Modelling Secure Systems Evolution: Abstract and Concrete Change Specifications

Jan Jürjens[1,2] , Martín Ochoa[1], Holger Schmidt[1], Loïc Marchal[3], Siv Hilde Houmb[4] and Shareeful Islam[5]

[1] Software Engineering, Dep. of Computer Science, TU Dortmund, Germany
[2] Fraunhofer ISST, Germany
[3] Hermès Engineering, Belgium
[4] Secure-NOK AS, Norway
[5] School of Computing, IT and Engineering, University of East London, UK
{jan.jurjens,martin.ochoa,holger.schmidt}@cs.tu-dortmund.de
loic.marchal@hermes-ecs.com
sivhoumb@securenok.com
shareeful@uel.ac.uk

Abstract. Developing security-critical systems is difficult, and there are many well-known examples of vulnerabilities exploited in practice. In fact, there has recently been a lot of work on methods, techniques, and tools to improve this situation already at the system specification and design. However, security-critical systems are increasingly long-living and undergo evolution throughout their lifetime. Therefore, a secure software development approach that supports maintaining the needed levels of security even through later software evolution is highly desirable. In this chapter, we recall the UMLsec approach to model-based security and discuss on tools and techniques to model and verify evolution of UMLsec models.

Keywords: Software Evolution, UMLsec, UMLseCh, Security

1 Introduction

Typically, a systematic approach focused on software quality – the degree to which a software system meets its requirements – is addressed during design time through design processes and supporting tools. Once the system is put in operation, maintenance and re-engineering operations are supposed to keep it running.

At the same time, successful software-based systems are becoming increasingly long-living [21]. This was demonstrated strikingly with the occurrence of the year 2000 bug, which occurred because software had been in use for far longer than its expected lifespan. Also, software-based systems are getting increasingly security-critical since software now pervades the whole critical infrastructures dealing with critical data of both nations and also private individuals. There is therefore a growing demand for more assurance and verifiable secure IT systems both during development and at deployment time, in particular also for

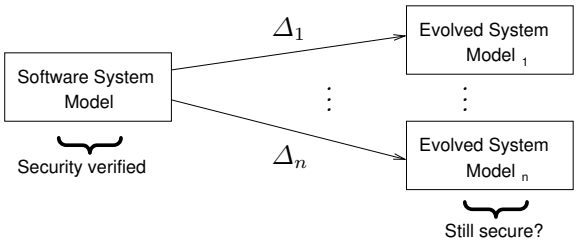

Fig. 1. Model verification problem for n possible evolution paths

long living systems. Yet a long lived system also needs to be flexible, to adapt to evolving requirements, usage, and attack models. However, using today's system engineering techniques we are forced to trade flexibility for assurance or vice versa: we know today how to provide security or flexibility taken in isolation. We can use full fledged verification for providing a high-level of assurance to fairly static requirements, or we can provide flexible software, developed in weeks using agile methodologies, but without any assurance. This raises the research challenge of whether and how we can provide some level of security assurance for something that is going to change.

Our objective is thus to develop techniques and tools that ensure "lifelong" compliance to evolving security requirements for a long-running evolving software system. This is challenging because these requirements are not necessarily preserved by system evolution [22]. In this chapter, we present results towards a security modelling notation for the evolution of security-critical designs, suitable by verification with formally founded automated security analysis tools. Most existing assessment methods used in the development of secure systems are mainly targeted at analysing a static picture of the system or infrastructure in question. For example, the system as it is at the moment, or the system as it will be when we have updated certain parts according to some specifications. In the development of secure systems for longevity, we also need descriptions and specifications of what may be foreseen as future changes, and the assessment methods must be specialized account for this kind of descriptions. Consequently, our approach allows to re-assess the impact that changes might have on the security of systems.

On one hand, a system needs to cope with a given change as early as possible and on the other hand, it should preserve the security properties of the overall system (Fig. 1). To achieve this, it is preferable to analyse the planned evolution before carrying it out. In this chapter we present a notation that allows to precisely determine the changes between one or more system versions, and that combined with proper analysis techniques, allows to reason about how to preserve the existing and new (if any) security properties due to the evolution. Reflecting change on the model level eases system evolution by ensuring effec-

tive control and tracking of changes. We focus in understanding and tracing the change notations for the system design model. Design models represent the early exploration of the solution space and are the intermediate between requirements and implementation. They can be used to specify, analyse, and trace changes directly.

