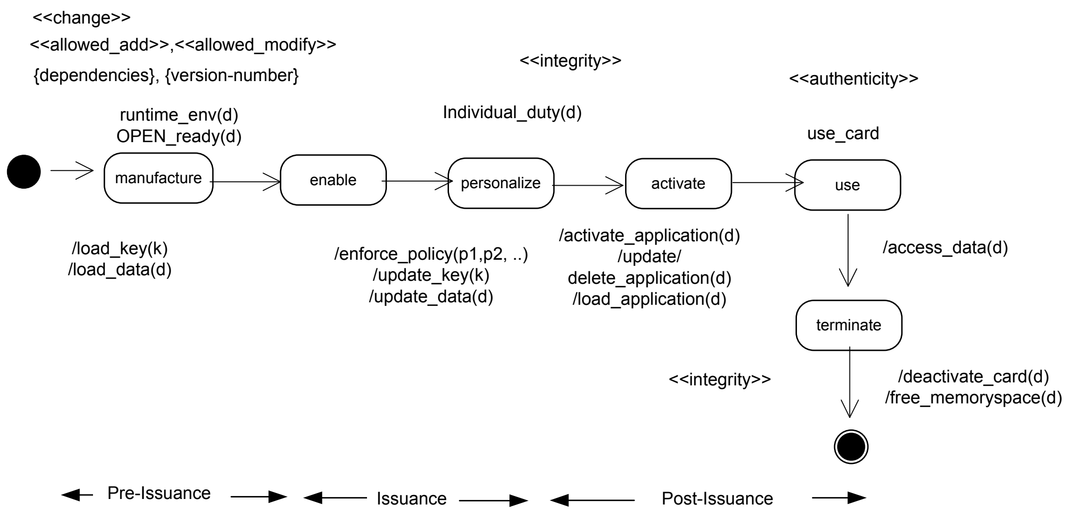


Figure 8.3: Statechart diagram of architectural design

### Class diagram

Class diagram focuses on the attributes and the relevant functions to support the e-purse application under the GP environment. The card manager is the main administrative module for this application. It contains information about its own security domain as well as other application security domain such as e-Purse and e-ticket. Several attributes for the card manager are: card holder global personal identification number (global PIN), privilege , card state, applications ID, and keys. Several operations performed by this class that focus mainly the card content management are: update, delete, and add new entry from the application, install application code, and all cryptographic operations. Note that every class must has certain privilege to perform any operation that related with other class. Transport and bank class also require certain common attributes such as ID, domain, state and privilege to share its service under the GP. Figure 8.4 shows the several class along with attributes and functions for the system actors. They have also their common functions such as check_availability(), check_balance() etc to perform their specific services.

**Modelling change** Card manager would not always responsible to handle all changes during the life time of the card. Sometimes applications have certain privilege under the Global Platform to mange the changes for the specific application within the card content. Therefore ensuring authenticity and integrity is mandatory within the context whenever any changed would like to be performed. Therefore we include « authenticity » and « integrity » stereotype for the class diagram. Additionally to adapt change, UMLseCh stereotypes such as « substitute » and « add » are included along with tag value ref.

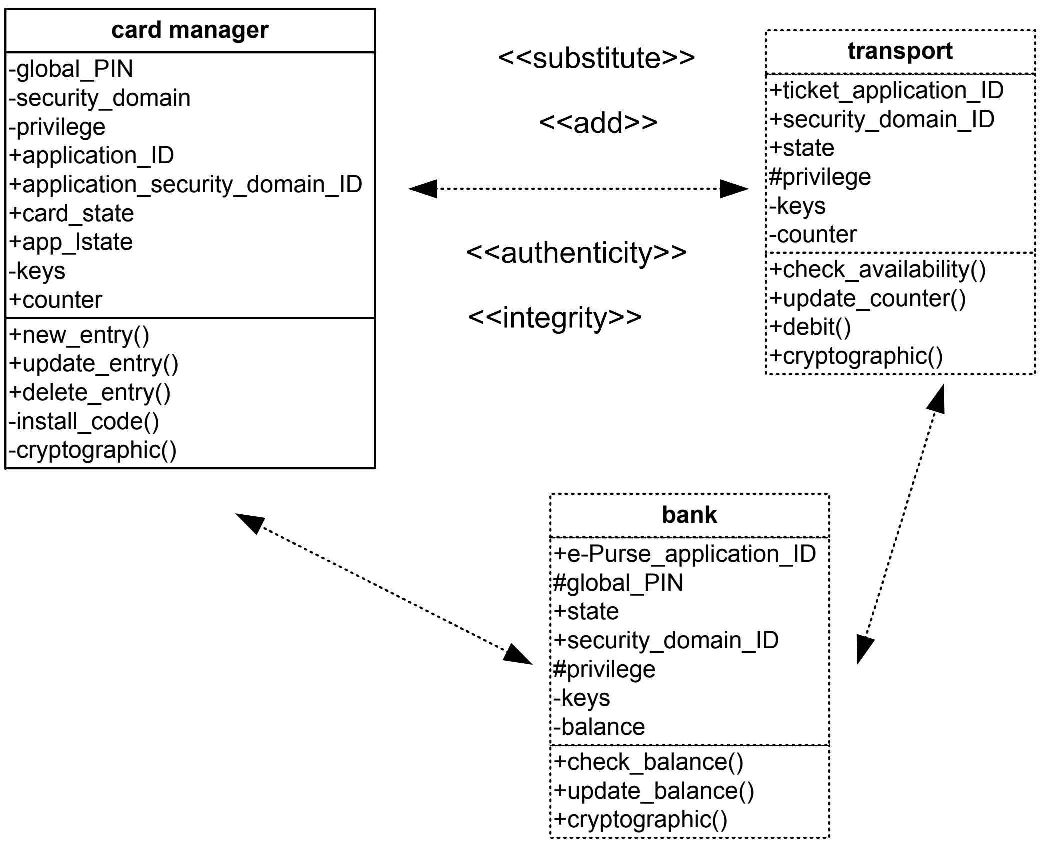


Figure 8.4: Class diagram of architectural design

## Deployment diagram

Deployment diagram mainly outlines the interactions of the participating entities through physical layer of the system. Security domain of the individual is considered as the main component for the deployment diagram. Furthermore we also need to identify node, process, and objects along with security domain. Several nodes such as card manager, bank, and transport perform their task to offer e-ticket, e-purse, and EMV services to the subscripted user. The figure shows three nodes instance card manager, bank and transport. The node instance card manager contains a component instance security domain to support both primary and additional security responsibilities and requirements. Furthermore this component instance supports service such as cryptography operations to support secure interaction among the nodes. Besides, card manager install the loaded application code from the bank and transport to the card. Several objects added within the security domain such as access control, confidentiality and integrity, etc.

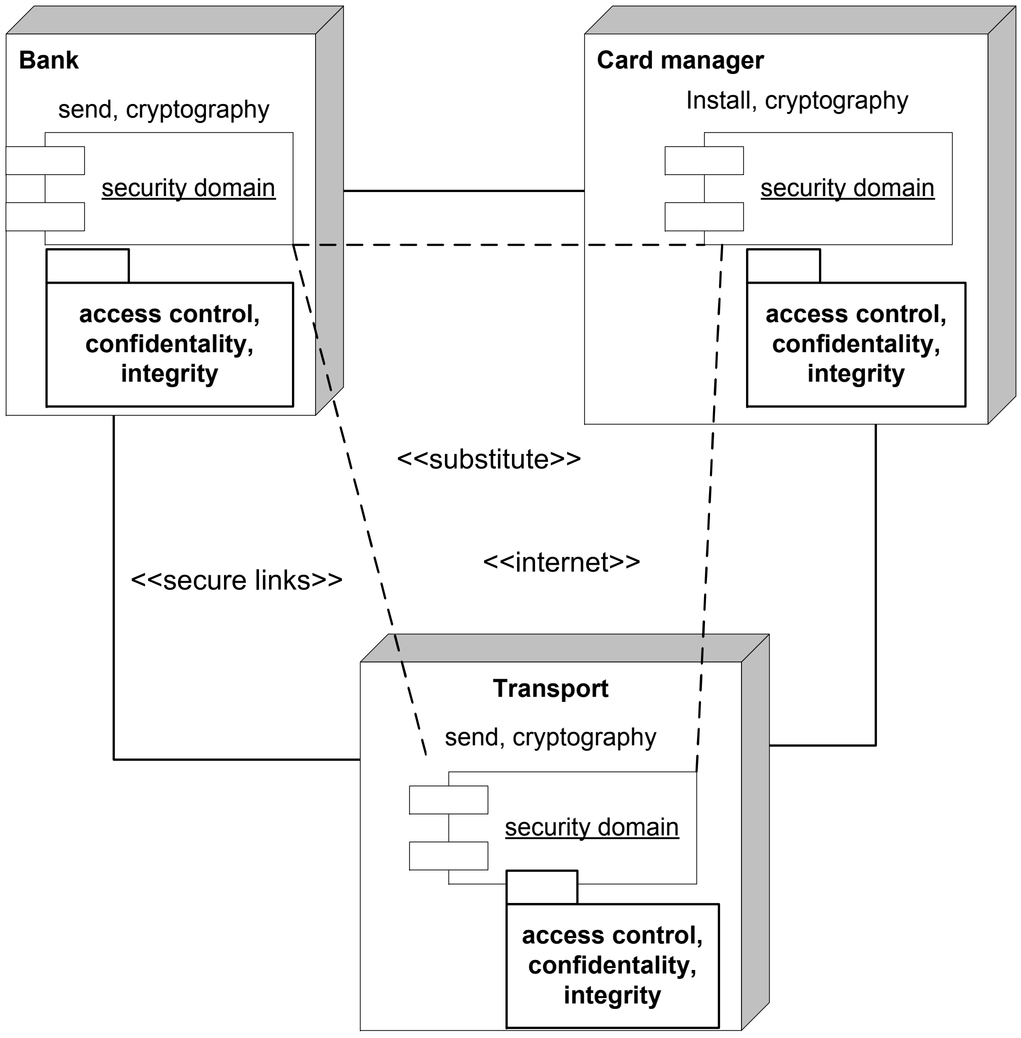


Figure 8.5: Deployment diagram of architectural design

**Modelling change** Similar like other diagram, deployment diagrams also adapt the change from the context. We ass UMLseCh stereotypes to support the change with in the subsystem. All node instances are connected by a link stereotype « Internet ». Figure 8.5 shows the physical architecture of the system. To ensure confidentiality and integrity of the data communicate within the established channel, we need to provide the link secure through « secure links » stereotype.

# 9 Preservation of security properties by system evolution

The evolution of a system is the result of applying a change on one or several of its components. The resulting system from that change will then be the composition of the elements that have been modified with the other elements (i.e. the unchanged elements). In this section, we show how a model component can be represented by a process and how processes can be composed to represent the new system.

As already introduced in Section 2, in UMLsec it is possible to make assumptions about the adversary power to manipulate messages over a network according to the type of link. In the case of an « Internet » link the adversary is able to delete, insert and modify messages between two distributed systems components. This notion is formalized by translating processes into first order logic to represent the constraints necessary for automated security verification [Jür06].

The main goal of the section is to describe a verification method that decides whether the composition of the processes verify or violate the security requirements, secrecy in particular.

## 9.1 A Domain-Specific Language for Cryptographic Protocols

In this section, we shortly recall the main elements of the domain-specific language for cryptographic protocols used in this paper, which was introduced in [Jür09] (where more details and explanation can be found). We consider concurrently executing processes interacting by transmitting sequences of data values over unidirectional FIFO communication channels. Communication is asynchronous in that transmission of a value cannot be prevented by the receiver. Processes are collections of programs that communicate through channels, with the constraint that for each of its output channels $c$ a given process $P$ contains exactly one program $p_c$ that outputs on $c$. This program $p_c$ may take input from any of $P$'s input channels. Intuitively, the program is a description of a value to be output on the channel $c$ in round $n+1$, computed from values found on channels in round $n$. Local state can be maintained through the use of feedback channels, and used for iteration (for instance, for *while* loops).

We assume disjoint sets $\mathcal{D}$ of data values, $\mathbf{Secret}$ of unguessable values (such as "nonces" – freshly generated values supposed to be used only once –, other random values, session keys, or similar), $\mathbf{Keys}$ of keys, $\mathsf{Channels}$ of channels and $\mathbf{Var}$ of variables. Write $\mathbf{Enc} \stackrel{\text{def}}{=} \mathbf{Keys} \cup \mathsf{Channels} \cup \mathbf{Var}$ for the set of *encryptors* that may be used for encryption or decryption. The values communicated over channels are formal *expressions* built from variables, values on input channels, and data values using concatenation. Precisely, the set $\mathbf{Exp}$ of expressions contains the empty expression $\varepsilon$ and the non-empty expressions generated by the grammar given in Figure 9.1. An occurrence of a channel name $c$ refers to the value found on $c$ at the previous instant. The empty expression $\varepsilon$ denotes absence of output on a channel at a given point in time. We write $\mathbf{CExp}$ for the set of *closed* expressions (those containing no subterms in $\mathbf{Var} \cup \mathsf{Channels}$). We write the decryption key corresponding to an encryption key $K$ as $K^{-1}$. In the case of asymmetric encryption, the encryption key $K$ is public, and $K^{-1}$ secret. For symmetric encryption, $K$ and $K^{-1}$ may coincide. We assume $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ for all $E \in \mathbf{Exp}, K, K^{-1} \in \mathbf{Keys}$ and $\mathcal{E}xt_K(\mathcal{S}ign_{K^{-1}}(E)) = E$ for all $E \in \mathbf{Exp}, K, K^{-1} \in \mathbf{Keys}$ (and we assume that no other equations except those following from these hold, unless stated otherwise).

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

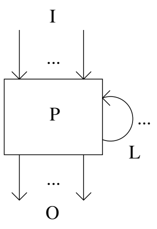
Figure 9.1: Grammar for simple expressions

Programs in the DSL are defined by the grammar given in Figure 9.2. Note that the grammar includes a non-deterministic choice operator. This allows one to use the DSL notation for specifications which admit underspecification, or to admit non-determinism at run-time. In the DSL grammar, variables are introduced in case constructs, which determine their values. The first case construct tests whether $E$ is a key; if so, $p$ is executed, otherwise $p'$. The second case construct tests whether $E$ is a list with head $x$ and tail $y$; if so, $p$ is evaluated, using the actual values of $x, y$; if not, $p'$ is evaluated. In the second case construct, $x$ and $y$ are bound variables. A program is *closed* if it contains no unbound variables. $while$ loops can be coded using feedback channels. From each assignment of expressions to channel names $c \in \mathsf{Channels}$ appearing in a program $p$ (called its *input channels*), $p$ computes an output expression.

| $p ::=$ | programs |
|---|---|
| $E$ | output expression ($E \in \mathbf{Exp}$) |
| *either p or p'* | nondeterministic branching |
| *if $E = E'$ then p else p'* | conditional ($E, E' \in \mathbf{Exp}$) |
| *case E of key do p else p'* | determine if $E$ is a key ($E \in \mathbf{Exp}$) |
| *case E of x :: y do p else p'* | break up list into head::tail ($E \in \mathbf{Exp}$) |

Figure 9.2: Grammar for programs in the Domain-Specific Language

A *process* is of the form $P = (I, O, L, (p_c)_{c \in O \cup L})$ where



- $I \subseteq$ Channels is called the set of its *input channels* and
- $O \subseteq$ Channels the set of its *output channels*,

and where for each $c \in \tilde{O} \overset{\text{def}}{=} O \cup L$, $p_c$ is a closed program with input channels in $\tilde{I} \overset{\text{def}}{=} I \cup L$ (where $L \subseteq$ Channels is called the set of *local channels*). From inputs on the channels in $\tilde{I}$ at a given point in time, $p_c$ computes the output on the channel $c$. We write $I_P$, $O_P$ and $L_P$ for the sets of input, output and local channels of $P$, $K_P \subseteq \mathbf{Keys}$ for the set of private keys and $S_P \subseteq \mathbf{Secret}$ for the set of unguessable values (such as nonces) occurring in $P$. We assume that different processes have disjoint sets of local channels, keys and secrets. Local channels are used to store local state between the execution rounds. Examples for processes specified using this DSL can be found in Section 9.5.

We now explain the translation from programs in the DSL to first-order logic (FOL) formulas. The formalization automatically derives an upper bound for the set of knowledge the adversary can gain. We use a predicate knows$(E)$ meaning that the adversary may get to know $E$ during the execution of the protocol. For any data value $s$ supposed to remain secret, one thus has to check whether one can derive knows$(s)$. The set of predicates defined to hold for a given program in the DSL is defined as follows.

For each publicly known expression $E$, one define knows$(E)$ to hold (in particular for the empty message $\varepsilon$). The fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows is captured by the formula in Fig. 9.3.

Given a program $p$ in our DSL, we define the FOL formula $\phi(p)$ that represents $p$ for the security analysis. The definition is given in Figure 9.4. For the usages of $E \in \mathbf{Exp}$ in Figure 9.4, the assumption is that there are $n$ occurrences of input expressions in $E$, and $E(i_1, \ldots, i_n)$ is the expression derived from $E$ by substituting these occurrences by the variables $i_1, \ldots, i_n$. Similarly, for $E' \in \mathbf{Exp}$, the assumption is that there are $m$ occurrences of input expressions in $E'$, and $E'(j_1, \ldots, j_m)$ is the expression derived from $E'$ by substituting these occurrences by the variables $j_1, \ldots, j_m$. Also, we assume that the predicate key(K) is true iff $K$ is a key. The formula formalizes the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues.

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

Figure 9.3: Structural formulas

Suppose we are given a process $P = (I, O, L, (p_c)_{c \in O})$ where $I, O, L$ are the sets of input, output, and local channels. For each program $p$ associated with a non-local output channel, this gives a predicate PRED($p$) where expressions of the form knows($i_k$) (where $i_k$ is an input received over a local input channel) are substituted by the expression sent($i_k$). For each program $p$ associated with a local output channel, the expression knows($E(i_1, \ldots, i_n)$) in the first line of Figure 9.4 is similarly substituted by sent($E(i_1, \ldots, i_n)$). Additionally, we have sent($\varepsilon$) as an axiom (where $\varepsilon$ represents the empty message). These modifications capture the fact that the adversary cannot read from or write to local channels.

Given a process $P$ specifying a crypto protocol, the logical formula $\psi(P)$ used in the security analysis is the conjunction of the formulas representing the publicly known expressions, the formula in Figure 9.3, and the conjunction of the formulas PRED($p$) for each program $p$ implementing a part of the protocol (with the modifications explained in the previous paragraph). The attack conjecture, for which the automated theorem prover will check whether it is derivable from $\psi(P)$, depends on the security requirements given. For the requirement that the data value $s$ is to be kept secret, the attack conjecture is knows($s$). Thus, we say that *the process $P$ preserves the secrecy of the data value $s$* if $\psi(P) \nvdash$ knows($s$) (i.e. it is not possible to derive knows($s$) from the formulas defined by a protocol). Also, given a condition $C$ on the input/output behavior of the process $P$, we say that *the process $P$ preserves the secrecy of the data value $s$ assuming $C$* if $\psi(P) \wedge C \nvdash$ knows($s$)

We then use an automated theorem prover (such as SPASS [WSH+07]) for verifying security protocols as a "black box": An input file (constructed using the translation to FOL defined above) is presented to the ATP and an output from the ATP is observed. No internal properties of or information from the ATP is used. This allows one to use different ATPs interchangingly (besides SPASS, e.g. e-SETHEO, Vampire and Waldmeister) which helps overcoming restrictions of a given ATP. The results of the theorem prover have to be interpreted as follows: If the conjecture stating for example that the adversary may get to know the secret can be derived from the axioms which formalize the adversary model and the protocol specification, this means that there may be an attack against the protocol. If the attack conjecture can be derived from the protocol axioms, we use an attack generation script programmed in Prolog to construct the attack scenario (which essentially implements the above FOL formalization in Prolog to be able to directly query the variable valuations). If the conjecture cannot be derived from the axioms, this constitutes a proof that the protocol is secure with respect to the security requirement which is the negation of the attack conjecture, because logical derivation is sound and complete with respect to semantic validity for first-order logic. Note that since first-order logic in general is undecidable, it can happen that the ATP is not able to decide whether a given conjecture can be derived from a given set of axioms. However, experience has shown that for a reasonable set of protocols and security requirements, our approach is in fact practical.

$$
\begin{aligned}
&\phi(E) = \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \\
&\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathsf{knows}(E(i_1,\ldots,i_n))\big] \\
&\phi(\mathit{either}\ p\ \mathit{or}\ p') = \phi(p) \wedge \phi(p') \\
&\phi(\mathit{if}\ E = E'\ \mathit{then}\ p\ \mathit{else}\ p') = \\
&\quad \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \ \Rightarrow \\
&\qquad\qquad [E(i_1,\ldots,i_n) = E'(i_1,\ldots,i_n) \ \Rightarrow \phi(p)] \\
&\qquad\qquad \wedge\ [E(i_1,\ldots,i_n) \neq E'(i_1,\ldots,i_n) \ \Rightarrow \phi(p')]\big] \\
&\phi(\mathit{case}\ E\ \mathit{of}\ \mathit{key}\ \mathit{do}\ p\ \mathit{else}\ p') = \\
&\quad \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \ \Rightarrow \\
&\qquad\qquad\qquad [\mathsf{key}(E(i_1,\ldots,i_n)) \ \Rightarrow \phi(p)] \\
&\qquad\qquad\qquad \wedge\ [\neg\mathsf{key}(E(i_1,\ldots,i_n)) \ \Rightarrow \phi(p')]\big] \\
&\phi(\mathit{case}\ E\ \mathit{of}\ x :: y\ \mathit{do}\ p\ \mathit{else}\ p') = \\
&\quad \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \ \Rightarrow \\
&\qquad\qquad \forall h,t.[E(i_1,\ldots,i_n) = h :: t \ \Rightarrow \phi(p[h/x,t/y])] \\
&\qquad\qquad \wedge\ [\neg\exists h,t.E(i_1,\ldots,i_n) = h :: t \ \Rightarrow \phi(p')]\big]
\end{aligned}
$$

Figure 9.4: Definition of $\phi(p)$.

A prototypical implementation of the tool-support, which performs a translation from a state machine representation of the protocol to the FOL formulas, can be downloaded as open source from [Too09].

## 9.2 System Evolution: Atomic Change

In this section we discuss which kind of atomic changes are possible in the context of the DSL defined in Section 9.1, and how to deal with the changes in the context of the security analysis approach explained in that Section.

First, we observe that any evolution from a DSL process $P$ to a process $P'$ can be broken down into a sequence of subsequent modifications of the following kinds:

- deletion of any of the DSL constructs from Figure 9.2,
- insertion of any of the DSL constructs from Figure 9.2.

Thus, we can reduce the impact analysis of evolution on our security analysis to the impact that any of the above atomic steps has on the analysis, which we will do in the following.

**Deletion** We start by considering the case of deletion.

For a given DSL program $p$, we write $\psi_{\mathcal{P},\mathcal{A}}(p)$ for the conjunction of $\phi(p)$, the formulas capturing the given public knowledge $\mathcal{P}$ and the given previous knowledge $\mathcal{A}$ of the adversary, and the general axioms defined in Section 9.1.

We write $p \succcurlyeq p'$ (pronounced "$p$ leaks more knowledge than $p'$") iff for each public knowledge $\mathcal{P}$ and previous attacker knowledge $\mathcal{A}$ and for each $E \in \mathbf{Exp}$ with $\psi_{\mathcal{P},\mathcal{A}}(p') \vdash \mathsf{knows}(E)$, we have $\psi_{\mathcal{P},\mathcal{A}}(p) \vdash \mathsf{knows}(E)$.

**Theorem 1.** *Assume that the program $p'$ evolved from the program $p$ where $p$ and $p'$ are related as in the following case distinctions. Assume we are given the public knowledge $\mathcal{P}$ and the previous adversary knowledge $\mathcal{A}$.*

$p = E$**,** $p' = \varepsilon$ *(where $E \in \mathbf{Exp}$ and $\varepsilon$ is the empty program): This implies $p \succcurlyeq p'$.*

$p = either\ p'\ or\ p''$**:** *This implies $p \succcurlyeq p'$ and $p \succcurlyeq p''$.*

$p = if\ E = E'\ then\ p'\ else\ p''$**:** *For any expression $X \in \mathbf{Exp}$ such that $p$ preserves the secrecy of $X$:*
*$p'$ preserves the secrecy of $X$ assuming $E = E'$ and*
*$p''$ preserves the secrecy of $X$ assuming $E \neq E'$.*

$p = case\ E\ of\ key\ do\ p'\ else\ p''$**:** *For any expression $X \in \mathbf{Exp}$ such that $p$ preserves the secrecy of $X$:*
*$p'$ preserves the secrecy of $X$ assuming $E \in \mathbf{Keys}$ and*
*$p''$ preserves the secrecy of $X$ assuming $E \notin \mathbf{Keys}$.*

$p = case\ E\ of\ x :: y\ do\ p'\ else\ p''$**:** *For any expression $X \in \mathbf{Exp}$ such that $p$ preserves the secrecy of $X$:*
*$p'$ preserves the secrecy of $X$ assuming $\exists x, y. E = x :: y$ and $p''$ preserves the secrecy of $X$ assuming $\neg\exists x, y.\ E = x :: y$.*

**Proof sketch:** The proof proceeds by case distinction along the definition of the grammar of the DSL in Figure 9.2 and the translation to FOL defined in Figure 9.4.

As we can observe, deletion of program elements in the way considered above preserve the level of confidentiality provided by the program, but only up to the conditions in conditionals (because for example, there may be data leakage "hidden" in conditional branches whose condition is never satisfied).

**Insertion** The case of insertion is more complex than deletion.

**Theorem 2.** *Assume that the program $p$ evolved from the program $p'$ where $p$ and $p'$ are related as in the following case distinctions.*

$p = E$, $p' = \varepsilon$ *(where $E \in \mathbf{Exp}$ and $\varepsilon$ is the empty program): For any expression $X$, we have* $\psi(p) \vdash \mathsf{knows}(X)$ *iff* $\psi(p') \wedge \big[\forall i_1, \ldots, i_n.[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow \mathsf{knows}(E(i_1, \ldots, i_n))]\big] \vdash \mathsf{knows}(X)$.

$p = \mathit{either}\ p'\ \mathit{or}\ p''$: *For any expression $X$, we have*
$\psi(p) \vdash \mathsf{knows}(X)$ *iff* $\psi(p') \wedge \psi(p'') \vdash \mathsf{knows}(X)$.

$p = \mathit{if}\ E = E'\ \mathit{then}\ p'\ \mathit{else}\ p''$: *For any expression $X$, we have* $\psi(p) \vdash \mathsf{knows}(X)$ *iff*

$$\big[\forall i_1, \ldots, i_n.[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow \\ (E(i_1, \ldots, i_n) = E'(i_1, \ldots, i_n) \Rightarrow \phi(p')) \\ \wedge (E(i_1, \ldots, i_n) \neq E'(i_1, \ldots, i_n) \Rightarrow \phi(p'))]\big] \vdash \mathsf{knows}(X).$$

$p = \mathit{case}\ E\ \mathit{of}\ \mathit{key}\ \mathit{do}\ p'\ \mathit{else}\ p''$: *For any expression $X$, we have* $\psi(p) \vdash \mathsf{knows}(X)$ *iff*

$$\big[\forall i_1, \ldots, i_n.[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow \\ (\mathsf{key}(E(i_1, \ldots, i_n)) \Rightarrow \phi(p)) \\ \wedge (\neg\mathsf{key}(E(i_1, \ldots, i_n)) \Rightarrow \phi(p'))]\big] \vdash \mathsf{knows}(X).$$

$p = \mathit{case}\ E\ \mathit{of}\ x :: y\ \mathit{do}\ p'\ \mathit{else}\ p''$: *For any expression $X$, we have* $\psi(p) \vdash \mathsf{knows}(X)$ *iff*

$$\big[\forall i_1, \ldots, i_n.[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow \\ \forall h, t.(E(i_1, \ldots, i_n) = h :: t \Rightarrow \phi(p[h/x, t/y])) \\ \wedge\ (\neg\exists h, t.E(i_1, \ldots, i_n) = h :: t \Rightarrow \phi(p'))]\big] \vdash \mathsf{knows}(X).$$

**Proof sketch:** Again, the proof proceeds by case distinction along the definition of the grammar of the DSL in Figure 9.2 and the translation to FOL defined in Figure 9.4.

| | |
|---|---|
| $[E](\vec{M}) = \{E(\vec{M})\}$ | where $E \in \mathbf{Exp}$ |
| $[either\ p\ or\ p'](\vec{M}) = [p](\vec{M}) \cup [p'](\vec{M})$ | |
| $[if\ E = E'\ then\ p\ else\ p'](\vec{M}) = [p](\vec{M})$ | if $[E](\vec{M}) = [E'](\vec{M})$ |
| $[if\ E = E'\ then\ p\ else\ p'](\vec{M}) = [p'](\vec{M})$ | if $[E](\vec{M}) \neq [E'](\vec{M})$ |
| $[case\ E\ of\ key\ do\ p\ else\ p'](\vec{M}) = [p](\vec{M})$ | if $[E](\vec{M}) \in \mathbf{Keys}$ |
| $[case\ E\ of\ key\ do\ p\ else\ p'](\vec{M}) = [p'](\vec{M})$ | if $[E](\vec{M}) \notin \mathbf{Keys}$ |
| $[case\ E\ of\ x :: y\ do\ p\ else\ p'](\vec{M}) = [p[h/x, t/y]](\vec{M})$ | |
| if $[E](\vec{M}) = h :: t$ where $h \neq \varepsilon$ and $h$ is not of the form $h_1 :: h_2$ for $h_1, h_2 \neq \varepsilon$ | |
| $[case\ E\ of\ x :: y\ do\ p\ else\ p'](\vec{M}) = [p'](\vec{M})$ | if $[E](\vec{M}) = \varepsilon$ |

Figure 9.5: Definition of $[p](\vec{M})$.

## 9.3 System Evolution: Architectural Change

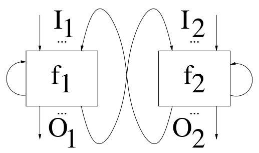
In this section, we investigate several types of evolution at a higher level of abstraction than the atomic changes considered in the previous section, namely architectural changes.

For this, we first need to recall some information about the formal semantics of our DSL from [Jür09].

For a formal semantics of the DSL, we use stream-processing functions (cf. [BS01a]). We write $\mathbf{Stream}_C \stackrel{\text{def}}{=} (\mathbf{CExp}^\infty)^C$ (where $C \subseteq \mathsf{Channels}$) for the set of $C$-indexed tuples of (finite or infinite) sequences of closed expressions. The elements of this set are called *streams*, specifically *input streams* (resp. *output streams*) if $C$ denotes the set of non-local input (resp. output) channels of a process $P$. Each stream $\vec{s} \in \mathbf{Stream}_C$ consists of components $\vec{s}(c)$ (for each $c \in C$) that denote the sequence of expressions appearing at the channel $c$. The $n^{th}$ element $x_n$ in such a sequence $\vec{s}(c) = (x_1, x_2, x_3, \ldots, x_n, \ldots)$ consisting of expressions $x_i$ is the expression appearing at time $t = n$. A function $f : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ from streams to sets of streams is called a *stream-processing function*.

The composition of two stream-processing functions $f_i : \mathbf{Stream}_{I_i} \to \mathcal{P}(\mathbf{Stream}_{O_i})$ $(i = 1, 2)$ with $O_1 \cap O_2 = \emptyset$ is defined as:

$f_1 \otimes f_2 : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$
(with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$,
$O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$).



where $f_1 \otimes f_2(\vec{s}) \stackrel{\text{def}}{=} \{\vec{t}\restriction_O : \vec{t}\restriction_I = \vec{s}\restriction_I \wedge \vec{t}\restriction_{O_i} \in f_i(\vec{s}\restriction_{I_i})\ (i = 1, 2)\}$ (where $\vec{t}$ ranges over $\mathbf{Stream}_{I \cup O}$). For $\vec{t} \in \mathbf{Stream}_C$ and $C' \subseteq C$, the restriction $\vec{t}\restriction_{C'} \in \mathbf{Stream}_{C'}$ is defined by $\vec{t}\restriction_{C'}(c) = \vec{t}(c)$ for each $c \in C'$. Since the operator $\otimes$ is associative and commutative [BS01a], we can define a generalised composition operator $\bigotimes_{i \in I} f_i$ for a set $\{f_i : i \in I\}$ of stream-processing functions.

A process $P = (I, O, L, (p_c)_{c\in O})$ is modelled by a stream-processing function $[\![P]\!] : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ from input streams to sets of output streams. For any closed program $p$ with input channels in $\tilde{I} \overset{\text{def}}{=} I \cup L$ and any $\tilde{I}$-indexed tuple of closed expressions $\vec{M} \in \mathbf{CExp}^{\tilde{I}}$ we define a set of expressions $[p](\vec{M}) \in \mathcal{P}(\mathbf{CExp})$ in Fig. 9.5, so that $[p](\vec{M})$ is the expression that results from running $p$ once, when the channels have the initial values given in $\vec{M}$. We write $E(\vec{M})$ for the result of substituting each occurrence of $c \in \tilde{I}$ in $E$ by $\vec{M}(c)$ and $p[E/x]$ for the outcome of replacing each free occurrence of $x$ in process $P$ with the term $E$, renaming variables to avoid capture. Then any program $p_c$ (for $c \in \mathsf{Channels}$) defines a stream-processing function $[p_c] : \mathbf{Stream}_{\tilde{I}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}})$ as follows. Given $\vec{s} \in \mathbf{Stream}_{\tilde{I}}$, let $[p_c](\vec{s})$ consist of those $\vec{t} \in \mathbf{Stream}_{\{c\}}$ such that

- $\vec{t}_0 \in [p_c](\varepsilon, \ldots, \varepsilon)$
- $\vec{t}_{n+1} \in [p_c](\vec{s}_n)$ for each $n \in \mathbb{N}$.

Finally, a process $P = (I, O, L, (p_c)_{c\in\tilde{O}})$ (where $\tilde{O} \overset{\text{def}}{=} O \cup L$) is interpreted as the composition $[\![P]\!] \overset{\text{def}}{=} \bigotimes_{c\in\tilde{O}}[p_c]$.

## 9.4 Reduction of non-determinism

We consider architectural change with the goal of reducing the non-determism admitted in the running system.

**Definition 1.** *For processes $P$ and $P'$ with $I_P = I_{P'}$ and $O_P = O_{P'}$ we define $P \rightsquigarrow P'$ if for each $\vec{s} \in \mathbf{Stream}_{I_P}$, $[\![P]\!](\vec{s}) \supseteq [\![P']\!](\vec{s})$ (i.e. each possible execution of $P'$ is also a possible execution of $P$).*

**Example** $(\mathit{either}\ p\ \mathit{or}\ q) \rightsquigarrow p$ and $(\mathit{either}\ p\ \mathit{or}\ q) \rightsquigarrow q$ for any programs $p, q$.

**Theorem 3.**

- *If $P$ preserves the secrecy of $m$ and $P \rightsquigarrow P'$ then $P'$ preserves the secrecy of $m$.*
- *If $P$ preserves the secrecy of $m$ assuming $C$ (for any condition $C$ on the input/output behavior of the process $P$) and $P \rightsquigarrow P'$ then $P'$ preserves the secrecy of $m$ assuming $C$.*

**Proof sketch:** The proof follows immediately from the definitions of secrecy and the $\rightsquigarrow$ relation, since secrecy is defined over the set of communicated values, and this set can only be reduced under $\rightsquigarrow$.

Note that the reduction of non-determism is technically related to the refinement by removing under-specification considered in [BS01a, Jür09], although it is used in a different context.

**Restriction of Interfaces** We consider architectural change with the goal of restricting the interface of a part of the system.

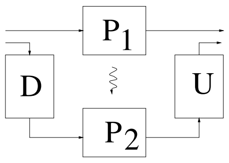
**Theorem 4.**

- *If $P$ preserves the secrecy of $m$ then $P$ preserves the secrecy of $m$ assuming $C$ (for any condition $C$ on the input/output behavior of the process $P$).*

**Proof sketch:** The proof follows immediately from the definitions of secrecy, since secrecy is defined over the set of possible system executions, and this set can only be reduced by imposing the condition $C$.

**Refactoring**

**Definition 2.** *Let $P_1, P_2, D$ and $U$ be processes with $I_{P_1} = I_D$, $O_D = I_{P_2}$, $O_{P_2} = I_U$, and $O_U = O_{P_1}$. We define $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ to hold if $P_1 \rightsquigarrow D \otimes P_2 \otimes U$.*



**Example** Suppose we the formulas in Figure 9.6 hold. Then we have $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$.

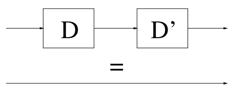
- $P_1 = (\{c\}, \{d\}, p_d \overset{\text{def}}{=} \textit{if } \mathsf{inp}(c) = 1 \textit{ then } 2 \textit{ else } 3)$,
- $P_2 = (\{c'\}, \{d'\}, p_{d'} \overset{\text{def}}{=} \textit{if } \mathsf{inp}(c') = 4 \textit{ then } 5 \textit{ else } 6)$,
- $D = (\{c\}, \{c'\}, p_{c'} \overset{\text{def}}{=} \textit{if } \mathsf{inp}(c) = 1 \textit{ then } 4 \textit{ else } \varepsilon)$ and
- $U = (\{d'\}, \{d\}, p_d \overset{\text{def}}{=} \textit{if } \mathsf{inp}(d') = 5 \textit{ then } 2 \textit{ else } 3$.

Figure 9.6: Example processes

For the next preservation result we need the following concepts. Given a stream $\vec{s} \in \mathbf{Stream}_X$ and a bijection $\iota : Y \rightarrow X$ we write $\vec{s}_\iota$ for the stream in $\mathbf{Stream}_Y$

obtained from $\vec{s}$ by renaming the channel names using $\iota$: $\vec{s}_\iota(y) = \vec{s}(\iota(y))$.

Given processes $D, D'$ with $O_D = I_{D'}$ and $O_{D'} \cap I_D = \emptyset$ and a bijection $\iota : O_{D'} \to I_D$ such that $[\![D]\!] \otimes [\![D']\!](\vec{s}) = \{\vec{s}_\iota\}$ for each $\vec{s} \in \mathbf{Stream}_{I_D}$, we say that $D$ is a *left inverse* of $D'$ and $D'$ is a *right inverse* of $D$.



**Example** $p_d \stackrel{\text{def}}{=} 0 :: \mathsf{inp}(c)$ is a left inverse of $p_e \stackrel{\text{def}}{=} \mathit{case}\ \mathsf{inp}(c)\ \mathit{of}\ h :: t\ \mathit{do}\ t\ \mathit{else}\ \varepsilon$.

We write $S \circ R \stackrel{\text{def}}{=} \{(x, z) : \exists y.(x, y) \in R \wedge (y, z) \in S\}$ for the usual composition of relations $R, S$ and generalize this to functions $f : X \to \mathcal{P}(Y)$ by viewing them as relations $f \subseteq X \times Y$. We note that a condition $C$ on the input/output behavior of a process $P$ can be viewed as a relation $C \subseteq \mathbf{Stream}_{O_P} \times \mathbf{Stream}_{I_P}$.

**Theorem 5.** *Let $P_1, P_2, D$ and $U$ be processes with $I_{P_1} = I_D$, $O_D = I_{P_2}$, $O_{P_2} = I_U$ and $O_U = O_{P_1}$ and such that $D$ has a left inverse $D'$ and $U$ a right inverse $U'$. Let $m \in (\mathbf{Secret} \cup \mathbf{Keys}) \setminus \bigcup_{Q \in \{D', U'\}}(S_Q \cup K_Q)$.*

- *If $P_1$ preserves the secrecy of $m$ and $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ then $P_2$ preserves the secrecy of $m$.*
- *If $P_1$ preserves the secrecy of $m$ assuming $C \subseteq \mathbf{Stream}_{O_{P_1}} \times \mathbf{Stream}_{I_{P_1}}$ and $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ then $P_2$ preserves the secrecy of $m$ assuming $[\![U']\!] \circ C \circ [\![D']\!]$.*

**Proof sketch:** This statement follows directly from Theorem 3 and the definition of refactoring.

This notion of refactoring considered here is similar on a technical level to interface refinement defined in [BS01a] and already considered in [Jür09], although again it is used for a different purpose.

**Change up to a Set of Behaviors $C$** We consider the situation where a process has changed arbitrarily, except that for a set of input/output behaviors $C$ it has remained the same.

**Definition 3.** *Let $P_1$ and $P_2$ be processes with $I_{P_1} = I_{P_2}$ and $O_{P_1} = O_{P_2}$. We define $P_1 \rightsquigarrow_C P_2$ for a total relation $C \subseteq \mathbf{Stream}_{O_{P_1}} \times \mathbf{Stream}_{I_{P_1}}$ to hold if for each $\vec{s} \in \mathbf{Stream}_{I_{P_1}}$ and each $\vec{t} \in [\![P_2]\!]$, $(\vec{t}, \vec{s}) \in C$ implies $\vec{t} \in [\![P_1]\!]$.*

**Example** $p \leadsto_C (\textit{if}\ \mathsf{inp}(c) = \mathbf{emergency}\ \textit{then}\ q\ \textit{else}\ p)$ for $C = \{(\vec{t}, \vec{s}) : \forall n. \vec{s}_n \neq \mathbf{emergency}\}$.

**Theorem 6.**
*Given total relations $C, D \subseteq \mathbf{Stream}_{O_P} \times \mathbf{Stream}_{I_P}$ with $C \subseteq D$, if $P$ preserves the secrecy of $m$ assuming $C$ and $P \leadsto_D P'$ then $P'$ preserves the secrecy of $m$ assuming $C$.*

**Proof sketch:** Again, this statement follows directly from Theorem 3 and the definition of change up to a set of behaviors $C$.

The notion of change up to a set of behaviors $C$ is technically similar to conditional refinement from [BS01a] already considered in [Jür09], although again it is used for a different purpose.

**Modular Reuse of Verification Results** In the situation where part of the system remains unchanged (as considered in the previous subsection), we would like to be able to reuse earlier verification results in a modular way by making use of security assertions for system components which are generated during the automated security analysis process. Note that our goal is not to investigate the problem of compositionality of security properties, but to collect the logical formulas generated from different program fragments together and store them in assertions to be reused in a later analysis.

A set of security assertions for a program part $\mathsf{p}$ consists of statements $\mathsf{derived}(\mathsf{L}, \mathsf{C}, \mathsf{E})$ where $\mathsf{L}$ is a list of variables, $\mathsf{C}$ is a condition over the variables in $\mathsf{L}$, and $\mathsf{E}$ is an expression which may contain free variables from $\mathsf{L}$.

These assertions mean that the set of adversary knowledge is contained in the set of expressions $\mathsf{E}$ that can be constructed by instantiating the variables from $\mathsf{L}$ with values that themselves can be derived this way for $\mathsf{p}$ and which fulfill the condition $\mathsf{C}$. This way of formulating modular assertions on parts of the code is inspired by the Assumption/Commitment specification approaches (see e.g. [BS01a]).

More formally, a program fragment $\mathsf{p}$ has an associated set $\mathcal{L}$ of statements $\mathsf{derived}(\mathsf{L}, \mathsf{C}, \mathsf{E})$ if according to the security analysis in Section 9.1, an adversary gets to know only those expressions that can be constructed recursively in the following way.

- For all valuations of the variables $\mathsf{v}$ in $\mathsf{L}$ fulfilling $\mathsf{C}$ and such that $\mathsf{knows}(\mathsf{v})$ holds, we have $\mathsf{knows}(\bar{\mathsf{E}})$ for the corresponding valuation $\bar{\mathsf{E}}$ of $\mathsf{E}$.

Note that for a single protocol run of $\mathsf{p}$, it is sufficient to use a finite set of such assertions, namely those defined in the following way:

- For each expression E in the initial knowledge set of the adversary or the set of previously known expressions for $\mathsf{p}$, we define an assertion of the form $\mathsf{derived}([], \mathsf{true}, \mathsf{E})$ where $[]$ is the empty list and $\mathsf{true}$ is the Boolean constant.
- For each list of predicates of the form

  $$\mathsf{PRED}(\mathsf{TR_k}) \;\equiv\; \bar{\mathsf{i}}_\mathsf{k} \wedge \mathsf{c_k} \;\Rightarrow\; \bar{\mathsf{a}}_\mathsf{k} \wedge \mathsf{PRED}(\mathsf{TR_{k+1}})$$

  defined for $\mathsf{p}$ in Section 9.1 for $\mathsf{k}$ up to some number $\mathsf{n}$, we define an assertion of the form $\mathsf{derived}(\mathsf{L}, \mathsf{C}, \mathsf{E})$ where L is the list of free variables in that list of predicates, C is a nested implication of the form

  $$\mathsf{IMP}(\mathsf{TR_k}) \;\equiv\; \mathsf{c_k} \;\Rightarrow\; \bar{\mathsf{a}}_\mathsf{k} \wedge \mathsf{IMP}(\mathsf{TR_{k+1}})$$

  (where $\bar{\mathsf{a}}_\mathsf{k} = \mathsf{a_k}$ in case $\mathsf{a_k}$ is of the form $\mathsf{localvar} = \mathsf{value}$ and $\bar{\mathsf{a}}_\mathsf{k} = \mathsf{true}$ otherwise), and E is the concatenation of the actions $\mathsf{a_k}$ that are of the form $\mathsf{ak} = \mathsf{outpattern}$.

For atoms that are freshly generated in each protocol run, we assume that these are given as methods with a sequence number of the protocol run as free variables. Then one can obtain a finite set of assertions bounding the adversary knowledge by closing the above assertions with forall quantification over the sequence number variables that they contain.

In order to make sure that the set of expressions generated by $\mathcal{L}$ does not contain any expressions that according to the security requirements for $\mathsf{p}$ are supposed to remain secret (because otherwise the security assertions would be practically unusable), the code fragment $\mathsf{p}$ is first analyzed using the approach in Section 9.1. We then say the $\mathcal{L}$ is a *secure bound generator for the adversary knowledge*. Of course, there can only be a secure bound generator for a program fragment $\mathsf{p}$ if $\mathsf{p}$ in fact fulfills its secrecy requirements.

To analyze a program fragment $\mathsf{p}$ carrying a set of assertions $\mathcal{L}$ one takes the formulas generated from the approach in Section 9.1 and adds for each assertion of the form $\mathsf{derived}(\mathsf{L}, \mathsf{C}, \mathsf{E})$ an axiom of the form as in Figure 9.7 to the input file that is to be

```
![v1,...,vn]: knows(v1) &...& knows(vn)
             & C(v1,...,vn) => knows(E)
```

Figure 9.7: Axiom

analyzed by the ATP (where L is assumed to be the list of variables $\mathsf{v1}, ..., \mathsf{vn}$ and $\mathsf{C}(\mathsf{v1}, ..., \mathsf{vn})$ is the instantiation of the condition C with the variables $\mathsf{v1}, ..., \mathsf{vn}$).

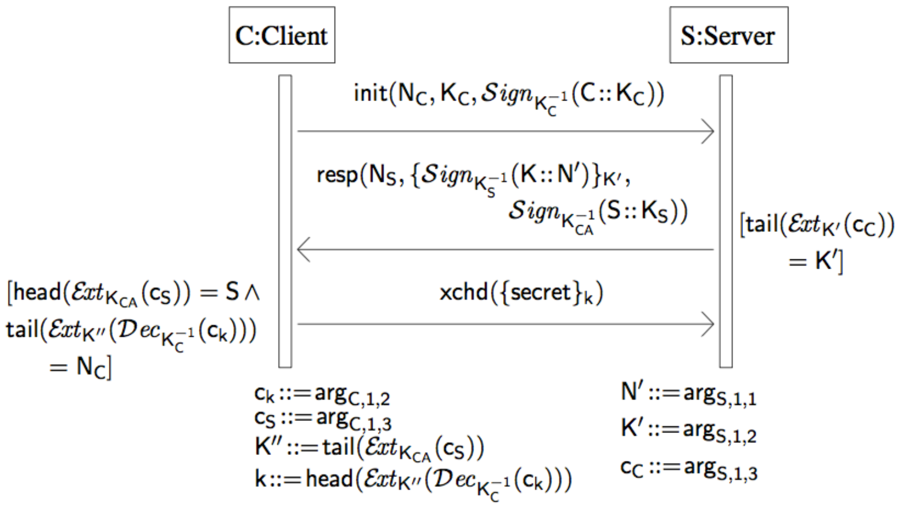

C:Client
S:Server
init(N_C, K_C, Sign_{K_C^{-1}}(C::K_C))
resp(N_S, {Sign_{K_S^{-1}}(K::N')}_{K'}, Sign_{K_CA^{-1}}(S::K_S))
xchd({secret}_k)
[tail(Ext_{K'}(c_C)) = K']
[head(Ext_{K_CA}(c_S)) = S ∧ tail(Ext_{K''}(Dec_{K_C^{-1}}(c_k))) = N_C]
c_k ::= arg_{C,1,2}
c_S ::= arg_{C,1,3}
K'' ::= tail(Ext_{K_CA}(c_S))
k ::= head(Ext_{K''}(Dec_{K_C^{-1}}(c_k)))
N' ::= args_{S,1,1}
K' ::= args_{S,1,2}
c_C ::= args_{S,1,3}

Figure 9.8: Variant of the TLS handshake

## 9.5 Example: Secure Channel Evolution

As an example for how our approach allows us to deal with system evolution, we consider the implementation of a secure channel $\mathbb{W}$ from a client $\mathbb{C}$ to a server $\mathbb{S}$ using the handshake protocol of a variant of TLS.

**Example: A Variant of the TLS Protocol** We specify a variant of the handshake protocol of TLS[2] as proposed at IEEE Infocom 1999[3] (note that this is not the variant of TLS in common use). To show applicability of our approach, we exhibit a security vulnerability, suggest a correction, and verify it. The goal of the protocol is to let a client send a secret over an untrusted communication link to a server in a way that provides secrecy and server authentication, by using symmetric session keys. The central part of the specification of this protocol is shown in Fig. 9.8. The two protocol participants client and server are connected by an Internet connection. The value secret which is exchanged encrypted in the last message of the protocol is required to remain secret.

Depicted in Fig. 9.8, the protocol proceeds as we explain in the following. Here we assume that the set $\mathbf{Var}$ contains elements $\mathsf{arg}_{O,l,n}$ for each $O \in \mathsf{Obj}(D)$ and numbers

[2]TLS (transport layer security) is the successor of the Internet security protocol SSL (secure sockets layer).

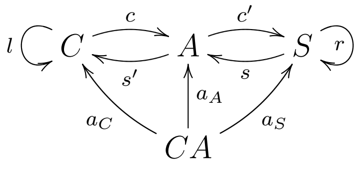
[3]V. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In Conference on Computer Communications (IEEE Infocom), pages 717–725. IEEE, Mar. 1999.

$l$ and $n$, representing the $n$th argument of the operation that is supposed to be the $l$th operation received by $O$ according to the sequence diagram $D$. The client $C$ initiates the protocol by sending the message $\mathsf{init}(\mathsf{N_C}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$ to the server $S$. Suppose that the condition $[\mathsf{tail}(\mathcal{E}xt_{\mathsf{K'}}(\mathsf{c_C}))=\mathsf{K'}]$ holds, where $K' ::= \mathsf{arg}_{S,1,2}$ and $\mathsf{c_C} ::= \mathsf{arg}_{\mathsf{S},1,3}$. That is, the key $K_C$ contained in the signature matches the one transmitted in the clear. In that case, $S$ sends the message $\mathsf{resp}\big(\mathsf{N_S}, \{\mathcal{S}ign_{\mathsf{K_S^{-1}}}(\mathsf{K} :: \mathsf{N'})\}_{\mathsf{K'}}, \mathcal{S}ign_{K_{\mathsf{CA}}^{-1}}(S :: K_S)\big)$ back to $C$ (where $\mathsf{N'} ::= \mathsf{arg}_{\mathsf{S},1,1}$). Then if the condition

$$[\mathsf{head}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))=\mathsf{S} \wedge \mathsf{tail}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))=\mathsf{N_C}]$$

holds, where $\mathsf{c_k} ::= \mathsf{arg}_{\mathsf{C},1,2}$, $\mathsf{c_S} ::= \mathsf{arg}_{\mathsf{C},1,3}$, and $\mathsf{K''} ::= \mathsf{tail}(\mathcal{E}xt_{K_{\mathsf{CA}}}(c_S))$ (that is, the certificate is actually for $S$ and the correct nonce is returned), $C$ sends $\mathsf{xchd}(\{\mathsf{s_i}\}_{\mathsf{k}})$ to $S$, where $\mathsf{k} ::= \mathsf{head}(\mathcal{E}xt_{\mathsf{K''}}(\mathcal{D}ec_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

The goal is thus to let a client $C$ send a master secret $m \in \mathbf{Secret}$ to a server $S$ in a way that provides confidentiality and server authentication. The protocol uses both RSA encryption and signing. Thus in this and the following section we assume also the equation $\{\mathcal{D}ec_{K^{-1}}(E)\}_K = E$ to hold (for each $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$). We also assume that the set of data values $\mathcal{D}$ includes process names such as $C, S, Y, \ldots$ and a message $abort$. The protocol assumes that there is a secure (wrt. integrity) way for $C$ to obtain the public key $K_{CA}$ of the certification authority, and for $S$ to obtain a certificate $\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)$ signed by the certification authority that contains its name and public key. The adversary may also have access to $K_{CA}$, $\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)$ and $\mathcal{S}ign_{K_{CA}^{-1}}(Z :: K_Z)$ for an arbitrary process $Z$. The channels between the participants are thus as depicted in Figure 9.9.



$$\begin{tikzcd}
C \arrow[loop left, "l"] \arrow[r, bend left, "c"] & A \arrow[l, bend left, "s'"] \arrow[r, bend left, "c'"] & S \arrow[l, bend left, "s"] \arrow[loop right, "r"] \\
& CA \arrow[ul, "a_C"] \arrow[u, "a_A"] \arrow[ur, "a_S"'] &
\end{tikzcd}$$

Figure 9.9: Channels between participants

Now we define the protocol using our domain-specific language in Figure 9.10 (here and in the following we denote a program with output channel $c$ simply as $c$ for readability). For readability we leave out a time-stamp, a session id, the choice of cipher suite and compression method and the use of a temporary key by $S$ since these are not relevant for the weakness. We use syntactic sugar by extending the case list construct to lists of finite length and by using pattern matching, and we also leave out some $case\ \ of\ key\ do\ \ else$ constructs to avoid cluttering. Similarly, we use the expression $\mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c')) \in \mathbf{Data} \cup \mathbf{Secret}$ as a short-hand for nested $if\ \ then\ \ else$ statements which iteratively check equality of $\mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c'))$ with all values in the

$$
\begin{aligned}
c &\overset{\text{def}}{=} \mathit{if}\ \mathsf{inp}(l) = \varepsilon\ \mathit{then}\ N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)\\
&\qquad \mathit{else\ case}\ \mathsf{inp}(s')\ \mathit{of}\ s_1 :: s_2 :: s_3\\
&\qquad\quad \mathit{do\ case}\ \mathcal{E}xt_{\mathsf{inp}(a_C)}(s_3)\ \mathit{of}\ S :: x\\
&\qquad\qquad \mathit{do\ if}\ \{\mathcal{D}ec_{K_C^{-1}}(s_2)\}_x = y :: N_C\ \mathit{then}\ \{m\}_y\\
&\qquad\qquad\quad \mathit{else}\ \text{abort}\\
&\qquad\qquad \mathit{else}\ \text{abort}\\
&\qquad\quad \mathit{else}\ \varepsilon\\
l &\overset{\text{def}}{=} 0\\
s &\overset{\text{def}}{=} \mathit{case}\ \mathsf{inp}(c')\ \mathit{of}\ c_1 :: c_2 :: c_3\\
&\qquad \mathit{do}\ \ \mathit{case}\ \mathcal{E}xt_{c_2}(c_3) \mathit{of}\ x :: c_2\ \mathit{do}\\
&\qquad\quad N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1)\}_{c_2} :: \mathsf{inp}(a_S)\\
&\qquad\qquad \mathit{else}\ \text{abort}\\
&\qquad \mathit{else}\ \varepsilon\\
r &\overset{\text{def}}{=} \mathit{if}\ \mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c')) \in \mathbf{Data} \cup \mathbf{Secret}\ \mathit{then}\\
&\quad \mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c'))\ \mathit{else}\ \varepsilon\\
a_C &\overset{\text{def}}{=} K_{CA}\\
a_A &\overset{\text{def}}{=} K_{CA} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S) :: \mathcal{S}ign_{K_{CA}^{-1}}(Z :: K_Z)\\
a_S &\overset{\text{def}}{=} \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)
\end{aligned}
$$

Figure 9.10: Protocol definition

finite set $\mathbf{Data} \cup \mathbf{Secret}$. Here the local channel $l$ of $C$ only ensures that $C$ initiates the handshake protocol only once (by sending out an arbitrary message ($0$) so that only at the start of the program execution the first condition in the definition of the program on channel $c$ will hold). The exchanged key is symmetric, i. e. we have $K_{CS}^{-1} = K_{CS}$. The values sent on $a_A$ signify that we allow $A$ to eavesdrop on $a_C$ and $a_S$ and to obtain the certificate issued by CA of some third party. The local channel $r$ of the server will contain the decrypted secret (which is assumed to be a value in the set $\mathbf{Data} \cup \mathbf{Secret}$) after it has been communicated successfully.

The ATP e-SETHEO returns as an output that the conjection $\mathsf{knows}(\mathsf{secret})$ can be derived from the defined rules (within three seconds[4]). For this example the attack tracking tool needs around 20 seconds to produce the attack which is visualized in Fig. 9.11.

[4]On a SunFire 3800 (4 processors, 6 GByte RAM, Solaris 9).

$$\begin{tikzcd}
C \arrow[r, "N_C::K_C::Sign_{K_C^{-1}}(C::K_C)"] & A \arrow[r, "N_C::K_A::Sign_{K_A^{-1}}(C::K_A)"] & S \\
C & A \arrow[l, "N_S::\{Sign_{K_S^{-1}}(K_{CS}::N_C)\}_{K_C}::Sign_{K_{CA}^{-1}}(S::K_S)"'] & S \arrow[l, "N_S::\{Sign_{K_S^{-1}}(K_{CS}::N_C)\}_{K_A}::Sign_{K_{CA}^{-1}}(S::K_S)"'] \\
C \arrow[r, "\{m\}_{K_{CS}}"] & A \arrow[r, "\{m\}_{K_{CS}}"] & S
\end{tikzcd}$$

Figure 9.11: Attack against TLS Variant

We can fix this problem as follows: Let $S'$ be the process derived from $S$ by substituting $K_{CS} :: c_1$ in the second line of the definition of $s$ by $K_{CS} :: c_1 :: c_2$. Change $C$ to $C'$ by substituting $y :: N_C$ in the fourth line of the definition of $c$ by $y :: N_C :: K_C$ as depicted in Figure 9.12. Now the new version can be verified by the automated

$$\begin{tikzcd}
C \arrow[r, "N_C::K_C::Sign_{K_C^{-1}}(C::K_C)"] & S \\
C & S \arrow[l, "N_S::\{Sign_{K_S^{-1}}(K_{CS}::N_C::\underline{K_C})\}_{K_C}::Sign_{K_{CA}^{-1}}(S::K_S)"'] \\
C \arrow[r, "\{m\}_{K_{CS}}"] & S
\end{tikzcd}$$

Figure 9.12: TLS fix

theorem prover approach. When e-SETHEO runs on the fixed version of the protocol it now gives back the result that the conjecture $\mathsf{knows}(\mathsf{secret})$ cannot be derived from the axioms formalizing the protocol, within 5 seconds. More specifically, within the e-SETHEO suite, the prover "eprover" was able to establish that there is no such derivation by exhaustively trying all possibilities. Note that this result means that there actually exists no such derivation, not just that the theorem prover is not able to find it. This means in particular that the attacker cannot gain the secret knowledge anymore.

From the point of view of secure systems evolution, this improvement now raises the question whether the modifcation may have a negative on the confidentiality of other data values in the protocol.

**Theorem 7.** *Let $P$ be the process which formalizes the (insecure) TLS variant considered above and $P'$ its patch suggested above. Then $P \succcurlyeq P'$ (that is, for every data value $d$ whose secrecy is preserved by $P$, the secrecy of $d$ is also preserved by $P'$).*

**Proof sketch:** Including the client's public key in the message of the server does not enlarge the adversary's knowledge since it is already public knowledge.

Adding another conjunct to the condition that is checked before the client sends out the second message does not enlarge the possible behavior of the client (but on the contrary restricts it). Therefore, it does not increase the adversary's potential for enlarging its knowledge.