In Sect. 2 we recall the *UMLsec* profile [10, 13], which is a UML [28] lightweight extension to develop and analyse security models, together with some applications. We present a change-modelling profile called *UMLseCh* in Sect. 3. We use UMLseCh design models for change exploration and decision support when considering how to integrate new or additional security functions and to explore the security implications of planned system evolution. To maintain the security properties of a system through change, the change can be explicitly expressed such that its implications can be analysed a priori. The UMLseCh stereotypes extend the existing UMLsec stereotypes so that the design models preserve the security properties due to change.

Although, the question of model evolution is intrinsically related with model-transformation, we do not aim to show an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [7, 1, 2, 25, 20]). However, we rely on UMLsec because it comes with sophisticated tool support[6], and our goal is to present an approach that is a) consistent with the UMLsec philosophy of extending UML b) is meant to be used on the UML fragment relevant for UMLsec.

In Sect. 4 we show some applications of UMLseCh to different diagram types and we discuss how this notation and related verification mechanisms could be supported by an extension of the UMLsec Tool Suite.

2 Background: Secure Systems Modelling with UMLsec

Generally, when using model-based development (Fig. 2a), the idea is that 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. In the model-based security engineering (MBSE) approach based on the UML [28] extension UMLsec, [11, 13], recurring security requirements (such as secrecy, integrity, authenticity, and others) and security assumptions on the system environment, can be specified either within UML specifications, or within the source code (Java or C) as annotations (Fig. 2b). This way we encapsulate knowledge on prudent security engineering as annotations in models or code and make it available to developers who may not be security experts.

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

[6] UMLsec tool suite: `http://www.umlsec.de/`

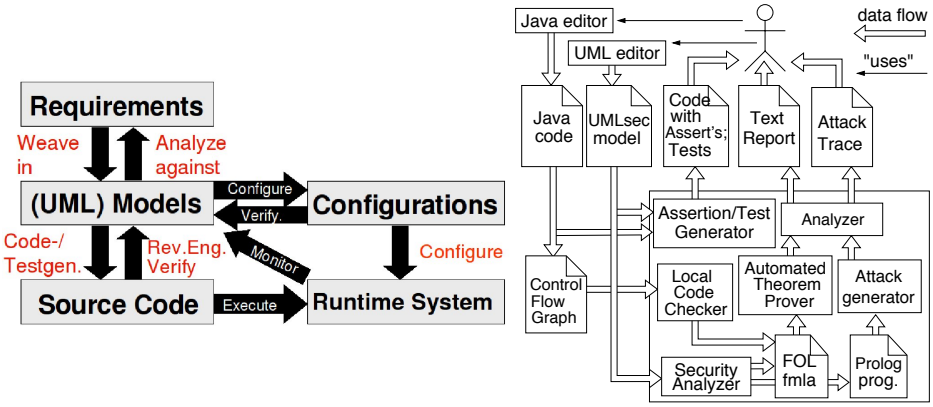

Fig. 2. a) Model-based Security Engineering; b) Model-based Security Tool Suite

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 assumptions on the physical level of the system, security requirements related to the secure handling and communication of data, and security policies that system parts are supposed to obey. The UMLsec tool-support in Fig. 2b) can 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 [29, 14, 18, 8]. 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 [13] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces similar to Broy's Focus model, whose behavior can be specified in a notation similar to that of Abstract State Machines (ASMs), and which is equipped with UML-type communication mechanisms. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined. To support stepwise development, we show secrecy, integrity, authenticity, and secure information flow to be *preserved* under refinement and the composition of system components. We have also developed an approach that supports the secure development of layered security services (such as layered security protocols). UMLsec can be used to specify and implement security patterns, and is supported by dedicated secure systems development processes, in particular an Aspect-Oriented Modeling approach which separates complex security mechanisms (which implement the security aspect model) from the core functionality of the system (the primary model) in order to allow a security verification of the particularly security-critical parts, and also of the composed model.