**Secure Channel Evolution** The initial requirement is that a client $\mathbb{C}$ should be able to send a message $msg$ on $\mathbb{W}$ with intended destination a server $\mathbb{S}$ so that $msg$ is not leaked to $A$. Before a security risk analysis the situation may simply be pictured in Figure 9.13. Since there are no unconnected output channels, the composition

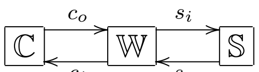


Figure 9.13: Situation before risk analysis

$\mathbb{C} \otimes \mathbb{W} \otimes \mathbb{S}$ obviously does not leak $msg$.

Suppose that the risk analysis indicates that the transport layer over which $\mathbb{W}$ is to be implemented is vulnerable against active attacks. This leads to the model in Figure 9.14.

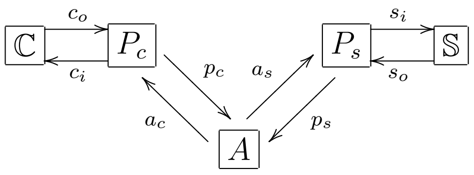


Figure 9.14: Model following risk analysis

We would like to implement the secure channel using the (corrected) variant of the TLS handshake protocol considered in Section 9.5. Thus $P_c$ resp. $P_s$ are implemented by making use of the client resp. server side of the handshake protocol. In Figure 9.15 we only consider the client side. We would like to provide an implementation $P_c$ such

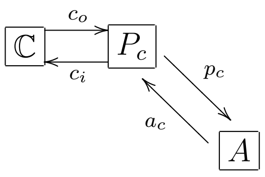


Figure 9.15: Client side of the TLS variant

that for each $\mathbb{C}$ with $msg \in S_{\mathbb{C}}$, $\mathbb{C} \otimes P_c$ preserves the secrecy of $msg$ (where $msg$ represents the message that should be sent to $\mathbb{S}$). Of course, $P_c$ should also provide functionality: perform the initial handshake and then encrypt data from $\mathbb{C}$ under the negotiated key $K \in \mathbf{Keys}$ and sent it out onto the network. As a first step, we may formulate the possible outputs of $P_c$ as nondeterministic choices (in order to constrain the overall behaviour of $P_c$). We also allow the possibility for $P_c$ to signal to $\mathbb{C}$ the readiness to receive data to be sent over the network, by sending $\mathrm{ok}$ on $c_i$ as depicted in Figure 9.16. Figure 9.17 $c_K$ denotes the following adaption of the

$$\begin{aligned} p_c &\stackrel{\mathrm{def}}{=} \textit{either if } \mathsf{inp}(c_o) = \varepsilon \textit{ then } \varepsilon \textit{ else } \{\mathsf{inp}(c_o)\}_K \\ &\quad\ \textit{or } c_K \\ c_i &\stackrel{\mathrm{def}}{=} \textit{either } \varepsilon \textit{ or } \mathrm{ok} \end{aligned}$$

Figure 9.16: Channel behaviour

(corrected) program $c$ defined in Section 9.5 (for readability, we allow to use syntactic “macros” here, the resulting program is obtained by “pasting” the following program text in the place of $c_K$ in the definition of $p_c$). For simplicity, we assume that $P_c$ has already received the public key $K_{CA}$ of the certification authority. We leave out the definition of $c_i$ since at the moment we only consider the case where $\mathbb{C}$ wants to sent data to $\mathbb{S}$. One can show that for any $\mathbb{C}$, the composition $\mathbb{C} \otimes P_c$ preserves the secrecy

```
$c_K \stackrel{\mathrm{def}}{=} \textit{either } N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)$
    $\textit{or case } \mathsf{inp}(a_c) \textit{ of } s_1 :: s_2 :: s_3$
    $\textit{do case } \mathcal{E}xt_{K_{CA}}(s_3) \textit{ of } S :: x$
        $\textit{do if } \mathcal{E}xt_x(\mathcal{D}ec_{K_C}(s_2)) = y :: N_C :: K_C \textit{ then } \{K\}_y$
                                                   $\textit{else } \mathrm{abort}$
        $\textit{else } \mathrm{abort}$
    $\textit{else } \mathrm{abort}$
```

Figure 9.17: Program definition for channel $c_k$

of the messages sent along $c_0$.

As a next step, we may split $P_c$ into two components: the client side $H$ of the handshake protocol (as part of the security layer) and program $P$ (in the application layer) that receives data from $\mathbb{C}$, encrypts it using the key received from $H$ and sends it out on the network as depicted in Figures 9.18 and 9.19.

Here we have a refactoring $P_c \overset{(D,U)}{\rightsquigarrow_T} P \otimes H$ up to the set of behaviors $T$ (cf. Section 9.4 and 9.4) where

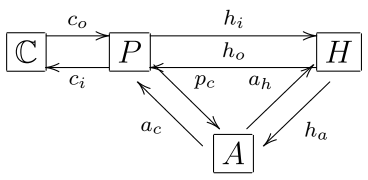


Figure 9.18: $P_c$ after split

```
$h_a \stackrel{\text{def}}{=}$ $\mathit{if}$ $\mathsf{inp}(h_i) = \varepsilon$ $\mathit{then}$ $N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)$
    $\mathit{else\ case}$ $\mathsf{inp}(a_h)$ $\mathit{of}$ $s_1 :: s_2 :: s_3$
      $\mathit{do\ case}$ $\mathcal{E}xt_{K_{CA}}(s_3)$ $\mathit{of}$ $S :: x$
        $\mathit{do\ if}$ $\{\mathcal{D}ec_{K_C}(s_2)\}_x = y :: N_C :: K_C$ $\mathit{then}$ $\{m\}_y$
                                        $\mathit{else}$ abort
        $\mathit{else}$ abort
      $\mathit{else}$ abort
$h_o \stackrel{\text{def}}{=}$ $\mathit{if}$ $\mathsf{inp}(h_i) = \varepsilon$ $\mathit{then}$ $\varepsilon$
    $\mathit{else\ case}$ $\mathsf{inp}(a_h)$ $\mathit{of}$ $s_1 :: s_2 :: s_3$
      $\mathit{do\ case}$ $\mathcal{E}xt_{K_{CA}}(s_3)$ $\mathit{of}$ $S :: x$
        $\mathit{do\ if}$ $\{\mathcal{D}ec_{K_C}(s_2)\}_x = y :: N_C :: K_C$ $\mathit{then}$ finished
                                        $\mathit{else}$ $\varepsilon$
        $\mathit{else}$ $\varepsilon$
      $\mathit{else}$ $\varepsilon$
$h_i \stackrel{\text{def}}{=}$ $0$
$p_c \stackrel{\text{def}}{=}$ $\mathit{if}$ $\mathsf{inp}(c_o) = \varepsilon$ $\mathit{then}$ $\varepsilon$ $\mathit{else}$ $\{\mathsf{inp}(c_o)\}_K$
$c_i \stackrel{\text{def}}{=}$ $\mathit{if}$ $\mathsf{inp}(h_o) =$ finished $\mathit{then}$ ok $\mathit{else}$ $\varepsilon$
```

Figure 9.19: Program channel description

- $T \subseteq \mathbf{Stream}_{O_{P_c}} \times \mathbf{Stream}_{I_{P_c}}$ consists of those $(\vec{s}, \vec{t})$ such that for any $n$, if $(\vec{s}(\tilde{c}_i)) \downharpoonright_i \neq finished$ for all $i \leq n$ then $(\vec{s}(\tilde{c}_o)) \downharpoonright_i = \varepsilon$ for all $i \leq n + 1$
- and $D$ and $U$ have channel sets $I_D = \{\tilde{c}_o, \tilde{a}_c\}$, $O_D = \{c_o, a_c, a_h\}$, $I_U = \{c_i, p_c, h_a\}$ and $O_U = \{\tilde{c}_i, \tilde{p}_c\}$ and are specified by

$$c_o \stackrel{\text{def}}{=} \mathsf{inp}(\tilde{c}_o), \qquad a_c \stackrel{\text{def}}{=} \mathsf{inp}(\tilde{a}_c), \qquad a_h \stackrel{\text{def}}{=} \mathsf{inp}(\tilde{a}_c),$$
$$\tilde{c}_i \stackrel{\text{def}}{=} \mathsf{inp}(c_i), \qquad \tilde{p}_c \stackrel{\text{def}}{=} \mathsf{inp}(h_a)$$

(after renaming the channels of $P_c$ to $\tilde{c}_o, \tilde{c}_i, \tilde{p}_c, \tilde{a}_c$).

Therefore, for any $\mathbb{C}$ with $[\![\mathbb{C}]\!] \subseteq T$, we have a refactoring $\mathbb{C} \otimes P_c \overset{(D,U)}{\rightsquigarrow} \mathbb{C} \otimes P \otimes H$. Since for any $\mathbb{C}$, the composition $\mathbb{C} \otimes P_c$ preserves the secrecy of the messages sent along $c_0$, as noted above, this implies that for any $\mathbb{C}$ with $[\![\mathbb{C}]\!] \subseteq T$, the composition $\mathbb{C} \otimes P \otimes H$ preserves the secrecy of these messages by Theorem 5 (since $D$ and $U$ clearly have inverses).

## 9.6 Composition and Secrecy

Following the definition of composition of processes as stream functions, it is possible to show that the FOL predicate of $P \otimes P'$ is just:

$$\psi(P \otimes P') = \psi(P) \wedge \psi(P')$$

It follows immediately that $\psi(P \otimes P') = \psi(P' \otimes P)$.

If we assume that both $P$ and $P'$ preserve the secrecy of the data value $s$, our goal is to show conditions so that:

$$\psi(P \otimes P') \nvdash \text{knows}(s).$$

In general this does not hold. For example consider a process $P$ which outputs $\{s\}_K$ and a process $P'$ which outputs $K^{-1}$. Independently this both processes preserve the secrecy of $s$, but when composed an adversary could trivially compute $s$.

We will try to understand when secrecy is preserved in a single process in order to understand when it is preserved under composition. To do that we will need some definitions.

**Definition 4** (Subterm)**.** *We say that a symbol x is a subterm of the symbol T and denote it x$\hat{\in}$ T when:*

*x=T*

*T=$\{$T'$\}_{\mathrm{K}}$ and x$\hat{\in}$ T'*

*T=$Sign_{\mathrm{K}}\{$T'$\}$ and x$\hat{\in}$ T'*

*T= h::k and x$\hat{\in}$ h or x$\hat{\in}$ k*

*Where by "=" we mean syntactic equality.*

For example $\mathsf{s} \,\hat{\in}\, \{\mathsf{s}\}_{\mathsf{K}}$ but is not true that $\mathsf{K} \,\hat{\in}\, \{\mathsf{s}\}_{\mathsf{K}}$. We denote this by $\mathsf{K} \,\hat{\notin}\, \{\mathsf{s}\}_{\mathsf{K}}$. Intuitively this means that an adversary could potentially compute s from $\{\mathsf{s}\}_{\mathsf{K}}$ but he could not compute K.

**Definition 5** (Inverse)**.** *Let x $\hat{\in}$ J. We define the cryptographic inverse of a symbol J with respect to x and denote it* $\mathsf{J}^{-1}(\mathsf{x})$ *in the following way:*

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

*If J=h::k and x $\hat{\notin}$ h then* $\mathsf{J}^{-1}(\mathsf{x})$=$\mathsf{k}^{-1}(\mathsf{x})$

*If J=h::k and x $\hat{\notin}$ k then* $\mathsf{J}^{-1}(\mathsf{x})$=$\mathsf{h}^{-1}(\mathsf{x})$

*If J=h::k and x $\hat{\in}$ k, x $\hat{\in}$ h then:*

$$\mathsf{J}^{-1}(\mathsf{x}) = \mathsf{and}(\mathsf{h}^{-1}(\mathsf{x}), \mathsf{k}^{-1}(\mathsf{x}))$$

*If J=$\{$J'$\}_{\mathsf{K}}$ or J=$Sign_{\mathsf{K}}\{$J'$\}$ then:*

$$\mathsf{J}^{-1}(\mathsf{x}) = \mathsf{or}(\mathsf{J}'^{-1}(\mathsf{x}), K^{-1})$$

For example let $\mathsf{J} = \{\{\mathsf{s}\}_{\mathsf{K}_1}\}_{\mathsf{K}_2}$. Then:

$$\mathsf{J}^{-1}(\mathsf{s}) = \mathsf{or}(K_1^{-1}, K_2^{-1})$$

which we will interpret later as "to preserve the secrecy s we need to preserve either $K_1^{-1}$ or $K_2^{-1}$".

**Definition 6.** *Let $\psi(P)$ be the first order logic formula associated to $P$. We define $\bar{\psi}(P)$ to be the set of instantiated formulas of $\psi(P)$ with all possible values satisfying the constraints in $\psi(P)$.*

It is possible to show by induction on the program constructs that $\bar{\psi}(P)$ consists of formulas $F_i$ of the form:

$$\mathsf{knows}(\mathsf{E}_i) \Rightarrow \mathsf{knows}(\mathsf{J}_i)$$

for closed expressions $\mathsf{E}_i$ and $\mathsf{J}_i$ (symbols).

**Definition 7.** *Let Pres(x,P) be the following inductively defined predicate:*

*((∀$\mathsf{F}_i \in \bar{\psi}(P)$ x $\hat{\notin}$ $J_i$ ) ⇒ Pres(x,P))*

*∧ ( ∀$\mathsf{F}_i \in \bar{\psi}(P)$ (x $\hat{\in}$ $\mathsf{J}_i$) ⇒ ((Pres($\mathsf{E}_i$,P) ∨ Pres($\mathsf{J}_{\mathsf{i}}{}^{-1}(\mathsf{x})$,P))*

*$\wedge$ ( (x = $\{$x'$\}_K$ $\vee$ x = $\mathcal{S}ign_K\{$x'$\}$) $\Rightarrow$ (Pres(x',P) $\vee$ Pres(K,P))*

*$\wedge$ ((x = h::k $\Rightarrow$ (Pres(h,P) $\vee$ Pres(k,P))*

*$\wedge$ ((x = and(h,k) $\Rightarrow$ (Pres(h,P) $\wedge$ Pres(k,P))*

*$\wedge$ ((x = or(h,k) $\Rightarrow$ (Pres(h,P) $\vee$ Pres(k,P)) )*

*$\Rightarrow$ Pres(x,P)*

*and $\neg$Pres($\epsilon$,P).*

We assume a closed world for this predicate, and therefore if we can't derive Pres(x,P) for some x, it follows $\neg$Pres(x,P).

**Claim**

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

**Proof**

Clearly $\neg$Pres($\epsilon$,P) since $\mathsf{knows}(\epsilon) \in \bar{\psi}(P)$ for all $P$. If $\forall$F$_i \in \bar{\psi}(P)$ x $\hat{\notin} J_i$ that means that there is no formula in $\bar{\psi}(P)$ containing x in a conclusive position, and therefore there is no way to derive knows(x) from the structural formulas.

Now assume it is possible to derive Pres(x,P). We have already covered the base cases so we can assume that in the formula:

( $\forall$F$_i \in \bar{\psi}(P)$ (x $\hat{\in}$ J$_i$) $\Rightarrow$ ((Pres(E$_i$,P) $\vee$ Pres(J$_i^{-1}$(x),P))

$\wedge$ ( (x = $\{$x'$\}_K$ $\vee$ x = $\mathcal{S}ign_K\{$x'$\}$) $\Rightarrow$ (Pres(x',P) $\vee$ Pres(K,P))

$\wedge$ ((x = h::k $\Rightarrow$ (Pres(h,P) $\vee$ Pres(k,P))

$\wedge$ ((x = and(h,k) $\Rightarrow$ (Pres(h,P) $\wedge$ Pres(k,P))

$\wedge$ ((x = or(h,k) $\Rightarrow$ (Pres(h,P) $\vee$ Pres(k,P)) )

$\Rightarrow$ Pres(x,P)

$\psi(P) \nvdash \mathsf{knows}(y)$ for all the Pres(y,P) y $\neq$ x needed in the precondition.

Since in this formulas all the cases where we could apply the Structural Formulas are covered, it is impossible to derive knows(x). The case $\neg$Pres(x,P) is similar.

$\square$

Note that the converse does not hold, that is $\psi(P) \nvdash \mathsf{knows}(x)$ does not mean we can derive Pres(x,P), because for some pathological cases we will have an infinite loop, for example for:

$\bar{\psi}(P) = \text{knows(x)} \Rightarrow \text{knows(x)}$.

It is although reasonable to expect this kinds of loops not to be present in practical cases. It would be an interesting matter of future work to completely classify this kind of cases. It is nevertheless possible to detect this loops in a machine implementation of the preservation predicate.

**Example**

Consider $P = (\{s\}_{h::t}, K_2^{-1})$ , $P' = (\{\text{h}\}_{\text{K}_2}, t)$.

Then $\bar{\psi}(P \otimes P') = \{\text{knows}(\{\text{s}\}_{\text{h::t}}), \text{knows}(K_2^{-1}), \text{knows}(\{\text{h}\}_{\text{K}_2}), \text{knows}(t)\}$

where we omit $\text{knows}(\epsilon) \Rightarrow$ before each formula, but we assume it is there formally.

Now it is easy to see from the definition that to derive Pres(s,P $\otimes$ P') we would need derive Pres(h::t,P $\otimes$ P') and that means deriving either Pres(h,P $\otimes$ P') or Pres(t,P $\otimes$ P'). Trivially it is not possible to derive Pres(t,P $\otimes$ P') because it inverse with respect to t is just $\epsilon$. Now, to derive Pres(h,P $\otimes$ P') we need Pres($K_2^{-1}$,P $\otimes$ P') which is not possible by the same argument we used for t.

So we conclude $\neg$Pres(s,P $\otimes$ P').

Matter of current work is to show how would be the predicate be applied to the formulas obtained by an incorrect implementation of the TLS protocol [Jür05a], and its corrected version.

## 9.7 Composition and evolution

From the inductively defined predicates Pres(x,P) and $\neg$Pres(x,P) is possible to record the (finitely many) symbols needed to verify the secrecy of x and a boolean value that states for the secrecy preservation of those symbols.

One could then store this trees for separated processes and use to re-verify secrecy of compositions whenever changes are made to single components.

It is matter of current work to find a way to merge this proof trees for two different processes $P$ and $P'$ representing composition in a way that is more efficient as simply recomputing the secrecy value for x.

# 10 The SecureChange Process and Integration of Design Models

The change driven security process contains the classical steps of well established security-processes [IR008], [Dor02]. What distinguishes our vision of a change-driven security process is that the process steps are initiated by change-events. These change events affect the state of model elements. This section mainly focuses on the conceptual core of the Secure Change process -- a state machine based model.

In [HD09] we identified four basic types of change in context of security management:

- *Planned evolution* refers to pre-empting small system changes.
- *Planned revolution* means planning major system changes.
- *Unplanned evolution* entails reacting to minor changes in context or requirements.
- *Unplanned revolution* necessitates a reaction to major changes in context or requirements.

Further, we identified the requirements for a framework supporting the process of secure change [HD09]. It should support

- An *integrated view*: To keep a complex and interconnected system running despite change a variety of stakeholders have to collaborate in their daily operations. The modelling environments should support the collaboration of these different stakeholders (e.g. Chief Information Officer, legal experts, system administrators, software developers) in their daily work.
- *Domains and Responsibilities*: A stakeholder carries responsibility for a specific Domain, i.e. a subset of the constituting elements of a system. It is important to have a clear understanding of these governance aspects to be able to handle change effectively in an organization. Depending on the type and extent of change, certain stakeholders need to cooperate to provide solutions to handle such change.
- *Change Propagation*: Change is perceived as an event which triggers a series of consecutive steps. A framework which supports a change-driven security process needs to provide a foundation for propagating change to the right stakeholders in an organization. Change is perceived as an event which triggers a series of consecutive steps.
- *Bidirectional Flow of Information between Models and Executing System*: On the one hand, the target architecture of a Living Security framework is a security infrastructure equipped with “sensors” and “agents” collecting information and feeding it back into the modelling environment. Once there, information is interpreted at the level of Model Elements in context of the System Model. On the other hand, the security infrastructure ought to be configurable from a modelling perspective.

- *Information Consistency and Retrieval*: Stakeholders need explicit support to appropriately visualize and query security related information in various contexts. Information Retrieval goes beyond the mere checking of constraints on model elements in that stakeholders can access semantically enriched information.

Based on these requirements, we also outlined the foundation of a framework for Secure Change roughly relying on four basic principles of Living Models [Bre10].

We present a running example for illustration purposes in Section 10.1, give a brief summary of these principles in Section 10.2 and give on outline of the state machine based model, the conceptual underpinning of the change driven process in Section 10.3.

## 10.1 Running Example

Our use case refers to a financial trading platform which allows traders to place orders in specific market segments. The trading platform is directly connected with the systems of major financial institutions who use the platform to place large volume orders. In addition, individual traders can use an online frontend or download a client to use the services. The trading platform is developed and operated by a medium sized specialized company which offers services and support for the platform and sells licenses to its users. The trading platform is developed in house by a team of software developers who develop, deploy and actively manage the systems which are hosted in an outsourced data centre. The trading platform uses standardized financial communication protocols and is realized as a SOA. The company is certified according to ISO 27001 to underline their emphasis on security. Currently, the company offers access to specialized niche markets, but is planning to extend its service to stock markets in the near future.

Subsequently, we illustrate these requirements with a running example for an evolving, security-critical large-scale system which will be extended to illustrate the principles and concepts as we go along in Sections 10.2 and 10.3.

## 10.2 Basic Principles of Living Security for Secure Change

### 10.2.1 Common System View

The framework supports stakeholders in their various daily operations. This happens through Stakeholder-Centric Modelling Environments, perspectives on the system's security status, customized to an appropriate level of abstraction. The analysis of security attributes requires the analysis of interdependencies across the layers ranging from IT management, software engineering and system management. Although the framework also facilitates the cooperation among the stakeholders (Chief Information Officer, Chief Security Officer, Network Administrator, Security Engineer etc.), it does not necessarily need to provide an integrated and homogeneous modelling environment. Rather, these stakeholder-centric modelling environments rely on a common meta-model, the Common System View (Figure 10.1). This Figure also shows an example of a system model composed of three layers of the modelling environment *Software Engineering*, namely *Requirements*, *SW-Architecture*, and *Code*.

Security related concepts like threats, risks, requirements etc. are introduced into the meta-model as Meta Model Plug Ins. Every element of the System Meta Model can be decorated with security-related semantics. Figure 10.2 shows the security meta-model that plugs into the System Meta Model as an extension for security. In addition other security relevant meta-models like the UMLsec notation can be used to extend the artefacts of the system model with security relevant information.

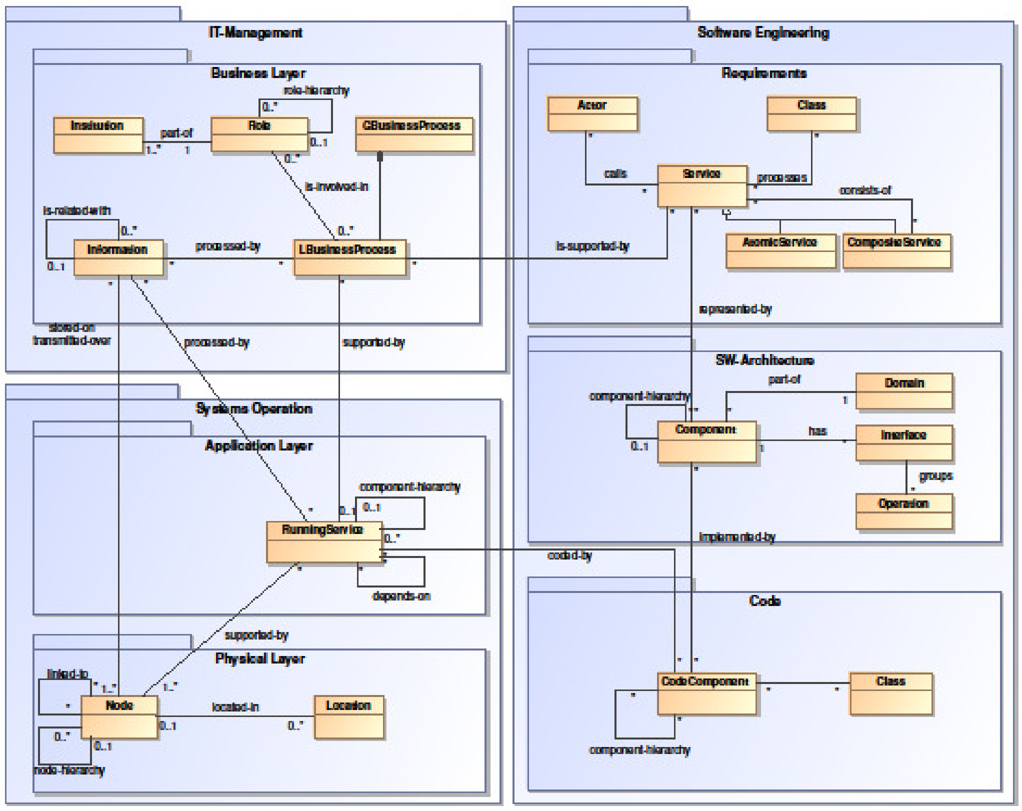


Figure 10.1: Sample System Meta Model

Other activities which focus on specific aspects like requirements engineering, verification, and testing similarly use a specific system model as a basis and extend this system model with the relevant concepts needed to fulfil the activity.

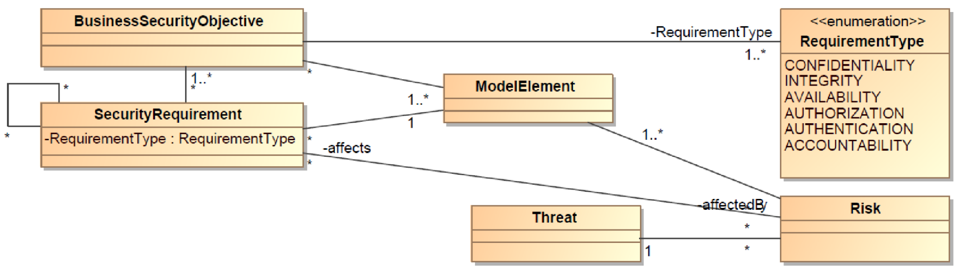
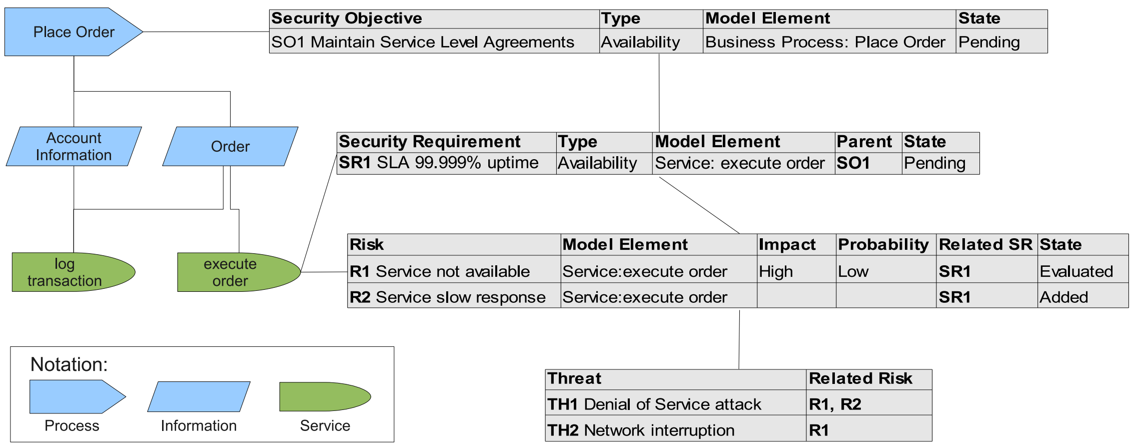


Figure 10.2: Security Plug In

**Example 1.** In order to realize the first requirement, an integrated view, a common system view is needed. The common system view provides on the one hand all concepts to depict the building blocks of the entire system, beginning with its infrastructural components and services, containing the deployed software artefacts, modules, and their interfaces. On the other hand via appropriate plug-ins it provides also concepts for modelling non-functional aspects such as security objectives, requirements, threats, security controls. The common system view relates all these concepts on different layers to each other. Consider as an example the business process “Place Order” as depicted in Figure 10.3. It requires many different running services in the trading platform, which in turn require system and network capacities. If a legal experts identifies a new requirement (e.g. new contractual obligations related to premium customers) which is valid for the business process “Execute order”, then she links this requirement to the concept business process. Referring to the example business process, the objective is “Maintain Service Level Agreements”. The dependencies and relation between the different layers serve as a means to identify which parts of the infrastructure are impacted by such a new requirement (e.g. the service “execute order” requires an uptime of 99.999%). Carrying on the example, a system operator might receive notice that certain parts of the infrastructure he is responsible for are impacted by this new requirement. Another advantage of the Common System View is the provision of stakeholder specific perspectives. In this example a software architect does focus mainly on the deployed components which realize the services and how these components communicate with each other.



| Security Objective | Type | Model Element | State |
|---|---|---|---|
| SO1 Maintain Service Level Agreements | Availability | Business Process: Place Order | Pending |

| Security Requirement | Type | Model Element | Parent | State |
|---|---|---|---|---|
| **SR1** SLA 99.999% uptime | Availability | Service: execute order | **SO1** | Pending |

| Risk | Model Element | Impact | Probability | Related SR | State |
|---|---|---|---|---|---|
| **R1** Service not available | Service:execute order | High | Low | **SR1** | Evaluated |
| **R2** Service slow response | Service:execute order | | | **SR1** | Added |

| Threat | Related Risk |
|---|---|
| **TH1** Denial of Service attack | **R1, R2** |
| **TH2** Network interruption | **R1** |

Figure 10.3: Example instance of a system model extended with a security plug-in

## 10.2.2 Model Element States

To depict changes and distinguish different states of an information object, we want to have the possibility to model not only the dependencies between business and technical artefacts, but furthermore we want to differentiate information objects with regard to their life-cycle. We model security relevant milestones in the lifecycle of model element as Model Element States. Changes of Model Elements States can propagate over the complex "network" of model elements as defined in instances of the System Meta Model and its plug ins. Figure 10.4 shows the state diagram of the meta model element *Security Requirement* defined in the Security Plug In.

**Example 2**. In the example outlined in Figure 10.3 the risks related to the service "execute order" have two different states. The first risk R1 has already undergone a risk evaluation and its state is therefore set to "evaluated". second risk R2 has only been identified but not evaluated yet, therefore its state is set to "pending". The related security requirement SR1 will remain in the state "pending" until all related risks (R1, R2) have reached the state "evaluated". Only then the security requirement SR1 will also reach the state "evaluated". Similarly, if the security requirements would be already in the state "evaluated" and a new risk R3 would be added, its state would immediately switch back to "pending", therefore indicating that a change occurred and additional steps are required to reach a new security state.

The integration of different meta-models will be based on the common system view using plug-ins to extend the original concepts. In the case of the UMLseCh profile the provided stereotypes, tags and constraints can be used to extend various elements of the sample system meta-model. Whereas the common system view is a domain specific language the UMLseCH profile is a generic extension to the UML.

To integrate the two meta-models and provide an appropriate plug-in for UMLseCh appropriate element types of the sample system meta-model will be extended using the concepts provided by UMLseCh.

Some of these concepts provided by UMLsec and UMLseCh are already defined in existing plug-ins and can potentially be reused, e.g. the concept of security requirement in the Security Plug In (c.f. Figure 10.2). Nevertheless the goal is to provide an appropriate view for each of the plug-ins containing exactly those concepts that are relevant.

Section 10.3 gives more details on how events affect the state of model elements thereby driving the Secure Change Process.

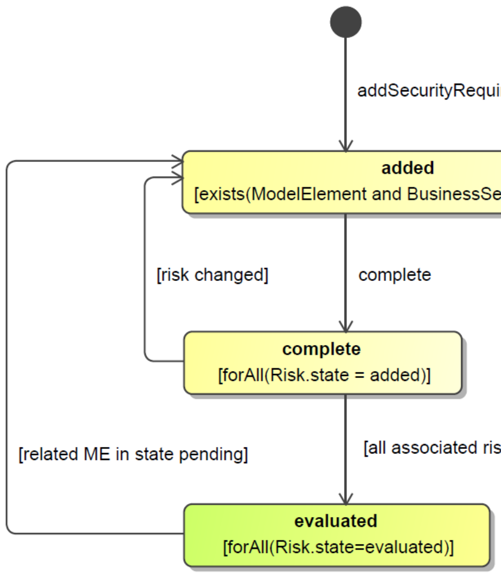

addSecurityRequirement
added
[exists(ModelElement and BusinessSecurityObjective)]
[risk changed]
complete
complete
[forAll(Risk.state = added)]
[related ME in state pending]
[all associated risks evaluated]
evaluated
[forAll(Risk.state=evaluated)]

Figure 10.4: State Machine of Meta Model Element Security Requirement

### 10.2.3 Persistence, Tracing

As instances of the System Meta Model, System Models are the actual targets of change. They capture the status (functional and non functional) of a system at all relevant levels of abstraction. But to indicate change and facilitate planning they have to be made persistent. This allows the definition of system (r)evolution as a sequence of modelled status snapshots. Like the model elements, the System Meta Model itself may undergo change and evolve over time.

**Example 3.** In our running example, the role of *Persistence* can be illustrated using two examples. First, consider a situation in which new security requirement has been identified as a change event and will be introduced in the common system view. The new requirement triggers a series of actions which are executed by different stakeholders. For instance, the software developer will be re-evaluating whether there are any new potential risks which might be related to the new security requirement. By keeping persistent versions of all the model snapshots, which are reflecting the ongoing security process it is on the one hand possible to provide an audit trail of the analysis and the resulting decisions. On the other hand it is possible to trace specific security solutions which are still in place back to a now possibly obsolete security requirement. Second, Persistence allows to model different future scenarios. Consider a new security requirement for which several options of security controls might be considered. Using different planning scenarios and snapshots of the model it is possible to evaluate the impact of the planned controls on the current system architecture.

### 10.2.4 Close Coupling of Models and Code

As the framework aims to support security engineering and management activities targeting the running system, the underlying models have to appropriately reflect the system's current security status. The Secure Change framework maintains a consistent state between models and evolving system in a changing environment through the tight coupling between models and the executing system.

Together with the Integrated View, this principle provides stakeholders with a Modelling Environment that is directly linked to the executing system.

**Example 4.** In the concrete example of the financial trading platform and the instance outlined in Figure 10.4, there might be specific sensors deployed in the system which monitor the uptime and performance of the service "execute order". The collected key indicators can be fed back to the model to enrich it with information reflecting the current status of the system.

## 10.3 State Transition and Change Propagation

In the System Model with its various views, change is propagated based on the interrelationship of a changing model element with other model elements.

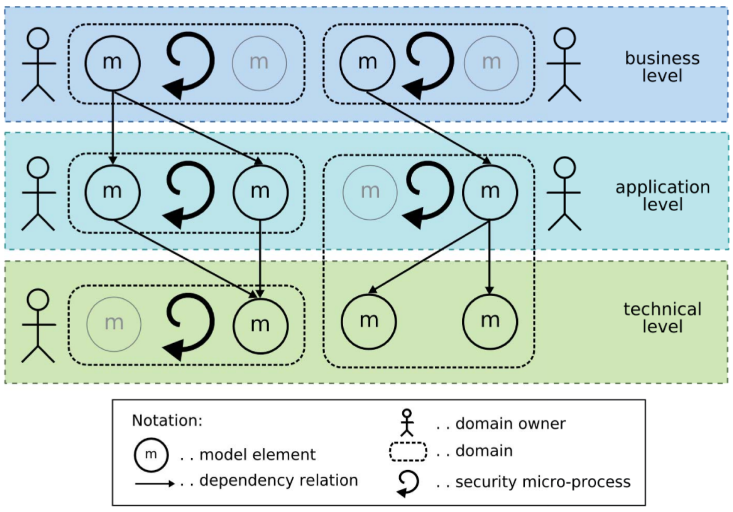


Figure 10.5: Concept of Distributed Security Process

Change events are sent to the current System Model where action is triggered and the effects percolate through the three layers and their sub layers. Change is handled according to the following procedure of the Change Driven Process:

A. State transition – A change event may induce a state transition of a model element. For instance, the state of a security requirement is changed from *evaluated* to *added* if the related model element (e.g. a software component) has been modified.

B. Change propagation – The state transition of the model element may trigger state transitions in related model elements according to stated propagation rules. For instance, the modification of a security requirement attached with a business process may cause state transitions in information objects and services supporting this business process. The propagation rules are specific to each meta model element.

C. Modification of task list – Each stakeholder is associated with a task list describing the pending action events of model elements he/she is responsible for. After each state transition new tasks may be pending and have to be added to the task list. Consequently fired action events (e.g. after the evaluation of a model element) are withdrawn from the task list.

Using the concept of change events and model element states it is possible to assign and distribute the tasks of the security process to the according stakeholders. Based on the concept of domains and responsibilities we are able to identify required tasks and assign them to the respective stakeholders.

In reality, this implies a distributed security micro-process which is executed by each of the stakeholders within his specific domains. Figure 10.5 highlights this concept of distributed instances of a security process. Of course the stakeholders do not work independently on their security related tasks, but a lot of coordination and cooperation is necessary.

**Example 5.** Referring to our sample process, the security process can be schematically described as follows. The identification of the new security objective “Maintain Service Level Agreements” was brought up by the legal experts based on new contractual obligations with premium clients. The legal experts introduced the new security objective in the common system view using a security plug in and attached it to the model element “Place Order”. Based on the dependencies of the system depicted in the common system view, this change event percolates through the processed information objects (e.g. “Account information” and “Order”) to the respective services (eg. “log transaction” and “execute order”).

The software architect whose domain and responsibility contains the services and the elaboration of the related security requirements receives a notification to evaluate the existing services according to the new security objective. She or he then identifies and translates the abstract security objective in the concrete security requirement “SLA 99.999% uptime”. This event which was triggered by the introduction of the new security objective by the legal experts again triggers new actions.

In the concrete case, a security engineer whose domain consists of the threats and risks related to services receives the notification to conduct a threat and risk analysis for the service “execute order” since a new security requirement with the status “pending” has been added. The security engineer then introduces two new risks (R1, R2) which are related to the security requirement SR1. As can be seen in the example the progress of the steps taken by the security engineer is also reflected as changes in the common system view. She or he has already evaluated the new risk R1, which state is set to “Evaluated”. As soon as the remaining risk R2 will be evaluated, the software architect will receive a notification that his or her security requirement SR1 has too reached the state “evaluated”. In this manner it becomes possible to translate change events in a series of tasks which have to be fulfilled by different domain owners. The progress of the different distributed actions will be reflected by the model element states and allow to analyze whether or not the whole system has again reached a stable security status.

## 10.4 Integrated Process

In order to integrate the solutions by all the different activities in the Work Packages we enhance the SecureChange process by an integrated view. This integrated view is based on the following concepts:

- Taxonomy: provides a classification of changes and attitudes to changes
- Artefacts: distinguishes on an abstract level the different models and artefacts which are used by the different Work Packages

The taxonomy can be used on the one hand to identify different basic change scenarios. These change scenarios on the other hand can be used to describe how change is handled on an abstract level by the different Work Packages.

The artefacts provide an overview of all the different types of models used by the various Work Packages of SecureChange. These artefacts are described on the meta level and abstract from concrete concepts. That way it is possible to treat method-specific conceptual models as black-boxes and plug-in different methods and approaches to the integrated SecureChange process. Examples for such artefacts are a system model which includes all artefacts related to the system, ie. architecture, code, constraints and others. Other specific artefacts are a verification model, a risk model, a requirements model and a test model.

Independently from which requirements engineering method and model is used, it is clear that a change in a requirement has to trigger some changes in the test model. Using the change scenarios derived from the taxonomy and case studies it can be described how the different artefacts are updated and trigger changes in other models. That way it is possible to outline how the results and solutions provided by one Work Package impact the other Work Packages.

### 10.4.1 Artefacts and relations between artefacts

As a first basis for distinguishing an abstract approach is followed. This is at the moment a non-exhaustive list which reflects the main types of artefacts which are treated by the different SecureChange Work Packages. The different artefacts are:

- System Model: The System Model includes all artefacts related to the system (from architecture to code, including constraints). It is a placeholder for the system model of Work Package 4 and all types of system models used throughout the other Work Packages.
- Verification Model: The Verification Model contains artefacts which are specific to Work Package 6.
- Risk Model: The Risk Model includes all artefacts related to risk analysis (e.g. assets, vulnerabilities, threats, controls, risk). It is a placeholder for different conceptual models of risk, such as the CORAS model, the THALES risk model or the ProSecO security model and therefore integrates mainly the artefacts from Work Package 2 and Work Package 5.
- Requirement Model: The Requirement Model reflects all artefacts which are related to requirements engineering. It is mainly related to Work Package 3.
- Test model: The Test Model contains the artefacts related to testing, and is related to Work Package 7.

Figure 10.6 outlines an overview of these different types of artefacts. Seen from an overall integrative perspective the different artefacts are strongly related to each other. The result of a requirement analysis will provide input for the test-engineers and be used to verify code and infrastructure components.

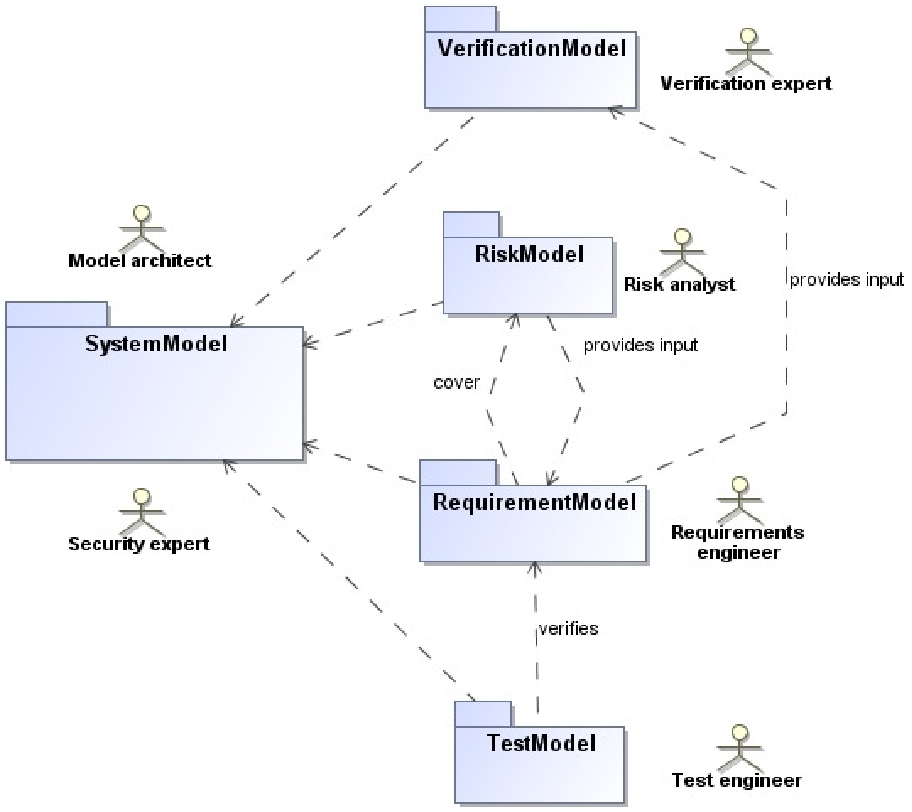


Figure 10.6: Integrated view of SecureChange artefacts and their dependencies

The meta model which is described in Section 4 is a working model in a specific context. It maps to the integrated view of SecureChange artefacts in the sense that it is a specific instantiation of a system model, partly a requirements model and a risk model. That way the specific working model can be mapped to one or more of the artefacts depicted in Figure 10.6.

## 10.4.2 Integrated SecureChange process metamodel

The description of the overall SecureChange process will deliver concepts of change derived from all the solutions of the different SecureChange Work Packages. The goal is to provide an integrated meta model of change related concepts which is independent from any Work Package specific solution (cf. 10.7).

Consider as an example a change in the infrastructure that requires a change in the system model. The system change triggers a system analysis to analyse the changes with the result of an updated system model. An updated system model might affect the current set of requirements and therefore triggers a requirements analysis resulting in a new updated requirement model. The update of the requirement model and the update of the system model both potentially impact the current test model. Therefore both changes trigger new test engineering with the result of a new updated test model.

The dependency relations between the different types of artefacts are the frame for change propagation.

Currently this change model is in a conceptual development phase and will be elaborated during Year 2 and Year 3. The different change related concepts provide a basis for the description of change handling in the integrated SecureChange process.

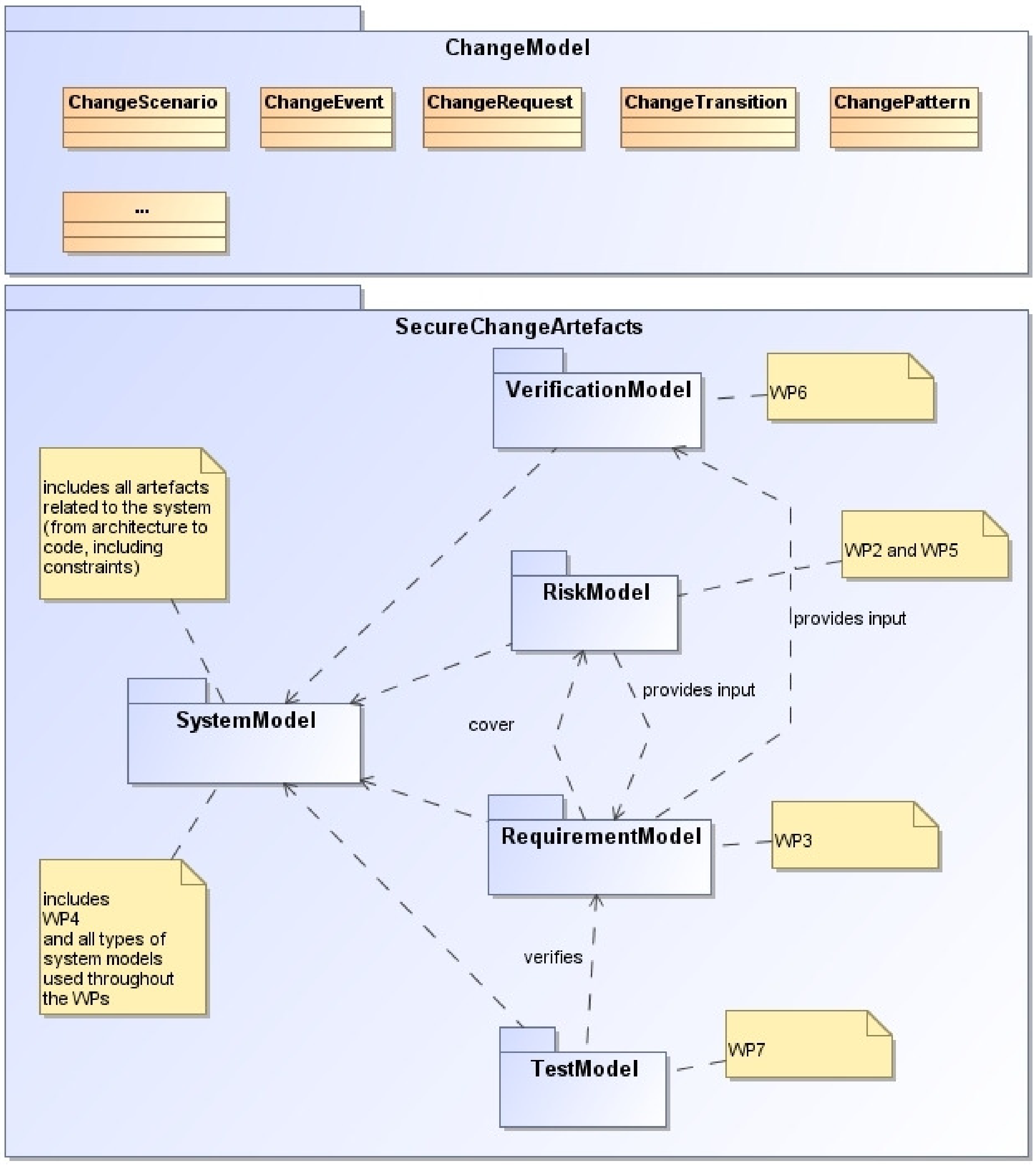


Figure 10.7: The integrated SecureChange process metamodel

At the moment we have identified the following list of change concepts in the various work packages (cf. Figure 10.7):

- ChangeScenario: is expressed at the requirements level and describes the change in the requirements. This change scenario will consist of a before and after requirements model.

- ChangePattern: consists of a specific change scenario, one or more solutions and a mapping between the elements from the change scenario and the architectural elements in the solution.
- ChangeEvent: is a general trigger of change which is derived from a set of change scenarios.
- ChangeRequest: is a general description of some change in the system.
- ChangeTransition: is a description of all the differences from one change to another.

Additional concepts which are candidates for the inclusion in the Change Model is the concept of perspectives which is used in Work Package 5, the concepts of Change Line, Version, Change Propagation and others.

Future tasks related to the refinement and further development of the integrated SecureChange process meta model include the collection of additional change related concepts throughout the other Work Packages. All these change concepts will be consolidated as an integration for all the Work Packages. In addition changes need to be classified to provide different basic categories of changes which might require a different handling. Activities in other Work Packages provide a sound basis for the development of such a Change Model, such as the Deliverable D3.2 of Work Package 3 and the Deliverable D5.2 of Work Package 5.

# 11 Modelling Security Requirements

In this section we present the approach to model requirements evolution and how changes in the requirements model can be propagated to UMLseCh models.

## 11.1 Meta-model for requirements representation

The conceptual model for evolving security requirements incorporates the state-of-art requirement modelling languages such as Secure Tropos and Problem Frames. As a unified extension to Secure Tropos and Abuse Frames, the conceptual model is explicit in representing target specifications where vulnerability can be revealed.
Essential elements such as threats are also made explicit in order to analyze attacks that are assumed to be present in a hostile operating environment. The overall goal of the model is to provide mechanisms for protecting valuable assets from damage. Using this conceptual model for security requirements, it is possible to construct arguments to examine the security of systems as they change.
Figure 11.1 represents the entities characterizing our meta-model to represent requirements and the relations between them.

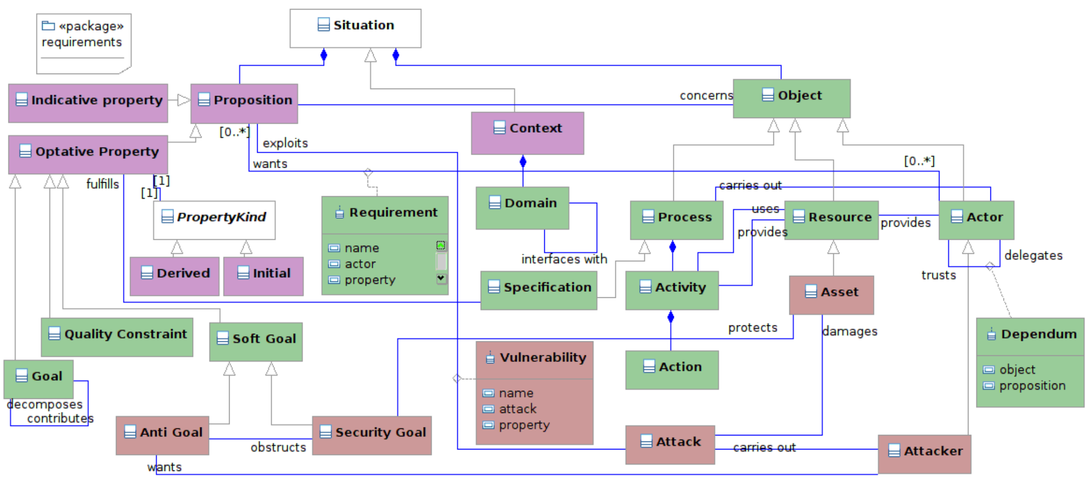


Figure 11.1: The meta-model for requirements elicitation

We have outlined in green the concepts inherited from the goal-oriented approaches, in magenta the concepts taken from Problem Frames approaches, and in red the concepts borrowed from risk analysis approaches.

A *situation* of our requirements model is expressed in terms of propositions and objects. *Propositions* are the sharable objects of attitudes and the primary bearers of truth and falsity [McG08]. A proposition can be an optative or an indicative property concerning objects. An *object* is an actor, a process or a resource. When a stakeholder (actor) *wants* a desired or optative property, it is modelled as initial *requirements,* which can be *refined* into *derived requirements. Therefore, requirements* are desired or optative properties that the system-to-be ought to have, as wanted explicitly by stakeholders. Initial requirements and derived requirements can be captured by *goals*, the objectives that the system-to-be should achieve. A derived requirement can also be a *soft goal*, which does not have a clear-cut evaluation of the truth value. . A security goal expresses that an asset needs to be protected from harms. An anti-goal is a goal of an attacker which may obstruct the achievement of a security goal. Both security *and anti-goals are soft goals*.

Unlike requirements, a *specification* fulfils certain requirements under given indicative domain properties.. It usually captures certain dynamic behaviour in order to satisfy software requirements; therefore specifications are modelled as processes.

*Objects* are entities used to describe a state of the world. An object can be dynamic or static. A static object can be an actor or a resource. An *actor* is an intentional entity such as a human, a device, a legacy software or software-to-be component that performs actions to achieve its own goals. We consider an *attacker* as a particular actor who wants an anti-goal to be satisfied. A *resource* is a physical or an informational entity which has no intention by itself. An *asset* is a resource which has a value and needs to be protected. V*ulnerability* is a weakness, a flaw or a deficiency that is exploited to carry out an attack which causes harm to or damages an asset. A dynamic object can be a *process* that consists of *activities*. An *activity* is a sequence of *actions* that can be performed by an actor to fulfill a goal.

A situation is a partial state of the world where some propositions are true and some other propositions are nor true nor false. Thus, a situation consists of objects and propositions concern these objects. Particular types of situations are *context,* the *domain*, and an *attack*. The *context* is a situation within which the system-to-be will operate. A context consists of several domains which interface with each other. An *attack* allows an attacker to fulfill an anti-goal. In particular, an attack is a situation in which vulnerability is exploited to cause damage on an asset.

For requirements analysis, these entities are related in the following seven basic types:

- *Trusts* is a relationship from one actor to another, which indicates the belief of one actor that the other will provide a resource or will perform a certain activity ;
- Delegates is a relationship from one actor to another which specifies that the fulfilment of a goal or the provisioning of an activity/resource;

Both trusts and delegates relationship are associated with a *dependum,* which specifies which object (resource/process) or which requirement (goals, softgoals) are trusted or delegated from one actor to another.

- *Provides* is a relationship either from an actor to a resource, which specifies that an actor provides a certain resource; or from an activity to resources .; *Uses* is the relationship opposite to Provides. ..
- *Carries Out* is a relationship either from an actor to a process, which specifies that an actor carries out a certain activity; *Carries Out* is a relationship either from an actor to a process, which specifies that an actor carries out a certain activity.

- *Fulfils* is a relationship from resources and activities to a goal, which specifies a goals is fulfilled by a combination of the resources and the activities;
- *Wants* is a relationship from actors to goals which associates an actor with its goals, including security and anti-goals.
- *Contributions* is a relationship among goals/security goals which indicates that a goal contribute to the satisfaction of another goal.
- *Decomposes* is a relationship from a goal to its subgoals, which indicates that a goal can be refined: AND-decomposition lists subgoals that must all be satisfied in order to satisfy the goal, whereas OR-decomposition suggests alternative ways to satisfy the goal.

For security requirement analysis, the following seven specific relationships are considered on an attack situation and a security goal:

- *Attacks* is a relationship from one situation to a vulnerable actor;
- *Damages* is a relation from an attack to the assets;
- *Exploits* is a relationship from an attack to a vulnerability, which is a (part of) specification that can be vulnerable to expose security problems;
- *Protects* is a relationship from a security goal to a set of valuable assets;
- *Obstructs* is a relation from an anti-goal to the corresponding security goal.

Such problem analysis for goal satisfaction can be done using proposition logic qualitatively, or using risk analysis quantitatively. In either way, arguments on the fulfilment of security requirements need to be acceptable after a negotiation process during which the trusted domain assumptions may not always hold. Therefore, the framework as such can support extensively evolving security requirements.

## 11.2 Meta-model of Security Requirements Evolution

After specifying the static view of situations about the security requirements, the next step in our methodology is to deal with the dynamic view. In a reactive view of the classification, situations are observed to change over time. Discrete changes have a sequence of change descriptions associated with timestamps, while continuous changes happen continuously in that the intervals can be arbitrarily further refined and the length can be arbitrarily prolonged for the security requirements in long-lived software systems.

In a nutshell, the situations that can change in the model for requirements include generally entities and the relationships between them. In particular we consider elementary types of changes, including the modification, the addition and the removal of an element (such as an entity or of a relationship). An example of possible change is the addition of a new actor (as the event that matches with the condition) that results in the addition to the model of a new entity representing the actor and new relationships such as the "wants" relationship to specify the goals the new actor wants to achieve or a "provides" relationship from the actor to the resources and activities that it offers.

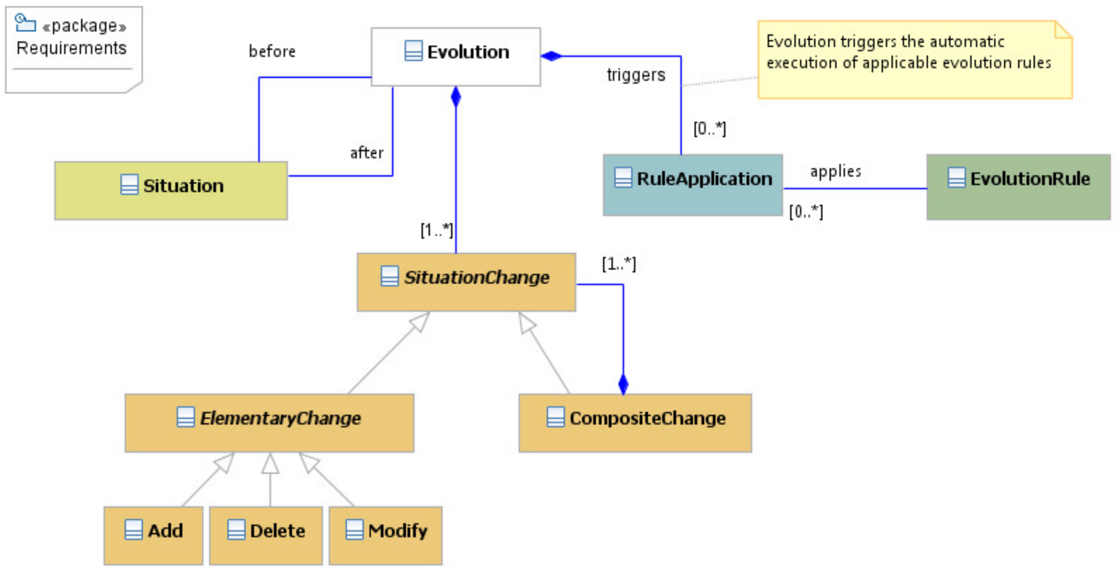


Figure 11.2: A generic meta-model for evolution of security requirements

A more complex kind of situation change can be described by a composite change, which is a transaction of elementary changes (or nested composites) that must happen together or not at all. For example, the deletion of an actor A may require the deletion of all the delegation relationships from A to another actor B, while finding for B alternative actors A' that can provide the same activities and resources delegated to A, otherwise the incomplete change may violate the intention of B. Therefore we record such complicated changes as a transformation that preserve the satisfaction of certain high-level requirements.

A natural way is to represent the change as a transition rule between two situations, denoted respectively as *before* and *after situations* (see Figure 11.2). Intuitively, the before/after *situation* represents the elements in the model the change has occurred at a given time before/after an adaptation has been applied. The outcome of changes is monitored by evolution rules to decide whether an adaptation action needs to be taken. If yes, the change will trigger the application of a general evolution rule in a concrete place in the model, possibly causing additional changes.