2.1 The UMLsec Profile

Because of space restrictions, we cannot recall our formal semantics here completely. Instead, we define precisely and explain the interfaces of this semantics that we need here to define the UMLsec profile. More details on the formal semantics of a simplified fragment of UML and on previous and related work in this area can be found in [9, 13]. The semantics is defined formally using so-called *UML Machines*, which is an extension of Mealy automata with UML-type communication mechanisms. It includes the following kinds of diagrams:

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called *object diagrams*.

Statechart diagrams (or *state diagrams*) give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions.

Sequence diagrams describe interaction between objects or system components via message exchange.

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statecharts and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of *packages*) integrate the information between the different kinds of diagrams and between different parts of the system specification.

There is another kind of diagrams, the use case diagrams, which describe typical interactions between a user and a computer system. They are often used in an informal way for negotiation with a customer before a system is designed. We will not use it in the following. Additionally to sequence diagrams, there are *collaboration diagrams*, which present similar information. Also, there are *component diagrams*, presenting part of the information contained in deployment diagrams.

The used fragment of UML is simplified to keep automated formal verification that is necessary for some of the more subtle security requirements feasible. Note that in our approach we identify system objects with UML objects, which is suitable for our purposes. Also, as with practically all analysis methods, also in the real-time setting [30], we are mainly concerned with instance-based models. Although, simplified, our choice of a subset of UML is reasonable for our needs, as we have demonstrated in several industrial case-studies (some of which are documented in [13]).

The formal semantics for subsystems incorporates the formal semantics of the diagrams contained in a subsystem. It

- models actions and internal activities explicitly (rather than treating them as atomic given events), in particular the operations and the parameters employed in them,

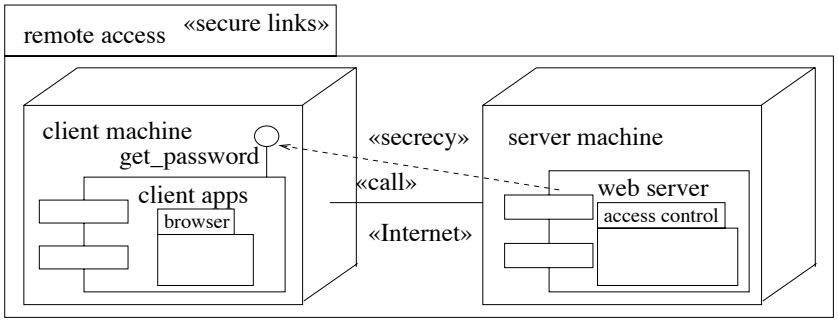

Fig. 3. Example *secure links* usage

– provides passing of messages with their parameters between objects or components specified in different diagrams, including a dispatching mechanism for events and the handling of actions, and thus
– allows in principle whole specification documents to be based on a formal foundation.

In particular, we can compose subsystems by including them into other subsystems.

For example, consider the following UMLsec Stereotype:

secure links This stereotype, which may label (instances of) subsystems, is used to ensure that security requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency d with stereotype $s \in \{\ll\texttt{secrecy}\gg, \ll\texttt{integrity}\gg, \ll\texttt{high}\gg\}$ between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype t such that

– in the case of $s = \ll\texttt{high}\gg$, we have $\mathsf{Threats}_A(t) = \emptyset$,
– in the case of $s = \ll\texttt{secrecy}\gg$, we have $\mathsf{read} \notin \mathsf{Threats}_A(t)$, and
– in the case of $s = \ll\texttt{integrity}\gg$, we have $\mathsf{insert} \notin \mathsf{Threats}_A(t)$.

Example In Fig. 3, given the *default* adversary type, the constraint for the stereotype $\ll\texttt{secure links}\gg$ is violated: The model does not provide communication secrecy against the *default* adversary, because the Internet communication link between web-server and client does not give the needed security level according to the $\mathsf{Threats}_{default}(Internet)$ scenario. Intuitively, the reason is that Internet connections do not provide secrecy against default adversaries. Technically, the constraint is violated, because the dependency carries the stereotype $\ll\texttt{secrecy}\gg$, but for the stereotype $\ll\texttt{Internet}\gg$ of corresponding link we have $\mathsf{read} \in \mathsf{Threats}_{default}(Internet)$.