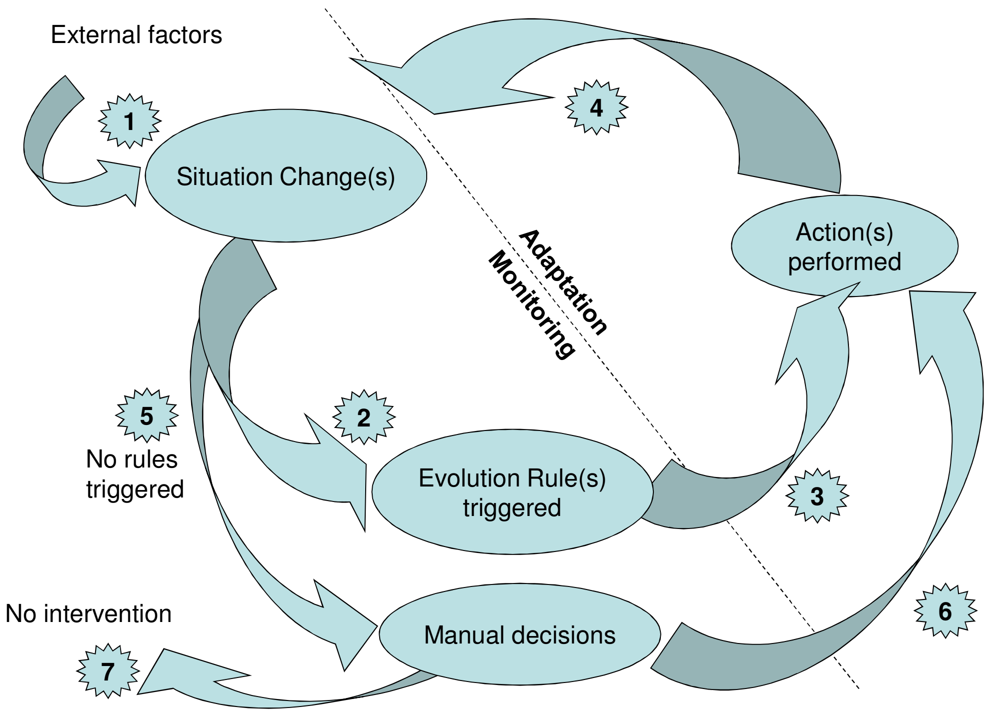


Figure 11.3: The continuous adaptation process to maintain the requirements

The envisioned workflow of maintaining design and requirement models through evolution requires the continuous adaptation of the models to react to changes. Changes in external factors – changed requirements, new threats, revised design decisions – can be introduced into the model by engineers (or automatic monitoring in some cases); this model change, however, may violate constraints and requirements, cause inconsistencies. Therefore reactions are required to handle the effects of change. While most reactions will remain responsibilities of engineers, evolution rules can be defined to automatically adapt and transform the model in some cases. Failing that, rule-based automatic mechanisms are expected to be able to initiate the process of adaptation in many cases, or at least indicate the problem to the engineers.

The applicability of automatic evolution rules is greatly enhanced by machine-understandable, domain-specific refinement of the general requirement modelling concepts appearing in this deliverable. Therefore SecureChange provides a general meta-model for security-related concepts, and suggests domain-specific subclassing where applicable to facilitate tool support and automated reaction mechanisms.

Figure 11.3 shows how the evolution rules are used in a feedback loop to deal with the evolution of security requirements. Changes of situation are initially caused by external factors (environment context) of the system. These changes can trigger evolution rules that perform automatic adaptation, or otherwise result in a manual change process. As a result of such an adaptation action, a new situation arises that one must iteratively reevaluate for automatic rule execution or manual intervention. Or else, if the new change does not trigger any further actions, or there is no further change, the control feedback loop can exit.

## 11.2 Example of requirements evolution modelling

In this section we show how we can represent the evolution of requirements that characterize the ATM case study by instantiating the meta-model presented in Section 11.1. We show how functional and security requirements of the actual ATM systems change due to the introduction of the AMAN queue management tool that supports ATCOs.

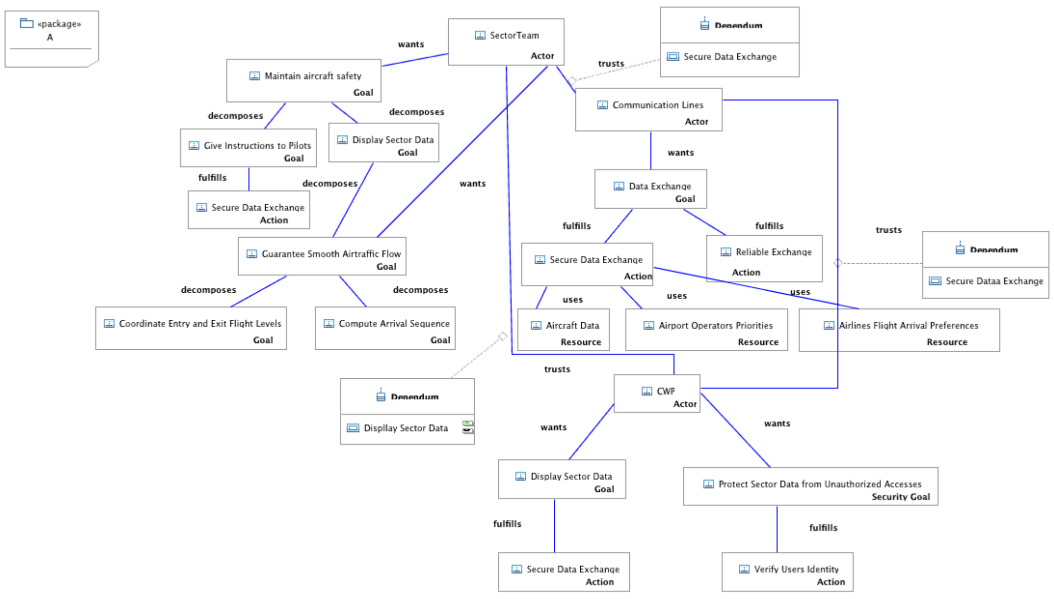

«package» A
SectorTeam Actor
Dependum Secure Data Exchange
wants
trusts
Maintain aircraft safety Goal
decomposes
decomposes
Give Instructions to Pilots Goal
Display Sector Data Goal
fulfills
decomposes
Secure Data Exchange Action
wants
Communication Lines Actor
wants
Data Exchange Goal
fulfills
fulfills
Guarantee Smooth Airtraffic Flow Goal
Secure Data Exchange Action
Reliable Exchange Action
trusts
Dependum Secure Dataa Exchange
decomposes
decomposes
uses
uses
uses
Coordinate Entry and Exit Flight Levels Goal
Compute Arrival Sequence Goal
Aircraft Data Resource
Airport Operators Priorities Resource
Airlines Flight Arrival Preferences Resource
trusts
Dependum Displlay Sector Data
CWP Actor
wants
wants
Display Sector Data Goal
Protect Sector Data from Unauthorized Accesses Security Goal
fulfills
fulfills
Secure Data Exchange Action
Verify Users Identity Action

Figure 11.4: Before Situation

Figure 11.4 represents the requirement model before the introduction of the AMAN. The main actors are the Sector Team at the destination airport composed by the Planning and the Tactical Controller, the CWP, and the dedicated communication lines (telephone, radio communications). The flight arrival management operations are performed by the Sector Team (Tactical and Planning Controllers) that has to compute the arrival sequence for the flights and give clearances for landing to the pilots flying in their sector on the basis of the information displayed by the CWP such air traffic, radar data, monitor displaying inbound/outbound traffic planned for the sector, telephone switchboards, airlines and airport operators preferences or priorities about arrival runways. The communication between the different ATM actors takes place over dedicated and secure communications lines. For example, for communication between the Sector Team and the pilots specific radio frequencies are used.

In this scenario, the security requirements are associated with the CWP and the Communication Lines:

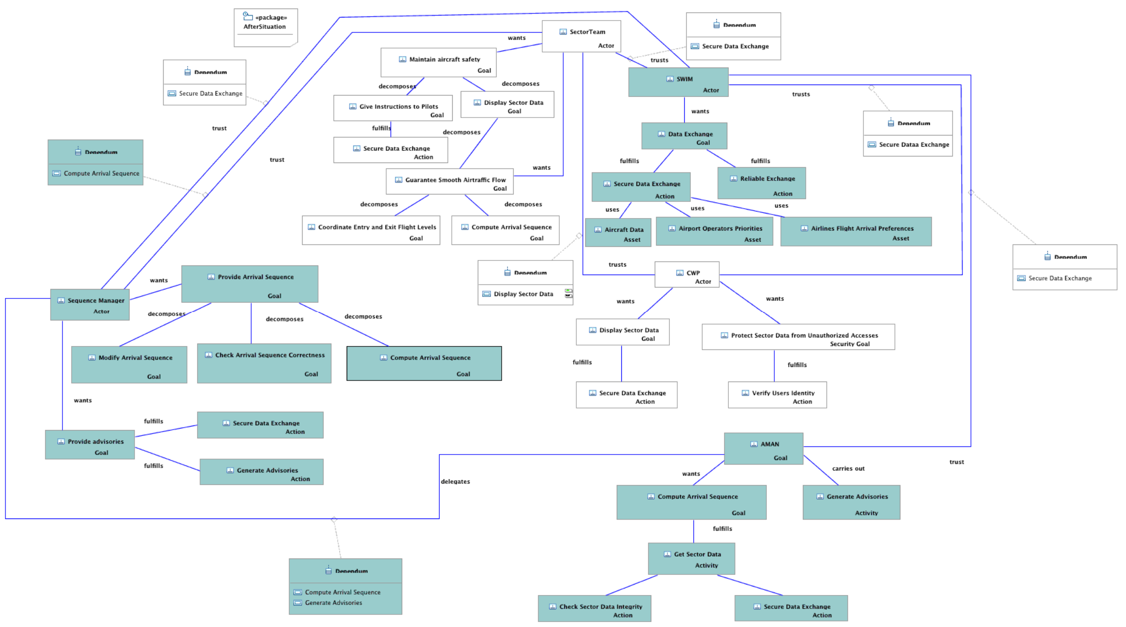
- The CWP shall provide an authentication mechanism to verify users identity

- The Communication Lines shall provide secure and reliable communication among ATM actors

As affect of the introduction of AMAN, ATM systems go under architectural, organizational, and operational changes. At architectural level, the AMAN supports the Sector Team by providing sequencing and metering capabilities for a runway, airport or constraint point, the creation of an arrival sequence using ‘ad hoc’ criteria, the management and modification of the proposed sequence, the support of runway allocation at airports with multiple runway configurations, and the generation of advisories for example on the time to lose or gain, or on the aircraft speed. At the organizational level, the introduction of the AMAN requires the introduction of a new type of ATCO, called Sequence Manager, who will monitor and modify the sequences generated by the AMAN and will provide information and updates to the Sector Team. At the operational level, on one side the AMAN interacts with the FDP, CNS, and Meteo services to collect the Airport Operators priorities for runaways usage the Airlines priorities in terms of flight arrivals, the Meteo condition, and the aircraft position that it uses to compute an ad hoc arrival sequence or to generate advisories. On the other side, the AMAN interacts with the Sequence Manager and the Sector Team through their CWPs monitor. The Sequence Manager can check the arrival sequence and the advisories generated by the AMAN, and if necessary can modify them, while the Sector Team ATCOs can only view them. Based on the information provided by the AMAN, the Sector Team gives clearances to the pilots flying in its sector. The communication between the different ATM actors does not take place over secure and dedicates lines: the actors are interconnected by the SWIM, an IP based data transport network that will replace the current point to point data systems.

In this scenario we have new security requirements that need to be satisfied:

- The CNS systems shall check the authenticity of aircraft tracks
- The AMAN shall provide selective access control for the different ATM actors (Sequence Manager, ATCOs,..)
- The AMAN shall disclose to another actor only the aircraft information necessary for the actor to perform its task (need to know principle)
- The AMAN shall check that the information coming from Meteo Services, Radars, Airlines and Airport Operators has not been alterated
- The SWIM shall require authentication sessions for users based on digitally signed certificates
- The SWIM shall be able to detect fake stakeholders and trace them in a blacklist
- The SWIM shall ensure data integrity and confidentiality.



«package» AfterSituation
SectorTeam Actor
Dependum Secure Data Exchange
Dependum Compute Arrival Sequence
Maintain aircraft safety Goal
Give Instructions to Pilots Goal
Display Sector Data Goal
Secure Data Exchange Action
Guarantee Smooth Airtraffic Flow Goal
Coordinate Entry and Exit Flight Levels Goal
Compute Arrival Sequence Goal
Dependum Display Sector Data
SWIM Actor
Data Exchange Goal
Secure Data Exchange Action
Reliable Exchange Action
Aircraft Data Asset
Airport Operators Priorities Asset
Airlines Flight Arrival Preferences Asset
Dependum Secure Dataa Exchange
CWP Actor
Display Sector Data Goal
Protect Sector Data from Unauthorized Accesses Security Goal
Secure Data Exchange Action
Verify Users Identity Action
Sequence Manager Actor
Provide Arrival Sequence Goal
Modify Arrival Sequence Goal
Check Arrival Sequence Correctness Goal
Compute Arrival Sequence Goal
Provide advisories Goal
Secure Data Exchange Action
Generate Advisories Action
AMAN Goal
Compute Arrival Sequence Goal
Generate Advisories Activity
Get Sector Data Activity
Check Sector Data Integrity Action
Secure Data Exchange Action
Dependum Compute Arrival Sequence Generate Advisories
wants
trusts
trust
decomposes
fulfills
uses
delegates
carries out

Figure 11.5: After situation

```
evolution rule NewActor {
 variables = (AMAN,CAS, SM, AS);
 event = appear {
       entity Component(AMAN);
       entity Component.Provides(AS)
                        }
 condition {
  entity DependencyGoal(CAS);
  relation DependencyGoal.Component(CAS->SM);
  entity Component(SM);
  relation Component.Requires(SM->AS);
     }
 action {
       create relation DependencyGoal.Component(CAS->AMAN);
              }
             }
```

Figure 11.6: After situation

The evolution rule represented in Figure 11.6 is an example of rules that can associated with the transformation of the requirement model represented in Figure 11.4 into the model shown in Figure 11.5. The rule is executed when the system detects the appearance of the change pattern ``The Sequence Manager depends on the AMAN for arrival sequences computation". The Condition part specifies that the Sequence Manager requires the arrival sequence to support the Sector Team in managing the air traffic in the sector. The Action part specifies that since the AMAN is now the component that provides the arrival sequences, a delegation of execution dependency for the ``Compute an Optimized Arrival Sequence" goal should be added between the Sequence Manager and the AMAN actors.

## 11.3 Mapping security requirements to UMLseCh stereotypes

In order to verify that UMLseCh design models comply with the changes in the requirements model, it is important to map the addition, removal and modification of entities or relationships to UMLseCh stereotypes.
To allow such mapping and traceability between requirements and UMLseCh models we can use change-driven transformations [Zav97]. The key concept of change-driven transformations is capturing and explicitly representing change operations, as model elements. The elements that correspond to future changes are *change commands*, while the ones that record already executed changes constitute a *change history model*. The latter kind can be automatically generated on-the-fly during the execution of model manipulation. Apart from basic change operations (creation, deletion, moving, value setting, etc.), user-defined domain-specific macro change types are also a that react to changes of the model by matching a single change operation and additional model elements, and create change commands to manipulate the target model. The created change command may be executed at a later time, even at a remote location. Thus rules are incremental and evaluated asynchronously to the update of the target model, and optionally asynchronously to the change of the source model that caused the change propagation. Change history is derived on-the-fly and automatically after the source model is updated, regardless whether the model manipulation was initiated by another (not necessarily change-driven) transformation, or by user interaction. Change history is asynchronously processed by transformation rules that should depend on the change history element, and potentially an extended condition involving the source model, but not the target. Instead of directly manipulating the target model, the transformation rules only create change commands to express the required modifications, thus allowing for deferred execution, remote processing, or piping through the runtime manipulation API of an application.
Figure 11.7 represents the process related to a change-driven transformation. MA and MA' are the requirement models before and after the change, and MB, MB' are the two states of the UMLsecCh model before and after the application of the change commands. CHMA is the change history model derived by observing the change of A, and CCB represents the change commands that affect the target model. Transformation and processing is indicated by circles.

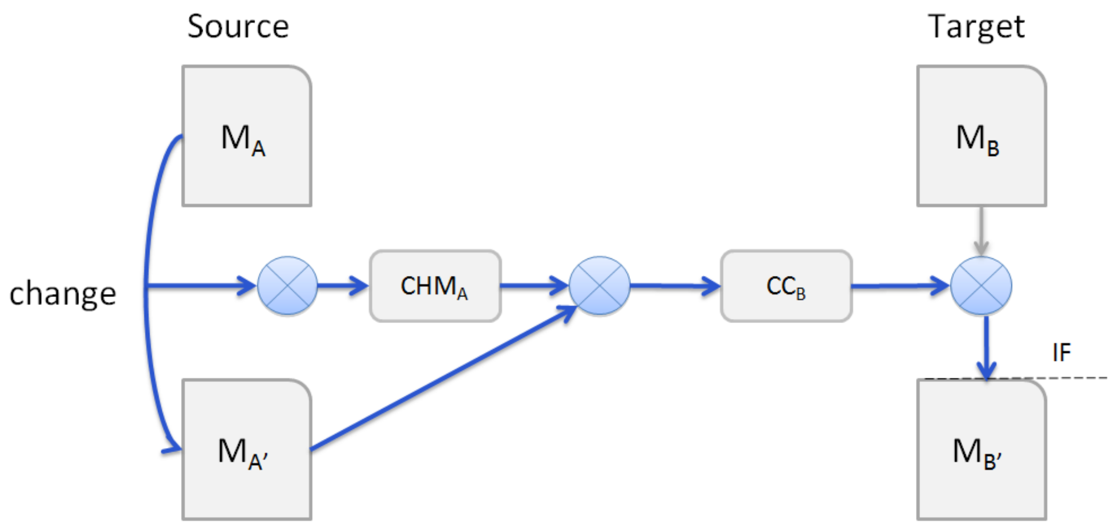


Figure 11.7: Change-driven workflow

Change driven transformation can also be used to automatically update the FOL formulas used for the security analysis when a change in the security requirements occurs. In fact, the attack conjecture is derived by security requirements and should be updated when a change in the requirements occurs in order to assess the security of the system with respect to the change. The Action part of the transformation rules will indicate the predicate(s) to be added to the formula and the type of logical connector. For example, if we refer to the ATM case study, let's assume that when the SWIM is introduced to replace the dedicated and secure communication lines, the SWIM has the requirement to enable the communication between the different ATM actors but it is not trusted by the ATM actors. Thus, a new confidentiality requirement is introduced to protect the data exchanged by means of SWIM network. This change in the security requirements should trigger the execution of a transformation rule which updates the FOL formula which describes the ATM design model by adding in AND the following predicate knows(s), where s represents the data exchanged via the SWIM network.

# 12 From Thales Security DSML to UMLseCh

In Thales environment, system security engineering classically involves:

1) The analysis and assessment of security risks encountered by the system
2) The specification of requirements for security measures to address those risks
3) The design, development, integration and validation of security architecture, functions and mechanisms that address those requirements.

Our present work is focusing on security engineering activities 1 and 2 above. Our objective is to provide adequate and efficient tooling to security engineers for an effective integration of security engineering in the process of critical system design; this will enable a better targeting of security specifications.

This section is organised as follows:

- Section 12.1 presents the principles of our approach to enhancing classical security analysis methods.
- Section 12.2 presents an overview of the Security DSML: the domain covered the actual contents of the DSML and conceptual models focussing in DSML Context Model.
- Section 12.3 present a simple web architecture example defined in DSML and an preview of related DSML Change Model
- Section 12.4 present how our DSML could be used to model UMLseCh specification in the Thales context.

## 12.1 Enhancing system security engineering in Thales

A comprehensive approach of security engineering starts with an analysis of the risks pending on the system. For critical systems, this analysis must be conducted with the biggest attention since the impacts can be very damaging for the populations in relation to those critical systems. On the other hand, overestimating risks may lead to excessive or unnecessary security measures, introducing undue rigidities and costs. The specification of security requirements builds upon this risk analysis, and aims at defining requirements that are commensurate with the risks.

Currently in our company, Security Analysis activities are carried out with the help of structured and proven methods that use referential repositories (of types of threats and vulnerabilities, of impacts and damages, attack scenarios, security functions etc.), standardized or not, and tabular, cross-matrix and dashboard based tools. EBIOS [EBI] and MEHAR [MEH] are the main methods employed at our company.

These methods imply a limited perception of the architecture of the system upon which the risk analysis is realised. In particular, we carried out a study of these methods and realized that they target on the one hand the business process supported by a system and on the other hand, in very little detail, technical and physical elements of the system (applications, databases, data files, servers, networks, mobile PCs etc.). Finer-grain knowledge of the architecture is not taken into account in these methods. The topology, data flows and functional dependencies throughout the system are especially not analyzed, which can lead to sub-optimal risk analyses and security requirements specifications.

Our work aims at developing a method that enables an enhancement of these classical risk analysis methodologies. As summarised in Figure 12.1, these enhancements rely on leveraging detailed knowledge of the targeted system in close integration with the mainstream system engineering process, and developing fine grain analyses of the actual risks at stake. This method builds upon the capacities provided by model-based development methods and techniques that are currently spreading in the systems engineering community, but are still poorly used in the security engineering domain.

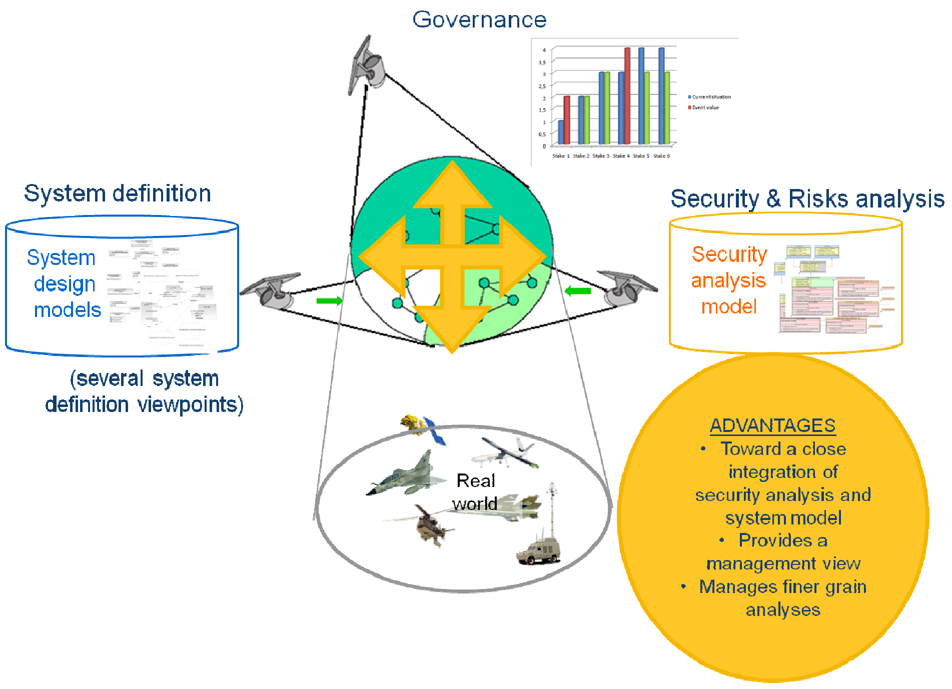

Governance
System definition
System design models
(several system definition viewpoints)
Security & Risks analysis
Security analysis model
Real world
ADVANTAGES
• Toward a close integration of security analysis and system model
• Provides a management view
• Manages finer grain analyses

Figure 12.1: Enhancing system security engineering methods

Our general objectives of enhancement are the following:

- *Objective1:* To optimize the qualification of the risks and the specification of security requirements and related security costs,
- *Objective 2:* To optimize the quality and the productivity of security engineering by capitalizing on data from one study to the next, and by proceeding to automatic calculation and consistency checking.
- *Objective 3:* To optimize the quality and the productivity of security engineering by sharing common models of the system between system design and security

analysis and thus by working on synchronized and consistent models of the system throughout the design process.

## 12.2 Thales Security DSML

### 12.2.1 Domain of interest

Our goal is to build a DSML allowing the support of finer grain, more formal security analyses that exploit formalized system architecture descriptions. The security architect formalizes security information and relates it to architecture components.

Figure 12.2 illustrates the scope and context of use for our “Security DSML”:

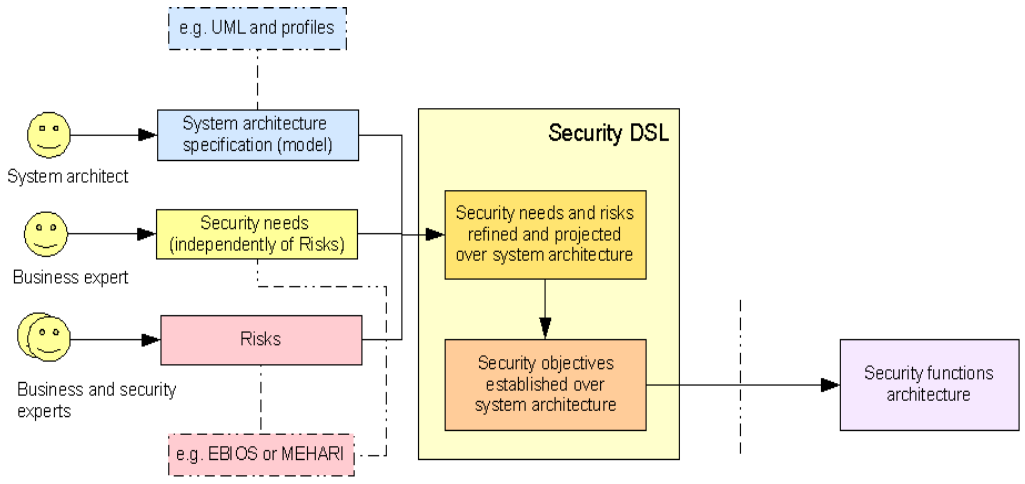

e.g. UML and profiles
System architect
System architecture specification (model)
Business expert
Security needs (independently of Risks)
Business and security experts
Risks
e.g. EBIOS or MEHARI
Security DSL
Security needs and risks refined and projected over system architecture
Security objectives established over system architecture
Security functions architecture

Figure 12.2: Scope of the Security DSML

- The System architecture model is built using distinct languages (like, for example, UML and/or UML profiles) by a System architect. *The Security DSML is defined with limited dependency upon the specific system architecture description formalism*. The objective is to be able to use the DSML concurrently with different languages for the specification of system architectures.
- Security needs are determined for each individual architecture components or groups of such, by a Business expert. A Security need is initially expressed intrinsically (e.g. “The document needs to be defence confidential”), without taking into account the risks, but only the impacts of unwanted actions and damages they may inflict.
- A Risk Analysis takes place, involving the collaboration of the Business and Security experts, in order to identify and value risks regarding system components and subcomponents.

The Security DSML shall support security needs and risks to be refined and projected on a System architecture model.

The DSML shall then support the experts work in determining which risks are unacceptable towards the specified security needs, either because they have a too important impact, a too big opportunity of happening, or both, making these components critical.

Security objectives are defined in order to reduce unacceptable risks and consequently bring the current level of security to a newly defined targeted one. The Security DSML shall support the capture of these security objectives and the assessment of their coverage of unacceptable risks pending on architectural components.

### 12.2.2 Overview

The Risk analysis model, security requirements model and context model are expressed in a dedicated DSML. As shown by Figure 12.3, these kinds of models are parts of static model:

- ***Requirement Model*** describes the specialization of Objectives into several Requirements and the links between those and the other elements of DSML (Risk, Context). Requirements are then stored in a common requirement Database which contains other kinds of requirements (safety, maintainability, etc). In Thales context, the official database of System Engineering Workbench for Requirement Management is Rational DOORS [Rat09] (Dynamic Object Oriented Requirements System). An overview of this mapping is defined in section 12.4, for further details see [Del10b].
- ***Context Model*** describes the System Architecture (Essential Elements and/or Target), related constraints and the links between those and the other elements of DSML (Risk, Requirement).
- ***Risk Model*** describes the risk characterization into threats, damages and vulnerabilities and the links between those and the other elements (Requirement, Context).

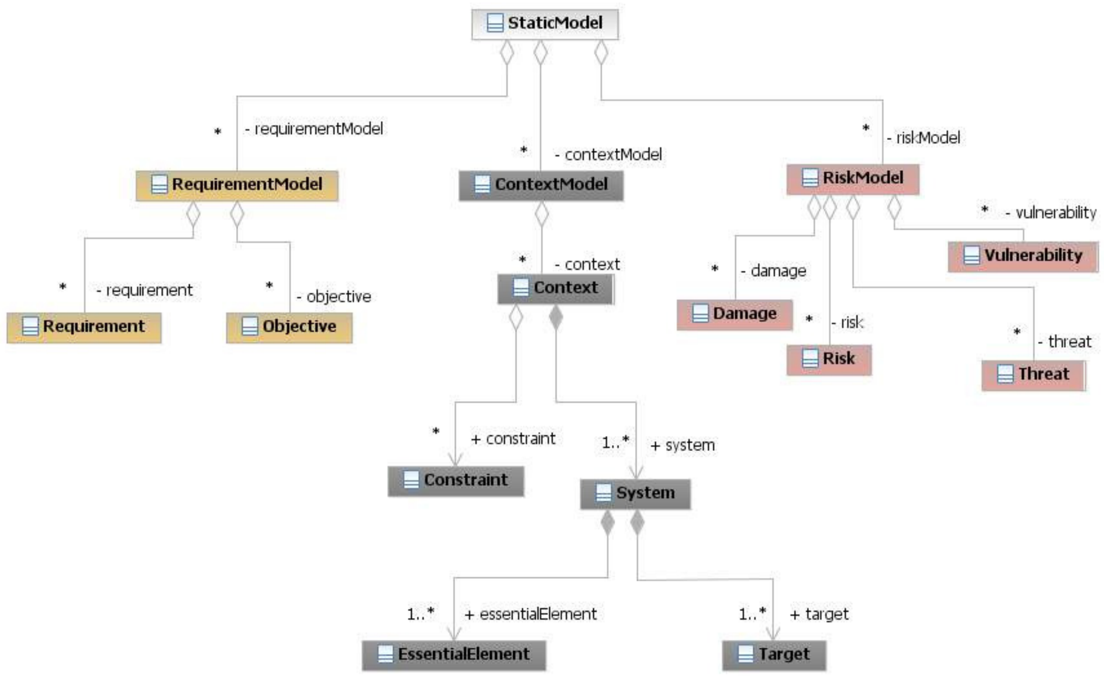


Figure 12.3: Security DSML Static Model description

To address change inside this DSML, we must consider a change model which could be mapped with all models included in static model. Figure 12.4 depicts the traceability relation between different models defined in DSML and the relation with the change model (presented in sections 12.2.3.2 and 12.3.2).

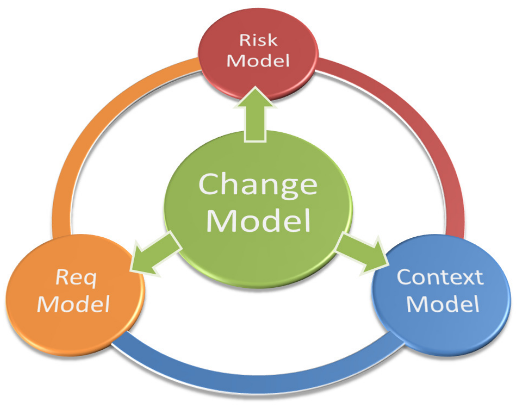


Figure 12.4: Relationship between DSML Static Models and Change Model

### 12.2.3 Conceptual Models

#### 12.2.3.1 Focus on Context Model

This deliverable cannot be the place for a detailed presentation of the context metamodel and syntax of our DSML. We are providing below representative extracts.

More details are provided in [NV09]. The core part of the conceptual model[1] is represented in Figure 12.5 below.

The system under analysis is considered to hold targets and essential elements. ***Targets*** are physical elements subject to vulnerabilities and damages by threats. ***Essential elements*** are usually more logical, functional elements: data and functions (or services, or capabilities depending on context) that are essential to the business stakes of the company, and therefore subject to security needs. Essential elements depend on targets for their implementation.

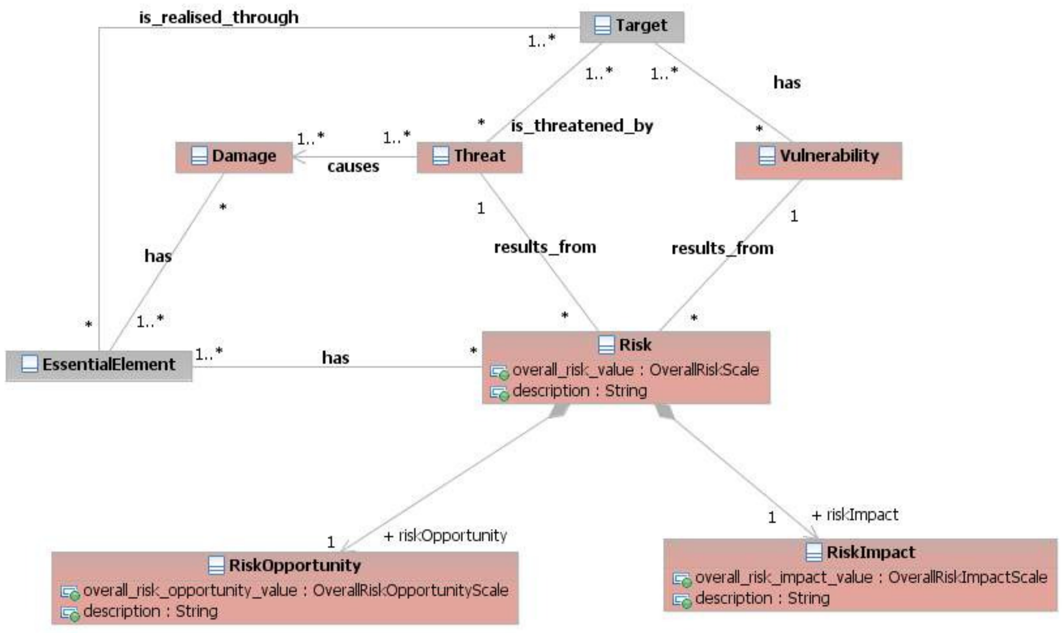


Figure 12.5: Risk Model – Conceptual Model

The central concept of our security analysis conceptual model is the one of ***risk***. A **risk** pertains to an essential element of the system. A **risk** comes from the combination of a ***threat*** that could exploit an opportunity to take advantage of a ***vulnerability*** of a target, with the ***essential element*** depending on the *target*. A risk is valuated based on its impact on the target and its opportunity to be triggered on the target.

The link between the risk analysis model and the system architecture model is illustrated in Figure 12.6.

Our approach was to introduce in the DSML placeholders for system architecture elements generic enough to support mapping to various types of architectures (whether business, system or technical). In our current prototype DSML, only three generic architectural concepts are used, as represented in the figure: components (***Component***), communication channels (***CommunicationChannel***) linking the components and data exchanged (***ExchangedData***) between components via the communication channels. Data correspond to essential elements while communication

[1] For readability sake, it is represented in the form of a conceptual model rather than a formal metamodel.

channels are considered as targets; components are considered as both essential elements and targets.

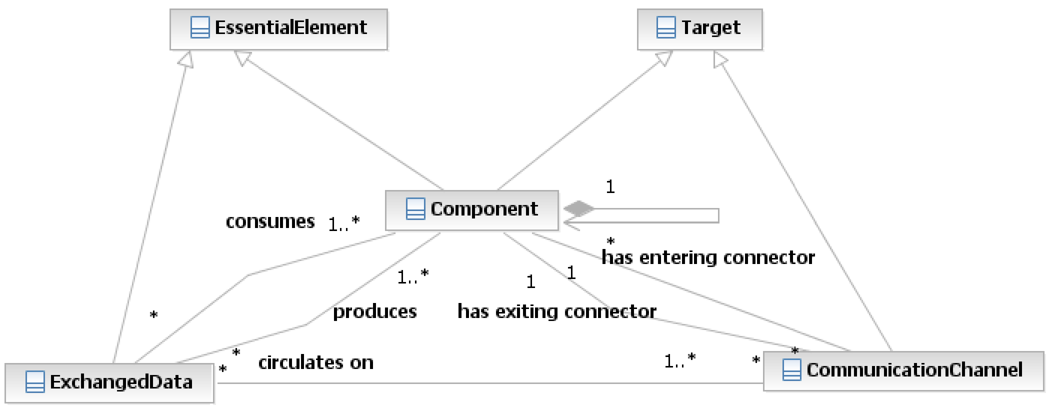


Figure 12.6: Linking architecture description to risk analysis – Conceptual Model

### 12.2.3.2Change Model

To represent traceability between changes and static model, we add a further Model into DSML: ***Change Model*** which is composed by several ***Change Lines***. As shown by Figure 12.7, a ***Change Line*** is considered as set of ***Changes*** and ***Change Transitions*** to grant consistency between successive changes which compose a Change Line.

***Change*** is described by a Change Trigger (e.g. discover a fault or a new threat) which activates a ***Change Request***. It's also possible to activate a Change Trigger by a threshold defined in an ***Evolution Function*** which monitors the static model of the system, for further detail on ***Evolution Function*** see [HD09].

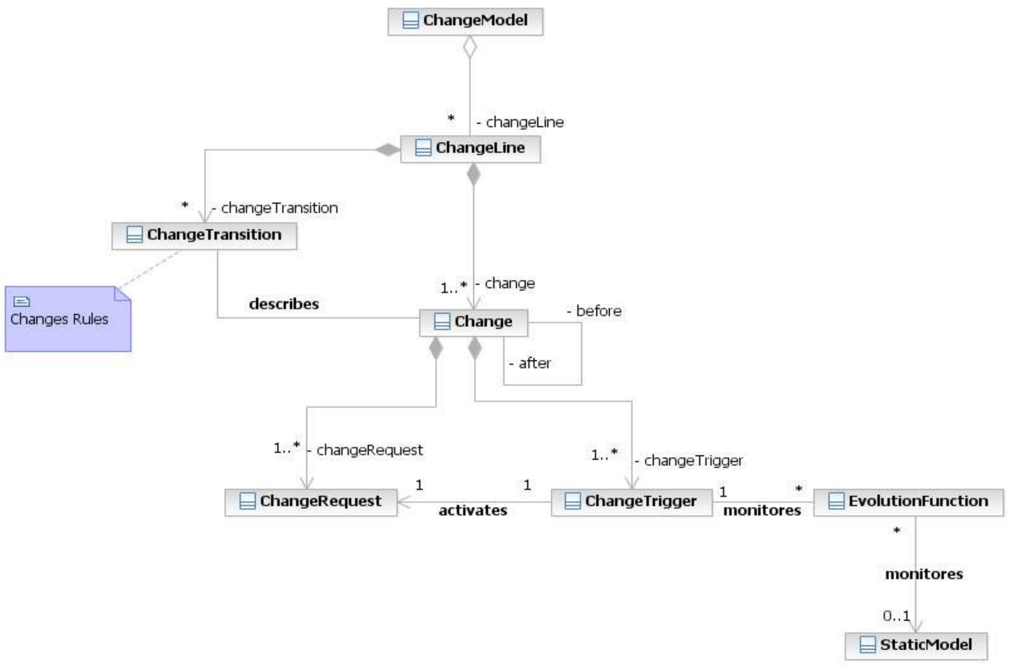


Figure 12.7: Change Model – Conceptual Model

As shown by Figure 12.8, a ***Change Request*** contains a PUID[2] to identify it and a status which represent the state of Change request (for further detail [Del10a]]). After the activation of Change Request by the Change Trigger, Change Request status is first defined in CCB (Configuration Control Board). The Configuration (or change) Control Board (CCB) is a periodic meeting between several actors of a development team (client, manager, quality, design, integration …) to define which change requests which are accepted, refused or postponed in the next version of system. The detailed behavior of Change Request is described in [Del10a].

To covers all kind of static model, ***Change Request*** is specialized into the following kinds:

- ***Requirement Change Request*** modifies Requirement Model (Requirement, Objectives). It's possible to map this kind of Change Request with DOORS Change Request.
- ***Context Change Request*** modifies Context Model (e.g. system architecture).
- ***Risk Change Request*** modifies Risk Model (Risk, Threat, Damage, and Vulnerability).

These three kinds of Change Request are dependants; a Requirement Change Request could impact on Risk Change Request and Context Change Request and vice versa. This is why we consider a traceability relation between those Change Requests. This relation is described by an "impacts_on" association (see Figure 12.8)**.**

[2] PUID = Product Unique Identifier

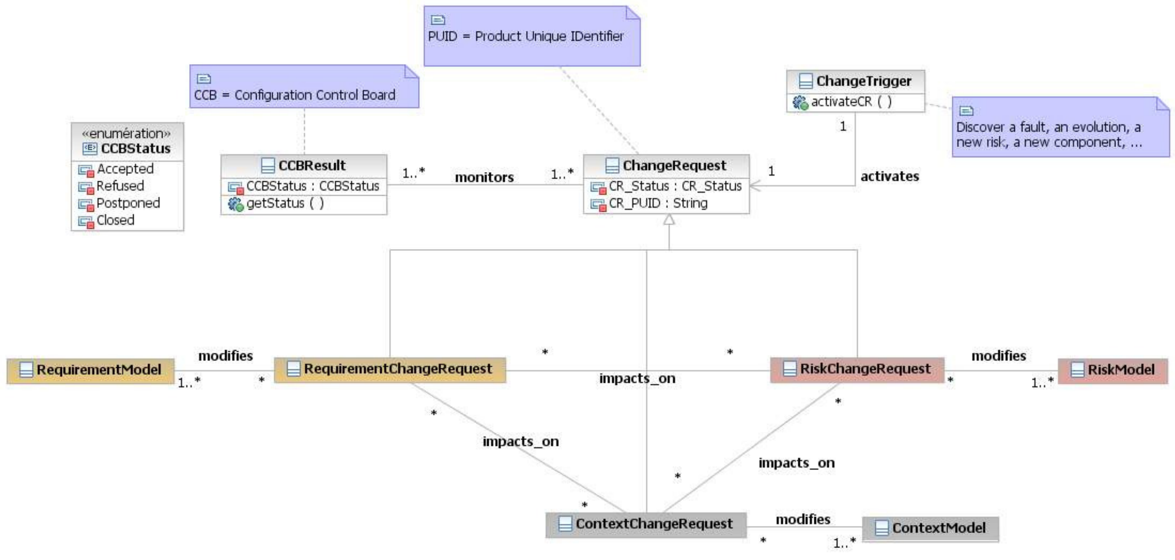


Figure 12.8: Change Request - Conceptual Model

To define correctly a ***Context Change Request***, Security Designer must first of all take into account related ***constraints*** of the context model. This relation is shown by association “respects” between Context Change Request and Constraint in Figure 12.9. These constraints describe how the service provided by the system should be realized in the context model. Independent of Platform, theses ***constraints are applied on Essential Elements*** of the System which describes the logical view of the system. These ***constraints are evaluated on specific targets*** which realize Essential Element in more concrete view dependant of the system platform. These constraints must be stored in change transition in order to preserve them on successive changes inside the change model.

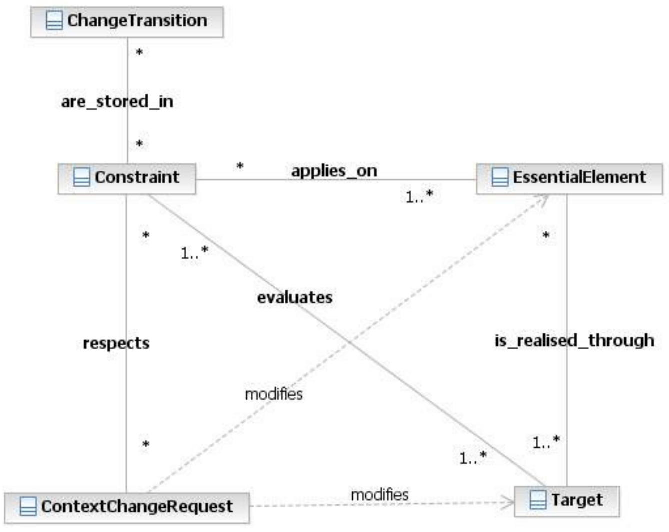


Figure12.9: Relations between Context Change Request and Context Model – Conceptual Model

## 12.3 Simple Web Architecture security implementation using the Security DSL

### 12.3.1 Static Model

Figure 12.10 shows the Simple Web Architecture[3] represented in the Security DSML Context Model. The main characteristic of the Context Model is that it shows the entire model through a "filter" that lets us view only the architectural components. The purpose of this diagram is to show an un-detailed view of the model, in which the security information shall trespass only lightly.

As we can see in Figure 12.10, architectural elements are expressed as boxes, data as discs and channels as arrows. Data linked to channels have the meaning that the data are transmitted through the channels. The same architectural component is shown more than once on the diagram, for readability and traceability reasons.

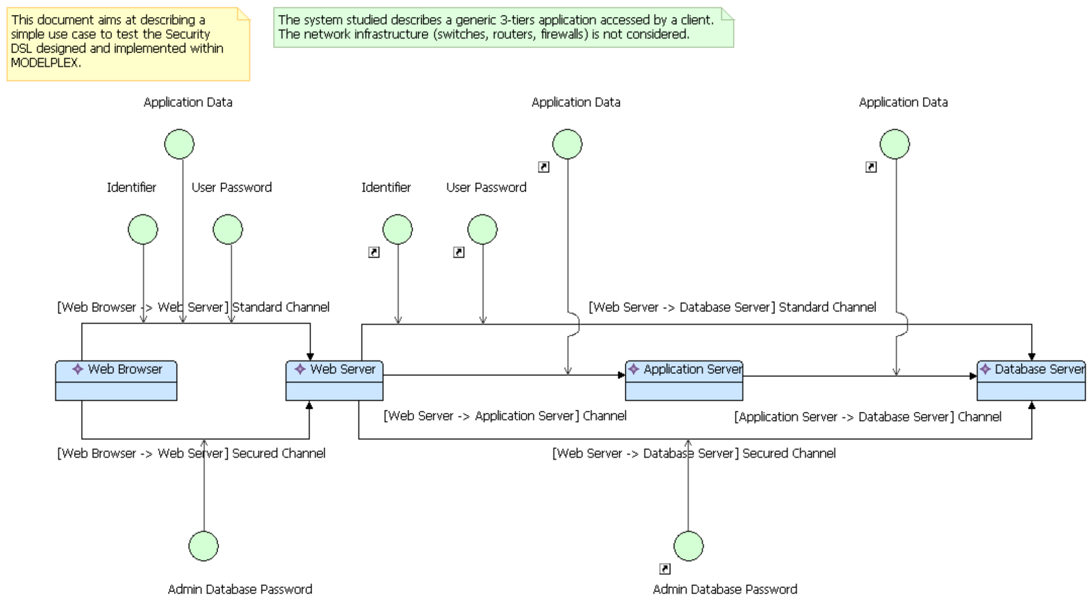
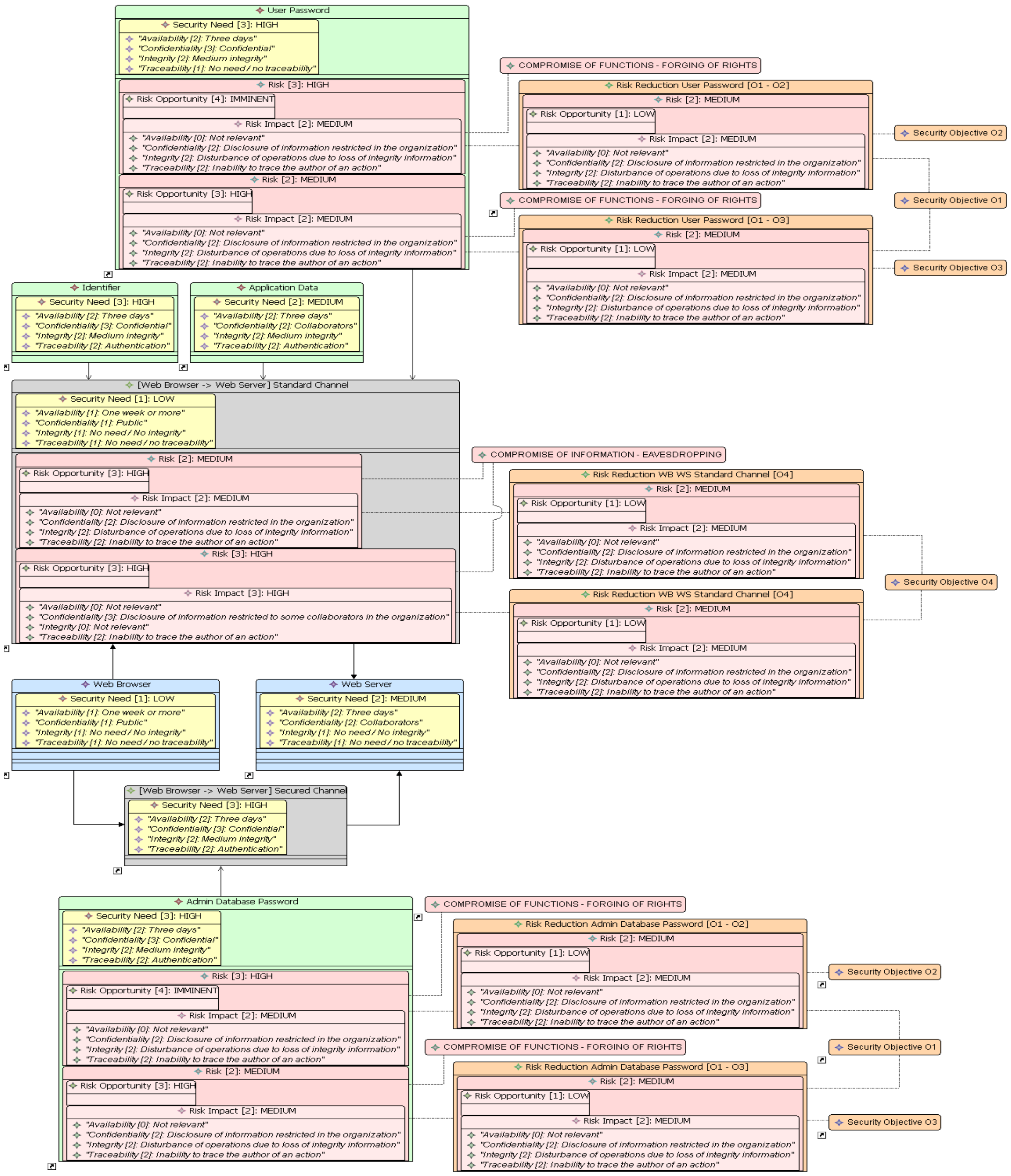

This document aims at describing a simple use case to test the Security DSL designed and implemented within MODELPLEX.
The system studied describes a generic 3-tiers application accessed by a client. The network infrastructure (switches, routers, firewalls) is not considered.
Application Data
Identifier
User Password
Application Data
Identifier
User Password
Application Data
[Web Browser -> Web Server] Standard Channel
[Web Server -> Database Server] Standard Channel
Web Browser
Web Server
Application Server
Database Server
[Web Server -> Application Server] Channel
[Application Server -> Database Server] Channel
[Web Browser -> Web Server] Secured Channel
[Web Server -> Database Server] Secured Channel
Admin Database Password
Admin Database Password

Figure 12.10: Simple Web Architecture – Context diagram (complete view)

The Security DSML Risk Model can be used to show partial views of the risk analysis of the model. They do not filter it and let users create and view all the security information predefined in the language. In Figure 12.11, we can see a partial view of the risk model of the Simple Web Architecture, represented in the Risk Model. This partial view treats the Web browser, the Web server and their connections and data transmissions. In the Risk Model, all architectural and security components are expressed as boxes (even channels). A color code is employed for simple observation: elements are blue, data are green, channels are gray; security needs are yellow, risks elements are red.

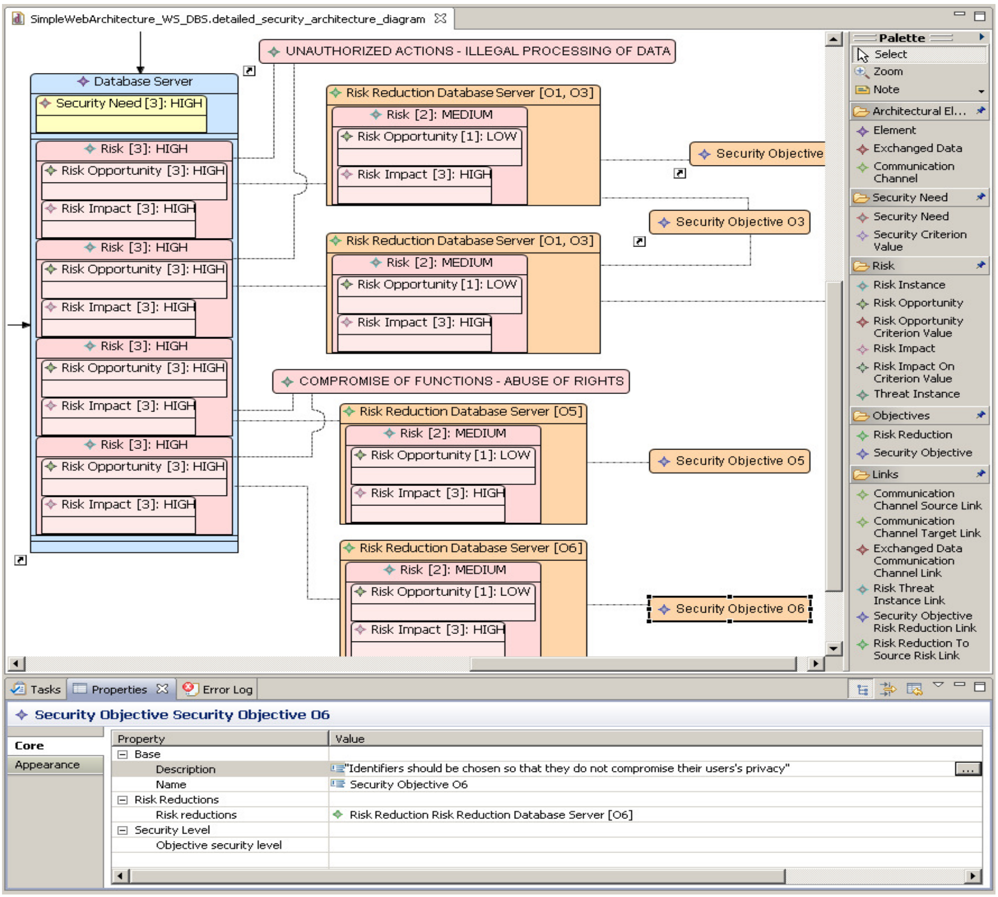
[3] The network infrastructure (switches, routers, firewall) is not considered in this example.



User Password
Security Need [3]: HIGH
"Availability [2]: Three days"
"Confidentiality [3]: Confidential"
"Integrity [2]: Medium integrity"
"Traceability [1]: No need / no traceability"
Risk [3]: HIGH
Risk Opportunity [4]: IMMINENT
Risk Impact [2]: MEDIUM
"Availability [0]: Not relevant"
"Confidentiality [2]: Disclosure of information restricted in the organization"
"Integrity [2]: Disturbance of operations due to loss of integrity information"
"Traceability [2]: Inability to trace the author of an action"
Risk [2]: MEDIUM
Risk Opportunity [3]: HIGH
COMPROMISE OF FUNCTIONS - FORGING OF RIGHTS
Risk Reduction User Password [O1 - O2]
Risk Opportunity [1]: LOW
Risk Reduction User Password [O1 - O3]
Security Objective O2
Security Objective O1
Security Objective O3
Identifier
"Traceability [2]: Authentication"
Application Data
Security Need [2]: MEDIUM
"Confidentiality [2]: Collaborators"
[Web Browser -> Web Server] Standard Channel
Security Need [1]: LOW
"Availability [1]: One week or more"
"Confidentiality [1]: Public"
"Integrity [1]: No need / No integrity"
"Traceability [1]: No need / no traceability"
Risk Impact [3]: HIGH
"Confidentiality [3]: Disclosure of information restricted to some collaborators in the organization"
"Integrity [0]: Not relevant"
COMPROMISE OF INFORMATION - EAVESDROPPING
Risk Reduction WB WS Standard Channel [O4]
Security Objective O4
Web Browser
Web Server
[Web Browser -> Web Server] Secured Channel
Admin Database Password
Risk Reduction Admin Database Password [O1 - O2]
Risk Reduction Admin Database Password [O1 - O3]

Figure 12.11: Simple Web Architecture – Risk Model (partial view WB – WS)

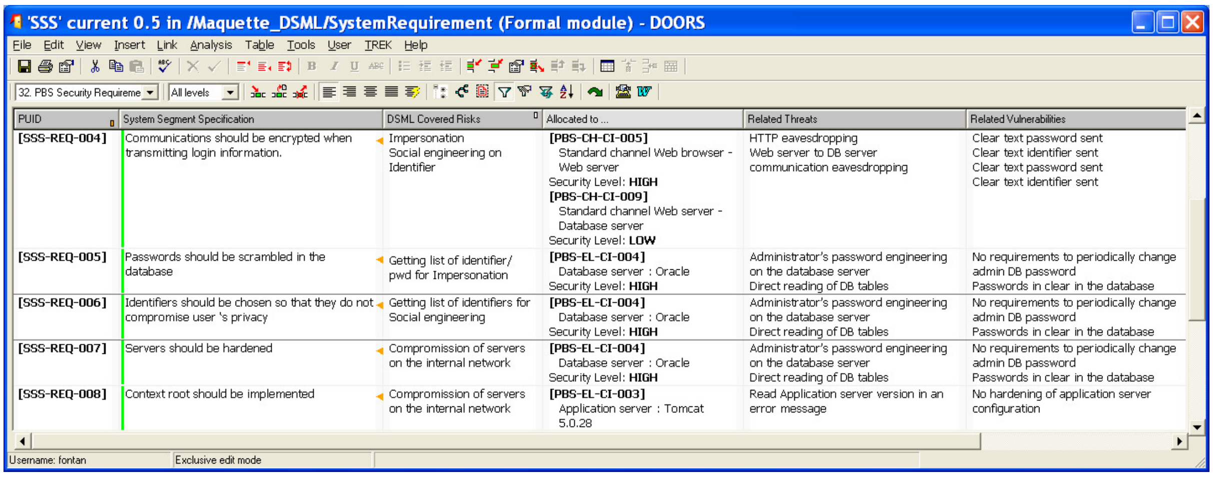
In the requirement model, all architectural and security components are expressed as boxes (even channels). A color code is employed for simple observation: elements are blue, data are green, channels are gray; security needs are yellow, and objectives are orange. Figure 12.12 depicts a close view on Security Objective O6 (Identifiers should be chosen so that they do not compromise user‘s privacy). Figure 12.13 presents the requirements derived from security objective in DOORS.


SimpleWebArchitecture_WS_DBS.detailed_security_architecture_diagram
UNAUTHORIZED ACTIONS - ILLEGAL PROCESSING OF DATA
Database Server
Security Need [3]: HIGH
Risk [3]: HIGH
Risk Opportunity [3]: HIGH
Risk Impact [3]: HIGH
Risk Reduction Database Server [O1, O3]
Risk [2]: MEDIUM
Risk Opportunity [1]: LOW
Risk Impact [3]: HIGH
Security Objective
Security Objective O3
COMPROMISE OF FUNCTIONS - ABUSE OF RIGHTS
Risk Reduction Database Server [O5]
Security Objective O5
Risk Reduction Database Server [O6]
Security Objective O6
Palette
Select
Zoom
Note
Architectural El...
Element
Exchanged Data
Communication Channel
Security Need
Security Need
Security Criterion Value
Risk
Risk Instance
Risk Opportunity
Risk Opportunity Criterion Value
Risk Impact
Risk Impact On Criterion Value
Threat Instance
Objectives
Risk Reduction
Security Objective
Links
Communication Channel Source Link
Communication Channel Target Link
Exchanged Data Communication Channel Link
Risk Threat Instance Link
Security Objective Risk Reduction Link
Risk Reduction To Source Risk Link
Tasks
Properties
Error Log
Core
Appearance

Security Objective Security Objective O6

| Property | Value |
|---|---|
| Base | |
| Description | "Identifiers should be chosen so that they do not compromise their users's privacy" |
| Name | Security Objective O6 |
| Risk Reductions | |
| Risk reductions | Risk Reduction Risk Reduction Database Server [O6] |
| Security Level | |
| Objective security level | |

Figure 12.12: Close view on Security Objective O6 emphasizing its Properties – Requirement Model


'SSS' current 0.5 in /Maquette_DSML/SystemRequirement (Formal module) - DOORS
File Edit View Insert Link Analysis Table Tools User TREK Help
32. PBS Security Requireme
All levels

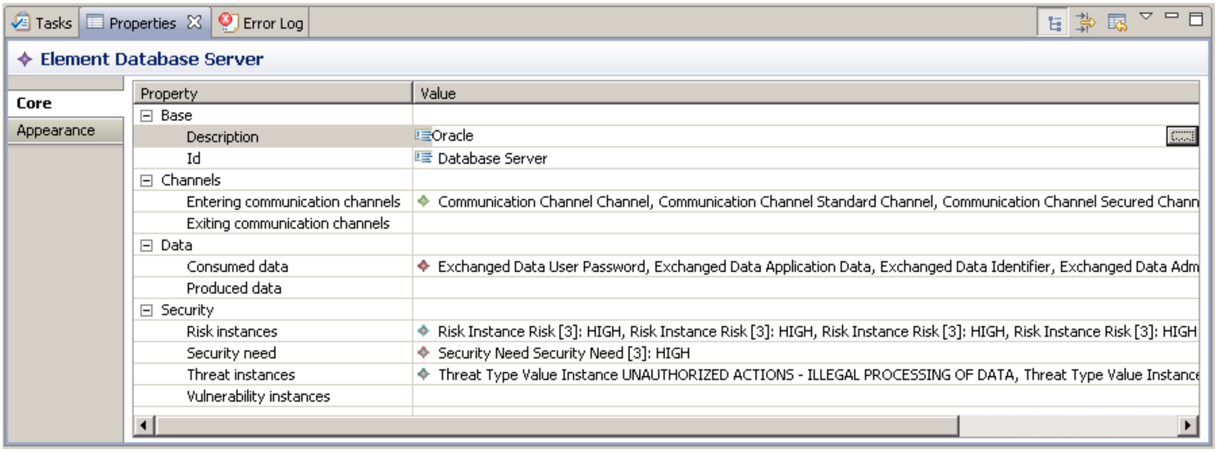
| PUID | System Segment Specification | DSML Covered Risks | Allocated to ... | Related Threats | Related Vulnerabilities |
|---|---|---|---|---|---|
| [SSS-REQ-004] | Communications should be encrypted when transmitting login information. | Impersonation<br>Social engineering on Identifier | [PBS-CH-CI-005]<br>Standard channel Web browser - Web server<br>Security Level: **HIGH**<br>[PBS-CH-CI-009]<br>Standard channel Web server - Database server<br>Security Level: **LOW** | HTTP eavesdropping<br>Web server to DB server communication eavesdropping | Clear text password sent<br>Clear text identifier sent<br>Clear text password sent<br>Clear text identifier sent |
| [SSS-REQ-005] | Passwords should be scrambled in the database | Getting list of identifier/ pwd for Impersonation | [PBS-EL-CI-004]<br>Database server : Oracle<br>Security Level: **HIGH** | Administrator's password engineering on the database server<br>Direct reading of DB tables | No requirements to periodically change admin DB password<br>Passwords in clear in the database |
| [SSS-REQ-006] | Identifiers should be chosen so that they do not compromise user 's privacy | Getting list of identifiers for Social engineering | [PBS-EL-CI-004]<br>Database server : Oracle<br>Security Level: **HIGH** | Administrator's password engineering on the database server<br>Direct reading of DB tables | No requirements to periodically change admin DB password<br>Passwords in clear in the database |
| [SSS-REQ-007] | Servers should be hardened | Compromission of servers on the internal network | [PBS-EL-CI-004]<br>Database server : Oracle<br>Security Level: **HIGH** | Administrator's password engineering on the database server<br>Direct reading of DB tables | No requirements to periodically change admin DB password<br>Passwords in clear in the database |
| [SSS-REQ-008] | Context root should be implemented | Compromission of servers on the internal network | [PBS-EL-CI-003]<br>Application server : Tomcat 5.0.28 | Read Application server version in an error message | No hardening of application server configuration |

Username: fontan    Exclusive edit mode

Figure 12.13: Derived Requirements expressed in DOORS

There is information that has been set not to be shown in diagrams, but which can be consulted in the Properties View (Description, constraints applied on it…), as can be seen in Figure 12.14. This properties view is also defined in DOORS as shown by Figure 12.15.

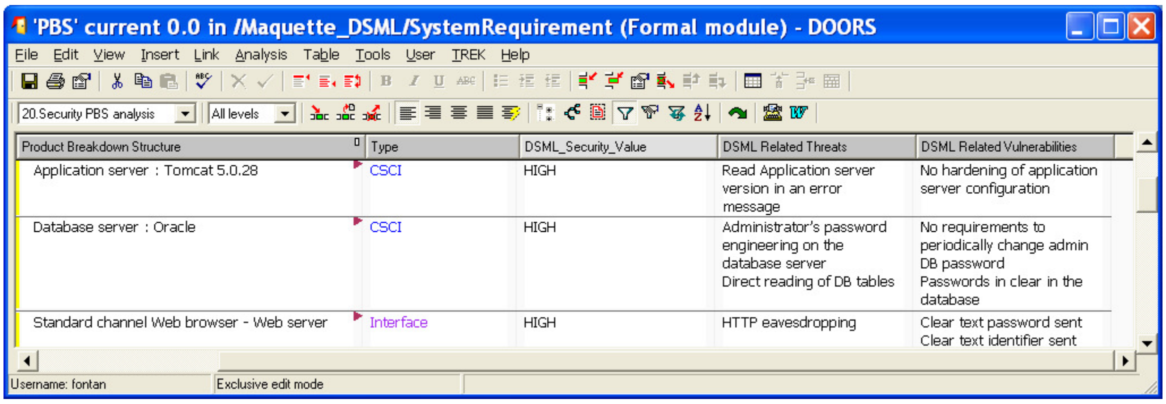
For all specificities of the concrete syntax and tooling of DSML, see [NV09]. For the description of the mapping between DOORS and DSML, see [Del10b].



Element Database Server

| Property | Value |
|---|---|
| Base | |
| Description | Oracle |
| Id | Database Server |
| Channels | |
| Entering communication channels | Communication Channel Channel, Communication Channel Standard Channel, Communication Channel Secured Chann |
| Exiting communication channels | |
| Data | |
| Consumed data | Exchanged Data User Password, Exchanged Data Application Data, Exchanged Data Identifier, Exchanged Data Adm |
| Produced data | |
| Security | |
| Risk instances | Risk Instance Risk [3]: HIGH, Risk Instance Risk [3]: HIGH, Risk Instance Risk [3]: HIGH, Risk Instance Risk [3]: HIGH |
| Security need | Security Need Security Need [3]: HIGH |
| Threat instances | Threat Type Value Instance UNAUTHORIZED ACTIONS - ILLEGAL PROCESSING OF DATA, Threat Type Value Instance |
| Vulnerability instances | |

Figure 12.14: Properties of the Database Server in DSML


File Edit View Insert Link Analysis Table Tools User TREK Help
20.Security PBS analysis
All levels

'PBS' current 0.0 in /Maquette_DSML/SystemRequirement (Formal module) - DOORS

| Product Breakdown Structure | Type | DSML_Security_Value | DSML Related Threats | DSML Related Vulnerabilities |
|---|---|---|---|---|
| Application server : Tomcat 5.0.28 | CSCI | HIGH | Read Application server version in an error message | No hardening of application server configuration |
| Database server : Oracle | CSCI | HIGH | Administrator's password engineering on the database server<br>Direct reading of DB tables | No requirements to periodically change admin DB password<br>Passwords in clear in the database |
| Standard channel Web browser - Web server | Interface | HIGH | HTTP eavesdropping | Clear text password sent<br>Clear text identifier sent |

Username: fontan    Exclusive edit mode

Figure 12.15: Database Server description in DOORS

### 12.3.2 Change Model

In this example, we will consider that security designer would to change the Database Server in DSML. Designer must open in first a Context Change Request inside the Change Model and afterwards includes it on Context Model. The Detailed Change Management process is described in [Del10a].

A Change Model is composed by several Change Lines which represent major release of the system (e.g. version 2.0). Inside each Change Line, Changes are represented by states linked by a transition. Transition enables to preserve postponed Change Requests and related constraints defined in context model. Transition grants consistency between successive changes which compose a Change Line.

In accordance to section 12.2.3.2, Change Requests are represented inside Changes. A Change corresponds to minor release of the system (e.g. version 2.1). For each change we define the related Change Requests defined in this release. As shown by Figure 12.16, Change Request contains an identifier and a status which represent to current state of Change Request inside Change Management Process (for further detail about Change Management Process see [Del10a].

In current example, we consider that a designer would to change Web Sever Oracle to Web Server IIS, first of all he open a new Context Change Request ("Context Change Request 101" in Figure 12.16). This change request is evaluated during a CCB and the corresponding status is set on Postponed. The change and its impact will be taken into account in the next minor release.

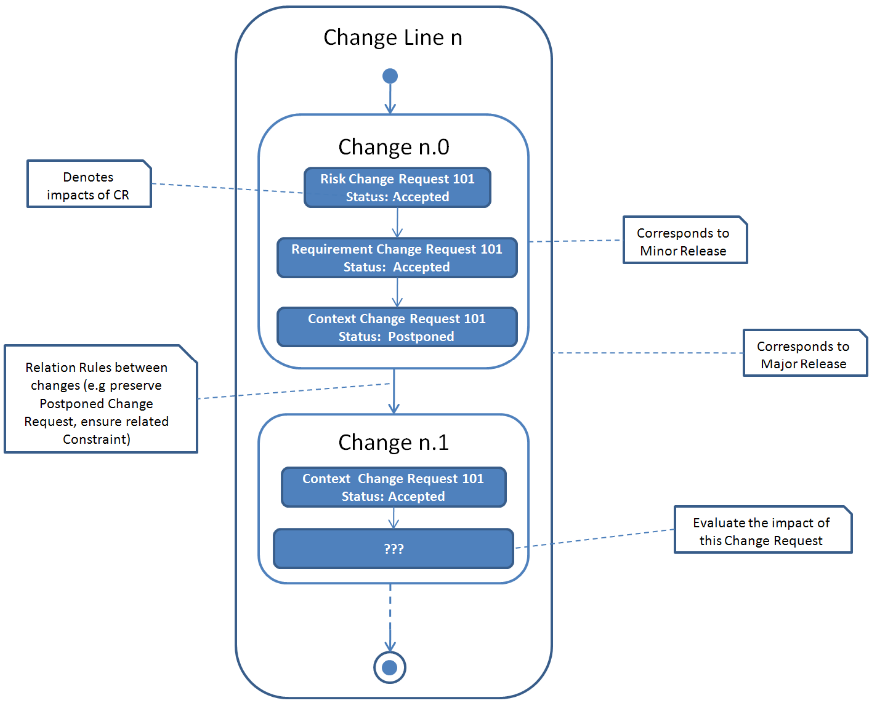

Change Line n
Change n.0
Risk Change Request 101
Status: Accepted
Requirement Change Request 101
Status: Accepted
Context Change Request 101
Status: Postponed
Change n.1
Context Change Request 101
Status: Accepted
???
Denotes impacts of CR
Corresponds to Minor Release
Corresponds to Major Release
Relation Rules between changes (e.g preserve Postponed Change Request, ensure related Constraint)
Evaluate the impact of this Change Request

Figure 12.16: DSML Change Model - preview

In order to ensure traceability of change, Context Change Request is also represented in Context Model with a textual description, as show by Figure 12.17. In this case, Web Server description is not changed in the targeted component because the current status of context change request is set up to "Postponed", it could be changed when the status will be set to "In progress". The Detailed Change Management process is described in [Del10a].

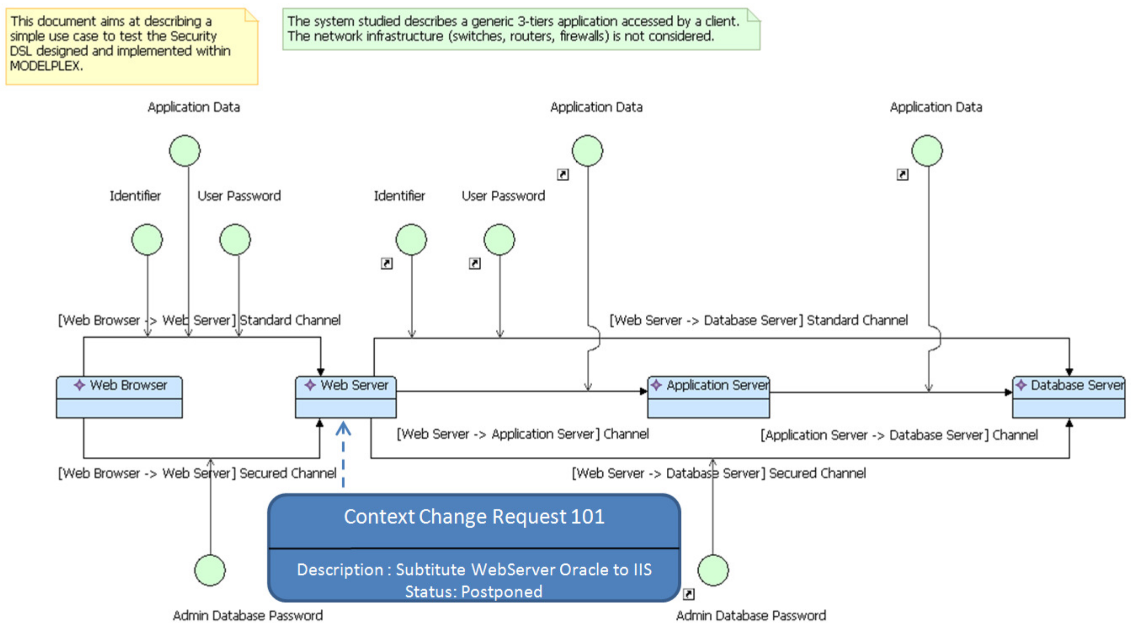

This document aims at describing a simple use case to test the Security DSL designed and implemented within MODELPLEX.
The system studied describes a generic 3-tiers application accessed by a client. The network infrastructure (switches, routers, firewalls) is not considered.
Application Data
Application Data
Application Data
Identifier
User Password
Identifier
User Password
[Web Browser -> Web Server] Standard Channel
[Web Server -> Database Server] Standard Channel
Web Browser
Web Server
Application Server
Database Server
[Web Server -> Application Server] Channel
[Application Server -> Database Server] Channel
[Web Browser -> Web Server] Secured Channel
[Web Server -> Database Server] Secured Channel
Context Change Request 101
Description : Subtitute WebServer Oracle to IIS
Status: Postponed
Admin Database Password
Admin Database Password

Figure 12.17: Connection between Context Change Request and Context Model - preview

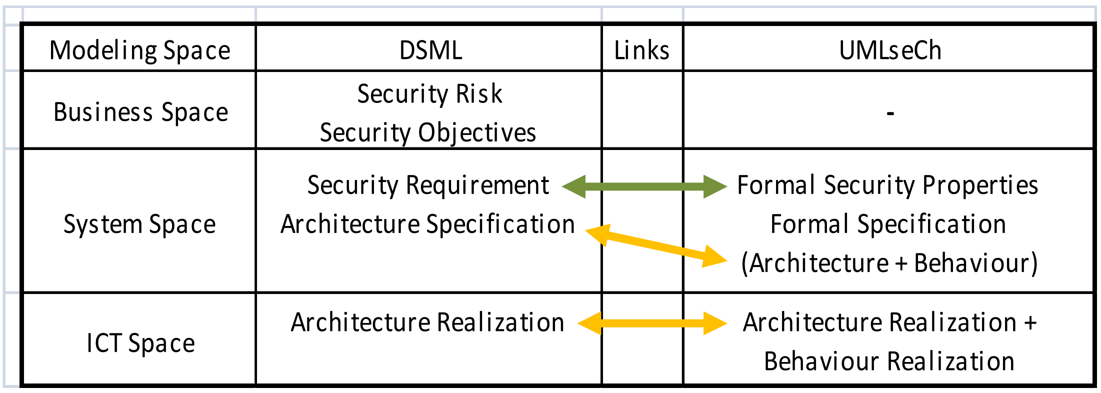
## 12.4 From Thales Security DSML to UMLseCh

### 12.4.1 UMLseCh vs DSML in Thales Architecture Framework

The Thales architecture Framework (TAF) is organized around three distinct modelling spaces:

- The Business space addresses the computation-independent analysis of the operational context: organisation, processes, information flows and interaction between logical business entities.
- The System space captures the system solution to realise the business capabilities, defined at a technology-independent level (such as Platform Independent Model).
- The ICT[4] space (equivalent to Technical space) captures the system solution at a technology-specific level, with focus on the definition of the platform-specific design for integration. It allows generating implementation code.

[4] Information Communication Technologies



| Modeling Space | DSML | Links | UMLseCh |
|---|---|---|---|
| Business Space | Security Risk<br>Security Objectives | | - |
| System Space | Security Requirement<br>Architecture Specification | ↔<br>↔ | Formal Security Properties<br>Formal Specification<br>(Architecture + Behaviour) |
| ICT Space | Architecture Realization | ↔ | Architecture Realization +<br>Behaviour Realization |

Figure 12.18: Comparison between Security DSML and UMLseCh profile in Modelling Spaces

Figure12.18 compares Security DSML and UMLseCh profile in the TAF modeling spaces. Security DSML is in higher space than UMLseCh which is positioned in System Space and ICT Space. Moreover Security DSML doesn't have behavior representation: these languages are complementary. Note that Change Model doesn't appear explicitly in TAF description: this process is considered as transverse process in TAF.

### 12.4.2 How to map Security DSML and UMLseCh?

Figure 12.19 shows the possible mapping between Thales Security DSML and UMLseCh profile, to do this we must consider two kinds of relations:

- ***Traceability relation*** between Security Requirement of Security DSML and Formal Security Properties of UMLseCh (e.g. Secrecy, Integrity …). This relation enables to connect results of Formal Verification with Security Requirements which are expressed by UMLseCh Formal Properties. Requirements are stored in a common requirement Database (DOORS TREK [Rat09]) which contains other kinds of requirements (safety, maintainability, etc).

- ***Consistency relation*** between Architecture Specification and Realization. This relation relies on *Model Driven Viewpoint Engineering* approach [BX09]. This approach consists to build a common architecture repository and several viewpoints[5] of this repository (e.g. DSML view and UMLseCh view are shared in the same architecture description). To ensure and prove consistency between those models, it's necessary to define consistency rules expressed by *Conjunctive Normal Form*[6] (CNF) Formulas [JP05], for further details see next section.

[5] In *Model Driven Viewpoint Engineering* approach, a view is an instance of a viewpoint.
[6] In boolean logic, a formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals. As a normal form, it is useful in automated theorem proving.

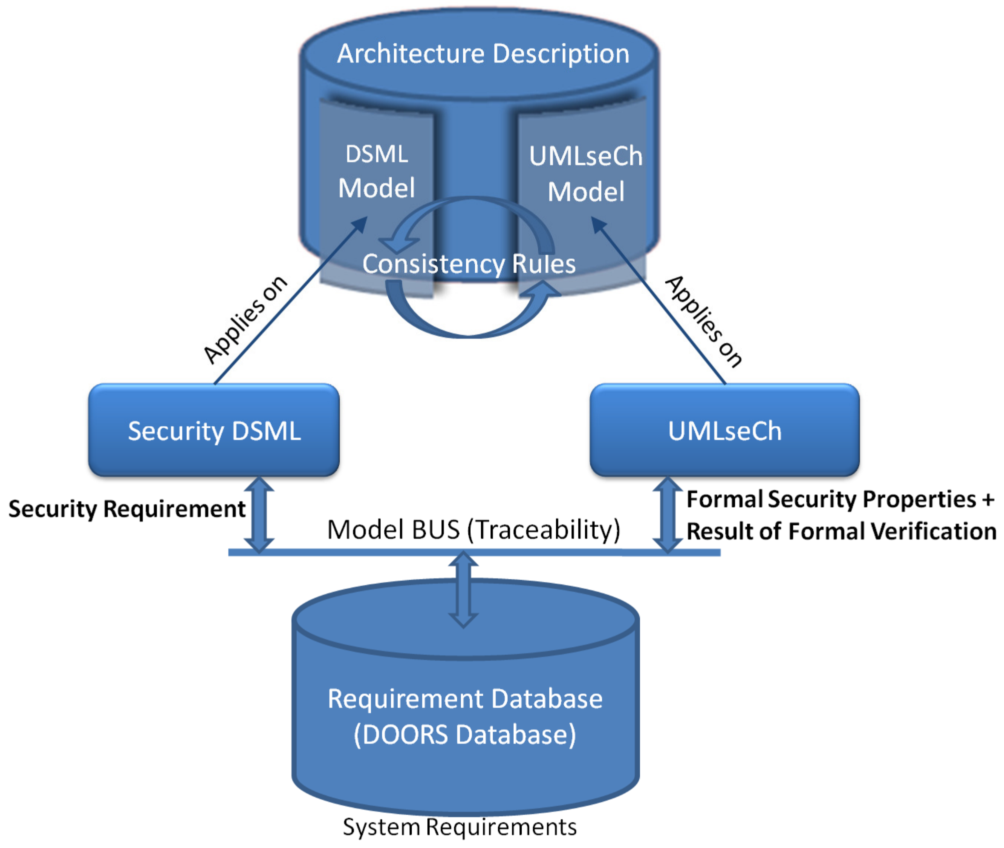


Figure 12.19: Mapping between Thales Security DSML and UMLseCh profile

### 12.4.3 Focus on Consistency Relations

As suggested by Figure 12.19, consistency relation between DSML and UMLseCh are only defined for architectural model, DSML doesn't include behavioural representation of the system. It's also possible to define consistency relations between Change Model of DSML and UMLseCh Change Diagram.

#### 12.4.3.1 Consistency analysis with the Praxis Tool

To check consistency between models, we have opted for the Praxis tool [BX08]. Based on building events of model, Praxis is an approach that deals with the detection and the diagnosis of inconsistencies. Praxis represents models as sequences of unitary model editing operations [BX08]. The formalism doesn't require the use of a unique metamodel, it allows for many models with many metamodels to be used at the same time.

In Praxis, inconsistencies are specified using first order logic over the sequence of unitary model editing operations. Moreover, Praxis allows for static analysis of the inconsistency rules that allow determining for each unitary modification the rules for which the consistency value can change.

The entire approach is integrated in the UML modeling tool from IBM, Eclipse Modeling Framework. This integration is based on the SWI-Prolog engine. Editing operations are represented by Prolog facts and inconsistency rules by Prolog query. The validation of Praxis has shown the following advantages and drawbacks:

- Performances: Praxis is quite fast. The use of static information making the detection incremental speeds up inconsistency detection during model evolution.
- Completeness: Praxis can be used to detect and to diagnosis both for structural and methodological inconsistencies.

Conjunctive Normal Form is represented in Praxis by EMF Text. Consistency Rules are expressed by a set of methods, for each rule correspond a specific method which is the conjunction of several properties. Figure 12.20 shows an elementary consistency rule between two classes (C1 and C2) of two different models ({1} and {2}). In this rule, C1 and C2 must have the same name.

```
rules myRuleSet {
["Classes {1} and {2} have the same name!"]
  public classesSameName(C1,C2) <=> and {
    create(C1, #class),
    create(C2, #class),
    setProperty(C1, #name, Name1),
    setProperty(C2, #name, Name2),
    'equal(Name1, Name2)
  }
}
```

Figure 12.20: Example of CNF properties

### 12.4.3.2Consistency relations between DSML and UMLseCh

Figure 12.21 summarizes the set of DSML elements and UMLseCh elements whose are taken in account for consistency rules. Note that the relation between security requirements of the DSML and formal properties of UMLseCh are based on traceability and are not described in this table.

As suggested by Figure 12.21, consistency relations between DSML and UMLseCh are only defined for system representation and Change representation: Risk and Requirement Models of the security DSML are not defined in UMLseCh. DSML doesn’t include behavioral representation (Statechart Diagram, Sequence Diagram) of the system. Theses languages are so complementary.

Inside Change representation, the Change Model defined in DSML is equivalent to UMLseCh Statechart Diagram. The Type of Change attribute of Change Request correspond to Change annotation in Class Diagram (the behavioral description of the system is not defined in the security DSML). This is why we define the application of this Consistency relation as partial.

| Representation | Security DSML | | UMLseCh | | Consistency Relation | |
|---|---|---|---|---|---|---|
| | DSML Models | DSML elements | UMLseCh Diagrams | UMLseCh element | Applicable? | Relation based on ? |
| Risk | Risk Model | - | Ø | Ø | No | Not Applicable |
| Requirement | Requirement Model | - | Ø | Ø | No | Not Applicable |
| System | Context Model | Essential Element | Use Case Diagram | Classifier (Use Case) | Yes | Name |
| | | Target | Class/Deployment Diagram | Classifier (Class, Node) | Yes | Name, Attribute, Methods |
| | | Communication Channel | | Association + AssociationClass (Interface) | Yes | Source and Target of association, list of signals in Interface |
| | | Exchanged Data | | Signal | Yes | Name, Attribute |
| | | Constraint | | Constraint | Yes | Contract expressed in CNF |
| | Ø | Ø | Statechart Diagram | - | No | Not Applicable |
| | Ø | Ø | Sequence Diagram | - | No | Not Applicable |
| Change | Change Model | Change | Statechart Diagram | State | Yes | Name |
| | | Transition | | Transition | Yes | Source/Target of Transition, Guard of Transitions |
| | | Change Request | Class Diagram | Change Annotation | Partial | Type of Change in DSML Change Request and kind of annotation in UMLseCh |

Table 12.21: Consistency relations between Security DSML and UMLseCh profile

# 13 From UMLseCh Design Models to Model-based Testing

The aim of Model-Based Testing (MBT) is to use a model in order to compute test cases that will then be executed on a concrete system. This kind of approach is complementary to the verification step (in UMLsec, the security analysis) that is supposed to be done before using the model for test generation. Usually, the test model that is considered is different from the model supposed to design the system. While the design model is generally very abstract, the test model is more concrete, closer to the implementation, which eases the concretization of the tests to be run on the system under test. Unfortunately, in a large majority of cases, the test model is a refinement of a design model, and design models can very rarely be used as they are to produce tests that aim at being run on the implementation.

In this section, we study the possible reuse of UMLseCh design models for model-based test generation, in the objective of gathering both verification and validation steps in a single model.

The Model-Based Testing approach, in the context of the SecureChange project, aims at two main objectives that can be related to the UMLseCh design models. The first objective is to automatically generate test suites dedicated to the validation of the software evolutions (see Deliverable 7.2 for more details). The second objective is to produce security-oriented tests that will be in charge of ensuring the conformance of the system under test to the security policy. These two aspects are complementary but independent. We focus in this section on the use of UMLseCh models for model based security testing.

This section first recalls the test generation principles used in our automated test generation Smartesting's Test Designer (TD) technology, used as a basis for the mode-based testing activity in Secure Change. We then compare the existing UMLseCh elements and discuss their reuse/adaptation to be employed with TD. We expose a forward-looking use of UMLseCh models for generating security-oriented tests. Finally, we discuss the UMLseCh notation for secured evolution and its possible use in a model-based testing process.

## 13.1 Current Model-Based Testing approach with Smartesting's Test Designer technology

Test Designer is a model based test generation technology, industrialized by the Smartesting Company. In the context of model based testing, a formal behavioural model (called the Test Model) is used to automatically compute the test cases that will later on be concretized to be run on the system under test (offline testing). The model also provides the oracle, namely the expected results of the tests, in terms of return values of operations and successive states reached by the test case.

The current Model-Based Testing approach with Smartesting's Test Designer is given in Figure 13.1.

The process starts by the analysis of the informal specification and requirement documents. A first modelling step consists in designing a test model that will be used for test generation. The test generation process covers the formal model, and generated test cases are made of sequences of operations calls. To produce the test cases, two strategies are possible. The first one consists in employing a model coverage criterion, namely a transition coverage, which is automatically applied by the tool. The second option is to manually design a test scenario as a regular expression combining sequences of operations and intermediate states that have to be reached by the tests. This latter technique makes it possible to relate the test scenario to high level properties that one may want to exercise on the system. Abstract test cases are then concretized to be run on the system under test (SUT).

In its current version, the Test Designer technology takes as input three UML diagrams, a subset of UML 2.1 (the latest version of UML) for model-based testing purposes. This subset allows formal behaviour models of the SUT to be designed, which can be mechanically interpreted to generate test suites. The subset uses class, instance and state diagrams, plus OCL expressions.

The **class diagram** describes the data model, organised in classes, containing attributes and operations, and relations between classes. The operations of the class diagram have to contain OCL constraints that describe their pre- and post-conditions. The post-conditions are expressed using classical OCL, but are interpreted as an action language, meaning that logical "and"s between predicates may represent a compound condition (if present in a decision – IF … THEN … ELSE … END) or a sequential assignment (when present in the THEN or ELSE parts). Finally, notice that TD class diagram forbid the use of inheritance between classes, and not object instance may be dynamically created. In practice, the validation engineer has to provide a special object, representing the system under test, which carries operations representing its API.

The **object diagrams** are used to specify the object instances existing in the system at the initial state. As dynamic object creation is not supported by the tool, this diagram provides all the test data that will be used during the execution of the system. In particular, the dynamic creation of objects can be simulated by isolated instances of objects that are not related to any other instance.

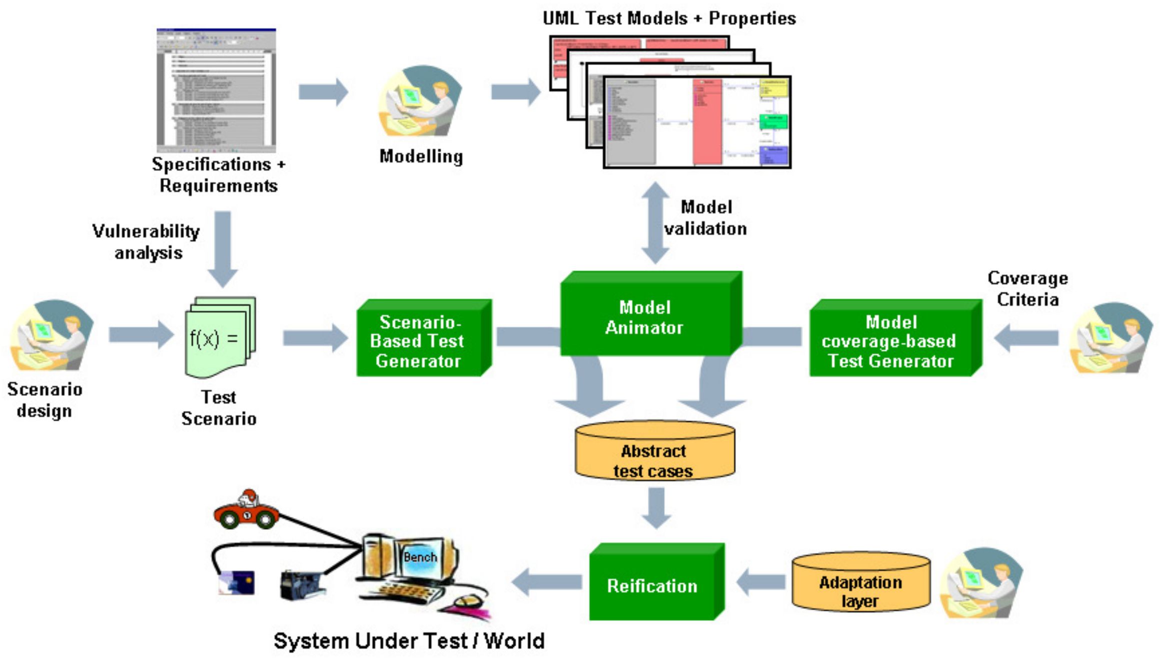


Figure 13.1: MBT with Smartesting's Test Designer

Third, and finally, Test Designer supports the use of layered **Statechart diagrams** that describes the evolution of the SUT object. The guard and effect of transitions are defined using the OCL language.

### 13.1.1 Example of diagrams used for TestDesigner

Figure 13.2 represents a class diagram used in TestDesigner. The diagram represents a SmartCard that contains a hierarchy of Security Domains (that can be assimilated to directories on a file system. Each security domain has a parent SD and possibly children. Also, security domains contain applications that are installed on the smart card.

The diagram contains a main entity that represents the System Under Test. This entity is supposed to carry all the operations that can be invoked when testing the system (a.k.a. control points).

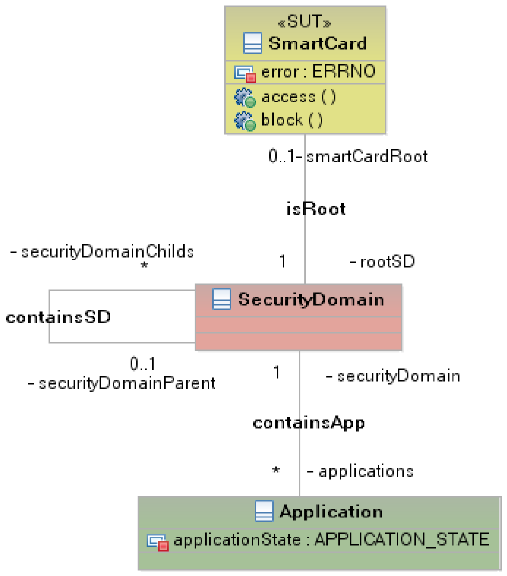


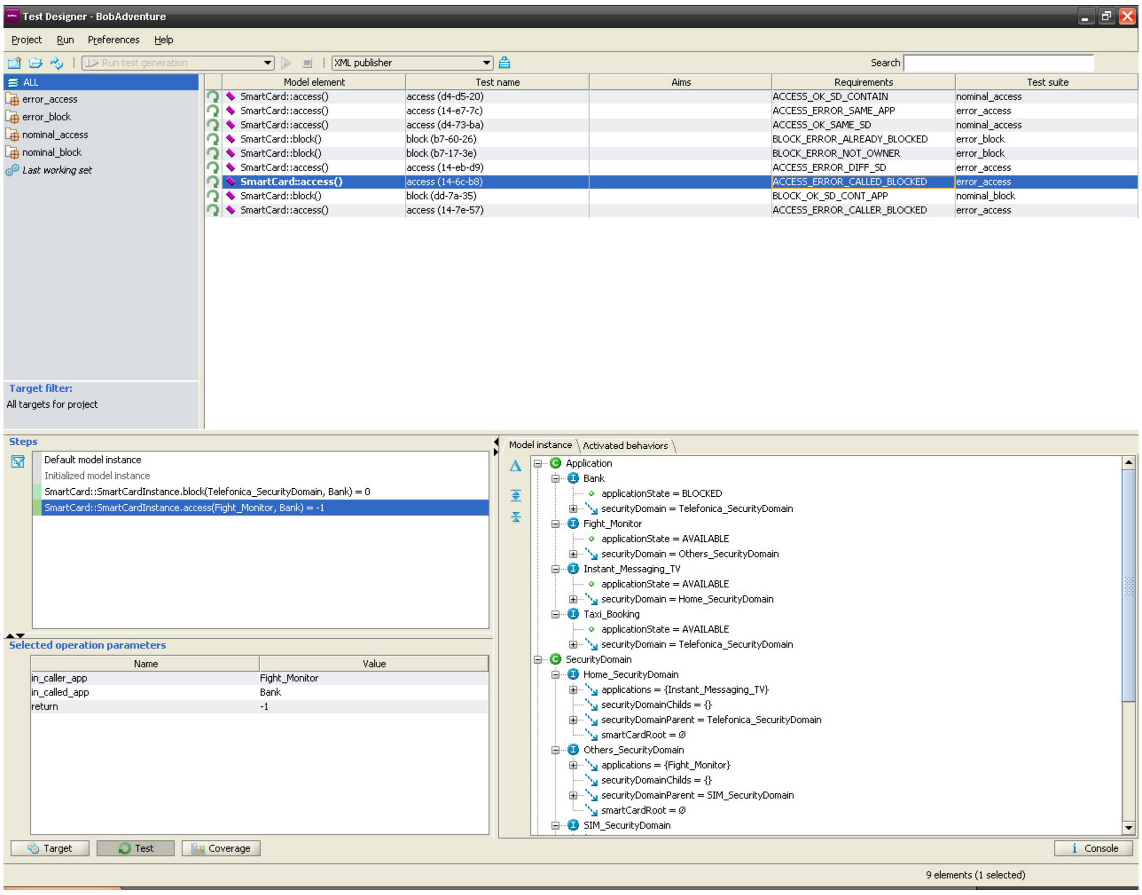
Figure 13.2 : Class diagram used in TestDesigner

In order to represent the dynamics of the system, OCL constraints have to be added to describe the behaviours of the SUT operations. Figure 13.3 gives the pre- and postconditions of the block() operation carried by the SmartCard. This operation contains specific tags (prefixed by ---@REQ) that mark the requirements that are expressed in the operation's behaviours. When employed, TestDesigner will produce tests that exercise these behaviours.

```
if in_app.applicationState = APPLICATION_STATE::BLOCKED then
        ---@REQ: BLOCK_ERROR_ALREADY_BLOCKED
        self.error = ERRNO::ALREADY_BLOCKED and
        result = -1
else
        if in_sd.applications->exists(app | app=in_app) then
                ---@REQ: BLOCK_OK_SD_CONT_APP
                in_app.applicationState = APPLICATION_STATE::BLOCKED and
                result = 0
        else
                ---@REQ: BLOCK_ERROR_NOT_OWNER
                self.error = ERRNO::NOT_OWNER and
                result = -1
        endif
endif
```

Figure 13.3: OCL constraints describing the Block() operation

Figure 13.4 gives a screenshot of the TestDesigner tool, in which the tests are classified according to the requirements they cover.



| Model element | Test name | Aims | Requirements | Test suite |
|---|---|---|---|---|
| SmartCard::access() | access (d4-d5-20) | | ACCESS_OK_SD_CONTAIN | nominal_access |
| SmartCard::access() | access (14-e7-7c) | | ACCESS_ERROR_SAME_APP | error_access |
| SmartCard::access() | access (d4-73-ba) | | ACCESS_OK_SAME_SD | nominal_access |
| SmartCard::block() | block (b7-60-26) | | BLOCK_ERROR_ALREADY_BLOCKED | error_block |
| SmartCard::block() | block (b7-17-3e) | | BLOCK_ERROR_NOT_OWNER | error_block |
| SmartCard::access() | access (14-eb-d9) | | ACCESS_ERROR_DIFF_SD | error_access |
| SmartCard::access() | access (14-6c-b8) | | ACCESS_ERROR_CALLED_BLOCKED | error_access |
| SmartCard::block() | block (dd-7a-35) | | BLOCK_OK_SD_CONT_APP | nominal_block |
| SmartCard::access() | access (14-7e-57) | | ACCESS_ERROR_CALLER_BLOCKED | error_access |

Steps

```
Default model instance
Initialized model instance
SmartCard::SmartCardInstance.block(Telefonica_SecurityDomain, Bank) = 0
SmartCard::SmartCardInstance.access(Fight_Monitor, Bank) = -1
```

Selected operation parameters

| Name | Value |
|---|---|
| in_caller_app | Fight_Monitor |
| in_called_app | Bank |
| return | -1 |

Model instance | Activated behaviors

```
Application
    Bank
        applicationState = BLOCKED
        securityDomain = Telefonica_SecurityDomain
    Fight_Monitor
        applicationState = AVAILABLE
        securityDomain = Others_SecurityDomain
    Instant_Messaging_TV
        applicationState = AVAILABLE
        securityDomain = Home_SecurityDomain
    Taxi_Booking
        applicationState = AVAILABLE
        securityDomain = Telefonica_SecurityDomain
SecurityDomain
    Home_SecurityDomain
        applications = {Instant_Messaging_TV}
        securityDomainChilds = {}
        securityDomainParent = Telefonica_SecurityDomain
        smartCardRoot = Ø
    Others_SecurityDomain
        applications = {Fight_Monitor}
        securityDomainChilds = {}
        securityDomainParent = SIM_SecurityDomain
        smartCardRoot = Ø
    SIM_SecurityDomain
```

Figure 13.4: Screenshot of the TestDesigner tool

## 13.2 Relationship between UMLseCh design models and TestDesigner models

In order to map UMLseCh design models into Test Designer, we can reason on the UMLseCh diagrams.

UMLseCh **class diagrams** can be used in TD. However, it is mandatory that these diagrams contain OCL constraints that will describe the behaviors of the operations. OCL is not exploited in UMLseCh, their addition to UMLseCh class diagrams can be done transparently and is thus not a problem at all.

UMLseCh **sequence diagrams** are used at two purposes, either to specify the behaviour of security mechanisms, in general, and, in particular, they can be used to describe cryptographic protocols, involving pre-defined functions, such as encryption, decryption, signature, hashing, etc.

At the current time, the use of UMLseCh sequence diagrams for protocol testing in TD is not possible. Nevertheless, test generation for UMLseCh sequence diagrams when describing security mechanisms is possible and discussed in the following part.

### 13.2.1 New security-oriented test generation techniques based on UMLseCh elements

The UMLseCh toolset works using an intermediate format named XMI (XML Metadata Interchange) that describes the content of a set of UML diagrams. Such a file can be exploited in order to extract security-relevant informations that would have been modelled by UMLseCh.

#### 13.2.1.1 Use of UMLseCh stereotypes

For the diagrams that can be reused in Test Designer without modifications, it is possible to take into account the following UMLseCh stereotypes, so as to produce new testing strategies:

- critical data with tags “integrity”, “secrecy” or “authenticity”: describe the fact that the considered data is somehow a critical data, that have to be protected. When verified with UMLseCh tools, the process will check that the security properties attached to this data is preserved on the model and though the possible evolutions. Nevertheless, there is no guarantee that these properties also exist on the System Under Test. Testing scenarios may try to exercise the model with the objective to exercise the property for these protected data. For example, testing scenarios involving the coverage of definitions and uses of a secret data may possibly reveal security faults on the SUT. This kind of approach necessary needs to be coupled with security monitors that will evaluate the preservation of security properties at test execution-time, revealing an error is the test cases leads the system to a state in which the secret data is read.

- rbac with tags “role”, “right” and “protected”: makes it possible to describe Role-Based Access control rules, in order to check that the model of the system enforces this access control policy. Such formalism can be used to derive testing strategies in which attempts will be made to access a private data in various unacceptable conditions.

#### 13.2.1.2 Use of UMLseCh models

In this section, we present how UMLseCh models can be exploited for test generation purposes.

Under certain conditions depending on their content, UMLseCh **activity diagrams** can be seen as labelled (guarded) transition systems, which can be explored, from its initial state to one of the final states, so as to produce test cases representing high-level interactions of objects. This graph can be covered using the “classical” graph coverage criteria such as *all-nodes* (producing a set of tests that goes through each node), *all transitions* (producing a set of tests that goes through each transition), *all k-paths* (producing a set of tests that covers all the transitions and iterates each loop k times), *all paths* (producing a set of tests that covers all the paths that can be explored). The latter criterion being very explosive and not realistic on large systems, its choice is questionable.

When **sequence diagrams** describe the behaviour of security mechanisms, they can be exploited similarly to activity diagrams, in order to extract test scenarios from the message sequence chart that is given.

## 13.3 Model-Based Testing for Changes

The UMLseCh extension describes the possible evolutions that may occur from a given version of a model to another. By analyzing the possible changes, the UMLseCh tool suite makes it possible to ensure that the security properties that applied on the initial model are preserved for any of the changes described in the UMLseCh model.

Model-based testing works on using models for generating tests. The test generation solution that we propose in SecureChange (Deliverable 7.2) consists in considering two models, one for version N, the other for version N+1. In this context, the use of a single UMLseCh model is not feasible, since we need to have an instantiation of the changes for our technique to work. Once again, we face the problem of the level of abstraction that differs between the two approaches.

Nevertheless, from a methodological point of view, it is not a problem. First, a model is designed using UMLseCh, it contains security properties that are verified using the UMLseCh tool set. This model may consider the description of possible changes, whose effects are checked as secure (also by the UMLseCh tool set). In the end, a possible instantiation of these changes is concretely done, it is thus assumed to also satisfy the initial security properties as long as the changes that were concretely made correspond to changes that were expected at the design level. The only verification necessary here is to check that the instantiation fits the evolution model. Once the two models exist and are both checked as secured, the test generation process (testing the evolutions on the system under test) takes place, considering the two models so as to derive the tests cases that are relevant w.r.t. the evolutions that happened.

# Conclusion

This document defines an extension (UMLseCh) to the selected security modelling notation (UMLsec) which allows to model secure system evolution in general, and that also includes a specific notation for modelling systems based on smart cards. Hints about how an analysis tool could implement this notation are given (first step towards Task 4.3 “Extend existing security analysis tools with adaptive security”).

To illustrate the use of the notation the GlobalPlatform and the ePurse application are modelled. This also shows how the methodologies presented in this deliverable can be applied in the context of the industrial application scenario POPS.

A formal analysis of the preservation of secrecy under composition is also included. This is a starting point for Task 4.2 “Provide formal foundation for evolutive security extension”: component modification is a common form of evolution, therefore results involving component composition are desirable.

Connections with the SecureChange lifecycle process, Security Requirements and Model-Based Testing are shown. This are all topics under development by other Work Packages in the SecureChange project. A comparison between UMLseCh and the Security DSML modelling tool used by the industrial partner Thales is also highlighted.

Deliverable 4.1 provides thus not only the expected results as specified in the Description of Work, but also includes several starting points towards Deliverable 4.2 (“Formally founded automated security analysis tools for this notation with a link to code-level verification”).

# Glossary

**Abstract Design**

**change** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that the modelling element and all its sub-elements has or is ready to undergo change.

**current_change** This stereotype can be attached to a subsystem, are used across all UML diagrams and specifies that one or more of the subsystem's sub-elements have been changed.

**future_change** This stereotype can be attached to a subsystem, are used across all UML diagrams and specifies that one or more of the subsystem's sub-elements are ready to undergo change.

**new** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that a new modelling element has been added to the model.

**modified** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that the modelling element have been changed.

**deleted** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that a modelling element have been deleted.

**allowed_addition** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that adding a new element to the model is an allowed kind of change.

**allowed_modify** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that modifying an existing element in the model is an allowed kind of change.

**allowed_delete** This stereotype can be attached to any elements in a model, are used across all UML diagrams and specifies that deleting an existing element in the model is an allowed kind of change.

### Concrete Design

**substitute** This stereotype can be attached to any model element and specifies a list of possible substitutive elements and a condition formulated in first order logic relative to other stereotypes and model elements in order to make the evolution coherent with the new elements.

**substitute-all** This stereotype can be attached to any subsystem and specifies a list of elements to be substituted and a list of substitutions. As in the stereotype « substitute » it allows a condition in FOL to control change. The use of a meta variable to better specify the changing model elements and the substitution is allowed.

**add** This stereotype is syntactic sugar for a substitution which preserves the model element as it is and adds some new element/stereotype to it.

**add-all** This stereotype is syntactic sugar for « substitute-all » which preserves a list of model element as they are and adds some new element/stereotype to them.

**del** This stereotype is syntactic sugar for a substitution which deletes a model element.

**del-all** This stereotype is syntactic sugar for « substitute-all » which deletes a list of model elements in a subsystem.

### Smart-cards

**secure runtime env.** This stereotype can be attached to subsystem or node through several UMLMsec diagrams to ensure proper memory management of the data and code of the applications and user data within Smart-card.

**secure interface** For proper implementation of the security mechanism(protect from bypass, deactivated, corrupted and circumvented), this stereotype would be added to node and link that represents the smart card components along with relevant security mechanism.

**legitimate process** This is a high level stereotype that specifies the enforcement of the required legal constraints and security policies under smart card components within the UMLsec diagrams.

**data filter** To enforce availability of data and service, the stereotype specifies the necessity to monitor and if require to filter the data by the node and throughout the established communication channel.

# References

[Bre10] R. Breu. Ten principles for living models - a manifesto for change-driven software engineering. In *4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2010). IEEE Computer Society Conference Proceedings.*, 2010.

[Bro99] Manfred Broy. A logical basis for component-based systems engineering. In *Calculational System Design. IOS*. Press, 1999.

[BS01a] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[BS01b] Manfred Broy and Gheorghe Stefănescu. The algebra of stream processing functions. *Theor. Comput. Sci.*, 258(1-2):99–129, 2001.

[BS01c] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on stream*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[BX08] Mounier I et al. Blanc X, Mougenot A. Detecting model inconsistency through operation-based model construction. In *In Proc. Int'l Conf. Software engineering (ICSE'08)*, volume 1, pages 511–520, 2008.

[BX09] Barais O. Labreuche C. Blanc X., Mougenot A. Model driven viewpoint engineering: State of the art., 2009.

[Com07] ISO 15408:2007 Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 2: Part 2; Security Functional Components, CCMB-2007-09-002, September 2007.

[Del10a] Deliverable 2.1: An architectural blueprint and a software development process for security-critical lifelong systems, 2010. Unpublished Draft Report ICT-FET-231101 D2.1, SecureChange (EU ICT-FET-231101).

[Del10b] Deliverable 3.2: Methodology for evolutionary requirements, 2010. Unpublished Draft Report ICT-FET-231101 D3.2, SecureChange (EU ICT-FET-231101).

[Dor02] C. J. Dorofee. *Managing information security risks: The OCTAVE approach*. Pearson Education, 2002.

[EBI] Ebios method. `http://www.ssi.gouv.fr/en/confidence/ebiospresentation.html`.

[GPS06] Globalplatform card specification version 2.2, March 2006.

[GPS08] Global platform uicc configuration version 1.0, October 2008.

[HD09] M. S. H. Dahl. Deliverable 5.2: Documentation of forecasts of future evolvement., 2009. Unpublished Draft Report ICT-FET-231101 D5.2, SecureChange (EU ICT-FET-231101).

[Inc03a] Sun Microsystems Inc. Application programming interface java card platform, version 2.2.1, June 2003.

[Inc03b] Sun Microsystems Inc. Runtime environment specification java card platform, version 2.2.1., June 2003.

[Inc03c] Sun Microsystems Inc. Virtual machine specification java card platform, version 2.2.1, June 2003.

[IR008] ISO/IEC 27005:2008 Information technology - Security techniques - Information security risk management. , 2008.

[IR09] Dániel Varró István Ráth, Gergely Varró. Change-driven model transformation. In *in Proc. of Int. Conf. on Model Driven Engineering Languages and Systems(MODELS), Denver, USA*, 2009.

[Isl09] *Software development risk management model: a goal driven approach*, New York, NY, USA, 2009. ACM.

[JM92] Brian A. Nixon: John Mylopoulos, Lawrence Chung. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Software Eng.*, 1992.

[JP05] Sheridan D. Jackson P. Clause form conversions for boolean circuits. *Lecture Notes in Computer Science 3542*, pages 183–198, 2005.

[Jür05a] J. Jürjens. *Secure Systems Development with UML*. Springer Verlag, 2005.

[Jür05b] J. Jürjens. Sound methods and effective tools for model-based security engineering with uml. *ICSE 2005,ACM*, pages 322–331, 2005.

[Jür06] J. Jürjens. Security analysis of crypto-based java programs using automated theorem provers. *ASE 2006, IEEE Computer Security*, pages 167–176, 2006.

[Jür09] Jan Jürjens. A domain-specific language for cryptographic protocols based on streams. *J. Log. Algebr. Program.*, 78(2):54–73, 2009.

[McG08] M. McGrath. *Propositions. In Edward N. Zalta, editor, The Stanford Encyclopedia of Phylosophy.* 2008.

[MEH] Mehari method. `https://www.clusif.asso.fr/fr/production/ouvrages/type.asp?id=METHODES`.

[MF09] A. Tedeschi M. Felici, V. Meduri. Secure Change - Review Story, Version 1.2, September 2009.

[NV09] Jitia C. Normand V., Felix E. A dsml for security analysis. *IST MODELPLEX project restricted deliverable 3.3.g.*, 2009.

[Rat09] Rational doors homepage, 2009. `http://www-01.ibm.com/software/awdtools/doors/`.

[Too09] UMLsec tool, 2001-09. http://mcs.open.ac.uk/jj2924/umlsectool.

[WSH+07] C. Weidenbach, R.A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: Spassversion 3.0. In *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520. Springer-Verlag, 2007.

# Appendix

In this appendix we include the following peer reviewed articles:

1). A. Bauer, J. Jürjens
*Run-time Verification of Cryptographic Protocols*
Accepted at the Journal for Computers & Security, to be published in 2010

Provides a foundation for the security monitors delivered in deliverable 4.3 (white-box approach).

2). A. Pironti, J. Jürjens
*Formally Based Black Box Monitoring of Security Protocols*
Accepted at International Symposium on Engineering Secure Software and Systems (ESSOS 2010). To be published in the Lecture Notes for Computer Science, Springer-Verlag, 2010.

Provides foundation for the security monitors delivered in deliverable 4.3 (black-box approach).

3). A. Bauer, J. Jürjens, Y. Yu
*Run-time Security Traceability for Evolving Systems* (submitted)

Provides a foundation for the security monitors delivered in deliverable 4.3. Explains how the white-box approach introduced in the earlier paper is adapted to the case of system evolution.

# Runtime Verification of Cryptographic Protocols

Andreas Bauer [a] and Jan Jürjens [b,*]

[a]*NICTA, and Australian National University*

[b]*Department of Computer Science, TU Dortmund (Germany)*

**Abstract**

There has been a significant amount of work devoted to the static verification of security protocol designs. Virtually all of these results, when applied to an actual implementation of a security protocol, rely on certain implicit assumptions on the implementation (for example, that the cryptographic checks that according to the design have to be performed by the protocol participants are carried out correctly). So far there seems to be no approach that would enforce these implicit assumptions for a given implementation of a security protocol (in particular regarding legacy implementations which have not been developed with formal verification in mind).

In this paper, we use a code assurance technique called "runtime verification" to solve this open problem. Runtime verification determines whether or not the behaviour observed during the execution of a system matches a given formal specification of a "reference behaviour". By applying runtime verification to an implementation of any of the participants of a security protocol, we can make sure during the execution of that implementation that the implicit assumptions that had to be made to ensure the security of the overall protocol will be fulfilled. The overall assurance process then proceeds in two steps: First, a design model of the security protocol in UML is verified against security properties such as secrecy of data. Second, the implicit assumptions on the protocol participants are derived from the design model, formalised in linear-time temporal logic, and the validity of these formulae at runtime is monitored using runtime verification. The aim is to increase one's confidence that statically verified properties are satisfied not only by a model of the system, but also by the actual running system itself. We demonstrate the approach at the hand of the open source implementation Jessie of the de-facto Internet security protocol standard SSL. We also briefly explain how to transfer the results to the SSL-implementation within the Java Secure Sockets Extension (JSSE) recently made open source by Sun Microsystems.

*Key words:* Security protocols, SSL, Java, temporal logic, static verification, runtime verification, security automata.

## 1. Introduction

With respect to cryptographic protocols (or short, crypto-protocols), a lot of successful work has been done to formally analyse abstract specifications of these protocols for security design weaknesses [26, 19, 40, 1, 14, 47, 7, 6]. What is still largely missing is an approach which provides assurance for *implementations* of crypto-protocols against security weaknesses. This is necessitated by the fact that so far, crypto-protocols are usually not generated automatically from formal specifications. So even where the corresponding specifications are formally verified, the implementations may still contain vulnerabilities related to the insecure use of cryptographic algorithms, or by some underlying assumptions in the abstract models that are invalidated by reality.

For example, recently a security bug was discovered in OpenSSL where several functions inside OpenSSL incorrectly checked the result from calling a signature verification function, thereby allowing a malformed signature to be (wrongly) treated as a good signature [1]. This was a serious vulnerability because it would enable an attacker in a "man in the middle" attack scenario (which is generally realistic on the Internet, which is why security protocols are needed in the first place) to present a malformed SSL/TLS signature to a client suffering from this vulnerability, who would wrongly accept it as valid.

The above vulnerability thus gives a motivating example for a scenario where checks performed at runtime can be indispensable for the secure functioning of a given crypto-protocol implementation. The goal of our work is to offer a formally based approach which is integrated with existing design models of the system (e.g., in UML), and which would allow the runtime assurance of properties supporting general security requirements (such as secrecy and authenticity), rather than being

* Corresponding author. Partially supported by the EU project and SecureChange (ICT-FET-231101).

*Email addresses:* `baueran@rsise.anu.edu.au` (Andreas Bauer), `http://jurjens.de/jan/` (Jan Jürjens).

[1] Cf. http://www.openssl.org/news/secadv_20090107.txt.

a quick fix for a particular known vulnerability. More specifically, the approach is based on the combination of:

– static security verification of UML specifications of crypto-protocols against security requirements using the UMLsec tool-support [48], which implements a Dolev-Yao style security analysis [26] [2] and
– a technique called *runtime verification* (cf. [25, 13]) which monitors assumptions that had to be made during the model-level analysis, on the implementation level at runtime, using monitors that can be automatically generated from linear-time temporal logic (LTL, [42]) formulae (cf. Section 4) using the open source LTL3TOOLS [49].

The combination of the two verification approaches thus allows us to ensure that Dolev-Yao type security properties will be enforced by the implementation at runtime.

We discuss runtime verification in detail in Section 4, but in a nutshell it works like this: we are given a formal specification of desired or undesired system behaviour. From this, a so called *monitor* is automatically generated, which is a software component that, at runtime, compares an observed behaviour of a system with the specified reference behaviour. If a mismatch is detected, the monitor signals an alarm. If at a certain point it becomes clear that the observed behaviour from then on will always satisfy the given reference behaviour, the monitor signals confirmation. Otherwise, it continues monitoring the system until one of the two situations mentioned above occurs (if ever).

Note that we are not concerned with the correct implementation of low-level crypto-algorithms (such as key generation or encryption). Instead, we would like to make sure that certain assumptions on the correct use of these algorithms within a protocol (e. g., that the validity of a signature is indeed checked at a certain point in the protocol), that have to be made when performing a static security analysis at the model level, are satisfied at runtime at the implementation level. In particular, our aim is not to verify the implementation against low-level weaknesses that cannot be detected using Dolev-Yao style verification (such as buffer overflows), for which other tools already exist that can be applied in addition to ours.

The goal of the proposed process is thus to ensure that the *implementation* of a crypto-protocol is conformant to its specification as far as the Dolev-Yao style security properties are concerned which have been verified against some security properties at the specification-level. To check conformance with respect to the security properties of the implementation against the specification, we have to ensure in particular that the implementation will only send out those (security-relevant) message parts that are permitted by the protocol specification, and only *whenever* they are permitted by the specification (e. g. after a certain signature check has been performed). Our approach allows us to enforce this conformance separately for each of the distributed participants in a protocol (such as client and server), by generating a security monitor for each of the parts that should be monitored in a distributed way. In particular, it allows the user to apply our runtime assurance approach only to *some* participants in the protocol: For example, those for which one has access to the source code, which is the level at which we will apply the approach in this paper (although in principle run-time verification can also be applied to system components available as a black box only, without access to the code).

For example, our approach can in particular enforce that the secret is only sent out on the network whenever specified by the protocol design, and only after encrypting it appropriately. As another example, the protocol specification may require that a session key may only be sent out encrypted under a public key that was received before, and *after* checking that the certificate for that public key is valid. In this paper, we will focus on examples of the latter kind that concern the kind of cryptographic checks that according to the protocol specification need to have been performed correctly before the next message in a protocol can be sent out.

It is interesting to note that our approach bears some similarities with the work of Schneider [43], who introduced *security automata* to detect violations of so called *safety properties* at runtime at the implementation level. Note, however, that safety properties form only a strict subset of those properties relevant to runtime verification of crypto-protocols, as demonstrated by our case study of the widely used SSL-protocol for Internet communication encryption. In particular, our approach to runtime verification strictly exceeds Schneider's security automata, as we will demonstrate in this paper, and allows one to generate monitors for properties that go beyond the "safety spectrum", as was necessary in our application to SSL. Also, this work seems to be the first application of runtime verification to crypto-protocol implementations.

Specifically, we explain our approach at the hand of JESSIE, an open source implementation of the *Java Secure Socket Extension* (JSSE), which includes the SSL-protocol. We first generate finite state machines as acceptors for the relevant properties defined at the specification-level, and then generate Java code from these state machines. The outcome then constitutes the executable monitor which watches over our implementation of Jessie. In addition to that, we also briefly explain how to transfer these results to Sun's own implementation of the JSSE, which was recently made open source.

Note that it is not in scope of the current paper to explain how the LTL formulas used in the run-time verification could be generated automatically from a UMLsec specification.

*Outline.* This article is a significantly extended version of our previous paper [11]. Additions in comparison to that paper include:

– the first part of the methodology, which performs the automated, static security verification on the model level, in a way that is tightly integrated with the later runtime verification part,

[2] A Dolev-Yao style security analysis of a cryptographic protocol is a security analysis of the interaction of the protocol participants with a man-in-the-middle attacker, at a relatively high level of abstraction which does not consider low-level properties (such as bit-level properties of cryptographic algorithms, or traffic analysis) but focusses on the correctness of the use of cryptography in the protocol.

– the integration of the overall approach including the two phases (static model verification and runtime implementation verification)
– the implementation of automated, formally based tool-support for both phases of the approach and detailed description of this tool-support, and
– an explanation of how the application of our approach was transferred to other libraries such as Sun's own JSSE implementation.

The remaining parts of this article are structured as follows. In the next section, we first outline related work. In Section 3, we introduce the SSL-protocol and identify relevant security properties. We then explain how protocol models in the security extension UMLsec of the Unified Modeling Language (UML) can be automatically verified against these security properties. In Section 4, we provide more details on runtime verification, and discuss how the approach we employ in our work exceeds Schneider's security automata in its formal expressiveness. Section 5 then identifies what we call *runtime security properties*, which we have derived from our specification used for static verification. We formalise these properties in the widely used temporal logic LTL [42], and discuss how to automatically derive a monitor from these formalised properties. Specifically, we provide details on the finite state machines that are generated first, and subsequently on the generated code. Moreover, we state briefly how our results developed at the hand of an application to the open source library Jessie, can be transferred to Sun's own SSL-implementation as part of the JSSE, which was recently made open source as well. Finally, in Section 6, we conclude.