Code Security Assurance [15, 16] Even if specifications exist for the implemented system, and even if these are formally analyzed, there is usually no guarantee that the implementation actually conforms to the specification. To deal with this problem, we use the following approach: After specifying the system in UMLsec and verifying the model against the given security goals as explained above, we make sure that the implementation correctly implements the specification with techniques explained below. In particular, this approach is applicable to legacy systems. In ongoing work, we are automating this approach to free one of the need to manually construct the UMLsec model.

Run-time Security Monitoring using Assertions A simple and effective alternative is to insert security checks generated from the UMLsec specification that remain in the code while in use, for example using the assertion statement that is part of the Java language. These assertions then throw security exceptions when violated at run-time. In a similar way, this can also be done for C code.

Model-based Test Generation For performance-intensive applications, it may be preferable not to leave the assertions active in the code. This can be done by making sure by extensive testing that the assertions are always satisfied. We can generate the test sequences automatically from the UMLsec specifications. More generally, this way we can ensure that the code actually conforms to the UMLsec specification. Since complete test coverage is often infeasible, our approach automatically selects those test cases that are particularly sensitive to the specified security requirements [19].

Automated Code Verification against Interface Specifications For highly nondeterministic systems such as those using cryptography, testing can only provide assurance up to a certain degree. For higher levels of trustworthiness, it may therefore be desirable to establish that the code does enforce the annotations by a formal verification of the source code against the UMLsec interface specifications. We have developed an approach that does this automatically and efficiently by proving locally that the security checks in the specification are actually enforced in the source code.

Automated Code Security Analysis We developed an approach to use automated theorem provers for first-order logic to directly formally verify crypto-based Java implementations based on control flow graphs that are automatically generated (and without first manually constructing an interface specification). It supports an abstract and modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. Currently, this approach works especially well with nicely structured code (such as created using the MBSE development process).

Secure Software-Hardware Interfaces We have tailored the code security analysis approach to software close to the hardware level. More concretely, we considered

the industrial Cryptographic Token Interface Standard PKCS 11 which defines how software on untrustworthy hardware can make use of tamper-proof hardware such as smart-cards to perform cryptographic operations on sensitive data. We developed an approach for automated security analysis with first-order logic theorem provers of crypto protocol implementations making use of this standard.

Analyzing Security Configurations We have also performed research on linking the UMLsec approach with the automated analysis of security-critical configuration data. For example, our tools automatically checks SAP R/3 user permissions for security policy rules formulated as UML specifications [13]. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software, such as firewalls or other access control configurations.

Industrial Applications of MBSE include:

Biometric Authentication For a project with an industrial partner, MBSE was chosen to support the development of a biometric authentication system at the specification level, where three significant security flaws were found [14]. We also applied it to the source-code level for a prototypical implementation constructed from the specification [12].

Common Electronic Purse Specifications MBSE was applied to a security analysis of the Common Electronic Purse Specifications (CEPS), a candidate for a globally interoperable electronic purse standard supported by organizations representing 90 % of the world's electronic purse cards (including Visa International). We found three significant security weaknesses in the purchase and load transaction protocols [13], proposed improvements to the specifications and showed that these are secure [13]. We also performed a security analysis of a prototypical Java Card implementation of CEPS.

Web-based Banking Application In a project with a German bank, MBSE was applied to a web-based banking application to be used by customers to fill out and sign digital order forms [6]. The personal data in the forms must be kept confidential, and orders securely authenticated. The system uses a proprietary client authentication protocol layered over an SSL connection supposed to provide confidentiality and server authentication. Using the MBSE approach, the system architecture and the protocol were specified and verified with regard to the relevant security requirements.

In other applications [13], MBSE was used ...

- to uncover a flaw in a variant of the Internet protocol TLS proposed at IEEE Infocom 1999, and suggest and verify a correction of the protocol.
- to perform a security verification of the Java implementation Jessie of SSL.
- to correctly employ advanced Java 2 or CORBA security concepts in a way that allows an automated security analysis of the resulting systems.

- for an analysis of the security policies of a German mobile phone operator [17].
- for a security analysis of the specifications for the German Electronic Health Card in development by the German Ministry of Health.
- for the security analysis of an electronic purse system developed for the Oktoberfest in Munich.
- for a security analysis of an electronic signature pad based contract signing architecture under consideration by a German insurance company.
- in a project with a German car manufacturer for the security analysis of an intranet-based web information system.
- with a German chip manufacturer and a German reinsurance company for security risk assessment, also regarding Return on Security Investment.
- in applications specifically targeted to service-based, health telematics, and automotive systems.

Recently, there has been some work analyzing trade-offs between security- and performance-requirements [24, 31].

3 Modelling evolution with UMLseCh

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 section, by "UML" we mean the core of the UML 2.0 which was conservatively included from UML 1.5[7].

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 [13]. In this section, we define the meta-model of UMLseCh in an analogous way.

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. Sect. 3.1), 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. Sect. 3.2). We discuss about possible verification strategies based on the concrete notation in Sect. 3.3.

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

3.1 Abstract Notation

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. The abstract stereotypes makes up three refinement levels, where the upper level is ≪`change`≫. This stereotype can be attached to subsystems and is used across all UML diagrams. The meaning of the stereotype is that 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:

1. ≪`past_change`≫ representing changes already made to the system (typically between two system versions).
2. ≪`current_change`≫ representing changes currently being made to a system.
3. ≪`future_change`≫ specifying the future allowed changes.

To track and ensure controlled change actions one needs to be explicit about which model elements are allowed to change and what kind of change 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 by refining the abstract notation into the concrete one.

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`≫ and ≪`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) [5].

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.

3.2 Concrete Notation

We further extend UMLsec by adding so called "concrete" stereotypes: these stereotypes allow to precisely define substitutive (sub) model elements and are

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

Fig. 4. UMLsecCh concrete design stereotypes

equipped with constraints that help ensuring their correct application. These differ from the abstract stereotypes basically because we define a precise semantics (similar to the one of a transformation language) that is intended to be the basis for a security-preservation analysis based on the model difference between versions.

Figure 4 shows the stereotypes defining table. The tags table is shown in Figure 5.

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

Fig. 5. UMLsecCh concrete design tags

Description of the notation We now describe informally the intended semantics of each stereotype.

substitute The stereotype substitute attached to a model element denotes the possibility for that model element to be substituted by a model element of the same type over the time. It has three associated tags, namely {`ref`}, {`substitute`} and {`pattern`}.

These tags are of the form

$$\{\,\mathsf{ref} = \mathsf{CHANGE\text{-}REFERENCE}\,\},$$
$$\{\,\mathsf{substitute} = \mathsf{MODEL\text{-}ELEMENT}\,\}$$
$$\text{and } \{\,\mathsf{pattern} = \mathsf{CONDITION}\,\}.$$

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 (e.g. a stereotype, {`substitute = ≪stereotype≫`}). To textually describe UML model elements one can use an abstract syntax as in [13] or any equivalent grammar. Ideally, tool support should help the user into choosing from a graphical menu which model elements to use, without the user to learn the model-describing grammar. 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 (as it will be the case for simultaneous changes where we can not attach the evolution stereotype to all targeted elements at once), the tag pattern must be used. This tag has a model element as value, which represents the model element to substitute if a change occurs. In general the value of pattern can be a function uniquely identifying one or more model elements within a diagram.

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:

$$\ll\texttt{substitute}\gg$$
$$\{\,\mathsf{ref} = \mathsf{encrypt\text{-}link}\,\}$$
$$\{\,\mathsf{substitute} = \mathsf{encrypted}\,\}$$
$$\{\,\mathsf{pattern} = \mathsf{Internet}\,\}$$

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 occurred, 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 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} (i.e. a list of model elements that we wish to add). The tag{pattern} has a slightly different meaning in this case. While with stereotype ≪substitute≫, the tag {pattern} represents the model element to substitute, within the stereotype ≪add≫ it does not represent the model element to add, but the parent model element to which the new (sub)-model element is to be added.