## 2. Related work

### 2.1. *Monitoring, runtime verification & security automata*

Monitoring systems is a widely used means to verify that the behaviour of a running system, i.e., a stream of isolated *events*, adheres to its intended behaviour. Many examples are described in different areas much older than the emerging field of runtime verification; for instance, [33, 51] describe the use of synchronous observers to dynamically verify control software, and one may even count classic debugging as a form of monitoring (where the system is a "glass-box system" as compared to a "black-box system", where only the outside visible behaviour can be observed). However, such approaches are typically less rigorous, and less structured than runtime verification, which is formally based on temporal logics, or other forms of regular languages to specify the properties one is interested in, formally. As a scientific discipline, runtime verification was pioneered by works of Havelund and Rosu [34] (see also [35, 36]), who described how to obtain efficient monitors for specifications given in (past-time) linear-time temporal logic (LTL, [42]). Note that for different "flavours" of temporal logic, different approaches to runtime verification exist (cf. [12, 31, 41, 13, 9] for an overview).

Moreover, via the *Property Specification Language* (PSL), there exists nowadays an IEEE industry standard, IEEE1850, for temporal logic which subsumes LTL as well. There exists a considerable amount of tool support for creating and verifying PSL specifications, in particular, with respect to chip design and integrated circuits. We believe that the adoption of PSL and temporal logic by industry is also beneficial for the adoption of our approach in security-critical environments in general.

The techniques used in runtime verification also bear a resemblance with the well-known *security automata* as introduced by Schneider [43], and already mentioned in the introduction. Formally, Schneider's work is based on temporal logic as well, however, imposes restrictions on the types of specifications which can be monitored (or "enforced" to put it in Schneider's own terms). Security automata are restricted to the so-called *safety fragment* (of LTL). For a formula which is from the safety fragment, the corresponding (possibly empty) set of words satisfying the formula (the so-called language) is of such a form that any word *not* in this set can be recognised by an automaton using a prefix of that formula only. Note that this is not always possible for LTL formulae in general, e. g., there exist formulae formalising so-called liveness properties, or co-safety properties, that are not safety properties. Because the properties we consider go beyond the pure safety fragment of LTL, our approach is strictly more expressive than Schneider's original work, and this also explains why in our case study (see Section 5), we could not simply use security automata in the first place.

There also exists work by Clarkson and Schneider [24], in which the scope of properties from describing merely a set of words to sets of sets of words is extended, i.e., to so-called *hyperproperties*. Although Clarkson and Schneider have been able to describe some important security policies using hyperproperties that cannot be described using the types of properties used in this paper, it is not clear to us whether hyperproperties can be also be operationalized in the same efficient way as non-hyperproperties. We therefore do not use hyperproperties as a specification formalism for our method in this paper, although this could indeed be interesting future work.

Another application of monitoring to security was presented in [50]. The paper proposes a caller-side rewriting algorithm for the byte-code of the .NET virtual machine where security checks are inserted around calls to security-relevant methods. The work is different from ours in that it has not been applied to the security verification of cryptographic protocols, which pose specific challenges (such as the correct use of cryptographic functions and checks). In another approach, [45] proposes to use formal patterns of LTL formulae that formalise frequently reoccurring system requirements as *security monitoring patterns*. Again, this does not seem to have been applied to cryptographic protocols so far.

### 2.2. *Code-level security hardening*

Approaches for code-level security hardening based on approaches other than runtime verification exist as well, including the following examples. Again, they differ from the work

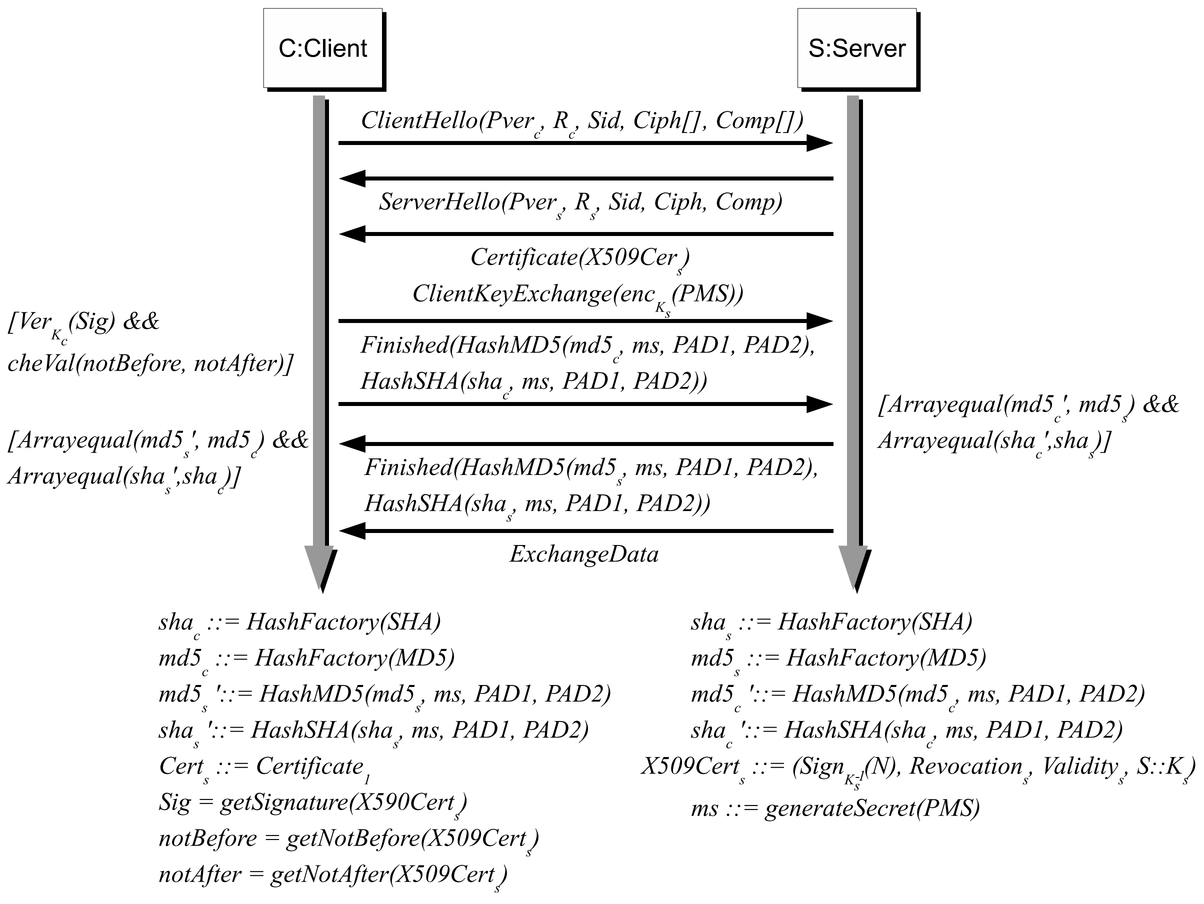

C:Client
S:Server
ClientHello(Pver_c, R_c, Sid, Ciph[], Comp[])
ServerHello(Pver_s, R_s, Sid, Ciph, Comp)
Certificate(X509Cer_s)
ClientKeyExchange(enc_{K_s}(PMS))
[Ver_{K_c}(Sig) &&
cheVal(notBefore, notAfter)]
Finished(HashMD5(md5_c, ms, PAD1, PAD2),
HashSHA(sha_c, ms, PAD1, PAD2))
[Arrayequal(md5_c', md5_s) &&
Arrayequal(sha_c',sha_s)]
[Arrayequal(md5_s', md5_c) &&
Arrayequal(sha_s',sha_c)]
Finished(HashMD5(md5_s, ms, PAD1, PAD2),
HashSHA(sha_s, ms, PAD1, PAD2))
ExchangeData
sha_c ::= HashFactory(SHA)
md5_c ::= HashFactory(MD5)
md5_s'::= HashMD5(md5_s, ms, PAD1, PAD2)
sha_s'::= HashSHA(sha_s, ms, PAD1, PAD2)
Cert_s ::= Certificate_1
Sig = getSignature(X590Cert_s)
notBefore = getNotBefore(X509Cert_s)
notAfter = getNotAfter(X509Cert_s)
sha_s ::= HashFactory(SHA)
md5_s ::= HashFactory(MD5)
md5_c'::= HashMD5(md5_c, ms, PAD1, PAD2)
sha_c'::= HashSHA(sha_c, ms, PAD1, PAD2)
X509Cert_s ::= (Sign_{K_s^{-1}}(N), Revocation_s, Validity_s, S::K_s)
ms ::= generateSecret(PMS)

Fig. 1. The cryptographic protocol implemented in SSLSocket.java

we present here in that they have not been applied to crypto protocol implementations and their specific requirements. [28] describes an approach for retrofitting legacy code with security functionality, specifically applied to authorisation policy enforcement. It can be used to identify security-sensitive locations in legacy servers in order to place reference monitor calls to mediate these locations. [53] shows how to apply aspect-oriented programming to implement security functionality such as access control and logging on the method level.

More recently, there has been a growing interest in formally verifying implementations of crypto-protocols against high-level security requirements such as secrecy with respect to Dolev-Yao attacker models (cf. [39, 32, 38, 15]). These works so far have aimed to verify implementations which were constructed with verification in mind (and in particular fulfil significant expectations on the way they are programmed) [32, 15], or deal only with simplified versions of legacy implementations [39, 38]. Our use of runtime verification is motivated by the observation that so far it has not seemed to be feasible to statically and formally verify legacy implementations of practically relevant complexity against high-level security requirements such as secrecy.

Other work on security verification on the code level includes [22, 18, 29, 8, 17, 20, 21]. Again, these approaches so far do not seem to have been applied to crypto-protocol implementations.

Other approaches to the model-based development of security-critical software include [10, 3, 54, 30, 37, 16]. These do not seem to have been used in connection with runtime verification so far.

## 3. Security properties of the SSL-protocol

SSL is the de-facto standard for securing http-connections and is therefore an interesting target for a security analysis. It may be interesting to note that early versions of SSL (before becoming a "standard" renamed as TLS in RFC 2246) had been the source of several significant security vulnerabilities in the past [2]. In this paper, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (cf. Figure 1).

As usual in the formal analysis of crypto-based software, the crypto-algorithms are viewed as abstract functions. In our application, these abstract functions represent the implementations from the *Java Cryptography Architecture* (JCA). The messages that can be created from these algorithms are then as usual formally defined as a term algebra generated from ground data, such as variables, keys, nonces, and other data using symbolic operations. These symbolic operations are the abstract versions of the crypto-algorithms.

We assume a set **Keys** of encryption keys disjointly partitioned in sets of *symmetric* and *asymmetric* keys. We fix a set **Var** of *variables* and a set **Data** of *data values* (which may include *nonces* and other secrets). The *algebra of expressions* **Exp** is the term algebra generated from the set $\mathbf{Var} \cup \mathbf{Keys} \cup \mathbf{Data}$ with the operations given in Figure 2. There, the symbols $E, E'$, and $E''$ denote terms inductively constructed in this way. Note that, as syntactic sugar, encryption $\mathsf{enc}(E, E')$ is often written more shortly as $\{E\}_{E'}$, $\mathsf{sign}(E, E')$ as $\mathcal{S}ign_{E'}(E)$, $\mathsf{conc}(E, E')$ as $E :: E'$, and $\mathsf{inv}(E)$ as $E^{-1}$. In that term algebra, one defines the equations dec(enc(E,K),inv(K))=E and ver(sign(E,inv(K)),K,E)=true for all terms E,K, and the usual laws regarding concatenation, head(), and tail().

| | |
|---|---|
| $\mathsf{enc}(E, E')$ | (encryption) |
| $\mathsf{dec}(E, E')$ | (decryption) |
| hash(E) | (hashing) |
| sign(E,E') | (signing) |
| ver(E,E',E") | (verification of signature) |
| kgen(E) | (key generation) |
| inv(E) | (inverse key) |
| conc(E,E') | (concatenation) |
| head(E) and tail(E) | (head and tail of concat.) |

Fig. 2. Abstract Cryptographic Operations

Note that the cryptographic functions in the JCA are implemented as several methods, including an object creation and possibly initialisation. Relevant for our analysis are the actual cryptographic computations performed by the digest(), sign(), verify(), generatePublic(), generatePrivate(), nextBytes(), and doFinal() methods (together with the arguments that are given beforehand, possibly using the update() method), so the others are essentially abstracted away. Note also that the key and random generation methods generatePublic(), generatePrivate(), and nextBytes() are not part of the crypto-term-algebra but are formalised implicitly in the logical formula by introducing new constants representing the keys and random values (and making use of the inv(E) operation in the case of generateKeyPair()).

In our particular protocol, setting up the connection is done by two methods: doClientHandshake() on the client side and doServerHandshake() on the server side, which are part of the SSL socket class in jessie-1.0.1/org/metastatic/jessie/provider. After some initialisations and parameter checking, both methods perform the interaction between client and server that is specified in Figure 1. Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake() rp. doServerHandshake() methods. The associated data is given in Figure 3.

| **Message name** | **Class of Message Type** | **Message Type** |
|---|---|---|
| `ClientHello` | `ClientHello` | `CLIENT_HELLO` |
| `ServerHello` | `ServerHello` | `SERVER_HELLO` |
| `Certificate*` | `Certificate` | `CERTIFICATE` |
| `ClientKeyExchange` | `ClientKeyExchange` | `CLIENT_KEY_EXCHANGE` |
| `Finished` | `Finished` | `FINISHED` |

Fig. 3. Data for the Handshake message

We must now determine for the individual data how it is implemented on the code level, to then be able to verify that this is done correctly. We explain this exemplarily for the variable randomBytes written by the method ClientHello to the message buffer. By inspecting the location at which the variable is written (the method write(randomBytes) in the class Random), we can see that the value of randomBytes is determined by the second parameter of the constructor of this class (see Figure 4).

```
Random(int gmtUnixTime, byte[] randomBytes)
{
  this.gmtUnixTime = gmtUnixTime;
  this.randomBytes = (byte[])randomBytes.clone();
}
```

Fig. 4. Constructor for random

| **in Model** | **Send: ClientHello** | **by Outputstream.write in** |
|---|---|---|
| | type.getValue() | Handshake.write |
| | (bout.size() >>> 16 & 0xFF) | Handshake.write |
| | (bout.size() >>> 8 & 0xFF) | Handshake.write |
| | (bout.size() & 0xFF) | Handshake.write |
| **Pver** $\rightarrow$ | major | ProtocolVersion.write |
| | minor | ProtocolVersion.write |
| | ((gmtUnixTime >>> 24) & 0xFF) | Random.write |
| | ((gmtUnixTime >>> 16) & 0xFF) | Random.write |
| | ((gmtUnixTime >>> 8) & 0xFF) | Random.write |
| | (gmtUnixTime & 0xFF) | Random.write |
| $R_C$ $\rightarrow$ | randomBytes | ClientHello.write |
| | sessionId.length | ClientHello.write |
| **Sid** $\rightarrow$ | sessionId | ClientHello.write |
| | ((suites.size() << 1) >>> 8 & 0xFF) | ClientHello.write |
| | ((suites.size() << 1) & 0xFF) | ClientHello.write |
| **Ciph[]** $\rightarrow$ | id[] | CipherSuite.write |
| | comp.size() | ClientHello.write |
| **Comp[]** $\rightarrow$ | comp[2] | ClientHello.write |

Fig. 5. Data in ClientHello message

Therefore the contents of the variable depends on the initialisation of the current random object and thus also on the program state. Thus we need to trace back the initialisation of the object. In the current program state, the random object was passed on to the ClientHello object by the constructor. This again was delivered at the initialisation of the Handshake object in SSLSocket. doClientHandshake() to the constructor of Handshake. Here (within doClientHandshake()), we can find the initialisation of the Random object that was passed on. The second parameter is generateSeed() of the class SecureRandom from the package java.security. This call determines the value of randomBytes in the current program state. Thus the value randomBytes is mapped to the model element $R_C$ in the message ClientHello on the model level. For this, java.security.SecureRandom.generateSeed() must be correctly implemented. To increase our confidence in this assumption of an agreement of the implementation with the model (although a full formal verification is not the goal of this paper),

$$\begin{aligned}\forall E_1, E_2.\ &\big(\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \ \Rightarrow\ \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\big)\\ &\wedge\big(\mathsf{knows}(E_1 :: E_2) \ \Rightarrow\ \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big)\\ &\wedge\big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \ \Rightarrow\ \mathsf{knows}(E_1)\big)\\ &\wedge\big(\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \ \Rightarrow\ \mathsf{knows}(E_1)\big)\end{aligned}$$

Fig. 6. Structural formulae

all data that is sent and received must be investigated. In Figure 5, the elements of the message ClientHello of the model are listed. Here it is shown which data elements of the first message communication are assigned to which elements in the doClientHandshake() method.

A crypto-protocol like the one specified in Figure 1 can then be verified at the specification level for the relevant security requirement such as secrecy and authenticity. This can be done using one of the tools available for this purpose, such as the UMLsec tool [48], which is based on the well-known Dolev-Yao adversary model for security analysis. The idea is here that an adversary can read messages sent over the network and collect them in her knowledge set. The adversary can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements can then be formalised using this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary.

We explain our translation from crypto-protocols specified as UML sequence diagrams to first-order logic formulae which can be processed by the automated theorem prover e-SETHEO [46]. The formalisation automatically derives an upper bound for the set of knowledge the adversary can gain.

The idea is to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know $E$ during the execution of the protocol. For any data value $s$ supposed to remain secret as specified in the UMLsec model, one thus has to check whether one can derive $\mathsf{knows}(s)$. The set of predicates defined to hold for a given UMLsec specification is defined as follows.

For each publicly known expression $E$, one defines $\mathsf{knows}(E)$ to hold. The fact that the adversary may enlarge her set of knowledge by constructing new expressions from the ones she knows (including the use of encryption and decryption) is captured by the formula in Figure 6.

For each object $O$ a given sequence diagram, our analysis defines a predicate $\mathsf{PRED}(O)$ which captures the behaviour of the object $O$ as relevant from the point of view of the attacker. Thus, for our purposes, a sequence diagram provides, for each object $O$, a sequence of command schemata of the form *await event e – check condition g – output event e'* represented as *connections* in the sequence diagrams. Connections are the arrows from the life-line of the source object $O$ to the life-line of a target object which are labelled with a message to be sent from the source to the target and a guard condition that has to be fulfilled.

Suppose we are given an object $O$ in the sequence diagram and a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ with:

– $\mathsf{source}(l) = O$,
– $\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and
– $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$,

where the parameters $arg_i$ of the guard and the message are variables which store the data values exchanged during the course of the protocol. Suppose that the connection $l'$ is the next connection in the sequence diagram with $\mathsf{source}(l') = O$ (i.e. sent out by the same object $O$ as the message $l$). For each such connection $l$, we define a predicate $\mathsf{PRED}(l)$ as in Figure 7. If such a connection $l'$ does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with $true$ in this formula.

$$\begin{aligned}&\mathsf{PRED}(l) =\\ &\forall exp_1, \ldots, exp_n. \big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n)\\ &\qquad \wedge\ cond(exp_1, \ldots, exp_n)\\ &\qquad\quad \Rightarrow\ \mathsf{knows}(exp(exp_1, \ldots, exp_n)\\ &\qquad\qquad \wedge\ \mathsf{PRED}(l')\big)\big)\end{aligned}$$

Fig. 7. Connection predicate

The formula formalises the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then she can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. With this formalisation, a data value $s$ is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulae defined by a protocol. This way, the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities and the message order). This means, that one will find all possible attacks, but one may also encounter "false positives", although this has not happened yet with any real examples. The advantage is that this approach is rather efficient.

For each object $O$ in the sequence diagram, this gives a predicate $\mathsf{PRED}(O) = \mathsf{PRED}(l)$ where $l$ is the first connection in the sequence diagram with $\mathsf{source}(l) = O$. The axioms in the overall first-order logic formula for a given sequence diagram are then the conjunction of the formulae representing the publicly known expressions, the formula in Figure 6, and the conjunction of the formulae $\mathsf{PRED}(O)$ for each object $O$ in the diagram. The conjecture, for which the automated theorem prover (ATP) will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value $s$ is to be kept secret, the conjecture is $\mathsf{knows}(s)$.

## 4. Runtime verification for systems monitoring

Verification on the specification level checks whether or not an abstract model satisfies predetermined security properties. Monitoring, on the other hand, checks whether or not the *implementation* of this model correctly and securely realises the specification. More generally, monitoring systems subsumes all techniques that help verify that the behaviour of a running system, i. e., a stream of isolated *events*, adheres to its intended behaviour. The approaches to monitoring range from ad-hoc to formal methods based on inference and logic (cf. Section 2). The latter are often referred to as runtime verification. In runtime verification, a reference behaviour is specified, typically in terms of a temporal logic language, such as linear time temporal logic (LTL, [42]), and then a so called monitor is generated which compares a system's observable behaviour against this specification. As such, it operates in parallel to the system and is intended not to influence its behaviour.

### 4.1. *Notions and notation*

In this section, we briefly recall some basic definitions regarding LTL and runtime verification thereof, and introduce the necessary notation. First, let $AP$ be a non-empty set of *atomic propositions*, and $\Sigma = 2^{AP}$ be an *alphabet*. Then infinite words over $\Sigma$ are elements from $\Sigma^\omega$ and are abbreviated usually as $w, w', \ldots$. Finite words over $\Sigma$ are elements from $\Sigma^*$ and are usually abbreviated as $u, u', \ldots$. As is common, we set $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$ as the set of all infinite and finite words over $\Sigma$.

We will adopt the following terminology with respect to monitoring LTL formulae. We will use the propositions in $AP$ to represent atomic system *actions*, which is what will be directly observed by the monitors introduced further below. Note that, by making use of dedicated actions that notify the monitor of changes in the system state, one can also indirectly use them to monitor whether properties of the system state hold. Thus, we can use the terms "action occurring" and "proposition holding" synonymously. An *event* will denote a set of propositions and a *word* will denote a *sequence of events* (i. e., a system's behaviour over time). The idea is that a monitor observes a stream of system events and that multiple actions can occur simultaneously.

To specify a system's behaviour (in order to define a monitor), we employ the temporal logic LTL which is defined as follows.

**Definition 1 (LTL syntax and semantics, [42])** *The set of LTL formulae over $\Sigma$, written LTL($\Sigma$), is inductively defined by the following grammar:*

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\ \boldsymbol{U}\ \varphi \mid \boldsymbol{X}\ \varphi, \quad p \in AP.$$

*The* semantics *of LTL formulae is defined inductively over its syntax via the satisfaction relation "$\models$" as follows. Let $\varphi, \varphi_1, \varphi_2 \in$ LTL($\Sigma$) be LTL formulae, $p \in AP$ an atomic proposition, $w \in \Sigma^\omega$ an infinite word, and $i \in \mathbb{N}$ a position in $w$. Let $w(i)$ denote the $i$th element in $w$ (which is a set of propositions):*

$$\begin{aligned}
&w, i \models true && \\
&w, i \models \neg\varphi && \Leftrightarrow w, i \not\models \varphi \\
&w, i \models p && \Leftrightarrow p \in w(i) \\
&w, i \models \varphi_1 \vee \varphi_2 && \Leftrightarrow w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
&w, i \models \varphi_1\ \boldsymbol{U}\ \varphi_2 && \Leftrightarrow \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
& && \qquad \forall i \leq l < k.\ w, l \models \varphi_1 \\
& && \qquad \text{("}\varphi_1 \textit{ until } \varphi_2\text{")} \\
&w, i \models \boldsymbol{X}\ \varphi && \Leftrightarrow w, i+1 \models \varphi \quad \text{("}\textit{next } \varphi\text{")}
\end{aligned}$$

*where $w, i$ denotes the $i$th position of $w$. We use $w(i)$ to denote the $i$th element in $w$ which is a set of propositions. Notice the difference between $w, i$ and $w(i)$, i.e., the former marks a position in the trace, whereas the latter denotes a set.*

When $w \models \varphi$ holds, we also say that $w$ is a *model* for the formula $\varphi$ in the logical sense of the word, , meaning that $w$ is a word which satisfies the formula. Intuitively, the statement $w, i \models \varphi$ is supposed to formalise the situation that the event sequence $w$ satisfies the formula $\varphi$ at the point when the first $i$ events in the event sequence $w$ have happened. In particular, defining $w, i \models true$ for all $w$ and $i$ means that $true$ holds at any point of any sequence of events. We also write $w \models \varphi$, if and only if $w, 0 \models \varphi$.

Further, as is common, we use **F** $\varphi$ as short notation for $true$ **U** $\varphi$ (intuitively interpreted as "eventually $\varphi$"), **G** $\varphi$ short for $\neg$**F** $\neg\varphi$ ("always $\varphi$"), and $\varphi_1$ **W** $\varphi_2$ short for **G** $\varphi_1 \vee (\varphi_1$ **U** $\varphi_2)$, which is thus a weaker version of the **U** -operator. For brevity, whenever $\Sigma$ is clear from the context or whenever a concrete alphabet is of no importance, we will use LTL instead of LTL($\Sigma$). Moreover, we make use of the standard Boolean operators $\Rightarrow, \wedge, \ldots$ that can easily be defined via the above set of operators.

**Example 1** *We give some examples of LTL specifications. Let $p \in AP$ be an action (formally represented as a proposition). Then **G F** $p$ asserts that at each point of the execution of any of the event sequences produced by the system, $p$ will afterwards eventually occur. In particular, it will occur infinitely often in any infinite system run.*

*For another example, let $\varphi_1, \varphi_2 \in$ LTL be formulae. Then the formula $\varphi_1$ **U** $\varphi_2$ states that $\varphi_1$ holds until $\varphi_2$ holds and, moreover, that $\varphi_2$ will eventually hold. On the other hand, **G** $p$ asserts that the proposition $p$ always holds on a given trace (or, depending on the interpretation of this formula, that the corresponding action occurs at each system update).*

It is worth emphasising the point that the "Until-operator" as defined above can be somewhat counterintuitive to the way that the word "until" is used in natural language; that is, the formula $a$ **U** $b$ is satisfied if and only if $b$ happens at some point in time in the future, and $a$ held until then. In the case of the weaker until, **W** , we do not demand occurrence of $b$, and the formula would be satisfied also by observing an infinitely recurring action $a$. Hence, when formalising natural language specifications in terms of LTL, the word "until" in the natural language will very often be translated to "weak until" in LTL.

## 4.2. *Monitorable languages*

An LTL formula gives rise to a set of infinite words, $L \subseteq \Sigma^\omega$, i. e., a language whose elements satisfy the formula according to the entailment relation of Definition 1. $L$ also naturally gives rise to another set, defined as $\Sigma^\omega - L$, i. e., the set of all words not contained in $L$. Let us now examine some properties of the languages (i.e. sets of words) that can be specified this way. We first introduce the notion of *bad* and *good* prefixes.

**Definition 2** *Let $L \subseteq \Sigma^\omega$ be a language of infinite words over alphabet $\Sigma$. A finite word $u \in \Sigma^*$ is called a* bad prefix *(with respect to L) iff there exists no infinite extension $w \in \Sigma^\omega$, such that $uw \in L$ (where $uw$ is the concatenation of the words $u$ and $w$). In contrast, the finite word $u$ is called a* good prefix *(with respect to L) iff for any infinite extension $w \in \Sigma^\omega$ it holds that $uw \in L$.*

We say that $u$ is a good (resp. bad) prefix of $w$ wrt. a language $L$ if $u$ is a prefix of $w$ and if $u$ is a good (resp. bad) prefix wrt. $L$ as defined above. Note that a word $w$ may have neither a good nor a bad prefix with respect to a language $L$ associated with a given formula $\varphi$. In this case, when monitoring this sequence of events, it will at no finite point in time be clear whether all extensions of the sequence of events monitored so far will satisfy the formula $\varphi$, or not.

**Definition 3** *$L$ is called a* safety language *iff all words $w \in \Sigma^\omega - L$ have a bad prefix.*

When we refer to the formula $\varphi \in LTL$ giving rise to a safety language, we will also use the term *safety property* to say that the logical models of $\varphi$ adhere to Definition 3.

**Definition 4** *$L$ is called a* co-safety language *iff all words $w \in L$ have a good prefix.*

Note that the complements of safety languages (i. e. the language $\Sigma^\omega - L$ where $L$ is a safety language) are exactly the co-safety languages. The proof of this fact is straightforward, by making use of the central observation that a word $w$ is a bad prefix with respect to the language $\Sigma^\omega - L$ exactly if it is a good prefix with respect to the language $L$, which follows directly from the definitions for good resp. bad prefixes.

Note that there exist languages which are both safety and co-safety languages, such as the empty set $\emptyset$, and the set of all infinite words, $\Sigma^\omega$. Similarly, there also exist languages that are neither safety nor co-safety languages. In fact, some such languages will play a crucial role for our application of monitoring security properties of the SSL-protocol, which cannot be specified using only safety or co-safety properties alone.

Some typical languages that are neither safety nor co-safety languages fall into a third class of languages, the liveness languages, introduced as follows.

**Definition 5** *$L$ is called a* liveness language *iff for every prefix $u \in \Sigma^*$ there exists an infinite extension $w \in \Sigma^\omega$, such that $uw \in L$.*

Note however that not all languages outside safety and co-safety are liveness languages, and that there are liveness languages that are at the same time co-safety languages. We revisit the examples given above to shed more light on the relationship between these three different classes of formal properties. Note further that there are other definitions of liveness found in the literature. We have adopted here the one given originally by [4].

**Example 2** *The formula $\boldsymbol{G}\ \boldsymbol{F}\ p$ from above (for a given $p \in AP$) asserts that at each point of a given event sequence produced by the system, $p$ will afterwards eventually occur. It does not assert a frequency of the occurrence but demands that $p$ occurs infinitely often. In particular, every observed behaviour, even if it does not contain an observation of $p$ yet, can be extended such that $\boldsymbol{G}\ \boldsymbol{F}\ p$ is satisfied. This means in particular that $\boldsymbol{G}\ \boldsymbol{F}\ p$ is not a safety property: there exist words that are not in the language $L$ defined by $\boldsymbol{G}\ \boldsymbol{F}\ p$, which do not have a bad prefix, because every of their prefixes can be extended to a word in $L$. In fact, this even holds for all words not in $L$, and indeed also for all words in $L$ as well, which proves that $\boldsymbol{G}\ \boldsymbol{F}\ p$ is a liveness property. It is not a co-safety property, because there exist words in $L$ which do not have a good prefix (in fact, none of the words in $L$ has a good prefix).*

*For given formulae $\varphi_1, \varphi_2 \in$ LTL, the formula $\varphi_1\ \boldsymbol{U}\ \varphi_2$ is a co-safety property as every word that satisfies this formula has a good prefix, i. e., can be detected via a finite word. This word is of a form such that it satisfies $\varphi_1$ finitely often, and then it indeed satisfies $\varphi_2$. The formula is not a safety property: One infinite counterexample is the word that satisfies $\varphi_1$, but not $\varphi_2$. Hence, not all counterexamples have a bad prefix as required by Definition 3. $\varphi_1\ \boldsymbol{U}\ \varphi_2$ is also not a liveness property: a word that does not satisfy $\varphi_1$ at its first position cannot be extended to a word that satisfies $\varphi_1\ \boldsymbol{U}\ \varphi_2$.*

*Finally, for a given $p \in AP$, the formula $\boldsymbol{G}\ p$ is a safety property as all counterexamples have a bad prefix. It is not a co-safety property: Intuitively speaking, one can never be sure that a given event sequence that satisfies $p$ up to a given finite point in time will always satisfy $p$. It is also not a liveness property because again, a word that does not satisfy $p$ at its first position cannot be extended to a word that satisfies $\boldsymbol{G}\ p$.*

The above three examples demonstrate that each of the classes of safety, co-safety resp. liveness properties contain properties not contained in any of the other two classes. One should also note that there exist properties that are not contained in any of the three classes.

In particular, there are properties which we believe to be security-relevant and which cannot be monitored using Schneider's security automata [43], which however can be monitored using our approach, as we will discuss in Section 4.3. For example, these can be properties of the form “an event will eventually happen” (as in the second example above). For this kind of property, the monitor cannot only just confirm that the relevant event has happened when it has happened. It can potentially also confirm at a given point in the execution of the system, that the event will not eventually happen (i.e. it will never happen from this point onwards). There are indeed security-relevant uses for such properties. One particularly important class is denial-of-service properties. Although this paper is not specifically concerned with denial of service, this kind of property does become relevant in our application here (cf. Section 3), where some properties aim to establish that a certain message in the protocol will, under certain conditions, eventually be sent out. If an adversary could detect conditions under which this is not the case, this might enable him to launch a denial of service attack against the protocol.

We will now briefly explain the approach for runtime verification used in this paper. Let $\varphi \in$ LTL be the specification we wish to monitor. The monitors generated by our approach monitor this formula by interpreting it using a 3-valued semantics that is defined with respect to a finite event sequence as follows.

**Definition 6** *Let $\varphi \in LTL$, and $u \in \Sigma^*$. Then, a monitor for $\varphi$ realises the following* entailment relation*:*

$$[u \models \varphi] := \begin{cases} \top & \text{if } u \text{ is a good prefix wrt. } L(\varphi) \\ \bot & \text{if } u \text{ is a bad prefix wrt. } L(\varphi) \\ ? & \text{otherwise.} \end{cases}$$

*The $[\cdot]$ is used to separate the 3-valued monitor semantics for $\varphi$ from the classical, 2-valued LTL semantics introduced in Definition 1.*

Note that monitoring only makes sense if there is hope that a conclusive answer (i. e., $\neq ?$) is obtainable at all. If the language does not allow for such an answer, then it makes little sense to monitor it, as all verdicts by such a monitor would yield $?$. Therefore, we define the set of languages that are *monitorable* using our approach as follows:

**Definition 7** *Let $MON$ be the set of all languages over an alphabet $\Sigma$, for which there exists some $u \in \Sigma^*$, such that $[u \models \varphi] \neq ?$.*

Thus, $MON$ is the set of all languages for which there exists a good or bad prefix. Note that not all words need to be recognisable for such a language via a good, resp. bad prefix—just some, such that the monitor can detect such a prefix should it occur. For such languages, our approach to runtime verification will produce a monitor that outputs $\top$ in case of a good prefix and $\bot$ in case of a bad prefix observed. While neither is observed, such a monitor will output $?$, meaning that the prefix seen so far does not allow a more conclusive answer as to whether the monitored language will be satisfied or violated when extending the prefix.

Notably, the generated monitors are able to detect good resp. bad prefixes of the monitored property $\varphi$, should they exist and occur, as early as possible. That is, if the observed trace $u \in \Sigma^*$ is a good resp. bad prefix for $\varphi$, then the monitor will not return $?$ as verdict. As such it detects minimal good resp. bad prefixes.

To provide a more precise comparison with Schneider's security automata in the next subsection, we also define the following notion of *syntactically monitorable* properties.

**Definition 8** *Let $MON_{syn}$ be the set of all languages that correspond to properties defined as conjunctions or disjunctions of safety and co-safety properties (i.e. properties $\phi_1 \wedge \ldots \wedge \phi_n$ or $\phi_1 \vee \ldots \vee \phi_n$ where the $\phi_i$ are safety or co-safety properties).*
Note that $MON_{syn}$ includes all safety and co-safety properties because $true$ is both a safety and a co-safety property, and recall that co-safety properties are exactly the negations of safety properties.

**Proposition 9** *We have $MON_{syn} \subseteq MON$.*

**PROOF.** [44, Th. 3.1] shows that the class of safety properties is closed under combination using $\vee$, and $\wedge$ (and by duality the same holds for co-safety properties). It is therefore sufficient to consider the case $\phi \wedge \psi$ where $\phi$ is a safety property and $\psi$ is a co-safety property (the case for $\vee$ again works by duality). By definition of safety properties, we know that all words that do not satisfy $\phi$ have a bad prefix. If there is no word that does not satisfy $\phi$, then $\phi \wedge \psi = \psi$. Otherwise, there is a word with a bad prefix for $\phi$, which is also a bad prefix for $\phi \wedge \psi$. □

## 4.3. *Expressiveness in comparison to security automata*

Schneider's security automata [43] which he proposes for the "enforcement" of system properties, accept exactly elements of the safety fragment of $\omega$-regular languages. $\omega$-regular languages, in contrast to regular languages (i.e., the languages which can be defined via commonly used regular expressions or finite automata), are those which can be defined via infinite automata, such as nondeterministic Büchi automata (cf. [23]). Moreover, the languages definable in LTL are $\omega$-regular, although not every $\omega$-regular language is definable in LTL. From the discussion in the previous subsection it then follows that, using complementation of a given system property (specified in LTL), security automata could also be used to "enforce" co-safety properties. That is, if $\varphi \in LTL$ is a safety property corresponding to desired system behaviour, then its negation, $\overline{\varphi}$, is a co-safety property corresponding to undesired system behaviour. The result of the automaton/monitor would then have to be inverted, accordingly.

As we have seen in the previous subsection, however, sometimes we may want to specify properties which may not fall into either category of safety or co-safety languages (cf. also our application in Section 5.1). Notably as such they also exceed the expressiveness of security automata, although we have managed to successfully create monitors for those. Let us therefore formally establish in what sense our approach to runtime verification exceeds the expressiveness of security automata. Schneider abbreviates the set of languages accepted by security automata as $EM$, which as pointed out above, coincides with the safety fragment of $\omega$-languages. For $EM$ he observes:

**Proposition 10 ([43])** *If the set of executions for a security policy $\varphi$ is not a safety language, then an enforcement mechanism from $EM$ does not exist for $\varphi$.*

Let us now relate this class $MON$ of properties which are monitorable in our approach with the class $EM$ of properties "enforceable" by security automata:

**Theorem 11** *It holds that $EM \subset MON_{syn} \subseteq MON$.*

**PROOF.** The statement $EM \subseteq MON_{syn}$ follows from the facts that $EM$ is contained in the class of safety properties (which was shown in Proposition 10 above), and that $MON_{syn}$ includes the class of safety properties, as noted above. (Note that the statement is still true if we use the "trick" explained above which would allow us to enforce co-safety properties using $EM$, since $MON_{syn}$ also includes the class of co-safety properties.)

To show that $MON_{syn}$ is strictly larger than $EM$, it suffices to exhibit a language in $MON_{syn}$ which is demonstrably not contained in $EM$. Consider the property $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$ where $a$ and $b$ are distinct atomic propositions. The corresponding language is in $MON_{syn}$ since $\neg\mathbf{F}\ a$ is a safety property and $\mathbf{F}\ b$ a co-safety property. $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$ is not in $EM$ because it is not a safety property (nor a co-safety property): Infinite event sequences containing $a$ but not $b$ do not satisfy $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$, but do not have a bad prefix (and infinite event sequences containing neither $a$ nor $b$ satisfy $\neg\mathbf{F}\ a \vee \mathbf{F}\ b$, but do not have a good prefix). $\square$

Note that Property 2 of Section 5 is an example which is contained in $MON$ but not in $EM$: it is neither safety, nor co-safety as there exists a model without good prefix as well as a counterexample without bad prefix for it (as explained in detail there). On the other hand, using our approach, there exists a monitor for it, which is depicted in Figure 10. Therefore, this property is indeed in $MON$.

## 5. Monitoring runtime security properties of SSL

In order to monitor properties of the SSL-protocol, we first need to determine how important elements at the model level are implemented at the implementation level. Basically, this can be done in the following three steps:

– Step 1: Identification of the data transmitted in the sending and receiving procedures at the implementation level.
– Step 2: Interpretation of the data that is transferred and comparison with the sequence diagram.
– Step 3: Identification and analysis of the cryptographic guards at the implementation level.

In Step 1, the communication at the implementation level is examined and it is determined how the data that is sent and received can be identified in the source code. Afterwards, in Step 2, a meaning is assigned to this data. The interpreted data elements of the individual messages are then compared with the appropriate elements in the model. In Step 3, it is described how one can identify the guards from the model in the source code.

To this aim, we first identify where in the implementation messages are received and sent out, and which messages exactly. In doing so, we exploit the fact that in most implementations of cryptographic protocols, message communication is implemented in a standardised way (which can be used to recognise exactly where messages are sent and received). The common implementation of sending and receiving messages in cryptographic protocols is through *message buffers*, by writing the data into type-free streams (i.e., ordered byte sequences), which are sent across the communication link, and which can be read at the receiving end. The receiver is responsible for reading out the messages from the buffer in the correct order and storing it into variables of the appropriate types. Accordingly, in case of the Java implementation JESSIE of the SSL-protocol, this is done by using the methods write() from the class java.io.OutputStream to write the data to be sent into the buffer and the method read() from the class java.io.InputStream to read out the received data from the buffer. Also note that the messages themselves are usually represented by message classes that offer custom write and read methods, and in which the write and read methods from the java.io are called, subsequently.

Moreover, according to the information that is contained in a sequence diagram specification of a cryptographic protocol, the monitor which is generated for performing runtime verification needs to keep track of the following information:

– *Which data is sent out?*, and
– *Which data is received?*

This, in turn, depends on where the monitor will be placed in the actual and possibly distributed implementation; that is, certain properties (of the SSL-protocol) are to be monitored at the client-side while others are to be monitored at the server-side. While our approach is not restricted to either point of view, server or client, the users have to make this decision based on the properties they would like to monitor, the available system resources, accessibility, and so forth.

The monitors, once in place, will then generally enforce that the relevant part of the implementation conforms to the specification in the following sense:

– The code should only send out messages that are specified to be sent out according to the specification and in the correct order, and
– these messages should only be sent out if the conditions that have to be checked first according to the specification are met.

In the next section, we give some example properties which highlight these two points wrt. the SSL-protocol. Note that it may depend on a given application and on the requirements to be monitored whether or not it may be possible to find monitorable properties whose violation at runtime indeed prevents the leaking of data (i.e. the detection occurs before the undesirable situation occurs). While this should be generally possible in many real-world applications where runtime monitoring is applicable, there may also be undesirable events that cannot be detected prior to their occurrence, i.e. there does not exist a suitable property that could be specified by the user or monitored by the system, such that its violation would help anticipate or prevent the undesirable event from happening.

### 5.1. *Example runtime security properties*

In this section, we consider the examples listed below for properties that should be enforced using runtime verification in the case of the SSL-protocol specified in Figure 1 in more detail. Each of these properties enforces on the implementation level the implicit assumption that had to be made for the model level security analysis that any of the messages in the protocol specified in Figure 1 is only sent out after the relevant protocol participant has satisfactorily performed the required checks on the message that was received just before.

(i) ClientKeyExchange$(enc_K, (PMS))$ is not sent by the client until it has received the Certificate$(X509Cer_s)$

message from the server, has performed the validity check for the certificate as specified in Figure 1, and this check turned out to be positive.

(ii) $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ is not sent by the server to the client before the MD5 hash received from the client in the message $\mathsf{Finished}(HashMD5(md5_c, ms, PAD1, PAD2))$ has been checked to be equal to the MD5 created by the server, and correspondingly for the SHA hash, but will send it out eventually after that has been established.

(iii) The client will not send any transport data to the server before the MD5 hash received from the server in the $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash.

Below we consider each of the three properties in detail.

*Property 1.* Step 1 and 2 of the above procedure, yield the two abstract messages $\mathsf{ClientKeyExchange}$ and $\mathsf{Certificate}$. Moreover, once we have identified in the source code where these messages are sent respectively received and evaluated, we can add at this point custom code that sets or unsets two "flags": the flag $\mathsf{ClientKeyExchange}(enc_K, (PMS))$ is set by the code if and only if the message $\mathsf{ClientKeyExchange}$ is sent by the client, and the flag $\mathsf{Certificate}(X509Cer_S)$ is set if and only if the certificate was received and positively checked. Otherwise, both flags are unset. Coming back to our formal model of LTL runtime verification (see Section 4), this yields the following set of atomic propositions:

$$AP = \{\mathsf{ClientKeyExchange}(enc_K, (PMS)), \\ \mathsf{Certificate}(X509Cer_S)\},$$

whose names correlate with the ones displayed in Figure 1. Notice that LTL as introduced in Section 4 does not cater for parameters, and parameters in an action's name are therefore not a semantic concept.

Based on $AP$ we can now formalise the required property in LTL as follows, using the "weak until" operator, which in particular allows for the fact that if the certificate is never received, then the formula is satisfied if in turn the message $\mathsf{ClientKeyExchange}(enc_K, (PMS))$ is never sent.

$$\varphi_1 = \neg\mathsf{ClientKeyExchange}(enc_K, (PMS)) \\ \mathbf{W}\ \mathsf{Certificate}(X509Cer_S).$$

This meets our intuitive interpretation of the "until" in the natural language requirement because if, for example, a man-in-the-middle attacker deletes any certificate message sent by the server, we cannot possibly demand that $\mathsf{ClientKeyExchange}(enc_K, (PMS))$ should be eventually sent by the client.

We can then use the approach to runtime verification described in detail in [13] and realised in the open source tool LTL3TOOLS to automatically generate a finite state machine for monitoring this formula. From this finite state machine, we subsequently generate Java code, i.e., the executable monitor (for details of this process, see Section 5.2), which watches over the protocol implementation while our client participates in an SSL-session. The executable monitor signals the value $\top$ ("property satisfied") once the certificate was received and checked, $\perp$ ("property violated") if the client sends the key without successful check, and it will signal the value ? ("inconclusive") as long as neither of the two conditions holds. Recall that the stream of events that is processed by the monitor consists of elements from $2^{AP}$ (i.e., the powerset of all possible system actions). That is, at each point in time, the monitor keeps track of *both* events: the sending of $\mathsf{ClientKeyExchange}(enc_K, (PMS))$ and the receiving of $\mathsf{Certificate}(X509Cer_S)$. Hence, as long as none of the events is observed, the monitor basically processes the empty event. Moreover, $\varphi_1$ is a classical safety property, because all counterexamples have a finite bad prefix, i.e., can be recognised as such after finitely many observations. As such, this property is also recognisable by a security automaton.

*Property 2.* In order to monitor the second requirement as given above, we now take the point of view of the server rather than that of the client. Let

$$AP = \{\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)), \\ \mathsf{Arrayequal}(md5_s, md5_c)\}$$

Notice, how we have not added $\mathsf{Arrayequal}(sha_s, sha_c)$ to $AP$. The reason for this is that we will be actually creating two monitors both operating over their own respective alphabets. We can now formalise the corresponding LTL property with respect to $AP$ as we did before:

$$\varphi_2 = (\neg\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)) \\ \mathbf{W}\ \mathsf{Arrayequal}(md5_s, md5_c)) \\ \wedge(\mathbf{F}\ \mathsf{Arrayequal}(md5_s, md5_c) \\ \Rightarrow \mathbf{F}\ \mathsf{Finished}(HashMD5(md5_s, ms, \ldots))).$$

To monitor the analogous statement for the SHA rather than the MD5, we define an additional formula, $\varphi_2'$, where all occurrences of the proposition $\mathsf{Arrayequal}(md5_s, md5_c)$ are replaced by the proposition $\mathsf{Arrayequal}(sha_s, sha_c)$, respectively. Neither of the two formulae are actually safety or co-safety properties, although there exist finite traces which violate, respectively, satisfy the formula: For instance, consider a trace $u = \emptyset; \{\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))\}$, which violates the first part of our conjunction since this implies $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)) = \top$, but $\mathsf{Arrayequal}(md5_s, md5_c)) = \perp$. On the other hand, the trace $v = \emptyset; \{\mathsf{Arrayequal}(md5_s, md5_c)\}$ is a model for $\varphi_2$, since our second observation in $v$ shows that the MD5 checksum was successfully compared, and until then, $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ did not hold. Note that $\emptyset$ means that all propositions in AP are interpreted as $\perp$ (i.e. that none of the monitored events occurs at that point in time), and we use the symbol ";" to denote the concatenation of events. However, as pointed out above, $\varphi_2$ is not a co-safety property since, besides the trace $v$ which has a good prefix, there also exists the infinite model $v' = \emptyset;\emptyset;\ldots$ without a good prefix, i.e., $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ never holds. Recall, the concept of co-safety asserts that *all* models possess a finite good prefix (Definition 4). Moreover, it is not a safety property since there exists the infinite counterexample $u' = \{\mathsf{Arrayequal}(md5_s, md5_c))\};\emptyset;\ \emptyset;\ldots$ without a bad prefix. Recall, the concept of safety asserts that *all* counterexamples can be recognised via a finite bad prefix (Definition 3), and $u'$, if infinitely extended with $\emptyset$ is, indeed, a counterexample, although it cannot be recognised as such after finitely many observations. Note that a monitor for this property is given in Section 5.2, although there exists no corresponding Schneider security automaton for it.

*Property 3*. Lastly, the above requirement 3. can be formalised as follows:

$$\varphi_3 = \neg\mathsf{Data}\ \mathbf{W}\ \mathsf{Arrayequal}(md5_s, md5_c),$$

where $AP$ is similar as in the previous example, but contains a proposition indicating the sending of data, $\mathsf{Data}$. Here we have a real safety property again since all traces of violating behaviour for $\varphi_3$ are recognisable after finitely many observations. It is not co-safety since the infinite trace $w = \emptyset;\emptyset;\ldots$ satisfies $\varphi_3$, which would be the case if an intruder has intercepted and kept the $\mathsf{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ message, such that it is never received at the server-side. Again, as in the previous example, we create a second monitor to check the outcome of the SHA-comparison.

Monitoring a property like $\mathbf{G}\,p$ means that the corresponding monitor would output ? as long as no violation occurred, but never $\top$, since all models are infinite traces without good prefixes. However, $\varphi_2$ and $\varphi_3$ are such that they do have models with good prefixes, hence the corresponding monitor can output all three values of $\{\top, \bot, ?\}$, depending on the observed system behaviour. The same holds for $\varphi_1$, which is actually a safety property. Note that, had we used the "strong until" operator $\mathbf{U}$ in $\varphi_1$ and $\varphi_2$, then both properties would be classical co-safety properties. Co-safety properties would be monitorable using security automata, however, only via the detour of complementing them and checking that the complements do *not* hold (i.e., one has to "invert" the semantics). Using LTL3TOOLS, this detour is not necessary. We can directly generate a monitor for $\varphi_1$ and $\varphi_2$ (also for the variants using the "strong until" operator).

## 5.2. *Implementation*

Once we have formalised the natural language requirements in terms of LTL formulae as above, we can then use our LTL3TOOLS [49] to automatically generate finite state machines (FSM) from which we derive the actual (Java) monitor code. The FSMs obtained from LTL3TOOLS are of type Moore, which means that, in each state that is reached, they output a symbol (i.e., ?, $\top$ (`TOP`), or $\bot$ (`BOT`)). States are changed as new system actions become visible to the monitor. The FSM generated for the runtime security property $\varphi_1$ is given in Figure 9. The initial state is $(0, 0)$ whose output is ?. If event $\{cert\}$ occurs, short for $\{\mathsf{Certificate}(X509Cer_S)\}$, then the monitor takes a transition into state $(1, -1)$ and outputs $\top$ to indicate that the property is satisfied. On the other hand, if neither $cert$ nor $cke$, short for $\mathsf{ClientKeyExchange}(enc_K, (PMS))$, occurs, then the automaton remains in $(0, 0)$ and outputs ?, indicating that so far $\varphi_1$ has not been violated, but also not been satisfied. A violation would be the reaching of $(-1, 1)$, if event $\{cke\}$ occurs (before $cert$), such that the monitor would output $\bot$. Generating code from this state machine is a straightforward exercise, which we only outline by giving a code example of the generated code (see Figure 8). This code in turn is embedded into an event loop, such that new events can enforce the triggering of transitions, and changing of states. The current state information is stored inside the variable $\mathsf{m\_state}$, whereas the events are abbreviated by definitions.

```
class MonitorPhi1
{
   ...

   void process_event(Event e)
   {
      if ( m_state.equals("(0,␣0)") )
      {
         System.err.println("?");

         if ( type_ID(e) == EMPTY )
         {
            m_state = "(0,␣0)";
            return;
         }
         else if ( type_ID(e) == CERT )
         {
            m_state = "(1,␣-1)";
            return;
         }
         ...  // process remaining events
      }
      else if ( m_state.equals("(1,␣-1)") )
      {
         System.err.println("TOP");
         ...  // process remaining states
      }
   }
}
```

Fig. 8. Fragment of the generated monitor code for $\varphi_1$

In order to determine which event has occurred, we use a set of program flags (i.e., Boolean variables) which are initially set to *false*. We set these to *true* as soon as an identified action has taken place. The combination of flags then determines the current event to process by the monitor. Hence, once the monitor code is generated, the only code that needs to be added to the main application is the code to set the flags, and the code to communicate them in terms of events to the monitor class. According to our experiences, an LTL property of the size that usually occurs in these kinds of application results into a monitor of 150 LOC in Java, including communication code between application and monitor.

The remaining two properties, $\varphi_2$ and $\varphi_3$ are implemented in the same manner. For brevity, we do not discuss their internals in detail, but give the automatically generated FSMs in Figures 10 and 11, respectively. Note that these monitors are good examples to demonstrate that it is, indeed, feasible to monitor properties beyond the "safety spectrum" that Schneider's security automata capture: All FSMs cover all the three different truth values, and not all of them are safety (or co-safety) formulae (as discussed above in detail).

Notice that other comprehensive Java programs but Jessie are often developed in parallel with an event logging library such as log4j [3], which can then directly be used for capturing all relevant system events that the monitors require. If no such library is used, as in the case of JESSIE, then the set of relevant events needs to be identified first, and all occurrences in the code instrumented accordingly.

### 5.3. *Application to Sun's JSSE implementation*

Let us also briefly explain how one can apply the runtime verification approach which we discussed above in terms of the open source implementation Jessie of the JSSE to Sun's very own implementation as a library in the standard JDK (since version 1.4), which was recently made open source. The source code of this library (after version 1.6) can be checked out from the OpenJDK repository [4]. To perform the runtime verification as explained above on this implementation, one only needs to modify the mappings between the specification elements and their implementations that provide the traceability of the model to the implementation level. For example, in the Jessie implementation, the doHandshake protocol is mainly implemented in the class SSLSocket of the Jessie (v. 1.0.1) library, whereas in the library implementation in the OpenJDK 1.6 (hereafter called JSSE 1.6), the protocol is mainly implemented in the class sun.security.ssl.HandshakeMessage. Nevertheless, the naming of the symbols can be traced to the implementation. Table 1 lists some mapping from the symbols in Figure 1 to their naming in the JSSE 1.6 library.

Table 1
Traceability mappings in JSSE 1.6

| Symbols | JSSE 1.6 |
|---|---|
| 1. $C$ | HandshakeMessage.ClientHello |
| 2. $S$ | HandshakeMessage.ServerHello |
| 3. $P_{\text{ver}}$ | protocolVersion |
| 4. $R_C$ | clnt_random |
| $R_S$ | svr_random |
| 5. $S_{\text{id}}$ | sessionId |
| 6. Ciph[ ] | cipherSuites |
| 7. Comp[ ] | compression_methods |
| 8. Veri | CertificateVerify.verify() |
| 9. $D_{\text{notBefore}}$ | cert.getNotBefore() |
| $D_{\text{notAfter}}$ | cert.getNotAfter() |

## 6. Conclusions

In this article, we successfully address the open problem of how to enforce that certain assumptions implicit in the Dolev-Yao style verification of crypto-protocol specifications are satisfied on the implementation level. Towards this goal, runtime verification allows us to monitor even complex, history-dependent specifications as they arise for security protocols. We have seen, in particular, that some crucial runtime correctness properties of our SSL-implementation could not be monitored using prior formal approaches to monitoring security-critical systems, since they exceed the expressiveness of the safety fragment (of LTL), which is the fragment monitorable by Schneider's security automata. As an example for a security weakness that can be found using our approach, the approach helped us detecting a missing signature check in the Jessie implementation of the SSL protocol.

However, as can also be seen by some of our properties, it is often difficult to decide whether or not a formula is a safety property, whether it is co-safety, or neither—even for the trained eye. In fact, it is known that given some formula $\varphi \in$ LTL, deciding whether $\mathcal{L}(\varphi)$ is safety (co-safety) is a PSPACE-complete problem [5, 44]. Hence, there are theoretical limitations to our approach, whenever it is not clear to the designer of a system whether or not some imposed security property is monitorable at all. However, due to the completeness result, we cannot hope to provide a more efficient or convenient way for performing this check. Also, from the practical point of view, the user can simply attempt to generate the monitor using the LTL3TOOLS. If that should fail, then the user has the option to syntactically simplify the input formula, or perhaps to generate multiple smaller monitors for the subformulae of the original input that will later operate in parallel.

Once the monitors are generated, then the resulting overhead from using monitors is minimal, in the sense that one can show that other approaches for generating monitors from LTL formulae cannot be any more efficient: For example, the particular approach described in [13] and realised by the LTL3TOOLS [49] creates monitors with (near to) *optimal* space complexity with respect to the property to be monitored. It should be pointed out, however, that the intermediate steps in generating a monitor involve a double exponential "blow up" in the length of the specification, but this does not necessarily affect runtime efficiency, when the monitors are minimised as a final step. On the other hand, sophisticated event logging libraries such as log4j can create a considerable space and time overhead, which is another reason why we have chosen to instrument our application manually in a more lightweight fashion by setting or unsetting a number of flags.

Although comprehensive performance tests in terms of the exact overhead of the instrumentation are yet to be carried

[3] http://logging.apache.org/
[4] https://openjdk.dev.java.net/svn/openjdk/jdk/trunk/j2se/src/share/classes/sun/security/ssl

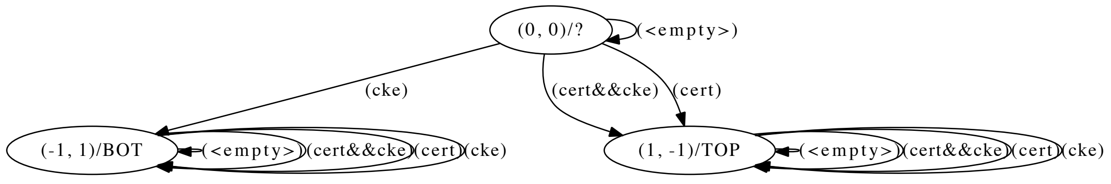

(0, 0)/?
(<empty>)
(cke)
(cert&&cke)
(cert)
(-1, 1)/BOT
(<empty>)(cert&&cke)(cert)(cke)
(1, -1)/TOP
(<empty>)(cert&&cke)(cert)(cke)

Fig. 9. Automatically generated FSM for $\varphi_1 = \neg \mathsf{cke}\ \mathbf{W}\ \mathsf{cert}$

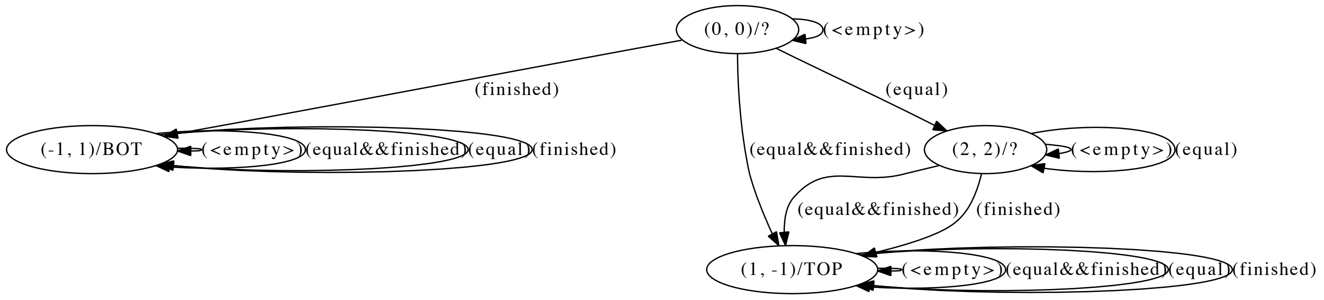


Fig. 10. Automatically generated FSM for $\varphi_2 = (\neg \mathsf{finished}\ \mathbf{W}\ \mathsf{equal} \wedge (\mathbf{F}\ \mathsf{equal} \Rightarrow \mathbf{F}\ \mathsf{finished}))$

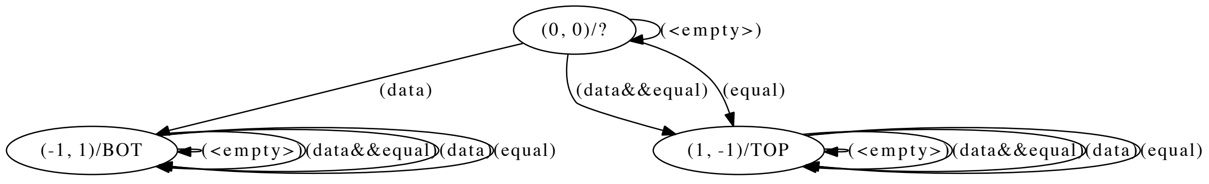

(0, 0)/?
(<empty>)
(data)
(data&&equal)
(equal)
(-1, 1)/BOT
(<empty>)(data&&equal)(data)(equal)
(1, -1)/TOP
(<empty>)(data&&equal)(data)(equal)

Fig. 11. Automatically generated FSM for $\varphi_3 = \mathsf{data}\ \mathbf{W}\ \mathsf{equal}$

out, we have collected some preliminary performance indicators using the comprehensive collection of LTL formulae [5] that is underlying Dwyer et al.'s commonly used *Software Specification Patterns* (cf. [27]). This collection of formulae, taken from real-world applications that employ formal specification using temporal logic, consists of a total of 447 formal specifications, of which 97 were given in terms of LTL. Although the 97 formulae were between 2 and 44 token in length, none of the generated (and minimised) monitors consisted of more than 6 states, which is at least an indication that the expected performance overhead for practical specifications, such as represented by Dwyer et al.'s formulae, is manageable, or even negligible as it was the case with our example properties.