The stereotype ≪add≫ is a syntactic sugar of the stereotype ≪substitute≫, as a stereotype ≪add≫ could always be represented with a substitute stereotype (substituting the parent element with a modified one). For example, in the case of Class Diagrams, if s is the set of methods and m a new method to be added, the new set of methods is:

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

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 substitute stereotype, as a stereotype ≪delete≫ could always be represented with a substitution. For example, if s is the set of methods and m a method to delete, the new set of methods is:

$$s' = s \setminus m$$

As with the previous stereotypes, the stereotype ≪delete≫ also has a constraint formulated in first order logic.

substitute-all The stereotype ≪substitute-all≫ is an extension 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 evolve 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 syntax of UMLsec, defined in [13].

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 this simultaneous substitution, consider 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.

Example Figure 6 shows the use of ≪`add-all`≫ and ≪`substitute-all`≫ on a package containing a class diagram and a deployment diagram depicting the communication between two parties through a common proxy server. The change reflects the design choice to, in addition to protect the integrity of the message `d`, enforce the secrecy of this value as well.

Complex changes In case of complex changes, that arise for example if we want to merge two diagrams having elements in common, we can overload the aforementioned stereotypes for accepting not only lists of elements but even lists of lists of elements. This is technically not very different from what we have described so far, since the complex evolutions can be seen as syntactic sugar for multiple coordinated single-model evolutions.

3.3 Security preservation under evolution

With the use of the UMLseCh concrete stereotypes, evolving a model means that we either *add*, *delete*, or / and *substitute* elements of this model explicitly. In other words, the stereotypes induce sets **Add**, **Del**, and **Subs**, containing the model elements to be added, deleted and substituted respectively, together with information about *where* to perform these changes.

Given a diagram M and a set Δ of such modifications we denote $M[\Delta]$ the diagram resulting after the modifications have taken place. So in general let P be a diagram property. We express the fact that M enforces P by $P(M)$. *Soundness* of the security preserving rules R for a property P on diagram M can be formalized as follows:

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

So to reason about security preservation, one has several alternatives, depending on the property P. For some static analysis, it suffices to show that simultaneous sub-changes contained in Δ preserve P. Then, incrementally, we can proceed until reaching $P(M[\Delta])$. This can be done by reasoning inductively inductively by cases given a security requirement on UML models, by considering incremental atomic changes and distinguishing them according to a) their *evolution* type (addition, deletion, substitution) and b) their *UML diagram* type.

For dealing with behavioural properties one could exploit the divide and conquer approach by means of compositionality verification results. This idea, originally described in general in [4] (and as mentioned before, used for safety

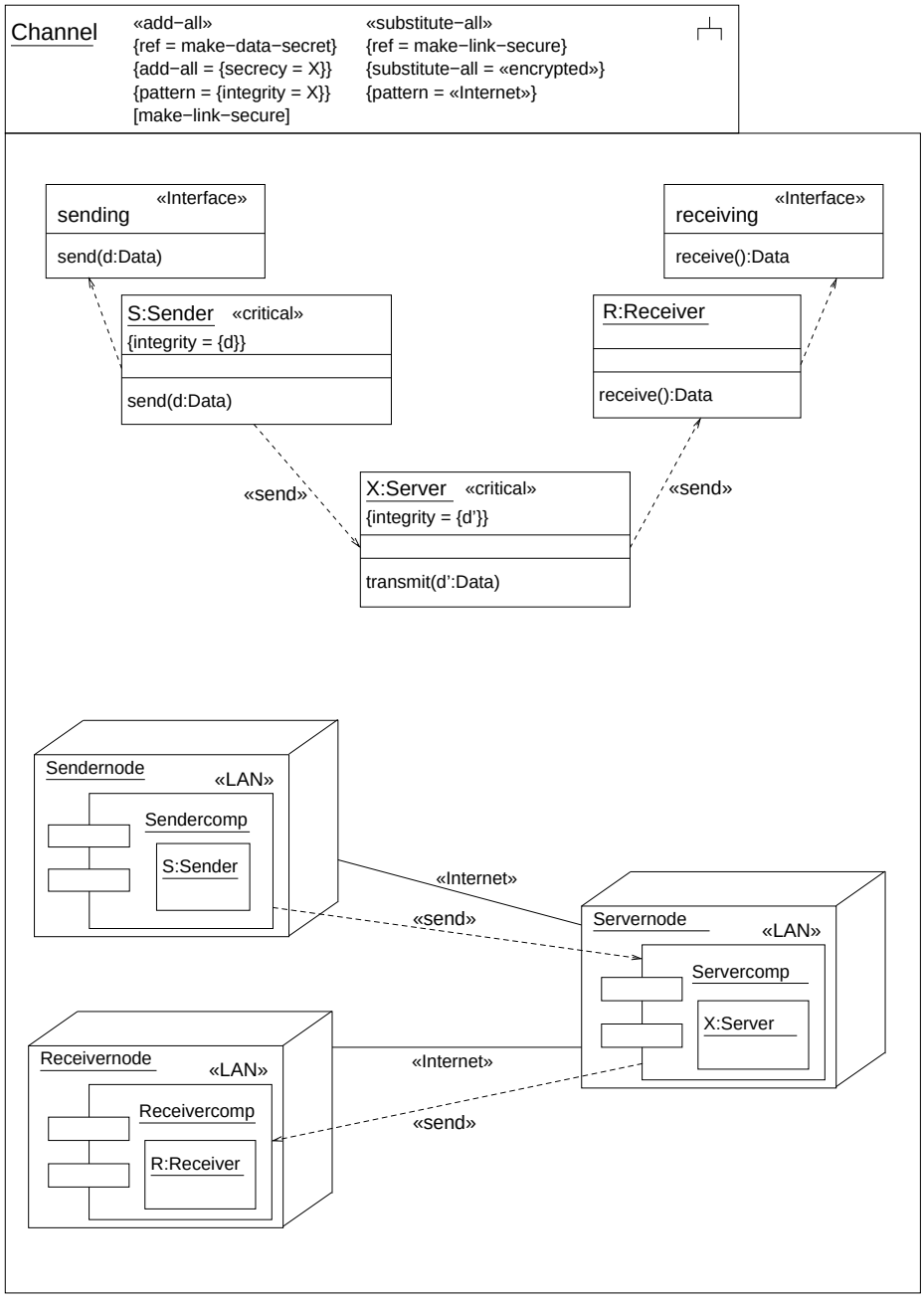

Fig. 6. A UMLseCh annotated diagram with simultaneous substitutions and additions

properties in [3]), is as follows: given components C and D, we denote with $C \otimes D$ its composition. Then, if we now that a security property P holds on both components separately and some set of rules R are satisfied then P holds in the composition, we can use this to achieve a more efficient verification under evolution, given R is easy enough to check. In practice, one often has to modify just one component in a system, and thus if one has:

$$P(C) \wedge P(D) \wedge R(C, D) \Rightarrow P(C \otimes D)$$

one can then check, for a modification Δ on one of the components:

$$P(C[\Delta]) \wedge P(D) \wedge R(C[\Delta], D) \Rightarrow P(C[\Delta] \otimes D) = P(C \otimes D)[\Delta]$$

and thus benefit from the already covered case $P(D)$ and the efficiency of R. Depending on the completeness of R, this procedure can also be relevant for an evolution of both components, since one could reapply the same decision procedure for a change in D (and therefore can be generalized to more than two components). The benefit consists in splitting the problem of verifying the composition (which is a problem with a bigger input) in two smaller subproblems. Some security properties (remarkably information flow properties like non-interference) are not safety properties, and there are interesting results for their compositionality (for example [23]).

4 Application Examples and Tool Support

4.1 Modelling change of UMLsec Diagrams

Secure Dependency This stereotype, used to label subsystems containing object diagrams or 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 along them, as given by the tags {`secrecy`}, {`integrity`}, and {`high`} of the stereotype ≪`critical`≫. More exactly, the constraint enforced by this stereotype is that if there is a ≪`call`≫ or ≪`send`≫ dependency from an object (or subsystem) C to an interface I of an object (or subsystem) D then the following conditions are fulfilled.