Moreover, for the purposes of this paper, we could only demonstrate how to monitor a selected set of properties. Many more properties would be important to monitor, such as, for example, monitoring that the data stream in an SSL transaction is indeed encrypted.

*Acknowledgements* We thank the reviewers for constructive feedback which led to a significant improvement of this paper.

## References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security (CCS 1997)*, pages 36–47, 1997.

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[3] M. Alam, M. Hafner, and R. Breu. Model-driven security engineering for trust management in SECTET. *Journal of Software*, 2(1), Feb. 2007.

[4] B. Alpern and F. B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[5] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[6] S. Andova, C. J. F. Cremers, K. Gjøsteen, S. Mauw, S. F. Mjølsnes, and S. Radomirovic. A framework for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008.

[7] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer-Verlag, 2005.

[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Pri-*

[5] cf. http://patterns.projects.cis.ksu.edu/documentation/specifications/ALL.raw

*vacy*, pages 387–401. IEEE Computer Society, 2008.

[9] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, editors, *ICSOC*, pages 193–202. ACM, 2004.

[10] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[11] A. Bauer and J. Jürjens. Security protocols, properties, and their monitoring. In B. D. Win, S.-W. Lee, and M. Monga, editors, *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems (SESS)*, pages 33–40, New York, NY, May 2008. ACM Press.

[12] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*. Accepted for publication.

[13] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4337 of *LNCS*. Springer, Dec. 2006.

[14] G. Bella, L. C. Paulson, and F. Massacci. The verification of an industrial payment protocol: the set purchase phase. In V. Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 12–20. ACM, 2002.

[15] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152. IEEE Computer Society, 2006.

[16] S. Braghin, A. Coen-Porisini, P. Colombo, S. Sicari, and A. Trombetta. Introducing privacy in a hospital information system. In Win et al. [52], pages 9–16.

[17] B. Braun. Save: static analysis on versioning entities. In Win et al. [52], pages 25–32.

[18] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In R. Büschkes and P. Laskov, editors, *DIMVA*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2006.

[19] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society, Series A*, 426(1871):233–271, December 1989. Also appeared as SRC Research Report 39 and, in a shortened form, in *ACM Transactions on Computer Systems 8*, 1:18–36 (February 1990).

[20] L. Cavallaro, A. Lanzi, L. Mayer, and M. Monga. Lisabeth: automated content-based signature generator for zero-day polymorphic worms. In Win et al. [52], pages 41–48.

[21] I. Chowdhury, B. Chan, and M. Zulkernine. Security metrics for source code structures. In Win et al. [52], pages 57–64.

[22] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46. IEEE Computer Society, 2005.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[24] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, Washington, DC, USA, 2008. IEEE Computer Society.

[25] S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[26] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[27] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. 2nd WS. on Formal methods in software practice*, pages 7–15. ACM, 1998.

[28] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE Symposium on Security and Privacy*, pages 214–229. IEEE Computer Society, 2006.

[29] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *ICSE*, pages 458–467. IEEE Computer Society, 2007.

[30] M. Gegick and L. Williams. On the design of more secure software-intensive systems by use of attack patterns. *Information & Software Technology*, 49(4):381–397, 2007.

[31] M. Geilen. On the construction of monitors for temporal logic properties. *ENTCS*, 55(2), 2001.

[32] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[33] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[34] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[35] K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[36] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Journal on Software Tools for Technology Transfer*, 2004.

[37] V. Horvath and T. Dörges. From security patterns to implementation using petri nets. In Win et al. [52], pages 17–24.

[38] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In S. Easterbrook and S. Uchitel, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM, 2006.

[39] J. Jürjens and M. Yampolskiy. Code security analysis with assertions. In D. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages

392–395. ACM, 2005.

[40] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[41] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *CCS*, pages 260–269. ACM, 2005.

[42] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[43] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[44] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[45] G. Spanoudakis, C. Kloukinas, and K. Androutsopoulos. Towards security monitoring patterns. In *SAC*, pages 1518–1525. ACM, 2007.

[46] G. Stenz and A. Wolf. E-SETHEO: An automated$^3$ theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 2000.

[47] S. G. Stubblebine and R. N. Wright. An authentication logic with formal semantics supporting synchronization, revocation, and recency. *IEEE Trans. Software Eng.*, 28(3):256–285, 2002.

[48] UMLsec tool, 2001-08. http://mcs.open.ac.uk/jj2924/umlsectool.

[49] LTL$_3$ Tools, 2008. http://ltl3tools.SourceForge.Net/.

[50] D. Vanoverberghe and F. Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In G. Barthe and F. S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 240–258. Springer, 2008.

[51] M. Westhead and S. Nadjm-Tehrani. Verification of embedded systems using synchronous observers. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 405–419, London, UK, 1996. Springer-Verlag.

[52] B. D. Win, S.-W. Lee, and M. Monga, editors. *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, SESS 2008, Leipzig, Germany, May 17-18, 2008*. ACM, 2008.

[53] B. D. Win, B. Vanhaute, and B. D. Decker. How aspect-oriented programming can help to build secure software. *Informatica (Slovenia)*, 26(2), 2002.

[54] L. Yu, R. B. France, I. Ray, and K. Lano. A light-weight static approach to analyzing UML behavioral properties. In *ICECCS*, pages 56–63. IEEE Computer Society, 2007.

# Formally-Based Black-Box Monitoring of Security Protocols[*]

Alfredo Pironti[1] and Jan Jürjens[2]

[1] Politecnico di Torino
Turin, Italy
http://alfredo.pironti.eu/research
[2] TU Dortmund and Fraunhofer ISST
Dortmund, Germany
http://jurjens.de/jan

**Abstract.** In the challenge of ensuring the correct behaviour of legacy implementations of security protocols, a formally-based approach is presented to design and implement monitors that stop insecure protocol runs executed by such legacy implementations, without the need of their source code. We validate the approach at a case study about monitoring several SSL legacy implementations. Recently, a security bug has been found in the widely deployed OpenSSL client; our case study shows that our monitor correctly stops the protocol runs otherwise allowed by the faulty OpenSSL client. Moreover, our monitoring approach allowed us to detect a new flaw in another open source SSL client implementation.

## 1 Introduction

Despite being very concise, cryptographic protocols are quite difficult to get right, because of the concurrent nature of the distributed environment and the presence of an active, non-deterministic attacker. Increasing the confidence in the correctness of security protocol implementations is thus important for the dependability of software systems. In general exhaustive testing is infeasible, and for a motivated attacker one remaining vulnerability may be enough to successfully attack a system. In this paper, we focus in particular on assessing the correctness of legacy implementations, rather than on the development of correct new implementations. Indeed, it is often the case in practice that a legacy implementation is already in use which cannot be substituted by a new one: for example, when the legacy implementation is strictly coupled with the rest of the information system, making a switch very costly.

In this context, our proposed approach is based on black-box monitoring of legacy security protocols implementations. Using the Dolev-Yao [6] model, we assume cryptographic functions to be correct, and concentrate on their usage within the cryptographic protocols. Moreover, we concentrate on implementations of security protocol actors, rather than on the high level specifications of

[*] This research was partially supported by the EU project SecureChange (ICT-FET-231101).

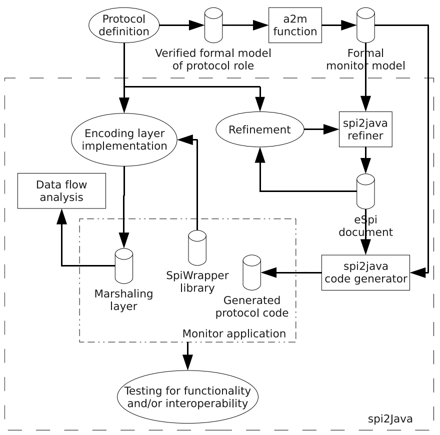


Fig. 1: Monitor design and development methodology.

such security protocols. That is, we assume that a given protocol specification is secure (which can be proven using existing tools); instead, by monitoring it, we want to asses that a given implementation of one protocol's role is correct with respect to its specification, and it is resilient to Dolev-Yao attacks.

The overall methodology is depicted in figure 1. Given the protocol definition, a specification for one agent is manually derived. By using the "agent to monitor" ($a2m$) function introduced in this paper, a monitor specification for that protocol role is automatically generated. Then the monitor implementation is obtained by using the model driven development framework called spi2java [13], represented by the dashed box in the figure. The spi2java internals will be discussed later on in the paper. The monitor application is finally ran together with the monitored protocol role implementation (not shown in the picture).

A monitor implementation differs from a fresh implementation of a security protocol, because it does not execute protocol sessions on behalf of its users. The monitor instead observes protocol sessions started by the legacy implementations, in order to recognize and stop incorrect sessions, in circumstances where the legacy implementations cannot be replaced.

For performance trade-offs, monitoring can be performed either "online" or "offline". In the first case, all messages are first checked by the monitor, and then forwarded to the intended recipient only if they are safe. In the second case, all messages exchanged by the monitored application are logged, and then fed to the monitor for later inspection. The online paradigm prevents a security property to be violated, because protocol executions are stopped as soon as an unexpected message is detected by the monitor, before it reaches the intended recipient. However, online monitoring may introduce some latency. The offline paradigm does not introduce any latency and is still useful to recognize compromised protocol sessions later, which can limit the damage of an attack. For example, if a credit card number is stolen due to an e-commerce protocol attack, and if offline monitoring is run overnight, one can discover the issue at most one day later, thus limiting the time span of the fraud.

In this paper, the main goal of monitors is to detect, stop and report incorrect protocol runs. Monitors are not designed for example to assist one in forensic diagnosis after an attack has been found.

The monitoring is "black-box" in that the source code of the monitored application is not needed; only its observable behaviour (data transmitted over the medium, or *traces*) and locally accessed data are required. Thus any legacy implementation can still be used in production as is, while being monitored. The correctness of this approach depends on the correctness of the generated monitor. Our approach leverages formal methods in the derivation of the monitor implementation, so that a trustworthy monitor is obtained.

Note that this approach can be exploited during the testing phase as well: One can run an arbitrary number of simulated protocol sessions in a testing environment, and use the monitor to check for the correct behaviour.

In order to validate the proposed approach, a monitor for the SSL protocol is presented. The generated monitor stops incorrect sessions that could, for example, exploit a recently found flaw in the OpenSSL implementation.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 illustrates the formal background used in the paper. Section 4 describes the function translating a Spi Calculus protocol agent's specification into a monitor specification for that agent. Then section 5 shows the SSL protocol case study. Finally section 6 concludes.

For brevity, this paper mainly concentrates on the description of the proposed approach and on its validation by means of a real-life size case study. An extended version of this paper that includes all the formal definitions and the source code of the presented case study and other case studies can be found in [12].

## 2 Related Work

Several attempts have been made to check that a protocol role implementation is correct w.r.t. its specification which can be grouped in four main categories: (1) Model Driven Development (MDD); (2) Static Code Verification; (3) Refinement Types; (4) Online Monitoring and Intrusion Detection Systems (IDSs).

The first approach consists of designing and verifying a formal, high-level model of a security protocol and to semi-automatically derive an implementation that satisfies the same security properties of the formal model [8,9,13]. However, it has the drawback of not handling legacy implementations of security protocols.

The second approach starts from the source code of an existing implementation, and extracts a formal model which is verified for the desired security properties [5, 11]. In principle, this approach can deal with legacy implementations, but their source code must be available, which is not always the case.

The third approach proves security properties of an implementation by means of a special kind of type checking on its source code [4]. Working on the source code, it shares the same advantages and drawbacks of the second approach.

The fourth approach comes in two versions. With online monitoring, the source code of an existing implementation is instrumented with assertions: program execution is stopped if any assertion fails at runtime [3]. Besides requiring the source code, the legacy implementation must be substituted with the instrumented one, which may not always be the case. IDSs are systems that monitor network traffic and compare it against known attack patterns, or expected average network traffic. By working on averages, in order not to miss real attacks, IDSs often report false positive warnings. In order to reduce them, sometimes the source code of the monitored implementation is also instrumented [10], sharing the same advantages and drawback of online monitoring.

Another branch of research focused on security wrappers for legacy implementations. In [1], a formal approach that uses security wrappers as firewalls with an encrypting tunnel is described. Any communication that crosses some security boundary is automatically and transparently encrypted by the wrappers. That is, the wrappers *add* security to a distributed legacy system. In our approach, the monitor *enforces* the security already present in the system. Technically, our approach derives a monitor based on the security requirements already present in the legacy system, instead of adding a boilerplate layer of security.

Analogously, in [7] wrappers are used, among other things, to transparently encrypt local files accessed by library calls. However, distributed environments are not taken into account. Finally, in [17] wrappers are used to harden software libraries. However, cryptography and distributed systems are not considered, and the approach is test-driven, rather than formally based.

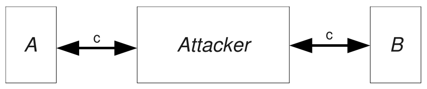
## 3 Formal Background

### 3.1 Network Model

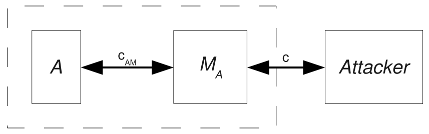
Many network models have been proposed in the Dolev-Yao setting. For example, sometimes the network is represented as a separate process [16]; the attacker is connected to this network, and can eavesdrop, drop and modify messages, or forge new ones. In other cases, the attacker is the medium [15], and honest agents can only communicate *through* the attacker. Even more detailed network models have been developed [19], where some nodes may have direct, private secured communication with other nodes, while still also being able to communicate through insecure channels, controlled by the attacker.

In general, it is not trivial to show that all of these models are equivalent in a Dolev-Yao setting, furthermore different network models and agents granularity justify different positions of the monitor with respect to the monitored agent, affecting the way the monitor is actually implemented. In this paper, we focus on a simple scenario that is usually found in practice, and is depicted in figure 2(a): the attacker is the medium, and every protocol agent communicates over a single insecure channel $c$, and private channels are not allowed. Moreover, agents are sequential and non-recursive.

Let us define $A$ as the (correct) model of the agent to be monitored, and $M_A$ as the model of its monitor. When the monitor is present, $A$ communicates



(a) Agents $A$ and $B$ with the attacker.



(b) Agent $A$ monitored by $M_A$ and the attacker.

Fig. 2: The network model.

| $L, M, N$ ::= | terms |
|---|---|
| $n$ | name |
| $(M, N)$ | pair |
| $0$ | zero |
| $suc(M)$ | successor |
| $x$ | variable |
| $M^{\sim}$ | shared-key |
| $\{M\}_N$ | shared-key encryption |
| $H(M)$ | hashing |
| $M^{+}$ | public part |
| $M^{-}$ | private part |
| $\{[M]\}_N$ | public-key encryption |
| $[\{M\}]_N$ | private-key signature |

(a) Spi Calculus terms.

| $P, Q, R$ ::= | processes |
|---|---|
| $\overline{M}\langle N\rangle .P$ | output |
| $M(x).P$ | input |
| $P\|Q$ | composition |
| $!P$ | replication |
| $(\nu n)\,P$ | restriction |
| $[M\ is\ N]\,P$ | match |
| $\mathbf{0}$ | nil |
| $let\ (x, y) = M\ in\ P$ | pair splitting |
| $case\ M\ of\ 0 : P\ suc(x) : Q$ | integer case |
| $case\ L\ of\ \{x\}_N\ in\ P$ | shared-key decryption |
| $case\ L\ of\ \{[x]\}_N\ in\ P$ | decryption |
| $case\ L\ of\ [\{x\}]_N\ in\ P$ | signature check |

(b) Spi Calculus processes.

Table 1: Spi Calculus grammar.

with $M_A$ only, through the use of a private channel $c_{AM}$, while $M_A$ is directly connected to the attacker by channel $c$, as depicted in figure 2(b). The dashed box denotes that $A$ and $M_A$ run in the same environment, for example they run on the same system with same privileges. Note that in $A$ channel $c$ is in fact renamed to $c_{AM}$.

### 3.2 The Spi Calculus

In this paper, the formal models are expressed in Spi Calculus [2]. Spi Calculus is amenable for our approach because it is a domain specific language tailored at expressing the behaviour of single security protocol agents, where checks on received data must be explicitly specified. Thus, from the Spi Calculus specifications of protocol agents, the $a2m$ function can derive precise and complete specifications of their monitors.

Briefly, a Spi Calculus specification is a system of concurrent processes that operate on untyped data, called terms. Terms can be exchanged between processes by means of input/output operations. Table 1(a) contains the terms defined by the Spi Calculus, while table 1(b) shows the processes.

A name $n$ is an atomic value, and a pair $(M, N)$ is a compound term, composed of the terms $M$ and $N$. The 0 and $suc(M)$ terms represent the value of zero and the logical successor of some term $M$, respectively. A variable $x$ represents any term, and it can be bound once to the value of another term. If a variable or a name is not bound, then it is free. The $M^{\sim}$ term represents a symmetric key built from key material $M$, and $\{M\}_N$ represents the encryption of the plaintext $M$ with the symmetric key $N$, while $H(M)$ represents the result of hashing $M$. The $M^+$ and $M^-$ terms represent the public and private part of the keypair $M$ respectively, while $\{[M]\}_N$ and $[\{M\}]_N$ represent public key and private key asymmetric encryptions respectively.

Informally, the $\overline{M}\,\langle N\rangle\,.P$ process sends message $N$ on channel $M$, and then behaves like $P$, while the $M\,(x)\,.P$ process receives a message from channel $M$, and then behaves like $P$, with $x$ bound to the received term in $P$. A process $P$ can perform an input or output operation iff there is a reacting process $Q$ that is ready to perform the dual output or input operation. Note, however, that processes run within an environment (the Dolev-Yao attacker) that is always ready to perform input or output operations. Composition $P|Q$ means parallel execution of processes $P$ and $Q$, while replication $!P$ means an unbounded number of instances of $P$ run in parallel. The restriction process $(\nu n)P$ indicates that $n$ is a fresh name (i.e. not previously used, and unknown to the attacker) in $P$. The match process executes like $P$, if $M$ equals $N$, otherwise is stuck. The nil process does nothing. The pair splitting process binds the variables $x$ and $y$ to the components of the pair $M$, otherwise, if $M$ is not a pair, the process is stuck. The integer case process executes like $P$ if $M$ is 0, else it executes like $Q$ if $M$ is $suc(N)$ and $x$ is bound to $N$, otherwise the process is stuck. If $L$ is $\{M\}_N$, then the shared-key decryption process executes like $P$, with $x$ bound to $M$, else it is stuck, and analogous reasoning holds for the decryption and signature check processes.

The assumption that $A$ is a sequential process, means that composition and replication are never used in its specification.

## 4 The Monitor Generation Function

The $a2m$ function translates a sequential protocol role specification into a monitor specification for that role; formally, $M_A \triangleq a2m(A)$. For brevity, $a2m$ is only informally presented here, by means of a running example. Formal definitions can be found in [12].

Before introducing the function, the concepts of *known* and *reconstructed* terms are given. For any Spi Calculus state, a term $T$ is said to be *known* by the monitor through variable $\_T$, iff $\_T$ is bound to $T$. This can happen either because the implementation of $M_A$ has access to the agent's memory location where $T$ is stored; or because $T$ can be read from a communication channel, and $M_A$ stores $T$ in variable $\_T$. A compound term $T$ (that is not a name or a variable) is said to be *reconstructed*, if all its subterms are known or reconstructed. For example, suppose $M$ is known through $\_M$ and $H(N)$ is known through $\_H(N)$. It is the case that $(H(N), M)$ is reconstructed by $(\_H(N), \_M)$. Note that, as terms become known, other terms may become reconstructed too. In the example given above, if $M$ was not known, then it was not possible to reconstruct $(H(N), M)$;

```
1a: A(M,k) :=
2a:  cAM<{M}k>.



3a:  cAM(x).
4a:  [x is H(M)]
5a:  0
```

(a) Agent $A$ specification.

```
1m: MA(k,_H(M)) :=
2m:  cAM(_{M}k).
3m:  case _{M}k of {_M}k in
4m:  [_H(M) is H(_M)]
5m:  c<_{M}k>.
6m:  c(x).
7m:  [x is _H(M)]
8m:  cAM<x>.
9m:  0
```

(b) Monitor specification derived from agent $A$ one.

Fig. 3: Example specification of agent $A$ along with its derived monitor $M_A$.

however, if later $M$ became known (for example, because it was sent over a channel), then $(H(N), M)$ would become reconstructed.

Note that the monitor implementation presented in this paper does not enforce that nonces are actually different for each protocol run. To enable this, the monitor should track all previously used values, in order to ensure that no value is used twice. Especially in the online mode, this overhead may not be acceptable. In order to drop this check, it is needed to assume that the random value generator in the monitored agent is correctly implemented. Also note that there may be cases where the monitor has not enough information to properly check protocol execution. These cases are recognised by the $a2m$ function, so that an error state is reached, and no unsound monitor is generated.

The $a2m$ function behaviour is now described by means of a running example. Agent $A$ sends some data $M$ encrypted by the key $k$ to the other party, and expects to receive the hash of the plaintext, that is $H(M)$. Note that the example focuses on the way the $a2m$ function operates, rather than on monitoring a security protocol, so no security claim is meant to be made on this protocol. Figure 3(a) shows the specification for agent $A$, and figure 3(b) its derived monitor specification $M_A$. Here, an ASCII syntax of Spi Calculus is used: the '$\nu$' symbol is replaced by the '`@`' symbol, and the overline in the output process is omitted (input and output processes can still be distinguished by the different brackets).

At line 1a the agent $A$ process is declared: it has two free variables, a message $M$ and a symmetric key $k$. At line 2a $A$ sends the encryption of $M$ with key $k$. Then, at line 3a it receives a message that is stored into variable $x$, and, at line 4a, the received message is checked to be equal to the hashing of $M$: if this is the case, the process correctly terminates.

At line 1m, the monitor $M_A$ is declared: to make this example significant, it is assumed that in the initial state the key $k$ used by $A$ is known by the monitor (through the variable $k$), while $M$ is not known (for example, because the monitor cannot access those data); however $H(M)$ is known through $\_H(M)$, that is the monitor has access to the memory location where $H(M)$ is stored, and this value is bound to the variable $\_H(M)$ in the monitor.

When line 2a is translated by $a2m$, lines 2m–5m are produced. The data sent by $A$ are received by the monitor at line 2m, and stored in variable $\_\{M\}_k$. Afterwards, some checks on the received value are added by the $a2m$ function. In general, each time a new message is received from the monitored application, it or its parts are checked against their expected (known or reconstructed) values. In this case, since $\{M\}_k$ is not known (by hypothesis) or reconstructed (because $M$ is not known or reconstructed), it cannot be directly compared against the known or reconstructed value, so it is exploded into its components. As $\{M\}_k$ is an encryption and $k$ is known, the decryption case process is generated at line 3m, binding $\_M$ to the value of the plaintext, that should be $M$. Since $M$ is not known or reconstructed, and it is a name, $\_M$ cannot be dissected any more; instead, $M$ becomes known through $\_M$, in other words, the term stored in $\_M$ is assumed by the monitor to be the correct term for $M$. Note that, before $M$ was known through $\_M$, $H(M)$ was known through $\_H(M)$, but it was not reconstructed. After the assignment of $\_M$, $H(M)$ becomes reconstructed by $H(\_M)$ too. The match process at line 4m ensures that known and reconstructed values for the same term are actually the same.

After all the possible checks are performed on the received data, they are forwarded to the attacker at line 5m. Then, line 3a is translated into line 6m. When translating an input process, the monitor receives message $x$ from the attacker on behalf of the agent and buffers it; $x$ is said to be known through $x$ itself. Then the monitor behaves according to what is done by the agent (usually checks on the received data, as it is the case in the running example). The received message stored in $x$ is not forwarded to $A$ immediately, because this could lead $A$ to receive some malicious data, that could for example enable some denial of service attack. Instead, the received data are buffered, and will be forwarded to $A$ only when necessary: that is when the process should end ($\mathbf{0}$ case), or when some output data from $A$ are expected.

Line 4a is then translated into line 7m, and finally, line 5a is translated into lines 8m and 9m. First, all buffered data ($x$ in this case) are forwarded to $A$, then the monitor correctly ends.

## 5 An SSL Server Monitor Example

### 5.1 Monitor Specification

As shown in figure 1, in order to get the monitor specification, a Spi Calculus specification of the server role for the SSL protocol is needed. The full SSL protocol is rather complex: many scenarios are possible, and different sets of cryptographic algorithms (called *ciphersuites*) can be negotiated. For simplicity, this example considers only one scenario of the SSL protocol version 3.0, where the same cipher suite is always negotiated. Despite these simplifications, we believe that the considered SSL fragment is still significant, and that adding full SSL support would increase the example complexity more than its significance.

In this paper, the chosen scenario requires the server to use a DSA certificate for authentication and data signature. Although RSA certificates are more com-

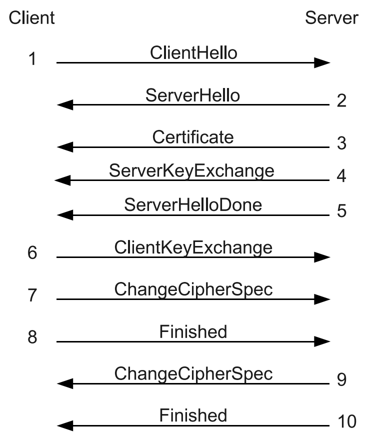


Fig. 4: Typical SSL scenario.

mon in SSL, using a DSA certificate allowed us to stress a bug in the OpenSSL implementation, showing that the monitor can actively drop malicious sessions that would be otherwise accepted as genuine by the flawed OpenSSL implementation. The more common RSA scenario has been validated through a dedicated case-study. However, it is not reported here for brevity; it can be found in [12].

The SSL scenario considered in this example is depicted informally in figure 4, while figure 5 shows a possible Spi Calculus specification of a server for the chosen scenario. The ASCII syntax of Spi Calculus is used in figure 5. Also, in order to model the Diffie-Hellman (DH) key exchange, the $EXP(L, M, N)$ term is added, which expresses the modular exponentiation $L^M \mod N$, along with the equation $EXP(EXP(g, a, p), b, p) = EXP(EXP(g, b, p), a, p)$.

To make the specification more readable, lists of terms like $(A, B, C)$ are added as syntactic sugar, and they are translated into left associated nested pairs, like $((A, B), C)$; a `rename n = M in P` process is introduced too, that renames the term $M$ to $n$ and then behaves like $P$.

The first message is the ClientHello, sent from the client to the server. It contains the highest protocol version supported by the client, a random value, a session ID for session resuming, and the set of supported cipher suites and compression methods. In the server specification, the ClientHello message is received and split into its parts at line 2S. In the chosen scenario, the client should send at least 3.0 as the highest supported protocol version, and it should send 0 as session ID, so that no session resuming will be performed. Moreover, the client should at least support the always-negotiated cipher suite, namely `SSL_DHE_DSS_3DES_EDE_CBC_SHA`, with no compression enabled. All these constraints are checked at lines 3S–4S.

In the second message the server replies by sending its ServerHello message, that contains the chosen protocol version, a random value, a fresh session ID and the chosen cipher suite and compression method. The server random value

```
 1S Server() :=
 2S   c(c_hello).  let (c_version,c_rand,c_SID,c_ciph_suite,c_comp_method) = c_hello in
 3S   [ c_version is THREE_DOT_ZERO ]  [ c_SID is ZERO ]
 4S   [ c_ciph_suite is SSL_DHE_DSS_3DES_EDE_CBC_SHA ]  [ c_comp_method is comp_NULL ]
 5S   (@s_rand)  (@SID)
 6S   rename S_HELLO = (THREE_DOT_ZERO,s_rand,SID,SSL_DHE_DSS_3DES_EDE_CBC_SHA,comp_NULL) in
 7S   (@DH_s_pri)  rename DH_s_pub = EXP(DH_Q,DH_s_pri,DH_P) in
 8S   rename S_KEX = ((DH_P,DH_Q,DH_s_pub),[{H(c_rand,s_rand,(DH_P,DH_Q,DH_s_pub))}]s_PriKey) in
 9S   c<S_HELLO,S_CERT,S_KEX,S_HELLO_DONE>.
10S   c(c_kex).  let (c_kexHead,DH_c_pub) = c_kex in  rename PMS = EXP(DH_c_pub,DH_s_pri,DH_P) in
11S   rename MS = H(PMS,c_rand,s_rand) in  rename KM = H(MS,c_rand,s_rand) in
12S   rename c_w_IV = H(KM,C_WRITE_IV) in  rename s_w_IV = H(KM,S_WRITE_IV) in
13S   c(c_ChgCipherSpec).  [ c_ChgCipherSpec is CHG_CIPH_SPEC ]
14S   c(c_encrypted_Finish).  case c_encrypted_Finish of {c_Finish_and_MAC}(KM,C_WRITE_KEY)~ in
15S   let (c_Finish,c_MAC) = c_Finish_and_MAC in  [ c_MAC is H((KM,C_MAC_SEC)~,c_Finish) ]
16S   let (final_Hash_MD5, final_Hash_SHA) = c_Finish in
17S   [ final_Hash_MD5 is H((c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex),C_ROLE,MS,MD5) ]
18S   [ final_Hash_SHA is H((c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex),C_ROLE,MS,SHA) ]
19S   c<CHG_CIPH_SPEC>.
20S   rename DATA = (c_hello,S_HELLO,S_CERT,S_KEX,S_HELLO_DONE,c_kex,c_Finish) in
21S   rename S_FINISH = (H(DATA,S_ROLE,MS,MD5),H(DATA,S_ROLE,MS,SHA)) in
22S   (@pad)  c<{S_FINISH,H((KM,S_MAC_SEC)~,S_FINISH),pad}(KM,S_WRITE_KEY)~>.
23S   0
```

Fig. 5: A possible Spi Calculus specification of an SSL server.

and the fresh session ID are generated at line 5S, then the ServerHello message is declared at line 6S. Again, in the chosen scenario, the server chooses protocol version 3.0, and always selects the `SSL_DHE_DSS_3DES_EDE_CBC_SHA` cipher suite, with no compression enabled. Then the server sends the Certificate message to the client: in the chosen scenario, this message contains a DSA certificate chain for the server's public key, that authenticates the server.

In the fourth message, named ServerKeyExchange, the server sends the DH key exchange parameters to the client, and digitally signs them with its public key. In the server specification, the DH server secret value `DH_s_pri` and the corresponding public value `DH_s_pub` are computed at line 7S. Then, at line 8S, the ServerKeyExchange message is declared: it consists of the server DH parameters, along with a digital signature of the DH parameters and the client and server random values, in order to ensure signature freshness.

The fifth message is the ServerHelloDone. It contains no data, but signals the client that the server ended its negotiation part, so the client can move to the next protocol stage. In the server specification, these four messages are sent all at once at line 9S.

In the sixth message, the client replies with the ClientKeyExchange message, that contains the client's DH public value. Note that there is no digital signature in this message, since the client is not authenticated. In the server specification the ClientKeyExchange is received at line 10S, where the payload is split from the message header too. Both client and server derive a shared secret from the DH key exchange. This shared secret is called Premaster Secret (PMS), and it is used by both parties to derive some shared secrets used for symmetric encryption of application data. The PMS is computed by the server at line 10S. By applying an SSL custom hashing function to the PMS and the client and server random data, both client and server can compute the same Master Secret (MS). The bytes of the MS are then extended (again by using a custom SSL hashing algorithm) to as many byte as required by the negotiated ciphersuite, obtaining the Key Material (KM) (line 11S). Finally, different subsets of bytes of the KM are used as shared secrets and as initialization vectors (IVs). Note that IVs, that are extracted at line 12S, are never referenced in the specification. They will be used as cryptographic parameters for subsequent encryptions, during the code generation step, explained in section 5.2.

The seventh message is the ChangeCipherSpec, received and checked at line 13S. This message contains no data, but signals the server that the client will start using the negotiated cipher suite from the next message on.

The client then sends its Finished message. Message Authentication Code (MAC) and encryption are applied to the Finished message sent by the client, as the client already sent its ChangeCipherSpec message. The client Finished message is received and decrypted at line 14S. The decryption key used `(KM,C_WRITE_KEY)~` is obtained by creating a shared key, starting from the key material `KM` and a marker `C_WRITE_KEY` that indicates which portion of the KM to use. At line 15S the MAC is extracted from the plaintext, and verified. The unencrypted content of the Finished message contains the final hash, that hashes all relevant session information: all exchanged messages (excluding the ChangeCipherSpec ones) and the MS are included in the final hash, plus some constant data identifying the protocol role (client or server) that sent the final hash. In fact, the Finished message includes two versions of the same final hash, one using the MD5 algorithm, and one using the SHA-1 algorithm. Both versions of the final hash are extracted and checked at lines 16S–18S. As Spi Calculus does not support different algorithms for the same hash, they are distinguished by a marker (`MD5` and `SHA` respectively) as the last hash argument, making them syntactically different.

Then the server sends its ChangeCipherSpec message to client (line 19S), and its Finished message, that comes with MAC and encryption too (lines 20S–22S). Encryption requires random padding to align the plaintext length to the cipher block size. This random padding must be explicitly represented in the server specification, so that the monitor can recognise and discard it, and only check the real plaintext. Otherwise the monitor would try to locally reconstruct the encryption, but it would always fail, because it could not guess the padding. The protocol handshake is now complete, and next messages will contain secured application data.

In order to verify any security property on this specification, the full SSL specification, including the client and protocol sessions instantiations is required. However, this is outside the scope of this paper; SSL security properties have already been verified, for example, by the AVISPA project [18]. Here it is assumed instead that the specification of the server is correct, and thus secure, so that the monitoring approach can be shown.

The *a2m* function described in section 4 is applied on the server specification, in order to obtain the online monitor specification for the server role. For brevity, the resulting specification is not shown here. It can be found, along with more implementation details, in [12]. It is assumed that the monitor has access to the server private DH value, which is then *known*, while it is not able to read the freshly generated server random value `s_rand`, the session ID `SID` and the random padding which are then not known nor reconstructed at generation time. Often, the server will generate a fresh DH private value for each session, and it will usually only store it in memory. In general, with some effort the monitor will be able to directly read this secret from the legacy application memory, without the need of the source code. Nevertheless, in a testing environment, if the source code of the monitored application happens to be available, it is possible to patch the monitored application, so that it explicitly communicates the DH private value to the monitor. Indeed, this is reasonable because the monitor is part of the trusted system, and is actually more trusted than the monitored application.

### 5.2 Monitor Implementation

The source code of the monitor implementation can be found in [12]. In order to generate the monitor implementation, the spi2java MDD framework is used [13]. Briefly, spi2java is a toolchain that, starting from the formal Spi Calculus specification of a security protocol, semi-automatically derives an interoperable Java implementation of the protocol actors. In the first place, spi2java was designed to generate security protocol actors, rather than monitors. In this paper, we originally reuse spi2java to generate a monitor.

In order to generate an executable Java implementation of a Spi Calculus specification, some details that are not contained in the Spi Calculus specification must be added. That is, the Spi Calculus specification must be refined, before it can be translated into a Java application.

As shown in figure 1, the spi2java framework assists the developer during the refinement and code generation steps. The spi2java refiner is used to automatically infer some refinement information from the given specification. All inferred information is stored into an eSpi (extended Spi Calculus) document, which is coupled with the Spi Calculus specification. The developer can manually refine the generated eSpi document; the manually refined eSpi document is passed back to the spi2java refiner, that checks its coherence against the original Spi Calculus specification, and possibly infers new information from the user given one. This iterative refinement step can be repeated until the developer is satisfied with the obtained eSpi document, but usually one iteration is enough.

The obtained eSpi document and the original Spi Calculus specification are passed to the spi2java code generator that automatically outputs the Java code implementing the given specification. The generated code implements the "protocol logic", that is the code that simulates the Spi Calculus specification by coordinating input/output operations, cryptographic primitives and checks on received data. Dealing with Java sockets or the Java Cryptographic Architecture (JCA) is delegated to the SpiWrapper library, which is part of the spi2java framework. The SpiWrapper library allows the generated code to be compact and readable, so that it can be easily mapped back to the Spi Calculus specification. For example, the monitor specification corresponding to line 2S of the server specification in figure 5 is translated as

```
/* c_0(c_hello_1). */
Pair c_hello_1 = (Pair) c_0.receive(new PairRecvClHello());
```

(each Spi Calculus term name is mangled to make sure there is a unique Java identifier for that term). To improve readability, the spi2java code generator outputs the translated Spi Calculus process as a Java comment too. In this example, the Java variable `c_0` has type `TcpIpChannel`, which is a Java class included in the SpiWrapper library implementing a Spi Calculus channel using TCP/IP as transport layer. This class offers the `receive` method that allows the Spi Calculus input process to be easily implemented, by internally dealing with the Java sockets. The `c_hello_1` Java variable has type `Pair`, which implements the Spi Calculus pair. The `Pair` class offers the `getLeft` and `getRight` methods, allowing a straightforward implementation of the pair splitting process. The spi2java translation function is proven sound in [14].

In order to get interoperable implementations, the SpiWrapper library classes only deal with the *internal* representation of data. By extending the SpiWrapper classes, the developer can provide custom marshalling functions that transform the internal representation of data into the external one.

In the SSL monitor case study, a two-tier marshalling layer has been implemented. Tier 1 handles the Record Layer protocol of SSL, while tier 2 handles the upper layer protocols. When receiving a message from another agent, tier 1 parses one Record Layer message from the input stream, and its contained upper layer protocol messages are made available to tier 2. The latter implements the real marshalling functions, for example converting US-ASCII strings to and from Java String objects. Analogous reasoning applies when sending a message. The marshalling layer functions only check that the packet format is correct. No control on the payload is needed: it will be checked by the automatically generated protocol logic.

The SSL protocol defines custom hashing algorithms, for instance to compute the MS from the PMS, or to compute the MAC value. For each of them, a corresponding SpiWrapper class has been created, implementing the custom algorithm. Moreover, the spi2java framework has been extended to support the modular exponentiation, so that DH key exchange can be supported.

Finally, it is worth pointing out some details about the IVs used by cryptographic operations (declared in the server specification at line 12S). For each term of the form $\{M\}_K$, the eSpi document allows its cryptographic algorithm (such as DES, 3DES, AES) and its IV to be specified. However, the IV is only known at run time. The spi2java framework allows cryptographic algorithms and parameters to be resolved either at compile time or at run time. If the parameter is to be resolved at compile time, the value of the parameter must be provided (e.g. AES for the symmetric encryption algorithm, or a constant value for the IV). If the parameter is to be resolved at run time, the identifier of another term of the Spi Calculus specification must be provided: the parameter value will be obtained by the content of the referred term, during execution. In the SSL case study, this feature is used for the IVs. For example, the `{c_Finish_and_MAC}(KM,C_WRITE_KEY)~` term uses the `H(KM,C_WRITE_IV)` term as IV. Technically, this feature enables support for cipher suite negotiation. However, as stated above, this would increase the specification complexity more than it would increase its significance, and is left for future work.

### 5.3 Experimental Results

The monitor has been coupled in turn with three different SSL server implementations, namely OpenSSL[3] version 0.9.8j, GnuTLS[4] version 2.4.2 and JESSIE[5] version 1.0.1.

Since the online monitoring paradigm is used in this case study, the monitor is accepting connections on the standard SSL port (443), while the real server is started on another port (4433). Each time a client connects to the monitor, the latter opens a connection to the real server, starting data checking and forwarding, as explained above.

It is worth noting that switching the server implementation is straightforward. In the testing scenario, assuming that the server communicates its private DH value to the monitor, it is enough to shut down the running server implementation, and to start the other one; the monitor implementation remains the same, and no action on the monitor is required. Otherwise, it is enough to restart the monitor too, enabling the correct plugin that gathers the private DH value from the legacy application memory. In other words, in a production scenario, the same monitor implementation can handle several different legacy server implementations; in the monitor, the only server-dependent part is the plugin that reads the DH secret value from the server application memory.

In order to generate protocol sessions, three SSL clients have been used with each server; namely the OpenSSL, GnuTLS, and JESSIE clients. During experiments, the monitor helped in spotting a bug in the JESSIE client: This client always sends packet of the SSL 3.1 version (better known as TLS 1.0), regardless of the negotiated version, that is SSL 3.0 in our scenario. The monitor correctly rejected all JESSIE client sessions, reporting the wrong protocol version.

When the OpenSSL or GnuTLS clients are used, the monitor correctly operates with all the three servers. In particular, safe sessions are successfully handled; conversely, when exchanged data are manually corrupted, they are recognized by the monitor and the session is aborted: corrupted data are never forwarded to the intended recipient.

In order to estimate the impact on performances of the online monitoring approach, execution times of correctly ended protocol sessions with and without the monitor have been measured. Thus, performances regarding the JESSIE client are not reported, as no correct session could be completed, due to the

[3] Available at: http://www.openssl.org/
[4] Available at: http://www.gnu.org/software/gnutls/
[5] Available at: http://www.nongnu.org/jessie/

| Client | Server | No Monitor [s] | Monitor [s] | Overhead [s] | Overhead [%] |
|---|---|---|---|---|---|
| OpenSSL | OpenSSL | 0.032 | 0.113 | 0.081 | 253.125 |
| GnuTLS | OpenSSL | 0.108 | 0.132 | 0.024 | 22.253 |
| OpenSSL | GnuTLS | 0.073 | 0.128 | 0.056 | 76.552 |
| GnuTLS | GnuTLS | 0.109 | 0.120 | 0.011 | 10.313 |
| OpenSSL | JESSIE | 0.158 | 0.172 | 0.014 | 8.986 |
| GnuTLS | JESSIE | 0.144 | 0.148 | 0.004 | 2.788 |

Table 2: Average execution times for protocol runs with and without monitoring.

discovered bug. That is, the measured performances all correspond to valid executions of the protocol only. Communication between client, server and monitor happened over local sockets, so that no random network delays could be introduced; moreover system load was constant during test execution. Table 2 shows the average execution times for different client-server pairs, with and without monitor enabled. For each client-server pair, the average execution times have been computed over ten protocol runs. Columns “No Monitor” and “Monitor” report the average execution times, in seconds, without and with monitoring enabled respectively. When monitoring is not enabled, the clients directly connect to the server on port 4433. The “Overhead” columns show the overhead introduced by the monitor, in seconds and in percentage respectively. In four cases out of six, the monitor overhead is under 25 milliseconds. From a practical point of view, a client communicating through a real distributed network could hardly tell whether a monitor is present or not, since network times are orders of magnitude higher. On the other hand, in the worst cases online monitoring can slow down the server machine up to 2.5 times. Whether this overhead is acceptable on the server side depends on the number of sessions per seconds that must be handled. If the overhead is not acceptable, the offline monitoring paradigm can still be used.

**The OpenSSL security flaw.** Recently, the client side of the OpenSSL library prior to version 0.9.8j has been discovered flawed, such that in principle it could treat a malformed DSA certificate as a good one rather than as an error.[6] By inspecting the flawed code, we were able to forge such malformed certificate that exploited the affected versions. This malformed DSA certificate must have the $q$ parameter one byte longer than expected. Up to our knowledge, this is the first documented and repeatable exploit for this flaw.

Without monitoring enabled, we generated protocol sessions between an SSL server sending the offending certificate, and both OpenSSL clients version 0.9.8i (flawed) and 0.9.8j (fixed). By using the `-state` command line argument, it is possible to conclude that the 0.9.8i version completes the handshake by reaching the “read finished A” state (after message 10 in figure 4); while the 0.9.8j version correctly reports an “handshake failure” error at state “read server certificate A”, that is immediately after message 3 in figure 4.

[6] http://www.openssl.org/news/secadv_20090107.txt

When monitoring is enabled, the malformed server certificate is passed to the monitor as an input parameter, that is, the server certificate is *known* by the monitor. In this case the monitor actually refuses to start. Indeed, when loading the server certificate, the monitor spots that it is malformed, and does not allow any session to be even started. If we drop the assumption that the monitor knows the server certificate, then the monitor starts, and checks the server certificate when it is received over the network. During these checks, the malformed certificate is found, and the session is dropped, before the server Certificate message is forwarded to the client. This prevents the aforecited flaw to be exploited on OpenSSL version 0.9.8i.

## 6 Conclusion

The paper shows a formally-based yet practical methodology to design, develop and deploy monitors for legacy implementations of security protocols, without the need to modify the legacy implementations or to analyse their source code. To our knowledge, this is the first work that allows legacy implementations of security protocol agents to be black-box monitored.

This paper introduces a function that, given the specification of a security protocol actor, automatically generates the specification of a monitor that stops incorrect sessions, without rising false positive alarms. From the obtained monitor specification, an MDD approach is used to generate a monitor implementation; for this purpose, the spi2java framework has been originally reused, and some of its parts enhanced.

Finally, the proposed methodology has been validated by implementing a monitor starting from the server role of the widely used SSL protocol. Core insights gained from conducting the SSL case study include that the same generated monitor implementation can in fact monitor several different SSL server implementations against different clients, in a black-box way. The only needed information is the private Diffie-Hellman key used by the server, in order to check message contents. Moreover, by reporting session errors, the monitor effectively helped us in finding a bug in an open source SSL client implementation.

The "online" monitoring paradigm proved useful in avoiding protocol violations, for example by stopping malicious data that would have otherwise exploited a known flaw of the widely deployed OpenSSL client. The overhead introduced by the monitor to check and forward messages is usually negligible. If the overhead is not acceptable, this paper also proposes an "offline" monitoring strategy that has no overhead and can still be useful to timely discover protocol attacks.

As future work, a general result about soundness of the monitor specification generating function would be useful. The soundness property should show that the generated monitor specification actually forwards only (and all) the protocol sessions that would be accepted by the agent's verified specification. Together with the soundness proofs of the spi2java framework, this would produce a sound monitor implementation, directly from the monitored agent's specification.

## References

1. Abadi, M., Fournet, C., Gonthier, G.: Secure implementation of channel abstractions. Information and Computation 174(1), 37–83 (2002)
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The Spi Calculus. Digital Research Report 149 (1998)
3. Bauer, A., Jürjens, J.: Security protocols, properties, and their monitoring. In: International Workshop on Software Engineering for Secure Systems. pp. 33–40 (2008)
4. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: Computer Security Foundations Symposium, IEEE. pp. 17–32 (2008)
5. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. In: Computer Security Foundations Workshop. pp. 139–152 (2006)
6. Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–207 (1983)
7. Fraser, T., Badger, L., Feldman, M.: Hardening COTS software with generic software wrappers. In: IEEE Symposium on Security and Privacy. pp. 2–16 (1999)
8. Hubbers, E., Oostdijk, M., Poll, E.: Implementing a formally verifiable security protocol in Java Card. In: Security in Pervasive Computing. Lecture Notes in Computer Science, vol. 2802, pp. 213–226 (2003)
9. Jeon, C.W., Kim, I.G., Choi, J.Y.: Automatic generation of the C# code for security protocols verified with Casper/FDR. In: International Conference on Advanced Information Networking and Applications. pp. 507–510 (2005)
10. Joglekar, S.P., Tate, S.R.: ProtoMon: Embedded monitors for cryptographic protocol intrusion detection and prevention. Journal of Universal Computer Science 11(1), 83–103 (2005)
11. Jürjens, J., Yampolskiy, M.: Code security analysis with assertions. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 392–395 (2005)
12. Pironti, A., Jürjens, J.: Online resources about black-box monitoring, available at: http://alfredo.pironti.eu/research/projects/monitoring/
13. Pironti, A., Sisto, R.: An experiment in interoperable cryptographic protocol implementation using automatic code generation. In: IEEE Symposium on Computers and Communications. pp. 839–844 (2007)
14. Pironti, A., Sisto, R.: Provably correct Java implementations of Spi Calculus security protocols specifications. Computers & Security (2009), in Press
15. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR (1997)
16. Schneider, S.: Security properties and CSP. In: IEEE Symposium on Security and Privacy. pp. 174–187 (1996)
17. Süßkraut, M., Fetzer, C.: Robustness and security hardening of COTS software libraries. In: IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 61–71 (2007)
18. Viganò, L.: Automated security protocol analysis with the AVISPA tool. Electronic Notes on Theoretical Computer Science 155, 61–86 (2006)
19. Voydock, V.L., Kent, S.T.: Security mechanisms in high-level network protocols. ACM Computing Surveys 15(2), 135–171 (1983)

doi:10.1093/comjnl/bxh000

# Run-Time Security Traceability for Evolving Systems*

ANDREAS BAUER[1,4], JAN JÜRJENS[2,3], YIJUN YU[2]

[1]*National ICT Australia (NICTA)*
[2]*Computing Department, The Open University, UK*
[3]*Microsoft Research (Cambridge)*
[4]*School of Computer Science, The Australian National University*
*Email: baueran@rsise.anu.edu.au, {j.jurjens,y.yu}@open.ac.uk*

**Security-critical systems are challenging to design and implement correctly and securely. A lot of vulnerabilities have been found in current software systems both at the specification and the implementation levels. This paper presents a comprehensive approach for model-based security assurance. Initially, it allows one to formally verify the design models against high-level security requirements such as secrecy and authentication on the specification level, and helps to ensure that their implementation adheres to these properties, if they express a system's run-time behaviour. As such, it provides a traceability link from the design model to its implementation by which the actual system can then be verified against the model while it executes. This part of our approach relies on a technique also known as run-time verification. The extra effort for it is small as most of the computation is automated, however, additional resources at run-time may be required. If during run-time verification a security weakness is uncovered, it can be removed using aspect-oriented security hardening transformations. Therefore, this approach also supports the evolution of software since the traceability mapping is updated when refactoring operations are regressively performed using our tool-supported refactoring technique. The proposed method has been applied to the Java-based implementation JESSIE of the Internet security protocol SSL, in which a security weakness was detected and fixed using our approach. We also explain how the traceability link can be transformed to the official implementation of the Java Secure Sockets Extension (JSSE) that was recently made open source by Sun.**

*Keywords: run-time verification, monitoring, IT security, cryptographic protocols, formal verification, security analysis, software evolution, requirements traceability*

*Received 08 November 2008; revised 31 March 2009 and 12 September 2009*

## 1. INTRODUCTION

There has been successful research over the last years to provide security assurance tools for the lower abstraction levels of software systems. However, these tools usually search for specific security weaknesses, such as buffer overflow vulnerabilities. What is so far largely missing is automated tool support which would support security assurance throughout the software development process, starting from the analysis of software design models (e.g., in UML) against abstract security requirements (such as secrecy and authentication), and tracing the requirements to the code level to make sure that the implementation is still secure.

This article presents a tool-supported approach that supports such a software security assurance, which can be used in the context of an approach for Model-based Security Engineering (MBSE) that has been developed in recent years (see e.g., [42, 43] for details and Figure 1 for a visual overview). In this approach, recurring security requirements (such as secrecy, integrity, authentication and others) and security assumptions on the system environment, can be specified either within a UML specification (using the UML extension UMLsec [42]), or within the source code (Java or C) as annotations. One can then formally analyse the UMLsec models against the security requirements using the UMLsec tool suite which makes use of model checkers and automated theorem provers for first-order logic (see Figure 2 and [47, 66]). The approach has been used successfully in a number of industrial applications (e.g., at BMW [13] and $O_2$ (Germany) [44]).

However, it is not enough that the specification is secure: we must also ensure that the implemented system is secure

*This work was partly supported by the Royal Society Industrial Fellowship on *Automated Verification of Security-Critical Software (VeriSec)* and the EU FP7 Integrated Project *Security Engineering for Lifelong Evolvable Systems*. A preliminary version of this paper was presented at the BCS08 Visions of Computer Science Conference, held on September 22-24, 2008.

as well. There are at least two ways to approach this problem: static code verification, or a technique called run-time verification [22, 10, 53] (see Section 3.1 for an introduction). In this paper, we focus on using (online) run-time verification for our purposes. It has an important advantage over static verification: In static verification, one can only verify the implementation on the basis of predefined assumptions. For example, these include assumptions on the behavioural semantics of the programming language, the compiler that will compile the source code to byte code, and/or byte code to machine code, the execution environment (operating system, hardware, physical environment), etc. When trying to apply static verification to complex implementations, one usually needs to make additional simplifying abstractions in order to make an automated formal verification of such implementations feasible in the first place, and one thus needs to make the additional assumption that these abstractions do not limit the scope of the verification. The verification procedure is then only known to be sound where these assumptions are fulfilled, and it is usually not feasible to verify formally whether they are fulfilled for a given execution environment. The advantage of run-time verification is now that the targeted execution environment itself is part of the verification environment (since verification is done at run-time anyway), so by construction the verification will be sound for the execution environment at hand. For this reason, run-time verification also does not suffer from the same scalability issues that static verification does as systems or system models become more complex: run-time verification always considers one concrete behaviour produced by the running system rather than the overall state-space of all the possible states it can be in.

Since run-time verification is a formal yet also *dynamic technique* (i.e., it operates on the running system as compared to a system model) there exist, besides similarities to other formal verification techniques such as model checking, some similarities to testing; however, the context and goals are different: Testing for complex implementations can usually not be applied exhaustively. In contrast to that, run-time verification ensures, by construction, that every system trace that will ever be executed will be verified—while it is executed. In the case of the cryptographic protocols that we consider, it is indeed sufficient to notice attempted security violations at run-time to still be able to maintain the security of the system: The monitor is constructed in such a way that, if it detects a violation, the current execution of the security protocol will be terminated before any secret information is leaked out on the network. Therefore, despite the similarities between testing and run-time verification, run-time verification can provide a level of assurance that goes beyond what testing can usually achieve when applied to highly complex security-critical software.

In practice, however, systems do not remain the same after they are deployed. On the contrary, many systems evolve over their life-time, and usually their life-time is significantly longer than expected when they were implemented (this became very apparent with the year 2000 bug). Manually re-establishing the verified traceability link for a new version of an implementation would be time-intensive. It would therefore be preferable if we enable our security assurance approach to cope automatically with the fact that systems will evolve at run-time, and still provide valid run-time security assurance. This is non-trivial to achieve: As the implementation or the used libraries evolve, the instrumentation may no longer guarantee the correct link to the protocol design. It is therefore important to have a way to perform refactoring steps in a traceable way.

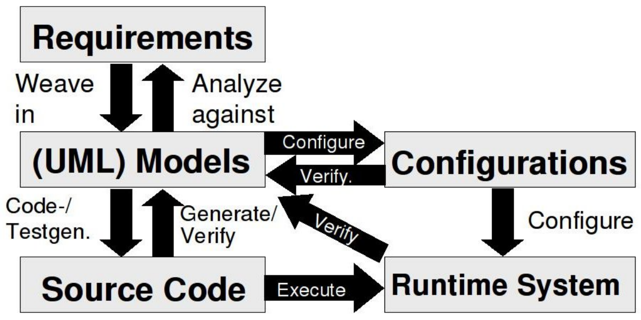


**FIGURE 1.** Model-based Security Engineering

In addition, we explain how to achieve security hardening through systematic instrumentations. As security vulnerabilities are often scattered throughout the implementation, we choose aspect-oriented programming (AOP) for security hardening.

One goal of our work is thus to maintain traceability between the design and the implementation of a crypto-based software through dedicated software refactoring approach which supports system evolution.

Note that an alternative approach could aim to generate complete implementations out of cryptographic protocol specifications, rather than establishing a link between the specification and an existing implementation, and hardening that implementation if necessary. If that would be possible, that would automatically also update the link between the specification and the implementation whenever the specification is changed, by just generating a new implementation. However, this is not our goal here. Rather, we would like the approach we develop here to be applicable to existing legacy implementations, rather than generating new implementations. The reason for this is that, in practice, there is often a strong desire to use a particular existing implementation. For example, that implementation might be conformant with certain standards or certifications, or satisfy stringent performance requirements (which an implementation automatically generated from a specification would usually not be able to satisfy). Since legacy implementations are usually too complex to verify statically, this again motivates the use of run-time verification and our approach.

One should note that our approach, at this point, focusses on a certain class of attacks which can be detected when observing the running implementation at a certain degree of abstraction, namely those attacks that rely on an interaction of the attacker with the protocol participants where the passive or active (man-in-the-middle) attacker can read, memorise, insert, change, and delete message parts into the communication between the protocol participants. In each case, we assume the actual cryptographic algorithms and their implementations (such as encryption and digital signature) to be secure, and we aim to detect insecurities in the way they are used in the context of a cryptographic protocol. We do not aim to detect attacks that rely on breaking these assumptions, such as statistical attacks or type confusion attacks.

### 1.1. A Brief Overview of the Approach

Let us briefly summarise the approach taken in this paper. Our approach supports the following steps:

**(1)** security protocols are specified and verified using the security extension UMLsec of UML,

**(2)** an implementation is linked to this UML model,

**(3)** temporal logic formulae are derived from the UML model,

**(4)** a security monitor is generated automatically from the temporal logic formulae created in step (3) in order to verify the implementation at run-time,

**(5)** the relation between the UML model and the code is maintained as the implementation evolves over time,

**(6)** errors in the implementation can be corrected using AOP, and

**(7)** the security monitor is updated with respect to the changes arising from step (6).

There are practical considerations why such a process is not fully automated, but rather has to be semi-automated; that is, some manual work is required and may be desired to have full control over the system as it evolves over time. However, steps (4) to (7) can be fully automated and are thus repeatable, given that the specifications from (1) to (3) are established manually. The time and effort spent on steps (1) to (3) can be considered as an overhead to the normal software development process, while steps (4) to (7) save the effort to accommodate changes in the evolving system. Moreover, it helps us in maintaining traceability links as this happens.

Note that our approach is interesting to apply not only to legacy systems (where there is often no alternative to manually re-engineering a specification, and static verification of the software is often not an option because of its complexity). It is also useful to apply our approach in a situation where model-based development techniques are used to develop a system: Experiences from practice indicate that, in such a context, changes are often done on the code level after the development of the model has been finished, which often means that the code becomes inconsistent with the model, which makes it necessary to monitor the code at run-time.

Our approach thus supports verified traceability that is robust under evolution at various stages of the system life-cycle:

- Verified traceability from security requirements to design: one includes security requirements as annotations into UML models and automatically verifies the models against these requirements (see Section 2).
- Verified traceability of security requirements from design to execution time: using run-time verification (see Section 3).
- Verified traceability of security requirements from one version of the implementation to another through system evolution (see Section 4).
- Traceable security hardening for code-level security vulnerabilities (see Section 5).

### 1.2. Advance over Prior State of the Art

In this section, we explain in which respect the work presented in this paper constitutes an advance over the prior state of the art in this field.

*New methodology:* We have developed a new integrated methodology for run-time security run-time verification of cryptographic protocol implementations that can handle system evolution.

Prior to our work there existed, to the extent of our knowledge no approach to security run-time verification of cryptographic protocols, and even less an approach that would be able to handle evolution. There exist other approach for run-time verification of security properties but to the extent of our knowledge they have not been applied to implementations of cryptographic protocols, which pose particular challenges for run-time verification that have thus not been addressed by other approaches.

In particular, we developed a new approach for model-based security assurance that covers properties from the design level all the way down to the implementations.

*Advance over prior work:* Some but not all parts of the methodology build on prior work, although that prior work needed to be further developed significantly in order to be applicable to run-time security monitoring of cryptographic protocol implementations that can handle system evolution, since neither of the prior work was able to deal with this task on the whole.

With regards to run-time verification, because of the particular challenges involved with run-time verification of cryptographic protocols (as opposed to other security-critical software), we have to develop a specialised approach based on the 3-valued run-time verification approach presented in [11]. Our discussion of which properties are monitorable by this particular run-time verification approach will demonstrate that this technique has significant advantages when applied to run-time verification of cryptographic protocol implementation. The approach has been implemented in terms of [68] by the first author of this paper. The implementation, which we also use in this paper, is available as open source software via a SourceForge web site.

Although refactoring and aspect-oriented programming are two well-known subjects and are well-supported by the integrated development environments, we are to the best of our knowledge the first to use refactoring to create and maintain traceability for secure software development that enables the use of AOP techniques to fix a security bug due to traceability mistakes.

The contribution of this work is then to show how these techniques can be combined in an efficient manner, and how they can be used in the context of developing security critical systems.

In this paper, we focus specifically on cryptographic protocols, since these are a compact yet highly security-critical and non-trivial to design part of a secure system, and thus serve as a particularly good example to demonstrate our approach.

*Significant practical applications* We have applied this new methodology in a significant new application to the SSL protocol implementations JESSIE and JSSE, which are industrial strength implementation with a large user base (in particular in the case of JSSE which is part of the standard Java security architecture).

More precisely, we demonstrate the approach by an application to the Java-based implementation JESSIE of the Internet security protocol SSL. We also explain how the traceability link can be transformed to the official implementation of the Java Secure Sockets Extension (JSSE) that was recently made open source by Sun.

Again, run-time verification of widely used crypto-protocol implementations such as JESSIE and JSSE have to the extent of our knowledge not been attempted so far, and they pose particular challenges since these implementations are significantly complex. In particular, this application allowed us to detect a previously unknown security vulnerability in one of the implementations, which was then hardened using our approach.

*Comparison to previous work* The work presented in this paper is new: although there has been a lot of work on formally verifying abstract specifications of cryptographic protocols, the only prior work on run-time verification for cryptographic protocols is (to our knowledge) the precursory conference paper [48], which however did not include support for automated security hardening, and for maintaining the verification results when the system evolves.

From a broader point of view, the goal of this work is to allow the use of formally based verification techniques (such as automated theorem provers and run-time verification) in practice by encapsulating them in an industrially accepted development approach (based on UML models) and apply them to an industrially used programming language (Java). We hope to thus contribute to dealing with the challenges faced when trying to use formal methods in a practical environment (cf. [35, 18, 65] for relevant discussions).

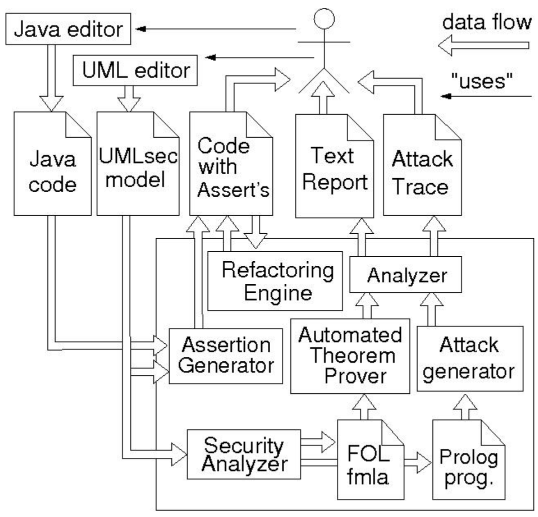


**FIGURE 2.** MBSE tool framework

The approach presented here has to be seen in the context of other approaches to model-based security based on UML developed over the last few years (see [42] for a more complete overview). There are also many other relevant approaches to model-based assurance of security-critical systems which are not based on UML, such as [58, 71]. The work presented here differs from that in that it is based on a modelling notation routinely used in industry today to facilitate uptake in practice, and that it includes a link to implementation level security assurance. Also related are several approaches to formally verifying implementations of cryptographic protocols developed recently, such as [46, 31, 15]. The current work is different in that it does not verify the implementation directly against security properties, but verifies specification models against security properties, and then verifies the implementation against the models with a focus on the security properties, using techniques including run-time security verification. The motivation for this two-step verification process is to facilitate application to complex legacy software.

See Section 6 for a more detailed comparison to previous research.

## 1.3. Outline of the Rest of the Paper

The rest of this paper proceeds as follows. In Section 2, we give an overview of model-based security engineering as a means of analysing *models* of security-critical systems in the UMLsec specification notation at design-time using first-order logic (FOL) theorem proving. We also explain how this technique was applied to the SSL protocol. Then, in Section 3, we discuss run-time verification as a dynamic verification technique in more detail, how to obtain run-time security properties of a system, and apply this approach to our case study, a Java implementation of the SSL protocol, JESSIE. As such we discuss the link between model and code of a system. In Section 4, we give a detailed account on how to maintain the links between models and code, in the face of system evolution, e.g., occurring program changes due to fixing bugs, or extending the functionality of a program. That is, we establish mappings between elements of our system models and code, and use automated refactoring techniques for changing an implementation. Finally, in Section 5, we outline an AOP-based approach that allows us to react upon security weaknesses detected in an implementation by *security hardening*.

Related work is discussed Section 6 and we draw conclusions from our work in Section 7.

## 2. MODEL-BASED SECURITY ANALYSIS

In this section, we give an overview of the part of our approach that applies to the specification level of a cryptographic protocol. We start by giving a general overview of the approach we use there (called model-based security engineering), then explain the relevant part of that approach in technical detail, and finally apply it to our running example, the SSL protocol.

### 2.1. Model-based Security Engineering

Model-based Security Engineering [42, 43, 17] provides a soundly based approach for developing security-critical software where recurring security requirements (such as secrecy, integrity, authentication and others) and security assumptions on the system environment can be specified either within a UML specification, or within the source code as annotations (cf. Figure 1). Various analysis plug-ins in the associated UMLsec tool framework [45, 66] (Figure 2) generate logical formulae formalising the execution semantics and the annotated security requirements. Automated theorem provers and model checkers are used to try to automatically establish whether the security requirements hold. (Note that security requirements in general are undecidable, so there may be worst-case examples which cannot be decided automatically, although in our experience most practical applications are unproblematic.) If not, a Prolog-based tool automatically generates an attack sequence violating the security requirement which can be examined to determine and remove the weakness. Thus we encapsulate knowledge on prudent security engineering and make it available to developers who may not be security experts. Since the analysis that is performed is too sophisticated to be done manually, it is also valuable to security experts.

Note that some of the activities contained in Figure 1 are done manually or supported with pre-existing tools outside the UMLsec tool suite, and therefore the relevant workflows do not appear in Figure 2. For example, to generate Java code from UML models (or vice versa) one can use the commercial tool suite Borland Together [67].

Part of the Model-based Security Engineering (MBSE) approach is the UML extension UMLsec for secure systems development which allows the evaluation of UML specifications for vulnerabilities using a formal semantics of a simplified fragment of the UML [40, 41, 42]. The UMLsec extension is given in form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate the security requirements and assumptions. *Constraints* give criteria that determine whether the requirements are met by the system design, by referring to a precise semantics of the used fragment of UML. The security-relevant information added using stereotypes includes security-relevant information covering the following aspects:

- Security assumptions on the physical system level, for example the stereotype ⟪encrypted⟫, when applied to a link in a UML deployment diagram, states that this connection has to be encrypted.
- Security requirements on the logical level, for example related to the secure handling and communication of data, such as ⟪secrecy⟫ or ⟪integrity⟫.
- Security policies that system parts are required to obey, such as ⟪fair exchange⟫ or ⟪data security⟫.

In each case, the assumptions, requirements, and policies are defined formally and precisely in [42] on the basis of a formal semantics for the used fragment of UML. We do not repeat these definitions here, since in this paper we will look at one specific security analysis scenario, where the assumptions and requirements are defined precisely at the level of the used formalisation in first-order logic.

The UMLsec tool-support (illustrated in Figure 2) can then be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [66, 43]. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes. The semantics for the fragment of UML used for UMLsec is defined in [42] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces and UML-type communication mechanisms. On this basis, important security requirements such as secrecy, integrity, authentication, and secure information flow are defined. To support stepwise development, one can show secrecy, integrity, authentication, and secure information flow to be *preserved* under refinement and the composition of system components. The approach also supports the secure development of layered security services (such as layered security protocols). See [42] for more information on the above.

### 2.2. Analysing Cryptographic Protocols

In the current paper, we concentrate on applying model-based security engineering to the special case of cryptographic protocols which are a particularly interesting target

| | |
|---|---|
| $\mathsf{enc}_{E'}(E)$ | (encryption) |
| $\mathsf{dec}_{E'}(E)$ | (decryption) |
| hash(E) | (hashing) |
| $\mathsf{sign}_{E'}(E)$ | (signing) |
| $\mathsf{ver}_{E'}(E,E'')$ | (verification of signature) |
| kgen(E) | (key generation) |
| inv(E) | (inverse key) |
| conc(E,E') | (concatenation) |
| head(E) and tail(E) | (head and tail of concat.) |

**FIGURE 3.** Abstract Cryptographic Operations

since they are compact pieces of highly security-critical software which are nevertheless highly non-trivial to design and implement correctly.

Using UML sequence diagrams, each message in a cryptographic protocol is specified by giving the sender, the receiver, the message, and possibly a precondition (in equational first-order logic (FOL)) which has to be fulfilled so that the message is sent out.

As usual in the formal analysis of cryptographic software, the cryptographic algorithms (such as encryption and decryption) are viewed as abstract functions. Our aim in this paper is not to verify the implementation of these algorithms, but we work on the basis of the assumption that these are correct, and aim to verify whether they are used correctly within the cryptographic protocol implementation.

We assume a set **Keys** of encryption keys disjointly partitioned in sets of *symmetric* and *asymmetric* keys. We fix a set **Var** of *variables* and a set **Data** of *data values* (which may include *nonces* and other secrets). The *algebra of expressions* **Exp** is the term algebra generated from the set **Var** ∪ **Keys** ∪ **Data** with the operations given in Figure 3. There, the symbols $E$, $E'$, and $E''$ denote terms inductively constructed in this way. Note that encryption $\mathsf{enc}_{E'}(E)$ is often written more shortly as $\{E\}_{E'}$, and that we sometimes use a specific notation $\mathsf{symenc}_{E'}(E)$ for symmetric encryption (although these alternative notations are both "syntactic sugar" without impact on the formalisation). In this term algebra, we impose the following equations, formalising the fact that decrypting with the correct key gives back the initial plain-text, and similarly for verification of signatures: $\mathsf{dec}_{K^{-1}}(\mathsf{enc}_K(E)) = E$ (for all $E \in$ **Exp** and $K \in$ **Keys**) and $\mathsf{ver}_{E'}(E,E'') = \mathsf{true}$ (for all $E \in$ **Exp** and $K \in$ **Keys**). We also assume the usual laws regarding concatenation, **head**(), and **tail**(), and that $K = K^{-1}$ for any symmetric encryption key $K$.

A cryptographic protocol can then be verified for the relevant security requirement such as secrecy and authentication using the UMLsec tools presented above, which rely on a translation from the UMLsec sequence diagram to a security-sensitive interpretation in FOL-based on the Dolev-Yao attacker model as explained in [43], which is then verified using automated theorem provers for FOL. The idea is here that an adversary can read messages sent over the network and collect them in his knowledge set. The adversary can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements can then be formalised using this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary.

We now explain how to analyse the UMLsec specification by making use of our translation from cryptographic protocols specified as UML sequence diagrams to FOL formulae which can be processed by the automated theorem prover e-SETHEO [63]. The formalisation automatically derives an upper bound for the set of knowledge the adversary can gain. The usage of the FOL generation explained in the following is complementary to the model-level security analysis mentioned above: Although using the approach described earlier one can make sure that the specification is secure, this does not imply that the implementation is secure as well, since we cannot make any assumptions on how it was constructed (as we would like to deal in particular with legacy implementations such as OpenSSL). The FOL-based approach described in the following therefore has the goal to verify the UML sequence diagram against the given security requirements such as secrecy.

The idea is to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know $E$ during the execution of the protocol. For any data value $s$ supposed to remain secret as specified in the UMLsec model, the FOL formalisation will thus compute all scenarios which would lead the attacker to derive $\mathsf{knows}(s)$.

The FOL rules generated for a given UMLsec specification are defined as follows. For each publicly known expression $E$, one defines $\mathsf{knows}(E)$ to hold. The fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows (including the use of encryption and decryption) is captured by the formula in Figure 4.

For our purposes, a sequence diagram is essentially a sequence of command schemata of the form *await event e – check condition g – output event e'* represented as *connections* in the sequence diagrams (where $e$ is a variable of the type **Exp** defined above and $e'$ is a term which evaluates to a value of type **Exp**). Connections are the arrows from the life line of a source object to the life line of a target object which are labelled with a message to be sent from the source to the target and a guard condition that has to be fulfilled.

Suppose we are given a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ in a sequence diagram with $\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$, where the parameters $arg_i$ of the guard and the message are variables which store the data values exchanged during the course of the protocol. Suppose that the connection $l'$ is the next connection in the

$$\begin{aligned}&\forall E_1,E_2.\big(\mathsf{knows}(E_1)\wedge\mathsf{knows}(E_2)\Rightarrow\mathsf{knows}(E_1::E_2)\wedge\mathsf{knows}(\{E_1\}_{E_2})\wedge\ \mathsf{knows}(\mathsf{sign}_{E_2}(E_1))\big)\\&\qquad\wedge\big(\mathsf{knows}(E_1::E_2)\Rightarrow\mathsf{knows}(E_1)\wedge\mathsf{knows}(E_2)\big)\wedge\big(\mathsf{knows}(\{E_1\}_{E_2})\wedge\mathsf{knows}(E_2^{-1})\Rightarrow\mathsf{knows}(E_1)\big)\\&\qquad\wedge\big(\mathsf{knows}(\mathsf{sign}_{E_2^{-1}}(E_1))\wedge\mathsf{knows}(E_2)\Rightarrow\mathsf{knows}(E_1)\big)\end{aligned}$$

**FIGURE 4.** FOL rules for attacker knowledge generation

$$\begin{aligned}\mathsf{PRED}(l)=\quad&\forall exp_1,\dots,exp_n.\big(\mathsf{knows}(exp_1)\wedge\dots\wedge\mathsf{knows}(exp_n)\wedge\ cond(exp_1,\dots,exp_n)\\&\qquad\Rightarrow\mathsf{knows}(exp(exp_1,\dots,exp_n)\wedge\ \mathsf{PRED}(l'))\big)\end{aligned}$$

**FIGURE 5.** FOL rule for attacker interaction

sequence diagram with $\mathsf{source}(l')=\mathsf{source}(l)$. For each such connection $l$, we define a predicate $\mathsf{PRED}(l)$ as in Figure 5. If such a connection $l'$ does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Figure 5.

The formula formalises the fact that, if the adversary knows expressions $exp_1,\dots,exp_n$ validating the condition $cond(exp_1,\dots,exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1,\dots,$ $exp_n)$ in exchange, and then the protocol continues. This way, the adversary knowledge set is approximated from above (e.g. one abstracts away from the message sender and receiver identities and the message order). In particular, one will find all possible Dolev-Yao type attacks on the protocol, but execution traces may also be generated that are not actually executable for a valid implementation. This has however not been a problem in practical applications of the approach.

For each object $O$ in the sequence diagram, this gives a predicate $\mathsf{PRED}(O)=\mathsf{PRED}(l)$ where $l$ is the first connection in the sequence diagram with $\mathsf{source}(l)=O$. The axioms in the overall FOL formula for a given sequence diagram are then the conjunction of the formulae representing the publicly known expressions, the formula in Figure 4, and the conjunction of the formulae $\mathsf{PRED}(O)$ for each object $O$ in the diagram. The conjecture, for which the automated theorem prover will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value $s$ is to be kept secret, the conjecture is $\mathsf{knows}(s)$. An example is given in the next section.

## 2.3. Application to SSL

We have applied the approach to the core part of the SSL 3.0 handshake protocol given in Figure 6 together with the open source Java implementation JESSIE (http://www.nongnu.org/jessie) of the Java Secure Socket Extension as will be presented as a running example throughout this paper. SSL is the de-facto standard for securing http-connections and is therefore an interesting target for a security analysis. It may be interesting to note that early versions of SSL (before becoming a "standard" renamed as TLS in RFC 2246) had been the source of several significant security vulnerabilities in the past [1]. In order to simplify the exposition, we concentrate on the fragment of SSL that uses RSA as the cryptographic algorithm and provides server authentication (there is no specific reason why we chose this particular fragment, and we concentrate on a fragment just to simplify the explanations). The protocol participants (here the instances C of class Client and S of class Server) are represented by vertical boxes, and the messages between them are represented by arrows. A logical expression next to an outgoing arrow is the guarding constraint that needs to be checked by the relevant protocol participant before the message is sent out. The assignments specified below the model in Figure 6 describe how the data that is received should be used by the receiving instance. Here the expression $\mathsf{arg}_{i,n,p}$ corresponds to the $p$th element of the $n$th message sent by the object instance $i$. For example, $R_S$‘:=$\mathsf{arg}_{S,1,1}$ means that the random number $R_S$, which was sent by the server in the message ServerHello, is stored in the variable $R_S$‘ at the Client, after receiving the message. In the guards, the local designations are used. The guard [ver(cert$_S$)] means that the certificate X509Cert_s previously received from the server must be verified. The guards [md5$_S$' = md5 $\wedge$ sha$_S$' = sha] and [md5$_C$' = md5 $\wedge$ sha$_C$' = sha] express the condition that the hash values of the instance which receives a Finished message have to agree with the hash values of the other instance. K is the symmetric session key which is created separately at each of the protocol partners, making use of the pre-master secret PMS. The values md5 and sha used as message arguments are created by the sender of the respective message by using the MD5 respectively SHA hash algorithm over the message elements received so far. ExchangeData represents the communication of data over the established channel once the handshake protocol is finished and also has an associated guard. For simplification, we specify the encryption of a compound message as the concatenation of the encryptions of the separate message elements (for example Finished(symenc$_K$(md5), symenc$_K$(sha)) rather than Finished(symenc$_K$(md5::sha))); we assume that type

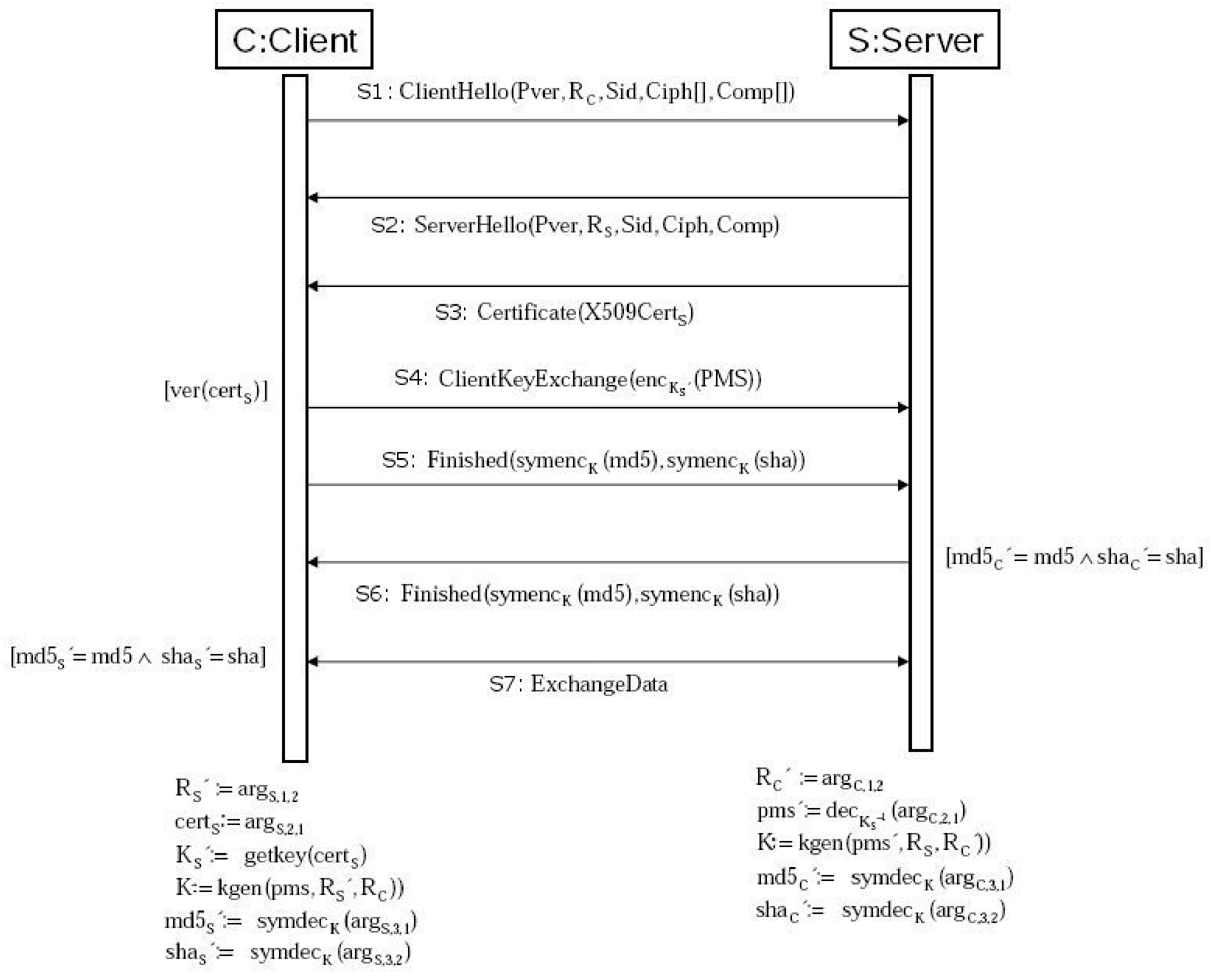

C:Client
S:Server
S1 : ClientHello(Pver, R_C, Sid, Ciph[], Comp[])
S2: ServerHello(Pver, R_S, Sid, Ciph, Comp)
S3: Certificate(X509Cert_S)
[ver(cert_S)]
S4: ClientKeyExchange(enc_{K_S'}(PMS))
S5: Finished(symenc_K (md5), symenc_K (sha))
[md5_C' = md5 ∧ sha_C' = sha]
S6: Finished(symenc_K (md5), symenc_K (sha))
[md5_S' = md5 ∧ sha_S' = sha]
S7: ExchangeData
R_S' := arg_{S.1.2}
cert_S' := arg_{S.2.1}
K_S' := getkey(cert_S)
K := kgen(pms, R_S', R_C))
md5_S' := symdec_K (arg_{S.3.1})
sha_S' := symdec_K (arg_{S.3.2})
R_C' := arg_{C.1.2}
pms' := dec_{K_S^{-1}} (arg_{C.2.1})
K := kgen(pms', R_S, R_C'))
md5_C' := symdec_K (arg_{C.3.1})
sha_C' := symdec_K (arg_{C.3.2})

**FIGURE 6.** Handshake protocol of SSL3 using RSA and Server Authentication

or message confusion attacks are ruled out using the usual protocol design rules not under investigation here.

We used the UMLsec tools to verify the UMLsec model of the SSL protocol (cf. Figure 6) against relevant security requirements such as secrecy. Verifying secrecy of a value *s* can be done by checking whether the statement knows(s) is derivable from the FOL formulae generated from the protocol specification. In each case, the properties were proved within less than a minute, e.g., the verification of the secrecy of the master secret communicated in the SSL protocol took 2 seconds.

## 3. LINKING MODELS TO CODE

We now explain how to link the formally verified specification to a crypto-based implementation which may not be trustworthy (for example, it might have been implemented insecurely from a secure specification, either maliciously or accidentally), in a way that enforces the security of the running system. That is, we use (online) run-time verification (cf. [22, 10, 53]; see also Section 3.1 for a detailed overview of this technique) to check whether or not the implementation conforms to our formal security properties while it executes. We currently focus on Java as the implementation language.

### 3.1. Run-time Verification using LTL

In a nutshell, run-time verification is a *formal* but *dynamic* technique to establish whether or not an executing system adheres to a predefined property (or a set thereof), by monitoring whether the system satisfies the property while it is used. Properties are typically specified in a temporal logic, such as LTL [57], and the object under scrutiny is the actual system and not its representation in terms of an abstract model or code as is the case with a static technique.

As such, run-time verification bears not only strong resemblance to testing since both techniques are dynamic, as already pointed out in the introduction, but also to rigorous formal verification methods such as (LTL-) model checking (cf. [20]), for instance. The idea of (LTL-) model checking is roughly as follows. A model of the system under scrutiny is checked against a formal correctness property, usually specified in terms of a temporal logic such as LTL, by verifying that all possible executions specified by the model adhere to the specified behaviour by the property. However, depending on the temporal logic used, the complexities of model checking range from polynomial in the size of the system model and property to PSpace-complete in the formula as is the case for LTL, which we are concerned with in this paper. While model checking has been successfully employed, e.g., for checking models of protocols (cf. [36]), using it to verify software in terms of low-level source code abstractions is still an active research subject due to the large state-spaces that result from using low-level models extracted from source code as compared to high-level behavioural models such as sequence diagrams or state machines (cf. [7, 30]). The advantage of model checking, however, is that once correctness has been established, we can be sure that the specified behaviour does, indeed, adhere to the intended behaviour—all possible executions have been checked. If model checking a system fails, then usually the model checker returns a counterexample in terms of a system execution that leads to the violation of the temporal logic property. In such a case, the system can be repaired with respect to the counterexample and perhaps model checked again.

Run-time verification (cf. [22, 10]) is similar to the above in the sense that it also employs, usually, a formal correctness property, specified in temporal logic to capture either intended behaviour (i.e., when one is interested to detect occurrence of a certain "good" behaviour), or unwanted behaviour (i.e., when one is interested in detecting when something "bad" has happened). However, model checking is a static verification technique as it operates on the model-level, whereas run-time verification operates directly on the system implementation. Moreover from a formal point of view, let $\mathcal{L}(M)$ be the language generated by some system model and $\mathcal{L}(\varphi)$ be the language of some formal property, $\varphi$. Then, model checking translates to checking whether or not the formal language generated by the system is *contained* in the formal language generated by the property, i.e., whether $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$ holds. Run-time verification, on the other hand, asks for the answer of a *word problem*: Let $u$ be the prefix of some potentially infinite word $w$, which resembles the system's behaviour, then we want to know whether or not $w \in \mathcal{L}(\varphi)$ after reading $u$. Or, in other words, we want to know, after seeing the finite sequence of behaviour $u$, whether or not for all possible extensions of $u$, our property will be satisfied, violated, or neither. Note that the last case simply means we have to wait for more behavioural observations until we can give a conclusive answer to this question. (For a more formal account on this form of run-time verification, see Sections 3.1.2 and 3.1.4.).

From a methodological point of view, in run-time verification, a so-called *monitor*, whose task it is to observe the system behaviour as it executes, is automatically generated from a security property (or a set thereof) formalised as an LTL formula, also referred to as an LTL property. This process is somewhat similar to constructing finite automata from regular expressions [3], which are also a formal means to define sequences of actions, i.e., system behaviour. If a monitor detects a violation of the security property it raises an alarm, if it detects that a security property was fulfilled it signals accordance, and otherwise keeps monitoring the executing system. Unlike regular expressions, temporal logic and, in particular, LTL-based temporal logic, has established itself in the area of formal verification and is nowadays frequently used also in industry to define the behaviour of systems (cf. [28]).

### *3.1.1. Definitions and Notation*

In what follows, we briefly recall some formal definitions regarding LTL and introduce the necessary notation. First, let $AP$ be a non-empty set of *atomic propositions*, and $\Sigma := 2^{AP}$ be an *alphabet*. Then infinite words over $\Sigma$ are elements from $\Sigma^\omega$ and are abbreviated usually as $w, w', \ldots$. Finite words over $\Sigma$ are elements from $\Sigma^*$ and are usually abbreviated as $u, u', \ldots$. The notion of infinite words makes sense when we consider the system under scrutiny being a reactive system, where the assumption is that the system is never switched off, and the words as a means to model the observable behaviour of that system. In run-time verification, however, we always observe only the prefix of a potentially infinite behaviour, hence we need a reasonable interpretation for LTL formulae over finite words as well. More specifically, our monitors adhere to the semantics introduced in [11] and realised by the open source monitor generator in [68]. It is explained also in Section 3.1.4.

We will adopt the following terminology with respect to monitoring LTL formulae. We will use the propositions in $AP$ to represent atomic system *actions*, which is what will be directly observed by the monitors introduced further below. As an example, an action may correspond to a specific function call, or a specific message that is sent or received by a participant in a protocol. This depends somewhat on the property being monitored, and the application at hand. A more comprehensive example is discussed in Section 3.3. Note also that, by making use of dedicated actions that notify the monitor of changes in the system state, one can also indirectly use them to monitor whether properties of the system state hold. Thus, we can use the terms "action occurring" and "proposition holding" synonymously. We will refer to a set of actions as an *event*, denoting the fact that certain actions may have occurred simultaneously, or that a certain state holds, described by a set of actions.

### *3.1.2. LTL Syntax and Semantics*

The set of LTL formulae over $\Sigma$, written $\mathrm{LTL}(\Sigma)$, is inductively defined by the following grammar:

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, \quad p \in AP.$$

The *semantics* of LTL formulae is defined inductively over its syntax as follows. Let $\varphi, \varphi_1, \varphi_2 \in \mathrm{LTL}(\Sigma)$ be LTL formulae, $p \in AP$ an atomic proposition, $w \in \Sigma^\omega$ an infinite word, and $i \in \mathbb{N}$ a position in $w$. Let $w(i)$ denote the $i$th element in $w$ (which is a set of propositions).

The (infinite word) *semantics* of LTL formulae is then defined inductively by the following logical statements.

$$
\begin{array}{lll}
w,i \models \mathit{true} & & \\
w,i \models \neg\varphi & \Leftrightarrow & w,i \not\models \varphi \\
w,i \models p & \Leftrightarrow & p \in w(i) \\
w,i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w,i \models \varphi_1 \vee w,i \models \varphi_2 \\
w,i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w,k \models \varphi_2 \wedge \\
 & & \forall i \leq l < k.\ w,l \models \varphi_1 \\
 & & \text{(``}\varphi_1 \text{ until } \varphi_2\text{'')} \\
w,i \models \mathbf{X}\varphi & \Leftrightarrow & w,i+1 \models \varphi \quad \text{(``next } \varphi\text{'')}
\end{array}
$$

where $w,i$ denotes the $i$th position of $w$. We also write $w \models \varphi$, if and only if $w,0 \models \varphi$, and use $w(i)$ to denote the $i$th element in $w$ which is a set of propositions, i.e., an event. (Notice the difference between $w,i$ and $w(i)$.)

Intuitively, the statement $w,i \models \varphi$ is supposed to formalise the situation that the event sequence $w$ satisfies the formula $\varphi$ at the point when the first $i$ events in the event sequence $w$ have happened. In particular, defining $w,i \models \mathit{true}$ for all $w$ and $i$ means that *true* holds at any point of any sequence of events.

Further, as is common, we use $\mathbf{F}\varphi$ as short notation for $\mathit{true}\mathbf{U}\varphi$ (intuitively interpreted as "eventually $\varphi$"), $\mathbf{G}\varphi$ short for $\neg\mathbf{F}\neg\varphi$ ("always $\varphi$"), and $\varphi_1\mathbf{W}\varphi_2$ short for $\mathbf{G}\varphi_1 \vee (\varphi_1\mathbf{U}\varphi_2)$, which is thus a weaker version of the $\mathbf{U}$-operator. For brevity, whenever $\Sigma$ is clear from the context or whenever a concrete alphabet is of no importance, we will use LTL instead of LTL($\Sigma$).

### 3.1.3. *Examples*

We give some examples of LTL specifications. Let $p \in AP$ be an action (formally represented as a proposition). Then $\mathbf{GF}p$ asserts that at each point of the execution of any of the event sequences produced by the system, $p$ will afterwards eventually occur. In particular, it will occur infinitely often in any infinite system run.

For another example, let $\varphi_1, \varphi_2 \in$ LTL be formulae. Then the formula $\varphi_1\mathbf{U}\varphi_2$ states that $\varphi_1$ holds until $\varphi_2$ holds and, moreover, that $\varphi_2$ will eventually hold. On the other hand, $\mathbf{G}p$ asserts that the proposition $p$ always holds on a given trace (or, depending on the interpretation of this formula, that the corresponding action occurs at each system update).

### 3.1.4. *Finite-Word Monitor Semantics*

To see how our monitors cope with the situation that at run-time only prefixes of potentially infinite words are observable, we also outline the semantics employed by the monitors, which is slightly different from the above LTL semantics, but based on it. Notably, it is a 3-valued semantics and defined as follows. Let $\varphi \in$ LTL, and $u \in \Sigma^*$. Then, a monitor for $\varphi$ returns the following values for a processed $u$, written $[u \models \varphi]$:

$$
[u \models \varphi] := \begin{cases} \top, & \text{if for all } v \in \Sigma^\omega \text{ we have } uv \models \varphi \\ \bot, & \text{if for all } v \in \Sigma^\omega \text{ we have } uv \not\models \varphi \\ ?, & \text{otherwise.} \end{cases}
$$

The $[\cdot]$ is used to separate the 3-valued monitor semantics for $\varphi$ from the classical, 2-valued LTL semantics introduced above.

In other words, this definition says that a monitor which was generated for a formula $\varphi$ will, upon reading some prefix $u$, return $\top$ if for all possible extensions of $u$ the infinite word semantics is fulfilled (i.e., $uv \models \varphi$), and $\bot$ if for all possible extensions of $u$ the infinite word semantics is violated (i.e., $uv \not\models \varphi$). Moreover, if there exists an extension $v'$ to $u$ such that $uv' \in \varphi$, and there exists another extension $v''$ such that $uv'' \notin \varphi$, then the monitor returns ? and keeps monitoring until $u$ is long enough to allow for a conclusive answer (i.e., $\top$ or $\bot$).

Such conclusive prefixes are also referred to as good (respectively bad) prefixes (cf. [10]) with respect to the monitored language that is given by $\varphi$. From that point of view, we can say that a monitor detects good (respectively bad) prefixes for the monitored property. Note, however, that not all properties that can be formalised in LTL necessarily have such a good or a bad prefix. Therefore, monitoring is often restricted to so called safety properties, where violations can be detected via bad prefixes (cf. [59]). In contrast, our monitoring procedure is not restricted to safety properties alone, but also to properties that lie outside this language-theoretic categorisation. For a more detailed comparison between the approach discussed in [59] and our monitoring framework, see [10].

## 3.2. Linking Cryptographic Protocol Models to Code

In this section, we explain how to approach the problem of creating a link between the cryptographic protocol model and its implementation.

Note that our aim is not to provide a fine-grained formal refinement from the specification to the code level. Such a refinement would require a formal behavioural semantics both of the model and the implementation of the protocol. Although such semantics exist in principle for our modelling notation (UMLsec) as well as the implementation language (Java), their treatment requires several chapters (in [42]) respectively even an entire book on its own (such as [62]). Such a treatment would exceed the goals of the current work.

Fortunately enough, for our purposes it is not necessary to construct a fine-grained refinement relation, but it is sufficient to create a link between the points in the specification and the code where a message is received, where the required cryptographic check is performed, and where the next message is sent out (as explained below). Since our goal is to use run-time verification, rather than static verification of the code, we only need to consider these points in the code, and therefore do not depend on a full formal semantics for all of Java: instead of referring to a static semantics of Java, we will refer to a given, concrete execution trace at run-time, and with respect to that, we only need to consider the messages that are received and sent out, and make sure that the necessary checks are performed in between. Indeed, this is one of the advantages in using run-time verification compared with static verification. We only need to know which library functions need to be called to receive or send messages from or to the network, and be able to determine whether the required cryptographic checks have been performed in between. The link between the relevant points in the model and the implementation is defined formally, although, as argued above, it is sufficient to do this on a syntactic (rather than semantic) level. An example for that is given in the next section in Table 1.

There is a distinct advantage, from a practical point of view, to work with a relatively abstract specification model, which is directly linked by a mapping to the implementation level: when the implementation changes (which usually happens quite frequently during the lifetime of a piece of software like a cryptographic protocol), this minimises the amount of changes that have to be done at the model level, but as far as possible localises the necessary changes to the model-code mapping itself. This is a practical advantage, in so far as the problem of keeping a model in synch with the changing code base is one of the major impediments to a larger update of rigorous model-based development approaches in practice.

As explained above, the cryptographic algorithms are viewed as abstract functions. In our application here, these abstract functions represent the implementations from the Java Cryptography Architecture (JCA). The messages that can be created from these algorithms are then as usual formally defined as a term algebra generated from ground data such as variables, keys, nonces, and other data using symbolic operations. These symbolic operations are the abstract versions of the cryptographic algorithms. Note that the cryptographic functions in the JCA are implemented as several methods, including an object creation and possibly initialisation. Relevant for our analysis are the actual cryptographic computations performed by the digest(), sign(), verify(), generatePublic(), and generatePrivate() methods which correspond to the abstract operations $\mathsf{hash}(E)$, $\mathsf{sign}_{E'}(E)$, $\mathsf{ver}_{E'}(E,E'')$, $\mathsf{kgen}(E)$ from Figure 3. Encryption and decryption are implemented in the JCA using the functions nextBytes() (encrypting or decrypting the next bytes of a message, depending on context), and doFinal() (finalising the encryption or decryption process). As mentioned above, our goal is not to provide a precise representation of the cryptographic generation process from the code level on the model level, but only to compare the values that were created at the points where they are received from or sent to the network.

First, we need to determine how important elements at the model level are implemented at the implementation level. This can be done in the following three steps:

- Step 1: Identification of the data transmitted in the sending and receiving procedures at the implementation level.
- Step 2: Interpretation of the data that is transferred and creation of a mapping to the relevant elements in the sequence diagram.
- Step 3: Identification and analysis of the cryptographic guards at the implementation level.

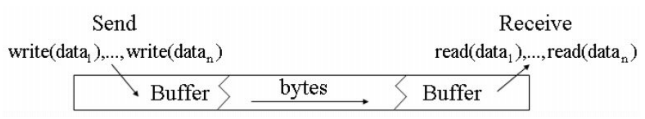

Send
write(data$_1$),...,write(data$_n$)
Receive
read(data$_1$),...,read(data$_n$)
Buffer
bytes
Buffer

**FIGURE 7.** Communication in the SSL protocol

In step 1, the communication at the implementation level is examined and it is determined how the data that is sent and received can be identified in the source code, with the goal to relate it to the model level. Afterwards, in step 2, a meaning is assigned to this data. The interpreted data elements of the individual messages are then linked to the appropriate elements in the model. In step 3, it is described how one can identify the guards from the model in the source code with the goal to ensure that the guards specified in the sequence diagram are correctly implemented in the code.

To be able to determine the data that is sent and received, it first needs to be identified at which points in the implementation messages are received and sent out, and which messages these exactly are. To be able to do this, we exploit the fact that in many implementations of cryptographic protocols, message communication is implemented in a standardised way (which can be used to recognise where messages are sent and received). The common implementation of sending and receiving messages in cryptographic protocols is through message buffers, by writing the data into type-free streams (ordered byte sequences), which are sent across the communication link, and which can be read at the receiving end. The receiver is responsible for reading out the messages from the buffer in the correct order in storing it into variables of the appropriate types. We assume that each message is represented by a message class (as done in many implementations such as JESSIE or JSSE). It stores the data to be written in the communication buffer. Conversely, this class can also read messages from the communication buffer (this communication principle is visualised in Figure 7). We found that this mechanism is implemented at the class level using the methods write() (for sending messages), and read() (for receiving them). Furthermore, the occurrences of the method write() (respectively, read()) which are called at the class java.io.OutputStream (respectively, java.io.InputStream) are used to identify the individual message parts within the communication procedure in the form of parameters that are delivered or the assignments made.

In the next subsection, we will explain how the ideas explained above were used in the application to the SSL Implementation JESSIE.

### 3.3. Security Monitoring the SSL Implementation JESSIE

We now explain how we applied run-time verification to the implementation of the Internet security protocol SSL in the project JESSIE, which is an open source implementation of the Java Secure Sockets Extension (JSSE). JESSIE 1.0.1 has 27271 lines of uncommented code in Java (measured using the `sloccount` utility).

First, we explain how we applied the approach for linking cryptographic protocol models to code (as explained in the previous section) to the case of JESSIE.

In our particular protocol, setting up the connection is done by two methods: doClientHandshake() on the client side and doServerHandshake() on the server side, which are part of the SSLsocket class in jessie − 1.0.1/org/metastatic/jessie/provider. After some initialisations and parameter checking, both methods perform the interaction between client and server that is specified in Figure 6. Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake(), respectively doServerHandshake(), methods.

As explained above, communication is implemented as follows: With the method call msg.write(dout, version), the message msg is written into the output buffer dout. Each occurrence of such a method call can be identified and associated with the specification of sending a message in a UMLsec sequence diagram (by an outgoing arrow from the life line of the sender). The method call dout.flush later flushes the buffer. The assignment msg = Handshake.read reads a message from the buffer during the handshake part of the protocol. As an example, the code fragment for initialising and sending the ClientHello message is given in Figure 8.

In order to be able to construct a link between the implementation with the abstract model, we must first determine for the individual pieces of data how they are implemented on the code level. For example consider the variable randomBytes written by the method ClientHello to the message buffer. By inspecting the location at which the variable is written (the method write(randomBytes) in the class Random), we can see how exactly the value of randomBytes is defined. In particular, the contents of the variable depends on the initialisation of the current random object and thus also on the program state. Thus we need to trace back the initialisation of the object. In the current program state, the random object was passed on to the ClientHello object by the constructor. This again was delivered at the initialisation of the Handshake object in SSLSocket.doClientHandshake() to the constructor of Handshake. Here (within doClientHandshake()), we can find the initialisation of the Random object that was passed on. The second parameter is generateSeed() of the class SecureRandom from the package java.security. This call determines the value of randomBytes in the current program state. Thus the value randomBytes is mapped to the model element $R_C$ in the message ClientHello on the model level. For this, java.security.SecureRandom.generateSeed() must be correctly implemented.

In the case of the SSL protocol, we had to link the symbols in its UMLsec specification in Figure 6 to their implementation in JESSIE version 1.0.1. To illustrate this,

**TABLE 1.** Mapping messages from symbols to program entities

| Symbols | Program entities |
|---|---|
| 1. $C$ | clientHello |
| 2. $S$ | serverHello |
| 3. $P_{\text{ver}}$ | session.protocol version |
| 4. $R_C$<br>$R_S$ | clientRandom<br>serverRandom |
| 5. $S_{\text{id}}$ | sessionId |
| 6. Ciph[ ] | session.enabledSuites |
| 7. Comp[ ] | comp |
| 8. Veri | Lines 1518–1557 |
| 9. $D_{\text{nb}}$<br>$D_{\text{na}}$ | getNotBefore()<br>getNotAfter() |

Table 1 presents nine example instances of this mapping. The first column shows the names of symbols as used in the cryptographic protocol model. The second column shows the names of corresponding program entities in the JESSIE library. Here one can also see that in general there does not need to be a one to one correspondence between the design and the code. For example, the design symbol Veri is implemented by a code fragment spread out over several lines of the code.

We now explain in particular how one can use run-time verification to increase one's confidence that the implementation adheres to the security properties previously demonstrated at the model and code levels. Note that our goal is not to provide a full formal verification of the correctness of the implementation against the specification, but to raise one's confidence in its security by demonstrating that certain particularly security-relevant parts (such as the checking of cryptographic certificates) are securely included into the implementation context. However, as discussed at the beginning of the last section, run-time verification can provide a higher level of assurance for crypto-based software than for example model-based testing, since full test coverage is in general not achievable for highly interactive and complex software like cryptographic protocols.

According to the information that is contained in a sequence diagram specification of a cryptographic protocol, the run-time verification needs to keep track of the following information:

(1) *Which data is sent out?* and
(2) *Which data is received?*

The run-time checks will enforce that the relevant part of the implementation conforms to the specification in the following sense.

(1) *The code should only send out messages that are specified to be sent out according to the specification and in the correct order*, and
(2) *these messages should only be sent out if the conditions that have to be checked first according to the specification are met.*

```
ClientHello clientHello = new ClientHello(session.protocol, clientRandom,
        sessionId, session.enableSuites, comp, extensions);
Handshake msg = new Handshake(Handshake.TYPE.CLIENT_HELLO, clientHello);
msg.write(dout, version);
```

**FIGURE 8.** Initialising and sending the CLIENT_HELLO message

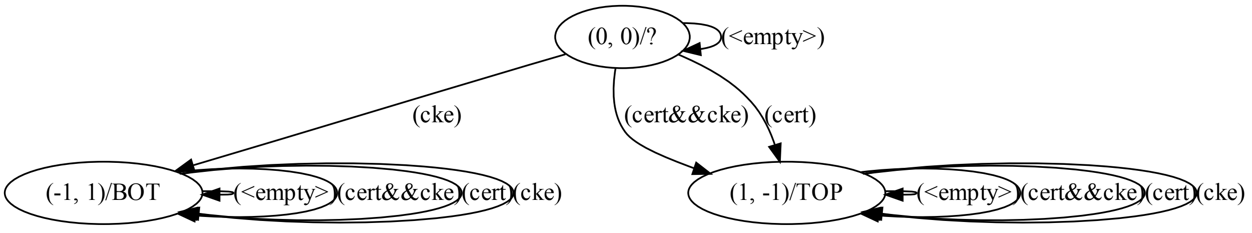

(0, 0)/?
(<empty>)
(cke)
(cert&&cke)
(cert)
(-1, 1)/BOT
(<empty>)(cert&&cke)(cert)(cke)
(1, -1)/TOP
(<empty>)(cert&&cke)(cert)(cke)

**FIGURE 9.** Automatically generated FSM for the property

An example of such a property in the case of the SSL-protocol specified in Figure 6 is given by the following requirement that arises from the above discussion:

"ClientKeyExchange$(enc_K, (PMS))$ is not sent by the client until it has received the Certificate$(X509Cer_s)$ message from the server, has performed the validity check for the certificate as specified in Figure 6, and this check turned out to be positive."

Next, we explain how to capture such a requirement using LTL. Together with Figure 6, this requirement gives rise to the following set of atomic propositions:

$$AP := \{\text{ClientKeyExchange}(enc_K, (PMS)), \text{Certificate}(X509Cer_S)\},$$

whose names correlate with the ones displayed in Figure 6. Notice that LTL as introduced above does not cater for parameters. Therefore, parameters in an action's name are not a semantic concept, but merely syntactic sugar to ease readability and establish a link with the names used in Figure 6. The link from symbol names to the actual names used in the monitor, and finally in the implementation code of JESSIE, is also exemplified by Table 5. Based on $AP$ we can now formalise the required property in LTL as follows:

$$\varphi := \neg\text{ClientKeyExchange}(enc_K, (PMS)) \; \mathbf{W}\text{Certificate}(X509Cer_S).$$

The formula uses the "weak until" operator, which in particular allows for the fact that if the certificate is never received, then the formula is satisfied if in turn the message ClientKeyExchange$(enc_K, (PMS))$ is never sent. This meets our intuitive interpretation of the "until" in the natural language requirement because if, for example, a man-in-the-middle attacker deletes any certificate message sent by the server, we cannot possibly demand that ClientKeyExchange$(enc_K, (PMS))$ should be eventually sent by the client. The derived monitor will later signal the value $\top$ ("property satisfied") once the certificate was received and checked, $\bot$ ("property violated") if the client sends the key without a successful check, and it will signal the value ? ("inconclusive") as long as neither of the two conditions holds. Recall that the stream of events that is processed by the monitor consists of elements from $2^{AP}$ (i.e., the powerset of all possible system actions). That is, at each point in time, the monitor keeps track of *both* events: the sending of ClientKeyExchange$(enc_K, (PMS))$ and the receiving of Certificate$(X509Cer_S)$. Hence, as long as none of the events is observed, the monitor basically processes the empty event.

Once we have formalised the natural language requirements in terms of LTL formulae as above, we can then use the tool from [68] to automatically generate finite state machines (FSMs) from which we derive the actual (Java) monitor code. The FSMs obtained from the tools are of type Moore, which means that, in each state that is reached, they output a symbol (i.e., ?, $\top$ (TOP), $\bot$ (BOT), or ?). States are changed as new system actions become visible to the monitor. In that sense, the states keep track of the context in which new actions are to be interpreted. For example, there may be an action, such as the sending of a secret key, which constitutes a security violation in one context, but is a necessary and desired action in another context, e.g., as part of a protocol. The monitor's states keep track of the current context, and the reaching of a new state means reaching a new context in which to interpret future actions. The FSM generated for the run-time security property $\varphi$ is given in Figure 9.

The initial state is $(0,0)$ whose output is ?. If event $\{cert\}$ occurs, short for $\{\mathsf{Certificate}(X509Cer_S)\}$, then the monitor takes a transition into state $(1,-1)$ and outputs $\top$ to indicate that the property is satisfied. On the other hand, if neither *cert* nor *cke*, short for $\mathsf{ClientKeyExchange}(enc_K,(PMS))$, occurs, then the automaton remains in $(0,0)$ and outputs ? anew, indicating that so far $\varphi$ has not been violated, but also not been satisfied. A violation would be the reaching of $(-1,1)$, if event $\{cke\}$ occurs (before *cert*), such that the monitor would output $\perp$. Here is an example run of the client which first yields ? as output for three time steps until, finally, $\top$ is returned in the fourth because the message was sent but the certificate also received and checked:

$$u := \langle \{\}, \{\}, \{\}, \{cert, cke\} \rangle.$$

At this point this particular monitor may stop monitoring for the remaining session. On the other hand, consider the following run:

$$u' := \langle \{\}, \{\}, \{\}, \{cke\} \rangle.$$

This run is indicative that the client has attempted to send $\mathsf{ClientKeyExchange}(enc_K,(PMS))$ prematurely, resulting in the monitor returning $\perp$ in the fourth time step. In formal terms, for all $v \in \Sigma^\omega$ we have $uv \models \varphi$ (i.e., $[u \models \varphi] = \top$), and for all $v \in \Sigma^\omega$ we have $u'v \not\models \varphi$ (i.e., $[u' \models \varphi] = \perp$). On the other hand, if we shortened $u$ and $u'$ by one observation, we obviously would have $[u \models \varphi] = ?$ and $[u' \models \varphi] = ?$.

Further properties as the ones above, which are monitorable in this particular application, are also discussed in [10]. The relationships between symbol names used in these specifications, the monitor FSMs, and the code are then given in Table 5 on page 21.

*A Note on Efficiency* The monitors we generate for each security property are *minimised* in a sense that we find a smallest possible state machine which corresponds exactly to the language of the security property by exploiting the well-known Myhill-Nerode equivalence relation between states of a finite automaton (cf. [37]). The latest version of our monitor generation tools [68] perform this minimisation and thus return the smallest monitor possible for a given language. In other words, it is not possible to find a smaller monitor without altering the monitored language, i.e., the generated monitors are *optimal*. Therefore, the efficiency of the proposed method solely depends on the respective security property chosen, i.e., the formal language it gives rise to and the means by which the state machines are implemented for the application at hand. Specifically, efficiency depends on

(1) the number of states in the monitor, which is the smallest number possible by the Myhill-Nerode equivalence,
(2) the time it takes to accept a system action and to change state in the monitor, and
(3) the time it takes for the monitor to emit the corresponding output symbol.

However, if the monitor contains only a very small number of states, as was the case in our examples, then it is very difficult to effectively measure items 2 and 3 in the above list of items, because they require only microseconds (or less) and exact measurements in these ranges can only be obtained reliably using real-time operating systems. However, due to the monitors being optimal in the above sense, we have a guarantee that the run-time overhead is minimal, which, indeed, resulted in no noticeable performance changes in our application. It may, however, be the case that for very involved specifications to be monitored in other application domains, that there is a noticeable overhead and that, indeed, additional resources are necessary to facilitate this technique. After all, the monitors are of worst-case exponential size with respect to the specification, and sometimes the worst case cannot be avoided, which is particularly limiting when the formula was already of a large size to begin with. Our experiences with the given application, however, did not reveal such cases, which leads us to believe that in many practical situations the worst-case behaviour can be avoided. Moreover, after minimalisation, the state-space of the generated monitors was $<= 10$ states, which is a good indication for how efficiently this method can be implemented.

Notice also that run-time verification is a method which scales well, in a sense that the size of the system under scrutiny does not impact on the efficiency of the method. Run-time verification operates on a concrete behaviour, whereas static verification techniques like model checking or ones which try to establish correctness with the help of a theorem prover, as we have laid them out in Section 2, explore the overall state-space of a system imposed by a model representation of it. Naturally, as system sizes increase, it affects the efficiency of such techniques, whereas run-time verification stays constant.

## 4. MAINTAINING TRACEABILITY UNDER EVOLUTION

There are two kinds of traceability associated with our use of run-time verification. First, as discussed in Section 3, we use it as a tool to trace high-level security properties beyond code and down to the actual execution level. Second, we have to tackle traceability within our run-time verification framework itself as security properties and the code change, which may affect the generated monitors. In this section, we focus mainly on the second kind of traceability with regard to run-time verification; that is, we examine how our LTL properties and monitors are affected by changes of the high-level security properties and of the implementation code.

We now explain how to maintain the traceability link constructed using the approach explained in the previous section in the presence of code evolution. We then explain how to apply the approach for run-time security verification explained in the previous section in this situation.

## 4.1. Evolution as Code Refactoring

Software *refactoring* [54] by definition changes the internal structure of an implementation without changing its externally observable behaviour. By this definition, refactoring transformations are program transformations that preserve externally observable program behaviour. Note that therefore transformations that change the externally observable behaviour of a program are beyond the definition of refactoring, and thus not considered in the following. Modification of the behaviour due to refactoring would be considered a bug of the refactoring engine that needs to be fixed eventually.

In practice, the refactoring engine in programming IDEs implements a subset of possible refactoring transformations which are commonly used in programming activities, such as renaming, extracting methods, etc.

For example, the general refactoring engine in Eclipse is provided by a set of plug-ins called the refactoring Language Toolkit (LTK)[70], which allows one (1) to perform refactoring operations, (2) to save the history of refactoring operations into an XML-based script, and (3) to apply a refactoring script automatically. The plug-ins are applicable to any programming or specification language. The Java Development Tool (JDT), for example, instantiates LTK with a number of Java-specific refactoring operations. Rather than refactoring Java source, another refactoring tool in the Plugin Development Environment (PDE) instantiates LTK with a number of refactoring operations specific for the plug-in metadata.

We use refactoring scripts to maintain traceability between a design and its evolving implementations. Modern IDEs such as Eclipse support refactoring by automated scripts, allowing users to perform, record and replay refactoring steps as if they were basic editing operations. The advantage over traditional editing scripts is that refactoring scripts preserve the externally observable behaviour of the program. Otherwise, Eclipse would reject the execution of an operation that might change the behaviour. For example, renaming class field x to y will change behaviour if there is already a local variable y in some method(s), because the renamed references to x will now become references to the local variable. This is carefully excluded by Eclipse.

However, such basic refactoring support is inadequate for our purpose, namely to maintain traceability between changing code bases. For example, adding or deleting a single space can make the *extract.method* (see below) operation inapplicable. To enhance reusability of refactoring operations regarding such kind of code changes, we extended the Eclipse Refactoring Language Toolkit (LTK) using a new approach to make the operating context of refactoring more tolerant to changes. To ease specifying these refactoring operations, we also implemented a utility to convert refactoring scripts saved from Eclipse into our specification language.

```
/* $workspace/abc/src/abc.java */
public class abc {
    public void main2(String args[]) {
        System.out.println("Hello");
    }
}
——— Step 1. rename.type ———
/* $workspace/abc/src/hello.java */
public class hello { ... }
——— Step 2. extract.method ———
public void main2(String args[]) {
   print_hello();
}
private void print_hello() {
   System.out.println("Hello");
}
——— Step 3. extract.temp ———
   String string = "Hello";
   System.out.println(string);
——— Step 4. rename.method ———
public class hello {
   public void main(String args[]) {
    print_hello();
   }
   private void print_hello() {
    String string = "Hello";
    System.out.println(string);
   }
}
```

**FIGURE 10.** A running example illustrates refactoring

*An Illustrative Example* To illustrate Java refactoring, Figure 10 shows a running example specific to Eclipse JDT, where a series of refactoring operations are applied to a small "Hello World" program.

Assume that initially the source file abc.java is located at a source folder src in the project abc. A series of refactoring operations are applied as follows. Step 1: The class abc is renamed to hello and abc.java is also renamed to hello.java, accordingly. This refactoring operation is called *rename.type*. Step 2: The statement System.out.println is extracted into the body of a new method print_ hello(). This operation is called *extract.method*. Step 3: The expression "Hello" is explicitly assigned to a new local variable string. This operation is called *extract.temp*. Finally, Step 4: The method main2 is renamed to a new method name main. This last operation is called *rename.method*.

After performing the above refactoring operations in Eclipse one can save the history into a refactoring script. Such a script can be automatically applied on the original code again to replay the changes. Figure 11 shows a snippet from the refactoring script in XML format. It briefly specifies the *rename.type* and *extract.method* operations used in the first two steps.

Every refactoring is recorded as an XML element refactoring, whose attributes specify the operation. Every operation has an identifier ID, indicating the type of the operation. Here, org.eclipse. jdt.ui.rename.type is the internal name used by JDT for *rename. type* refactoring. For readability, we omit the common prefix in the following and call it rename.type. The target of a refactoring operation for rename.type is a new class name, whereas the target for extract.method is a new method name. They are completely specified by the name

```
<?xml version=”1.0”?>
<session version=”1.0”>
<refactoring comment=”...”
   id=”org.eclipse.jdt.ui.rename.type”
   description=”Rename type ’abc’”
   project=”abc” input=”/src&lt;abc.java[abc”
   name=”hello” ... />
<refactoring comment=”...”
  description=”Extract method ’print_hello’”
  id=”org.eclipse.jdt.ui.extract.method”
  project=”abc” input=”/src&lt;{hello.java”
  name=”print_hello” selection=”64 28”
  ... />
...
</session>
```

**FIGURE 11.** Operations of Eclipse refactoring script (cf. Figure 10)

attribute. On the other hand, the source of a refactoring operation is suggested by attributes including project, input and optionally selection. The values of these attributes typically indicate the context of an operation. The project attribute specifies the subject project of the refactoring operation; the input attribute specifies the source folder, package and class name in which the source element is refactored; the selection attribute, when used, specifies the exact offset and length of the string selected for the refactoring.

In our example the extract.method refactoring is applicable only if the selection of a substring of 28 characters starting from the offset 68 in hello.java matches the statement to extract, character by character. Given such strict specifications of refactoring contexts in Eclipse, we can see that existing refactoring scripts are inadequate if source code has been modified by evolution or by previously applied refactoring operations, or when source code from a different library implementation is used. For example, it is required to modify the offset/length value if an extract.temp operation was applied earlier.

### 4.2. Maintaining Model-Code Traceability

In this subsection, we explain how to maintain traceability between a UMLsec specification of a cryptographic protocol and its implementation while the code evolves (cf. Figure 12).

We present our new refactoring engine that overcomes the limitation of the native Eclipse JDT refactoring operations, while making the refactoring operations reusable for maintaining design traceability in different legacy code.

Specifically, we need to map any symbolic name $S$ that appears in the design model to an identifier $I$ on the implementation level.

Refactoring scripts are used for maintaining such traceability: they guarantee that the externally observable behaviour of the program is preserved as far as expressed in the traceability links to the model level. We can apply the mapping in a round-trip fashion:

(1) to convert the program entities to names on the design level and

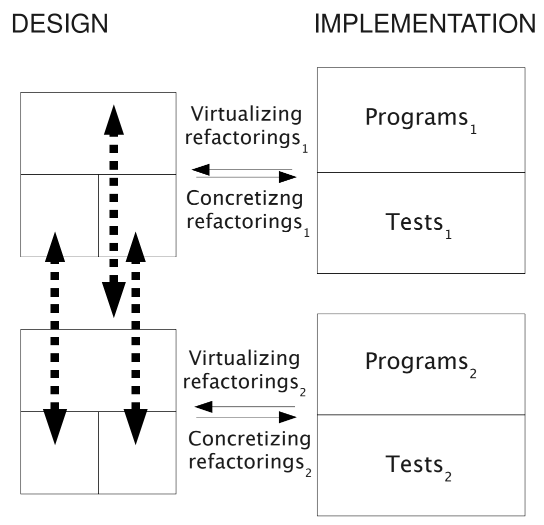

DESIGN
IMPLEMENTATION
Virtualizing refactorings$_1$
Concretizng refactorings$_1$
Programs$_1$
Tests$_1$
Virtualizing refactorings$_2$
Concretizing refactorings$_2$
Programs$_2$
Tests$_2$

**FIGURE 12.** Traceability for reuse

(2) to convert the names on the design level to names in the implementation.

When a relation between a symbol $S$ and an entity $I$ in the program is established, it will be maintained through a number of refactoring operations that transform every occurrence and update every reference of $I$ into $S$.

When the program entity already has an identifier in a form of class, method, field or local variable, renaming operations such as *rename.type*, *rename.method*, *rename.field* and *rename.local.variable* can be used; when the program entity does not have an associated identifier, then extracting operations such as *extract.method*, *extract.temp* and *extract.field* can be used to directly extract an identifier named by $S$.

The renaming operations, when applied in low granularity (e.g., *rename.local.variable*), are typically change sensitive as a selection offset/length is required to specify the exact context of source. The extracting operations, by definition, always need to specify the context of the source explicitly.

The mapping between symbols and program entities is not one to one. The same symbol from the design model may be implemented differently in different contexts. Therefore more than one refactoring operation can be applied to resolve the symbol names. Since a symbol may even be called differently in different parts of the program, the actual program entities have to be checked to find out whether they are the same throughout the design. Such checks must in particular ensure that the name can be differentiated by using the context of the messages.

If $S$ is a complex design element, such as a message in a message sequence chart, its mapping may at the same time require a mapping from its arguments to their corresponding identifiers. In order to create such a mapping, a sequence of basic refactoring operations needs to be performed. Therefore, such dependencies among refactoring operations need to be respected.

```
SPECIFICATION := OPERATION SPECIFICATION
OPERATION := '@' '{' NAME ',' FIELDS '}'
NAME := Identifier [ '.' NAME ]
FIELDS := FIELD [',' FIELDS]
FIELD := KEY '=' VALUE
KEY := Identifier
VALUE := String
```

**FIGURE 13.** The extended BNF for the syntax of our refactoring language

```
@{org.eclipse.jdt.ui.rename.type,
  project="abc", source="src", package="",
  class="abc", name="hello"
}
@{org.eclipse.jdt.ui.extract.method,
  project="abc", source="src", package="",
  class="hello", method="main",
  toclass="hello", name="print_hello",
  regexp="S.*(\"Hello\");",
  count="1"
}
```

**FIGURE 14.** Our specification for refactoring (cf. Figure 11)

As we have mentioned, the refactoring of *I* to *S*, when applied after other editing/refactoring operations, have to be carried out independently of the previous changes.

### 4.3. Reuse Support for Traceability Refactoring

One can reuse the traceability information discovered when linking the implementation to the UML model. For example, this can be done if one wants to apply the refactoring operations defined for one version of the implementation to a different version of that implementation, or to a different library. To this end, we create a refactoring plug-in that can apply parameterised refactoring operations[1]. Our refactoring tool is implemented on top of LTK refactoring plug-ins, which support languages beyond Java. In order to limit the changes to the existing refactoring engine, we invoke the context-specific refactoring operations in JDT by instantiating a scripting template with the parameters derived from our specifications.

In [51], Krueger classified software reusability as five connected facets: abstraction, classification, selection, specialisation and integration. Our traceability refactoring engine supports this view.

**Abstraction.** An extended BNF grammar of the abstract refactoring language is given in Figure 13.

A specification consists of one to many refactoring operations. Every operation has a name indicating the class that handles the refactoring and one to many fields. A field is a pair of a key identifier and a value string. It is up to the refactoring class to decide the concrete list of fields to be used.

Our declarative specification language abstracts away context-sensitivity of existing refactoring operations and can describe generally any refactoring operation supported by LTK, beyond Java. The refactoring context is parameterised to remove certain change-resisting dependencies (e.g., selection in Figure 11). Similar in format to that of BibTeX, a specification consists of a list of entries. Each entry is made of a list of fields, separated by a comma. The first field is a key, which matches one type of the existing refactoring operations (as in JDT). The remaining fields are in the form of name="string" pairs, where the quoted string can span multiple lines. As in Java, every quotation in the string must be escaped. Depending on the type of refactoring operations, the number of required fields may vary. The main reason why we chose this format is to support variability for recording refactoring operations.

Corresponding to Figure 11, the snippet in Figure 14 lists two refactoring operations in our specification language.

**Selection and Specialisation.** Most fields have evident meaning and usage as they correspond to the attributes in the Eclipse refactoring scripts. We introduce the new fields to compute the context (input) of the source element, such as source, package. The fields regexp and count in this specification indicate a selection to be refactored that is matching a regular expression, counted from the beginning. Our regular expression-based selection for context-sensitive refactoring operations increases the chance of reusability when changes happen to the code. In the implementation, we can actually construct a regular expression from a normal one by replacing white spaces with an arbitrary number of white spaces. In this way, even if a programmer or a code formatter inserted some indentation, the selection can still be matched. Introducing count is done mainly to be able to selectively refactor some instances of matching selection rather than the first one. When unspecified, the first matching selection will be chosen. The selection parameter is specialised from the other parameters by parsing the Java source file, searching for the method name in the given class to obtain the offset to the method in the source range, and then searching for the local variable in the source of the selected method and adding its relative offset to the method to obtain the absolute offset to the file.

**Classification.** As refactoring consists of a sequence of operations, we *classify* existing refactoring operations by context-sensitivity and discuss its impact on exchangeability and invertibility.

Context-free operations are more reusable whereas context-resistant or sensitive ones require more care. Since it is more likely to have the other parts of the code changed rather than the pattern of regular expressions, our new refactoring operation becomes less sensitive to code changes. According to our experience, when relaxed patterns are used in the regular expression, the context specification of refactoring operation is more tolerant to changes.

In practice we have found that if one performs larger-granularity refactoring operations (say, *a*) earlier

[1]These automated refactoring tools (ART), including their source code and examples in the paper, can be downloaded from the project subversion repository linked from [66].

**TABLE 2.** Refactoring operations parameterised by our refactoring tool

| ID | change resistant? | context | source selection | specified in Eclipse | our specification |
|---|---|---|---|---|---|
| org.eclipse.jdt.ui.rename.project | no | workspace | project | project | project |
| org.eclipse.jdt.ui.rename.folder | no | project | folder | folder | folder |
| org.eclipse.jdt.ui.rename.package | no | folder | package | package | package |
| org.eclipse.jdt.ui.rename.type | no | package | class | class | class |
| org.eclipse.jdt.ui.rename.method | no | class | method | method | method |
| org.eclipse.jdt.ui.move.method | no | class | method | method | method |
| org.eclipse.jdt.ui.extract.method | yes | class | statements | (offset, len) | (regexp [, count]) |
| org.eclipse.jdt.ui.rename.local.variable | yes | method | variable | (offset, len) | (regexp [, count]) |
| org.eclipse.jdt.ui.extract.local.variable | yes | method | expression | (offset, len) | (regexp [, count]) |
| ... | ... | ... | ... | ... | ... |

than smaller-granularity ones (say, $b$), the reusability of refactoring operation sequences can be increased. To allow reordering operations, side-effects of an operation on the context of another must be captured by changes to their parameters, i.e. $a \otimes b = b' \otimes a'$. For example, the two operations in our running example are not exchangeable. If one were to swap their order, one needs to accordingly apply the latter refactoring to the refactoring script of the former one. If one would apply the *extract.method* operation first, then the rename.type operation should be applied to the specification such that it is a method in the class abc rather than the class hello being extracted.

In Table 2, we list some JDT refactoring operations that have been parameterised in our refactoring engine. We also show which JDT operations are considered change resistant and a brief description on how such limitations are resolved.

**Integration.** After selection and specialisation, our tool delegates the domain-specific (here Java) refactoring integration tasks to LTK in Eclipse. We also support both *interactivity* and *transparency* for programmers to preview the effects of a refactoring if they choose to, and to avoid manually constructing the specification from the saved refactoring history in Eclipse.

The implementation of our refactoring plug-in adds two command buttons to the Eclipse GUI: one of them performs all refactoring operations automatically, while the other brings up a dialogue for each operation to preview the effects of a refactoring. This allows us to verify if there are any potential maintenance problems arising from the operation. For example, when renaming a variable to R_C, we can see a warning message from the Eclipse IDE that, by programming convention, it is not recommended to let the name of a variable start with capital letters. However, since our purpose is to facilitate the reuse of traceability in security analysis, such a renaming does not affect programmers because they can always edit the original source code.

Another utility program we implemented is a transformation that converts an XML-based refactoring script from the Eclipse IDE into our own specification language. By such a conversion, a string selected by offset and length is replaced with a regular expression and its count of its matching occurrence. For the string selected, the utility generates a regular expression with wildcards and an occurrence count such that it could match precisely with the selection string in the refactoring context (e.g., a method body), while being agnostic to the change to other parts of the program. For example, if the *extract.method* is applied to a set of statements, they will be remembered by the generated regular expression so that the method can be matched even when the other part of the method is changed.

After translation, the resulting specification is still further customisable. We also implemented a headless tool to invoke the functionality of the automated button as an RCP command. The argument of the command provides the name of a refactoring specification file.

### 4.4. The SSL Case Studies

In general, the run-time verification of a protocol like SSL should be invariant to implementation changes if the properties that are to be monitored are derived from the specification of the protocol (unless, of course, the specification of the protocol changes). In other words, if we have a security property that we want a correctly operating implementation of the SSL-protocol to adhere to at run-time, we want a modified implementation to also adhere to this property regardless of how it achieves it internally. This view asserts that run-time verification considers the system under scrutiny as a "black box".

#### *4.4.1. Evolution in the* JESSIE *Case Study*

To perform the model-based security analysis as explained above on a different version of JESSIE, one only needs to modify the specifications of the refactoring operations that provide the traceability of the model to the implementation level, without making any other adjustments to our refactoring engine. In particular, we considered the two versions JESSIE 1.0.0 (released on June 9, 2004 according to its CVS repository) JESSIE 1.0.1 (released on October 12, 2005 according to its CVS repository).

However, in the case of monitoring JESSIE, we linked the run-time verification to code elements and, therefore, are not immune to changes in the code to such an extent.

For example, in Section 3, we have defined an abstract run-time security property in LTL and, in doing so, have performed a mapping from elements in the design model (and therefore also in the LTL formula) to elements used in the monitor code. Moreover, there also exists a mapping from elements used in the monitor code to elements used in JESSIE's code. Table 5 exemplifies these mappings for three different run-time security properties (where the first is our running example, discussed in Section 3).

*Evolution* Inside the org.metastatic.jessie.provider package in JESSIE the 1.0.1 version has got 24 code block differences compared to that of 1.0.1 version. These changes cause that the selection-sensitive operations in the refactoring history script saved from Eclipse cannot be applied to JESSIE 1.0.0. After converting the script into our specification language, all of the refactoring operations (some of which are listed in Table 3) become reusable in our enhanced refactoring engine (cf. the column JESSIE 1.0.0). The only necessary change made to our original refactoring specification for JESSIE 1.0.1 was a global substitution of the project attribute for all operations from jessie-1.0.1 to jessie-1.0.0.

As part of the library release, two model-based unit tests for the message sequences in JESSIE 1.0.1 were provided: testclient.java and testserver.java. After refactoring, we were able to reuse them for the two other implementation libraries as well.

Since the "hooks" required in the code are different but conceptually the same, we focussed on only the first of the three properties in this paper given again in Table 5. All of the mentioned code can be found inside SSLSocket.java. The first column of Table 5 shows the name of an entity on the design model level as well as its symbol name in the corresponding LTL formula, the second column shows the action symbol as it is used within the generated monitor, and the last column displays the corresponding entity in the JESSIE source code. Notably, some properties share symbol names, which has to be respected by potential refactoring steps. That is, when we apply refactoring steps that affect elements in the given table, we have to make the according changes there as well to notify the run-time verification framework of the occurring changes. Internally, our refactorings reflect the links represented by that table and its crucial symbol names and code segments. This gives us a straightforward, but manageable, means to evolve our monitors along with code changes. As the properties that we monitor are fairly generic properties that are derived from the SSL-protocol specification itself, they have not changed between versions 1.0.1 and 1.0.0 of the SSL-protocol implementation JESSIE. However, this type of book-keeping does not in general give us any indication when refactorings or other code changes do *not* affect our monitors, because code which has been identified relevant to the monitors may have become "dead code" by a modification outside the scope of our book-keeping. Note however that dead code detection can typically also be automated, e.g. using static analysis (cf. [19]).

*Evolution Beyond Simple Refactoring* In order to preserve externally observable behaviour, the refactoring steps defined in previous sections represent relatively small and simple changes on the code base (e.g. consistent renaming of identifiers). However, in practice, systems often undergo more significant evolutions which may in particular not be behaviour-preserving. In these cases, the definition of the LTL formula to be monitored may have to be adapted manually to account for the system evolution. In this section, we demonstrate this in terms of an example.

We consider the situation where an initial version of a protocol implementation does not provide for dedicated error handling in the case that one of the cryptographic checks in the protocol is violated. We investigate how a monitor for such an implementation will have to evolve if the implementation is adjusted to provide dedicated error handling, to make sure that the error handling leads to a fail-safe system state. This will prevent the protocol implementation from proceeding with an insecure protocol execution, e.g. by sending out secret information even though the cryptographic checks were violated.

We therefore distinguish between the following cases:

(1) The system fails and does not reach a fail-safe state (monitor returns $\bot$).
(2) The system succeeds *or* reaches a fail-safe state, assuming an error occurred (monitor returns $\top$).
(3) Neither of the two conditions holds (monitor returns ?).

What we have done is, basically, added an *exception* to our rule, and thereby mapped two different events to one truth value, namely $\top$. This, however, is not uncommon in specifying behaviour of software and systems. For example, exceptions are incorporated into many different specification languages that are based on LTL and regular languages (cf. [28, 12, 6]). The one presented in [12], SALT, introduced the **accepton** $x$ directive for this purpose, where $x$ represents the exception.

As we are using LTL directly in this paper, we give a straightforward extension of our previously used specification (see Section 3) that caters for such an exception and demonstrates the concept:

$$\begin{aligned}\varphi_{fs} := \; & \neg\mathsf{ClientKeyExchange}(enc_K, (PMS)) \\ & \mathbf{W}(\mathsf{Certificate}(X509Cer_S) \\ & \quad \vee \mathsf{failsafe}).\end{aligned}$$

Using our monitor generator [68], it is easy to verify in terms of the resulting monitor FSM that this extension has the desired effect. Had we used the LTL meta-language SALT as introduced in [12], we could have simply added an exception as follows

$$\begin{aligned}\varphi'_{fs} := \; & (\neg\mathsf{ClientKeyExchange}(enc_K, (PMS)) \\ & \mathbf{W}\mathsf{Certificate}(X509Cer_S)) \\ & \quad \mathbf{accepton}\ \mathsf{failsafe},\end{aligned}$$

which would then have been translated into the above LTL formula. While with short formulae such as the above, it does not seem to make any difference as to whether

**TABLE 3.** Refactorings for the traceability to the protocol (cf. Figure 6)

| Messages in sequence | op. | diff | Time (sec) |
|---|---|---|---|
| S1: $C \rightarrow S : (P_{\text{ver}}, R_C, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$ | 7 | 31 | 13.891 |
| S2. $S \rightarrow C : (P_{\text{ver}}, R_S, S_{\text{id}}, \text{Ciph}[\,], \text{Comp}[\,])$ | 5 | 20 | 9.437 |
| S3. $S \rightarrow C$ : Certificate[X509Cert$_s$] | 2 | 2 | 1.474 |
| S4. $C$ : Veri(X509Cert$_s$) | 2 | 2 | 3.854 |
| ... | ... | ... | ... |
| Total of 7 messages and 3 checks | 27 | 86 | 40.303 |

meta-level constructs like **accepton** are employed, which subsequently "weave" the exception into all subformulae of a given specification, more comprehensive formulae may be difficult to specify in LTL alone and without such directives. Semantically, however, a language such as SALT is equally expressive to LTL. We therefore abstain from discussing it further in this paper as the resulting monitors are the same.

For all the 19 symbols, 7 messages and 3 checks in Figure 6, in total we have defined 27 refactoring steps in the specification to maintain the traceability between the protocol design and the JESSIE 1.0.1 code. The third column of Table 3 shows the count of changed segments by the refactoring steps. Using `diff`, each block of changes, even when they contain multiple lines, is counted as one. When the number of changed blocks is larger than the number of steps, changes have happened to more than one places on average. The last column shows the performance, i.e., how much time in seconds it took to perform the refactoring steps using our tools. Note the time required for the refactoring steps varies depending on its type and the number of occurrences in the code. For example, renaming a sessionId field into S_id took only 0.141ms whereas renaming a local variable sessionId into S_id took 1.484ms. The automatic execution of all the steps took about 40 seconds running our plug-ins inside Eclipse SDK 3.3 on a dual-core laptop (with a CPU running at 2 x 1.8GHz). Given the significant pay-off provided by the fact that the externally observable behaviour of the code is preserved during the complex refactoring steps, such performance figures do not impose a bottleneck within the overall process. On the contrary, much more time is spent on the security analysis and the manual creation of the refactoring steps, which will be paid back by reusing the scripts on different implementations.

### 4.4.2. *Maintaining Monitor-Code Traceability*

To illustrate this refactoring mapping, Table 4 presents some instances of such a mapping for our example implementation. The first column shows the names of symbols as used in the cryptographic protocol model. The second column shows the names of the corresponding program entities in the implementation. The third column shows the identifiers that are the target names of the refactoring operations. The type of the refactoring operation is shown in the last column. The implementation and execution of these refactoring operations is done using refactoring scripts. These scripts only allow a limited kind of refactoring which guarantees that the externally observable behaviour of the program is preserved (e.g. renaming identifiers in a way that is ensured not to create any conflicts). Each refactoring operation is declared as a transformation from a program entity (a collection of executable statements or declarations) to a symbolic entity which is named after the corresponding symbol in the design model. For example, the clientRandom variable is mapped to the symbol R_C in the protocol.

**TABLE 4.** Refactoring program entities in a traceable way

| Symbols | Program entities | Identif. | Refactoring op. |
|---|---|---|---|
| 1. $C$ | clientHello | C | rename.type |
| 2. $S$ | serverHello | S | rename.type |
| 3. $P_{\text{ver}}$ | session.protocol version | P_ver | extract.temp |
| 4. $R_C$ | clientRandom | R_C | rename.local.variable |
| $R_S$ | serverRandom | R_S | rename.local.variable |
| 5. $S_{\text{id}}$ | sessionId | S_id | rename.field |
| | sessionId | S_id | rename.local.variable |
| 6. Ciph[ ] | session.enabledSuites | Ciph | extract.temp |
| 7. Comp[ ] | comp | Comp | extract.temp |
| 8. Veri | Lines 1518–1557 | Veri | extract.method |

### 4.4.3. *Reusing* JESSIE *Refactoring Transformations for* JSSE

We also investigated on how to reuse the model-code traceability links for SSL from the JESSIE project for JSSE, another implementation of SSL. JSSEis part of Sun's Java Secure Sockets Extension (JSSE), a library in the standard JDK since version 1.4, released by Sun from version 1.6 onwards as an open source project called OpenJDK. Specifically, we considered JSSE 1.6 (released on May 8, 2007). The source code of the JSSE library can be checked out from its Subversion repository: https://openjdk.dev.java.net/svn/openjdk/jdk/trunk/j2se/src/share/classes/sun/security/ssl. In this case, we found that most of the refactoring operations cannot be applied as is. The doHandshake protocol is mainly implemented in the class SSLSocket of the JESSIE 1.0.1 library, whereas in the JSSE library implementation in the OpenJDK 1.6 (hereafter called JSSE 1.6), the protocol is mainly implemented in the class sun.security.ssl.HandshakeMessage. Nevertheless, the naming of the symbols can be traced to the implementation.

Table 6 lists the mappings from the symbols in Table 1 to their naming in the JSSE library. To reuse the existing refactoring operations, we have to instantiate their specifications with different parameters for its source (i.e., project, folder, package, class) and its context (i.e., regexp, count). In some cases even the type of refactoring operation needs to be changed. For example, Veri(X509Cert$_s$) is refactored by the *extract.method* operation in JESSIE (Table 4). However, to obtain the same symbol, a *rename.method* operation in JSSE is required (Table 6).

**TABLE 5.** Mapping model elements to monitor code and JESSIE (SSLSocket.java)

| Model / LTL symbol | Monitor | Concrete representation in JESSIE |
|---|---|---|
| 1: $\neg$ClientKeyExchange$(enc_K,(PMS))$ **W** Certificate$(X509Cer_S)$ | | |
| ClientKeyExchange$(enc_K,(PMS))$ | cke | `ProtocolVersion v = (ProtocolVersion) session.enabledProtocols.last(); byte[] b = new byte[46]; session.random.nextBytes (b); preMasterSecret = Util.concat(v.getEncoded(), b); EME_PKCS1_V1_5 pkcs1 = EME_PKCS1_V1_5.getInstance((RSAPublicKey) serverKex); BigInteger bi = new BigInteger(1, pkcs1.encode (preMasterSecret, session.random)); bi = RSA.encrypt((RSAPublicKey) serverKex, bi); ClientKeyExchange ckex = new ClientKeyExchange (Util.trim(bi));` |
| Certificate$(X509Cer_S)$ | cert | `Certificate serverCertificate = (Certificate) msg.getBody(); X509Certificate[] peerCerts = serverCertificate.getCertificates();` |
| 2: $(\neg$Finished$(HashMD5(\ldots$ **W** Arrayequal$(md5_s,md5_c)) \wedge ($**F** Arrayequal$(md5_s,md5_c) \Rightarrow$ **F** Finished$(HashMD5(md5_s,ms,\ldots)))$ | | |
| Finished$(HashMD5(md5_s,ms,PAD1,PAD2))$ | finished | `finis = generateFinished(version, (IMessageDigest) md5.clone(), (IMessageDigest) sha.clone(), true); msg = new Handshake(Handshake.Type.FINISHED, finis);` |
| Arrayequal$(md5_s,md5_c)$ | equal | `if (!Arrays.equals(finis.getMD5Hash(), verify.getMD5Hash()) \|\| !Arrays.equals(finis.getSHAHash(), verify.getSHAHash())) ...` |
| 3: $\neg$Data **W** Arrayequal$(md5_s,md5_c)$ | | |
| Arrayequal$(md5_s,md5_c)$ | equal | `if (!Arrays.equals(finis.getMD5Hash(), verify.getMD5Hash()) \|\| !Arrays.equals(finis.getSHAHash(), verify.getSHAHash())) ...` |
| Data | data | (Various stream read and write methods.) |

Such changes, however, do not influence the target name attribute for the operations because they are derived from the same protocol design. Modifying the refactoring specifications might seem a lot of work. However, we experienced little difficulty in applying them with the help of automated execution of the declarative refactoring specification. The benefit of such an effort is that we can reuse the model-based security test cases.

## 5. SECURITY HARDENING

Using the approach to run-time security verification explained in the previous sections, one can raise an alarm at run-time in case of a security violation, and terminate the given protocol execution, before the secret is leaked out to the network. In such a situation, it would however be even more useful if one could go a step further, and remove the security vulnerability in the implementation that has been detected in this way to make sure the same problem will not appear again. In this section, we explain how this is achieved in an approach making use of automated instrumentation techniques.

**TABLE 6.** Symbol-code mappings for JSSE

| Symbols | JSSE 1.6 |
|---|---|
| 1. $C$ | HandshakeMessage.ClientHello |
| 2. $S$ | HandshakeMessage.ServerHello |
| 3. $P_{\text{ver}}$ | protocolVersion |
| 4. $R_C$ $R_S$ | clnt_random svr_random |
| 5. $S_{\text{id}}$ | sessionId |
| 6. Ciph[ ] | cipherSuites |
| 7. Comp[ ] | compression_methods |
| 8. Veri | CertificateVerify.verify() |
| 9. $D_{\text{notBefore}}$ $D_{\text{notAfter}}$ | cert.getNotBefore() cert.getNotAfter() |

One often has to fix a vulnerability in multiple places of the code, making it difficult to maintain the changes consistently. In order to automate the vulnerability fix for

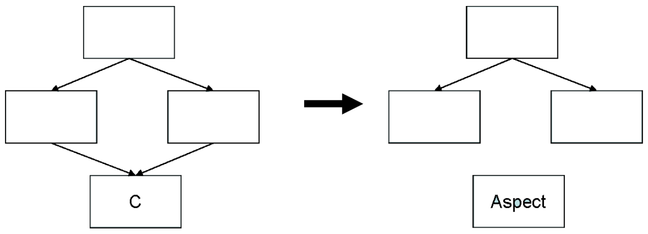


**FIGURE 15.** Comparing component-based with aspect-oriented systems in light of the inverse of control principle

security hardening, we therefore choose to apply aspect-oriented programming (AOP) techniques.

In the following subsection, we explain the basic concepts of aspect-oriented programming (AOP) in detail.

## 5.1. Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP, [50, 64]) separates crosscutting concerns that tangle the code into aspect modules. The tangled code at various control flow points (so-called "joinpoints") are encapsulated into a module when they match with the signatures of pointcut expressions. The functionality of the existing code can be altered by weaving additional statements (so-called "advices") before, after or around the existing joinpoints. AOP has advantages for maintenance as one can change the crosscutting behaviour of the system without directly modifying the source. AOP is supported in systems such as aspectJ [50] and Hyper/J [64] and fully supported in Java IDEs such as Eclipse through the AJDT project.

*AOP Principles* In a previous paper, we compared the fundamental difference in the methods reflected by component-based programming and AOP [73]. Figure 15 illustrates two modularisations to divide a problem into subproblems and to compose their solution later. In the component-based manner (left), the composition requires at various points an explicit invocation of the component module, whilst in the AOP manner (right), the aspect module has two parts: *pointcuts* and *advices*. The pointcuts are expression for the aspect module to figure out the various points that would be otherwise scattered in the components; and the advices are instrumentations that need to be weaved into the base system by *automatically* composing the advices at points (i.e., *joinpoints*) that match with the pointcuts expression. One of the major advantages of AOP is that the scattered joinpoints are modularised by the pointcut expressions, thus reducing the complexity in code. This principle is also known as *Inversion Of Control (IOC)*.

AOP can reduce the complexity given that the base system cannot be easily disentangled and the joinpoints scattered among them as crosscuts. As one often sees, similar security vulnerability are often scattered in the code thus making them good candidates for joinpoints. By weaving the advices into these scattered places, AOP can help one

```
/* HelloWorld.java */
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello␣world!");
  }
}

/* GoodbyeWorld.java */
public class GoodbyeWorld {
  public static void main(String[] args) {
    System.out.println("Goodbye,␣world!");
  }
}

/* HelloFromAspectJ.aj */
public aspect HelloFromAspectJ {
  pointcut mainMethod() :
    execution(public static void
        main(String[]));
  after() returning : mainMethod() {
    System.out.println("Hello␣from␣
        AspectJ");
  }
}
```

**FIGURE 16.** An illustrative aspectJ program

harden the security in the design and implementation of the base system.

*AOP with aspectJ* There are several implementations of AOP, among which `aspectJ` for Java is the most widely used. To convey the basic concepts of AOP, and also to explain the example used in this paper, we illustrate the syntax and semantics of the AOP language using the following illustrative example.

Two Java classes "HelloWorld" and "GoodbyeWorld" serve as the original system which an aspect "HelloFromAspectJ.aj" implements an advice that instrument the original program to print an additional message "Hello from AspectJ" (see Figure 16).

In the aspectJ module, for example, the pointcut expression `mainMethod()` matches the *main* methods in both Java classes according to the interface signature of the method. The specification of the advice introduces the boolean pointcut expression by the keyword "after". Because it matches with the two joinpoint methods in the Java classes, the statement in the body of the advice will be inserted *after* the invocation of *main* method. According to the semantics of the aspectJ language, one can also specify "before" and "around" advices. Namely, the *before* advice will be executed before the execution of the method at the joinpoint, and the *around* advice will be executed instead of the execution of the method at the joinpoint. Therefore, it is clear that AOP can completely change the behaviour of the original method, making it suitable to fix security vulnerability as opposed to the refactoring transformations.

## 5.2. Traceability under Evolution in the Presence of Aspects

The joinpoint model of AOP such as aspectJ is powerful: according to the specification an aspect specified in aspectJ can match with methods of classes. On the other hand, it does not include support for loops, super calls, throws clauses, multiple statements, etc. According to the literature [49], a joinpoint model at the method level has advantage over one at the lower statement level to ensure modularity of the code. Using aspectJ, therefore, we cannot base our solution on a statement level joinpoint model. If one wants to alter the behaviour of a group of statements, a necessary step is to perform a refactoring operation such as *extract.method*.

Another issue is that when expressed in an aspect, the pointcuts must match with names in a particular library. If one does not change the function names or naming conventions used in the pointcut expressions, the aspects can be harder to reuse for a different library. In order to improve the reusability of such security aspects, we therefore abstract away the names from the implementation by substituting them with the corresponding symbolic name in the design model. These again require refactoring operations.

Therefore to exploit the refactoring traceability, we need to make sure that the traceability-preserving refactoring also preserves the aspect-oriented joinpoints used in that approach. Thus we need to define joinpoints in terms of the symbol names and the joinpoint model in aspectJ (methods and fields). Such joinpoints must be aware of the context of the method invocations or field accesses.

When the identifiers are methods or fields, then they can already be matched by pointcut expressions in the aspects. Otherwise, more refactoring operations need to be performed to prepare for AOP instrumentations. As the joinpoint model in aspectJ does not support the instrumentation of a group of statements inside a method, for example, it is necessary to apply more refactoring operations such as *extract.method* to group these statements into a method. Having the joinpoints symbols refactored as methods and fields, they can now be used to define aspect pointcut expressions.

As long as program changes are captured by changing the refactoring scripts, one can maintain the pointcut expression unchanged. Similarly, if one wants to apply the same aspect to a different library where the symbols are implemented differently, the reusability of such security aspects eliminates the need to change the definition of the aspects. This effort for maintaining the traceability has a payoff only when a mapping can be used to express security aspects which otherwise would be non-reusable.

Since refactoring operations can improve the internal structures, these mappings can be performed selectively on the joinpoints that are made immediately useful for the aspects.

## 5.3. The SSL Case study: Fixing a Vulnerability in JESSIE

Since the places that need to get changed to fix a security vulnerability are often scattered across the code, it can be difficult and error-prone for humans to manually and consistently update the code.

We demonstrate how we use aspects for security hardening with an example from the JESSIE project.

In the JESSIE implementation, we found a significant security vulnerability as the certificate verification Veri(X509Cert_s) is not always invoked when the certificate message is received, which is an essential security check according to the protocol specification. It is needed because otherwise a man-in-the-middle attacker could insert a forged certificate containing his own public key into the communication and thereby decrypt the session key that is encrypted using that key, and thus eavesdrop on the encrypted communication in that session without being noticed by the communication partners. Therefore the current implementation of the SSL protocol in the JESSIE project does not enforce its security requirements. Below, we explain how this vulnerability arises and how one can use our approach to insert additional checks into the protocol implementation to harden its security.

Table 7 highlights the vulnerability by showing the execution log of four different test cases. In this table, the eight steps on the handshake protocol message sequence chart are shown by the rows. The second column shows the code corresponding to these steps that has been tested by the test cases. The third column highlights the differences in the instances of the four test cases.

If the certificate was checked at step S4, in Cases 3 and 4, the cheVal should report false in a correct implementation. However, we found they reported true instead.

Additional checks can be inserted into the protocol to harden its security. For example, using an aspect to crosscut every joinpoint of the program where a certificate is received, we found nothing is called by the program to check the issuing date. Therefore we find it is necessary to instrument the program with the functionality to check validity of the certificate against its date range issued by OpenSSL. Interestingly, this functionality was defined in JESSIE as a utility method checkValidity() in X509CertBridge.java. However it was never called, as indicated by a warning message in Eclipse.

Besides fixing the vulnerability by weaving an aspect into the refactored code we can also apply it to the implementation of the original program: After renaming checkValidity to cheVal, the aspect in Figure 17 is enabled to insert an additional check on the validity of certificate date (cheVal). Also, the refactored Veri is called right *after* a certificate is obtained through the pointcut expression certificate(). Without these refactoring operations, this aspect cannot be weaved through the original program.

This aspect whose design is derived from the protocol design model introduced earlier assumes the existence of a method for Veri. This method is created from the given

**TABLE 7.** Test cases for assessing security

| Seq. | Tested Code | Example Test Case |
|---|---|---|
| S1 | C = **new** ClientHello(P_pre, R_C, S_id, Ciph, Comp); | Case1: ClientHello(TLSv1, clientRandom1, [B@b012a558, enabledSuites1, zlib) Case2−4: ClientHello(TLSv1, clientRandom2, [B@b01b0558, enabledSuites2, zlib) |
| S2 | S.ServerHello(P_ver, R_S, S_id, Ciph, Comp); | Case1: ServerHello(TLSv1, serverRandom1,[B@b0134ed8, TLS_DHE_RSA_WITH_AES_256_CBC_SHA, zlib) Case2−4: ServerHello(TLSv1, serverRandom2,[B@b01baed8, TLS_DHE_DSS_WITH_AES_256_CBC_SHA, zlib) |
| S3 | C.Certificate(X509Cert_s) | Case1−4: Certificate(serverCertificate) |
| S4 | cheVal(D_notBefore, D_notAfter) | Case1,2: cheVal((107,2,2),(108,3,2))==True Case3: cheVal((107,2,1),(107,3,1))!=False Case4: cheVal((107,2,3),(107,3,1))!=False |
| S5 | Ver_K_CA(Sig) | Case1−4: sigVerity((1.2.840.113549.1.1.5 Signature)) |
| S6 | clientKeyExchange(ckex) | Case1−4: ClientKeyExchange(ckex1) |
| S7 | S.finished(md5_C, sha_C) | Case1−4: finished(gnu.java.security.hash.MD5@b00d27f8, gnu.java.security.hash.Sha160@b00d2f78) |
| S8 | C.finished(verifyData) | Case1−4: finished("6a:df:3d:90:ec:0b:33:bc:2d:ce:ef:aa") |

implementation by extracting 58 lines of code from the doClientHandshake method into a new public method Veri in the SSLSocket class. The extracted Veri method is then called in the advice to reimplement the already existing check. In addition to this check, we introduced an additional cheVal method into the aspect module. After weaving in this aspect, the date validity check is performed before the existing certificate check.

Using the test aspect, we were able to detect that the certificate() pointcut crosscuts three call sites with different argument settings (see accordingly the wildcard signature call(* C.certificate(..)) defined for this pointcut in our aspect definition above). One of them is without any arguments, whilst the other two are instantiated with arguments. From the execution log, we found all are executed after weaving our security aspect. However, if our aspect is not woven in (i.e. in the original JESSIE implementation), the original library only invokes the function of Veri when certificate is called without argument. In other words, the aspect has placed the check on all obtained certificates whilst the existing implementation misses some of them, which clearly results in a significant security vulnerability as explained earlier.

When weaving in the security aspect at the JSSE implementation, we could determine that it did not further harden the security for JSSE beyond the existing implementation since the security check implemented in the aspect is already correctly enforced in JSSE. This is confirmed by the logs of the two test cases that were reused. These test cases also helped us to verify that the messages are sent and received in a way that is consistent with the sequence diagram in Figure 6.

### 5.4. Continuous Integration

Continuous integration[26] has been adopted by our process where the regression test subprocess is augmented with the regressive refactoring: whenever code or model are changed in the repository – e.g., a developer committed a set of changes – the continuous integration script will check out the change set into a sandbox to conduct various automated builds and tests. Adding our refactoring scripts to the continuous integration script allows us to integrate our security assurance approach with the continuous integration framework. The error report subprocess is also augmented with an explanation of the counter-example of potential attack traces and the mismatch between the UMLsec model and the implementation code. Therefore we can incorporate a continuous integration process to make sure that whenever there is a change to the artefacts in the repository, a sequence of actions will be triggered to fully integrate the otherwise separate security tools.

For example, the usual compilation and function test steps are integrated with the additional actions in our proposed framework. Whenever there is a change in the design or in the implementation of the system, or there is a change to the refactoring scripts, the automated refactoring tool (ART) is called to check whether this causes the traceability links to be broken. If so, then the run-time verification tools will be

```
public aspect testCryptoProtocolSecurity {
  pointcut certificate():
      call(* Certificate.Certificate(..));
  Object around(): certificate() {
    X509Certificate[] X509Cert_s =
       (X509Certificate[]) proceed();
    if (! X509Cert_s.checked) {
     System.out.println("Problematic␣
         traceability␣found!");
    }
  }
}
public aspect CryptoProtocolSecurity {
  pointcut certificate():
      call(* Certificate.Certificate(..));
  Object around(): certificate() {
      X509Certificate[] X509Cert_s =
       (X509Certificate[]) proceed();
      SSLSocket s = (SSLSocket)
          thisJoinPoint.getThis();
      for (int m=0; m<pCs.length;m++) {
        assert cheVal(pCs[m].D_nb(),
                      pCs[m].D_na()):
          "+++␣The␣date␣is␣invalid␣+++";
      }
      s.Veri(X509Cert_s);
      return X509Cert_s;
  }
}
```

**FIGURE 17.** Aspect to check vulnerable certificates

invoked to check whether the new system still has correct traceability between design and implementation. If not, the developers will be informed to obtain a new refactoring script through further analysis.

The CruiseControl system is one of the most widely used continuous integration systems. A CI process in CruiseControl is driven by an XML-based build script for the Java-based build tool Apache Ant[2]. By default, the script would periodically monitor the designated repository for any changes. Then based on the Ant build dependencies, these changes may trigger a sequence of actions, normally including building (compilation, packaging, deploying) and testing.

We extend the CruiseControl system by adding a few more tasks to the Ant build and test scripts. A daemon process on the build/test machine periodically monitors whether there is any change to the repository. Whenever changed artifacts (including the code, the model, the test cases, the refactoring scripts and the security aspects and assurance test cases) are committed, the event triggered a run of the extended Ant build.xml script, cf. the following example:

[2]http://ant.apache.org

```
<project name="jessie"
   default="test"
   basedir="jessie">
  <target name="build" depends="refactoring"/>
  <target name="test" depends="build"/>
   // the following tasks are augmented
  <target name="umlsec"/>
  <target name="refactoring"/>
  <target name="saspect" depends="test"/>
</project>
```

The first parameter specifies an environment variable for the Eclipse *headless* build process. Since our refactoring and aspect tools have dependencies on the basic Eclipse platform and JDT, in order to run the scripts for refactoring and security aspects it is necessary to start Eclipse without GUI.

The dependencies between the targets of the build.xml are straightforward. Before one can build the new system, the modified code must be refactored such that the changes committed by the programmers are synchronised with the model. The UMLsec security check for model vulnerabilities is performed after the system is built and the refactoring is done. Based on the UMLsec model and the LTL formulae to be monitored, an updated security monitor can now be generated automatically, if required by the system changes.

Integrating with the rest of the system through continuous integration, these aspects are thus reusable whenever a change to the design or the code does not affect the traceability.

## 6. RELATED WORK

### 6.1. Formal Security Verification and Model-based Security

*Model-based Security* [25] uses UML for the risk assessment of an e-commerce system within the CORAS framework for model-based security risk assessment. This framework is characterised by an integration of aspects from partly complementary risk assessment methods. [29] proposes an extension of the i*/Tropos requirements engineering framework to deal with security requirements. [9] shows how UML can be used to specify access control in an application and how one can then generate access control mechanisms from the specifications. The approach is based on role-based access control and gives additional support for specifying authorisation constraints. [4] presents the SECTET framework for Model Driven Security which is then specialised towards a domain-specific approach for healthcare scenarios, including the modelling of access control policies, a target architecture for their enforcement, and model-to-code transformations. [72] presents an approach for the transformation of security requirements to software architectures.

In an approach for model-based development of cryptographic protocols, [56] explains how to generate "provably correct" implementations from formal models.

*Formally Verifying Cryptographic Protocol Implementations:* There have recently been some approaches towards formally verifying implementations of cryptographic protocols against high-level security requirements such as secrecy, for example [46, 31, 15].

## 6.2. Security Traceability and Maintenance

*Traceability and Model Synchronisation* Software maintenance makes use of related models at different stages of development. Example models are goal trees for requirements, UML diagrams for design and source code for implementation. When some model elements change, it is necessary to *synchronise* the change on related elements in order to maintain model consistency [38]. Existing traceability approaches aim to recover traceability links that connect elements of certain software engineering artifacts in requirements, design and implementation [5, 27, 21, 39]. Search-based techniques recover traceability links between documents and code with a precision below 100% [5, 39]; a probability-model based approaches relies on a softgoal-interdependency graph to recover traceability links between functional and non-functional requirements [21]; a scenario-driven approach generates traceability links from observations of system executions [27]. Other work on requirements tracing includes [61]. In general, none of them can recover accurate requirements traceability links. Though efficient techniques have been proposed to account for incremental update of traceability links recovered from search-based approaches, these incrementally maintained traceability links are still inaccurate [39]. Graph transformation-based techniques [38] may accurately trace structural semantics, yet another mechanism is required to trace behavioural semantics.

*Reverse Engineering* Existing reverse engineering frameworks were proposed to improve accuracy of traceability for reference architecture [55] and for known design patterns [14]. In our previous work [74], refactoring was proposed to enable accurate abstraction of behavioural implementations such that they can be compared to the goal-oriented requirements. In this work, refactoring is not only used for comparing the source and target, but also for transforming the source into the target.

*Refactoring Scripts* Dig et al. [23] first studied the evolution of component APIs that can be replayed as refactoring steps. They argued that the refactoring of library components may indeed change the behaviour of the overall system especially when the client of the components are not refactored accordingly. For example, a function 'foo' may be renamed to 'bar' in the library, yet the call site of the function may still try to invoke 'foo', only to find broken contracts. Therefore, it is useful to keep track of (or detect in Dig's case) the refactoring steps as a script such that they can be replayed at the client side. Our tool supports tracking refactoring steps by translating the refactoring steps recorded by the IDE into change resilient refactoring specifications. Comparing with [23]'s work, our use of refactoring is not for replaying the changes, rather for maintaining the traceability between design elements and implementation regardless of changes. Though the RefactorCrawler tool [23] cannot be used directly, we can make use of the refactoring preview dialog code in the MolhadoRef tool [24].

*Refactoring for Aspects* In [32, 52], specialised refactoring actions are defined mainly for aspect-orientation. In this work, we expand the scope to any general-purpose refactoring steps supported by existing tools. We have exploited the opportunity to perform aspect-oriented instrumentation in order to harden the security that require general-purposed refactoring actions. In [16], Binkley et al. proposed a number of aspect-aware refactoring transformations to convert object-oriented programs into aspect-oriented ones. If the design element is implemented by crosscutting code, then Binkley et al.'s technique may be applied to our work to maintain the traceability between such elements. Since refactoring alone does not change the behaviour of the system, aspects derived from such refactoring transformations must not change the behaviour. Consequently, they cannot improve the security of existing implementation. In our work, we employ AOP to instrument the code with additional functionality to enforce security hardening. Therefore our aspect is introduced for a different purpose.

## 6.3. Run-time Verification for Traceability

In this work we employ run-time verification as a tool to trace security requirements not only to the source code level, but beyond to the level of the execution of code. There are various reasons as to why this is advantageous. For example, assumptions that are inherent in design models may not adequately address real-world challenges, such as assumptions about attacker behaviour or the correctness of an implementation. Run-time verification as used in Section 3 has become a popular tool to verify that a system's execution adheres to a set of predefined properties.

As far as we know, this is the first work in which run-time verification is used for the traceability of high-level security properties in evolving systems.

Work in the area of run-time verification such as [34, 33, 11] consider it foremost from a theoretical point of view; that is, the complexity of the underlying problems, the theoretical expressiveness of the formalism used to express monitoring properties, or the efficiency of the generated monitors. In contrast, we focus on the methodological aspects of this technique for achieving traceable security beyond the source-code level.

As such, there are two aspects to be considered:

(1) the use of run-time verification for traceability of security properties in evolving systems, and
(2) the evolution of the run-time verification "layer" itself in terms of changing properties, monitor code, etc.

Regarding 1), although there seems to be no prior work on run-time security verification for evolving systems, there is some previous work on run-time security verification. The techniques used in run-time verification bear a resemblance with the well-known *security automata* as introduced by Schneider [59]. Formally, Schneider's work is based on temporal logic as well, however, it imposes restrictions on the types of specifications which can be monitored (or "enforced" to put it in Schneider's own terms). Security automata are restricted to the so-called *safety fragment* (of LTL). Because the properties we consider go beyond the pure safety fragment of LTL, our approach is strictly more expressive than Schneider's original work (see [10] for a discussion of this). Another application of monitoring to security was presented in [69]. The paper proposes a caller-side rewriting algorithm for the byte-code of the .NET virtual machine where security checks are inserted around calls to security-relevant methods. The work is different from ours in that it has not been applied to the security verification of cryptographic protocols, which pose specific challenges (such as the correct use of cryptographic functions and checks). In another approach, [60] proposes to use formal patterns of LTL formulae that formalise frequently reoccurring system requirements as *security monitoring patterns*. Again, this does not seem to have been applied to cryptographic protocols so far.

Regarding 2), an important step regarding evolvable systems was recently also made by Barringer et al. in [8]. However, their view on evolution differs from the one presented in this paper. Notably, their approach to run-time verification is not just passive, but active, in that a failing system is modified by a monitor noticing the failure. As such, the failing system evolves, and the monitors continuously adapt. In contrast, the evolution of our systems is sparked by comparably major changes in the software's implementation, e.g., triggered by new requirements that warrant a new release of a system, or specific rewrites for efficiency gains. As a consequence, our use of run-time verification is not as tightly integrated as that presented in [8], formally and practically.

## 7. CONCLUSIONS

We have used an approach for model-based security verification in which a design model in the UML security extension UMLsec can be formally verified against high-level security requirements such as secrecy and authentication. An implementation of the specification can then be verified against the model by making use of run-time verification. Using the approach to run-time security verification, one can raise an alarm at run-time in case of a security violation, and terminate the given protocol execution, before the secret is leaked out to the network. We also explained how to remove the security vulnerability in a implementation that has been detected in this way to make sure the same problem will not appear again, making use of techniques from aspect-oriented programming (AOP).

Despite the similarities between testing and run-time verification, run-time verification can provide a level of assurance that goes beyond what testing can usually achieve when applied to highly complex security-critical software: While testing complex systems can usually not be exhaustive, run-time verification ensures, by construction, that every system trace that will ever be executed will be verified – while it is executed. In the case of the cryptographic protocols that we consider, it is indeed sufficient to notice attempted security violations at run-time to still be able to maintain the security of the system: The monitor is constructed in such a way that, if it detects a violation, the current execution of the security protocol will be terminated before any secret information is leaked out on the network.

In practice, systems do not remain unchanged after they are being used but may evolve over their life-time. We have therefore enabled our security assurance approach to cope automatically with the fact that systems will evolve at run-time, and still provide valid run-time security assurance.

We demonstrated the approach at the hand of an application to the Java-based implementation JESSIE of the Internet security protocol SSL, in which a security weakness was detected and fixed using our approach. We also explained how the traceability link can be transformed to the official implementation of the Java Secure Sockets Extension (JSSE) that was recently made open source by Sun.

There are a number of possible directions for future work.

- Although run-time verification is quite effective, sometimes it would be preferable to be able to statically verify at least a particularly critical part of the code, to further increase its trustworthiness. In future work we plan to investigate how to combine run-time security verification with static compositional software verification such as [2].
- In another direction, it would be interesting to see whether it would be possible to expand the kinds of attacks that could be detected by this approach, for example by including weaknesses in the implementations of cryptographic algorithms (such as encryption and digital signature). It remains, however, to be seen which impact this would have on the performance of the monitors.
- The current monitoring approach relies on the assumption of having access to the source code of the monitored software. It would be interesting to see whether one could develop a monitor approach that does not rely on this assumption but is still sufficiently precise and performant.

*Acknowledgements* Discussions with Martin Leucker about a draft of this paper are gratefully acknowledged, as well as constructive comments by the reviewers which helped improving the presentation significantly.

## REFERENCES

[1] Abadi, M. and R. Needham (1996). Prudent engineering practice for cryptographic protocols. *IEEE Trans. on Software Engineering 22*(1), 6–15.

[2] Abramsky, S., D. Ghica, A. Murawski, and C.-H. Ong (2004). Applying game semantics to compositional software modeling and verification. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 421–435. Springer-Verlag, Berlin.

[3] Aho, A. V., R. Sethi, and J. D. Ullman (1988). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

[4] Alam, M., M. Hafner, and R. Breu (2007). Model-driven security engineering for trust management in SECTET. *Journal of Software 2*(1), 47–59.

[5] Antoniol, G., G. Canfora, G. Casazza, A. de Lucia, and E. Merlo (2002). Recovering traceability links between code and documentation. *IEEE Trans. on Software Engineering 28*(10), 970–983.

[6] Armoni, R., L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar (2002). The ForSpec temporal logic: A new temporal property-specification language. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 2280 of *Lecture Notes in Computer Science*, pp. 296–211. Springer-Verlag, Berlin.

[7] Ball, T., R. Majumdar, T. Millstein, and S. K. Rajamani (2001). Automatic predicate abstraction of C programs. *SIGPLAN Not. 36*(5), 203–213.

[8] Barringer, H., D. M. Gabbay, and D. E. Rydeheard (2007). From runtime verification to evolvable systems. In O. Sokolsky and S. Tasiran (Eds.), *Intl. Workshop on Runtime Verification*, Volume 4839 of *Lecture Notes in Computer Science*, pp. 97–110. Springer-Verlag, Berlin.

[9] Basin, D., J. Doser, and T. Lodderstedt (2006). Model driven security: From UML models to access control infrastructures. *ACM Trans. on Software Engineering and Methodology 15*(1), 39–91.

[10] Bauer, A. and J. Jürjens (2008). Security protocols, properties, and their monitoring. In *4th Int. Workshop on Software Engineering for Secure Systems (SESS)*, pp. 33–40. ACM Press, New York, NY.

[11] Bauer, A., M. Leucker, and C. Schallhart (2006). Monitoring of real-time properties. In *26th Intl. Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Volume 4337 of *Lecture Notes in Computer Science*, pp. 261–273. Springer-Verlag, Berlin.

[12] Bauer, A., M. Leucker, and J. Streit (2006). SALT—Structured Assertion Language for Temporal logic. In *Eighth Intl. Conference on Formal Engineering Methods (ICFEM)*, Volume 4260 of *Lecture Notes in Computer Science*, pp. 757–776. Springer-Verlag, Berlin.

[13] Best, B., J. Jürjens, and B. Nuseibeh. (2007). Model-based security engineering of distributed information systems using UMLsec. In *Intl. Conference on Software Engineering (ICSE)*, pp. 581–590. ACM Press, New York, NY.

[14] Beyer, D., A. Noack, and C. Lewerentz (2005). Efficient Relational Calculation for Software Analysis. *IEEE Trans. on Software Engineering 31*(2), 137–149.

[15] Bhargavan, K., C. Fournet, A. Gordon, and S. Tse (2006). Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop*, pp. 139–152. IEEE Computer Society.

[16] Binkley, D., M. Ceccato, M. Harman, F. Ricca, and P. Tonella (2006). Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Trans. on Software Engineering 32*(9), 698–717.

[17] Breu, R., K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz (2003). Key issues of a formally based process model for security engineering. In *16th Intl. Conference "Software & Systems Engineering & their Applications" (ICSSEA)*. IEEE Computer Society.

[18] Calder, M. (1998). What use are formal design and analysis methods to telecommunications services? In *5th Intl. Conference on Feature Interactions in Telecommunications and Software Systems*, pp. 23–31. IOS Press.

[19] Chess, B. and J. West (2007). *Secure Programming with Static Analysis*. Addison-Wesley Professional.

[20] Clarke, E. M., O. Grumberg, and D. A. Peled (1999). *Model Checking*. Cambridge, Massachusetts: The MIT Press.

[21] Cleland-Huang, J., R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina (2005). Goal-centric traceability for managing non-functional requirements. In *Intl. Conference on Software Engineering (ICSE)*, pp. 362–371. ACM Press, New York, NY.

[22] Colin, S. and L. Mariani (2004). Run-time verification. In *Model-Based Testing of Reactive Systems*, Volume 3472 of *Lecture Notes in Computer Science*, pp. 525–555. Springer-Verlag, Berlin.

[23] Dig, D., C. Comertoglu, D. Marinov, and R. Johnson (2006). Automated detection of refactorings in evolving components. In *20th European Conference on Object-Oriented Programming (ECOOP)*, Volume 4067 of *Lecture Notes in Computer Science*, pp. 404–428. Springer-Verlag, Berlin.

[24] Dig, D., K. Manzoor, R. Johnson, and T. N. Nguyen (2007). Refactoring-aware configuration management for object-oriented programs. In *Intl. Conference on Software Engineering (ICSE)*, pp. 427–436. ACM Press, New York, NY.

[25] Dimitrakos, T., B. Ritchie, D. Raptis, J. Aagedal, F. den Braber, K. Stølen, and S. Houmb (2002). Integrating model-based security risk management into ebusiness systems development: The CORAS approach. In *Second IFIP Conference on E-Commerce, E-Business, E-Government (I3E)*, pp. 159–175. Kluwer Academic Publishers.

[26] Duvall, P., S. Matyas, and A. Glover (2007). *Continuous integration: improving software quality and reducing risk.* Addison-Wesley Professional.

[27] Egyed, A. (2003). A scenario-driven approach to trace dependency analysis. *IEEE Trans. on Software Engineering 9*(2), 116–132.

[28] Eisner, C. and D. Fisman (2006). *A Practical Introduction to PSL (Series on Integrated Circuits and Systems).* Springer New York, Inc.

[29] Giorgini, P., F. Massacci, and J. Mylopoulos (2003). Requirement engineering meets security: A case study on modelling secure electronic transactions by VISA and Mastercard. In *22nd Intl. Conference on Conceptual Modeling (ER)*, Volume 2813 of *Lecture Notes in Computer Science*, pp. 263–276. Springer-Verlag, Berlin.

[30] Godefroid, P. (2005). Software model checking: The verisoft approach. *Form. Methods Syst. Des. 26*(2), 77–101.

[31] Goubault-Larrecq, J. and F. Parrennes (2005). Cryptographic protocol analysis on real C code. In *6th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, Volume 3385 of *Lecture Notes in Computer Science*, pp. 363–379. Springer-Verlag, Berlin.

[32] Hannemann, J. (2006). *Role-based refactoring of crosscutting concerns.* Ph. D. thesis, Vancouver, BC, Canada.

[33] Havelund, K. and G. Rosu (2002). Synthesizing Monitors for Safety Properties. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 2280 of *Lecture Notes in Computer Science*, pp. 342–356. Springer-Verlag, Berlin.

[34] Havelund, K. and G. Rosu (2004). Efficient monitoring of safety properties. *Software Tools for Technology Transfer 6*, 158– 173.

[35] Hoare, C. (1996). How did software get so reliable without proof? In *Formal Methods Europe (FME'96)*, Volume 1051 of *Lecture Notes in Computer Science*, pp. 1–17. Springer-Verlag, Berlin.

[36] Holzmann, G. J. (1991). *Design and validation of computer protocols.* Prentice-Hall, Inc.

[37] Hopcroft, J. E. and J. D. Ullman (1979). *Introduction to Automata Theory, Languages and Computation* (First ed.). Addison-Wesley.

[38] Ivkovic, I. and K. Kontogiannis (2004). Tracing evolution changes of software artifacts through model synchronization. In *20th IEEE Intl. Conference on Software Maintenance (ICSM)*, pp. 252–261. IEEE Computer Society.

[39] Jiang, H., T. N. Nguyen, and I. Chen (2008). Incremental latent semantic indexing for effective, automatic traceability link evolution management. In *Intl. Conference on Software Engineering (ICSE)*, pp. 59–68. ACM Press, New York, NY.

[40] Jürjens, J. (2000). Secure information flow for concurrent processes. In *11th Intl. Conference on Concurrency Theory (CONCUR)*, Volume 1877 of *Lecture Notes in Computer Science*, pp. 395–409. Springer-Verlag, Berlin.

[41] Jürjens, J. (2002). Formal semantics for interacting UML subsystems. In *5th Intl. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pp. 29–44. International Federation for Information Processing (IFIP): Kluwer Academic Publishers.

[42] Jürjens, J. (2004). *Secure Systems Development with UML.* Springer-Verlag, Berlin.

[43] Jürjens, J. (2005). Sound methods and effective tools for model-based security engineering with UML. In *Intl. Conference on Software Engineering (ICSE)*, pp. 322–331. ACM Press, New York, NY.

[44] Jürjens, J., J. Schreck, and P. Bartmann (2008). Model-based security analysis for mobile communications. In *Intl. Conference on Software Engineering (ICSE)*, pp. 683–692. ACM Press, New York, NY.

[45] Jürjens, J. and P. Shabalin (2004). Automated verification of UMLsec models for security requirements. In *The Unified Modeling Language (UML)*, Volume 2460 of *Lecture Notes in Computer Science*, pp. 412–425. Springer-Verlag, Berlin.

[46] Jürjens, J. and M. Yampolskiy (2005). Code security analysis with assertions. In *20th Intl. Conference on Automated Software Engineering (ASE)*, pp. 392–395. ACM Press, New York, NY.

[47] Jürjens, J. and Y. Yu (2007). Tools for model-based security engineering: Models vs. code. In *22nd Intl. Conference on Automated Software Engineering (ASE)*, pp. 545–546. ACM Press, New York, NY.

[48] Jürjens, J., Y. Yu, and A. Bauer (2008). Tools for traceable security verification. In *Proceedings of the BCS International Academic Conference 2008—Visions of Computer Science*, Swindon, UK, pp. 367–378. The British Computer Society.

[49] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold (2001). *An overview of AspectJ*, Volume 2072/2001 of *Lecture Notes in Computer Science*, pp. 327–355. Springer-Verlag, Berlin.

[50] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In *17th European Conference on Object-Oriented Programming (ECOOP)*, Volume 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer-Verlag, Berlin.

[51] Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv. 24*(2), 131–183.

[52] Laddad, R. (2006). *Aspect Oriented Refactoring*. Addison-Wesley Professional.

[53] Leucker, M. and C. Schallhart (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*. in press.

[54] Mens, T. and T. Tourwe (2004). A survey of software refactoring. *IEEE Trans. on Software Engineering 30*(2), 126–139.

[55] Murphy, G. C., D. Notkin, and K. J. Sullivan (2001). Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. on Software Engineering 27*(4), 364–380.

[56] Pironti, A. and R. Sisto (2007). An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *IEEE Symposium on Computers and Communications*, pp. 839–844. IEEE Computer Society.

[57] Pnueli, A. (1977). The temporal logic of programs. In *18th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pp. 46–57. IEEE Computer Society.

[58] Ryan, P., S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe (2001). *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley.

[59] Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur. 3*(1), 30–50.

[60] Spanoudakis, G., C. Kloukinas, and K. Androutsopoulos (2007). Towards security monitoring patterns. In *ACM Symposium on Applied Computing (SAC)*, pp. 1518–1525. ACM Press, New York, NY.

[61] Spanoudakis, G., A. Zisman, E. Pérez-Miñana, and P. Krause (2004). Rule-based generation of requirements traceability relations. *Journal of Systems and Software 72*(2), 105–127.

[62] Stärk, R., J. Schmid, and E. Börger (2001). *Java and the Java virtual machine – definition, verification, validation*. Springer-Verlag.

[63] Stenz, G. and A. Wolf (2000). E-setheo: An automated theorem prover. In *Intl. Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, Volume 1847 of *Lecture Notes in Computer Science*, pp. 436–440. Springer-Verlag, Berlin.

[64] Tarr, P., H. Ossher, W. Harrison, and S. S. Jr. (1999). Degrees of separation: Multi-dimensional separation of concerns. In *Intl. Conference on Software Engineering (ICSE)*, pp. 107–119. ACM Press, New York, NY.

[65] Thomas, M. (2004). Engineering judgement. In *9th Australian workshop on Safety critical systems and software (SCS)*, pp. 43–47. Australian Computer Society, Inc.

[66] Tool (2001-08). Security analysis tools. http://computing-research.open.ac.uk/jj/sectracetool.

[67] Tool (2009a). Borland Together. http://www.borland.com/us/products/together/.

[68] Tool (2009b). LTL$_3$ Tools. http://ltl3tools.SourceForge.Net/.

[69] Vanoverberghe, D. and F. Piessens (2008). A caller-side inline reference monitor for an object-oriented intermediate language. In *10 Intl. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Volume 5051 of *Lecture Notes in Computer Science*, pp. 240–258. Springer-Verlag, Berlin.

[70] Widmer, T. (2007, Feb). Unleashing the power of refactoring. *Eclipse Corner Articles*.

[71] Woodcock, J., S. Stepney, D. Cooper, J. Clark, and J. Jacob (2008). The certification of the Mondex electronic purse to ITSEC Level E6. *Formal Aspects of Computing 20*(1), 5–19.

[72] Yskout, K., R. Scandariato, B. D. Win, and W. Joosen (2008). Transforming security requirements into architecture. In *3rd Intl. Conference on Availability, Reliability and Security (ARES)*, pp. 1421–1428. IEEE Computer Society.

[73] Yu, Y., J. C. S. do Prado Leite, and J. Mylopoulos (2004). From goals to aspects: Discovering aspects from requirements goal models. In *12th IEEE Intl. Conference on Requirements Engineering (RE)*, pp. 38–47. IEEE Computer Society.

[74] Yu, Y., Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite (2005). Reverse engineering goal models from legacy code. In *13th IEEE Intl. Conference on Requirements Engineering (RE)*, pp. 363–372. IEEE Computer Society.