- For any message name n in I, n appears in the tag {`secrecy`} (resp. {`integrity`}, {`high`}) in C if and only if it does so in D.
- If a message name in I appears in the tag {`secrecy`} (resp.{`integrity`}, {`high`}) in C then the dependency is stereotyped ≪`secrecy`≫ (resp. ≪`integrity`≫ , ≪`high`≫).

If the dependency goes directly to another object (or subsystem) without involving an interface, the same requirement applies to the trivial interface containing all messages of the server object.

This property is specially interesting to verify under evolution since it is local enough to re-use effectively previous verifications on the unmodified parts and its syntactic nature makes the incremental decision procedure relatively straightforward.

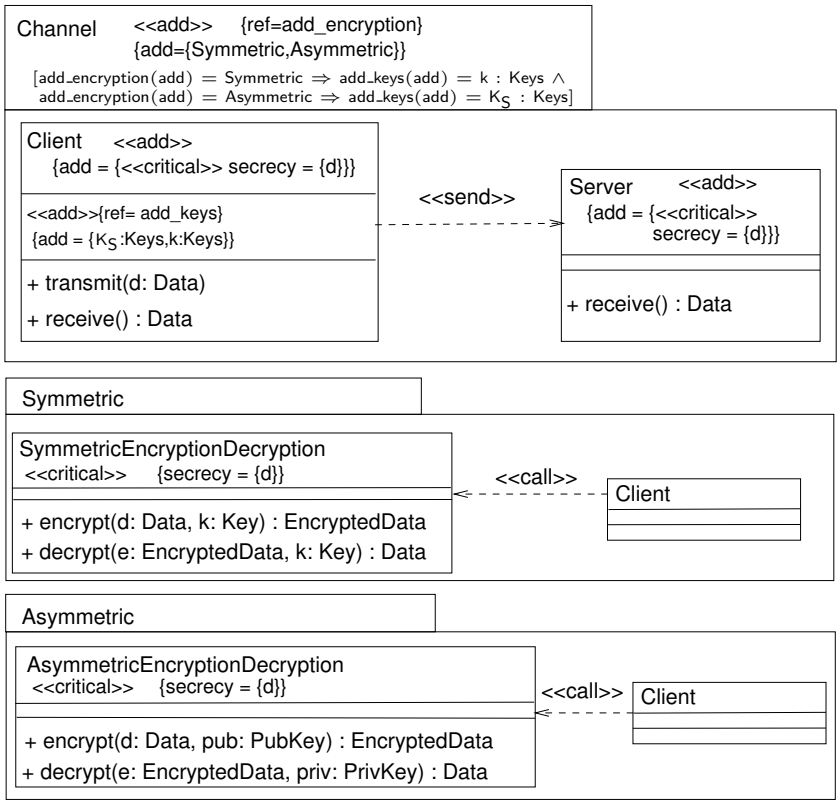

Fig. 7. An evolving class diagram with two possible evolution paths

Example The example in Fig. 7 shows the Client side of a communication channel between two parties. At first (disregarding the evolution stereotypes) the communication is unsecured. In the packages Symmetric and Asymmetric, we have classes providing cryptographic mechanisms to the Client class. Here the stereotype ≪add≫ marked with the reference tag {ref} with value add_encryption specifies two possible evolution paths: merging the classes contained in the current package (Channel) with either Symmetric or Asymmetric. There exists also a stereotype ≪add≫ associated with the Client class adding either a pre-shared private key k or a public key K_S of the server. To coordinate the intended evolution paths for these two stereotypes, we can use the following first-order logic constraint (associated with add_encryption):

$$[\mathsf{add\_encryption}(\mathsf{add}) = \mathsf{Symmetric} \Rightarrow \mathsf{add\_keys}(\mathsf{add}) = \mathsf{k} : \mathsf{Keys} \ \wedge$$
$$\mathsf{add\_encryption}(\mathsf{add}) = \mathsf{Asymmetric} \Rightarrow \mathsf{add\_keys}(\mathsf{add}) = \mathsf{K_S} : \mathsf{Keys}]$$

The two deltas, representing two possible evolution paths induced by this notation, can be checked incrementally by case distinction. In this case, the evo-

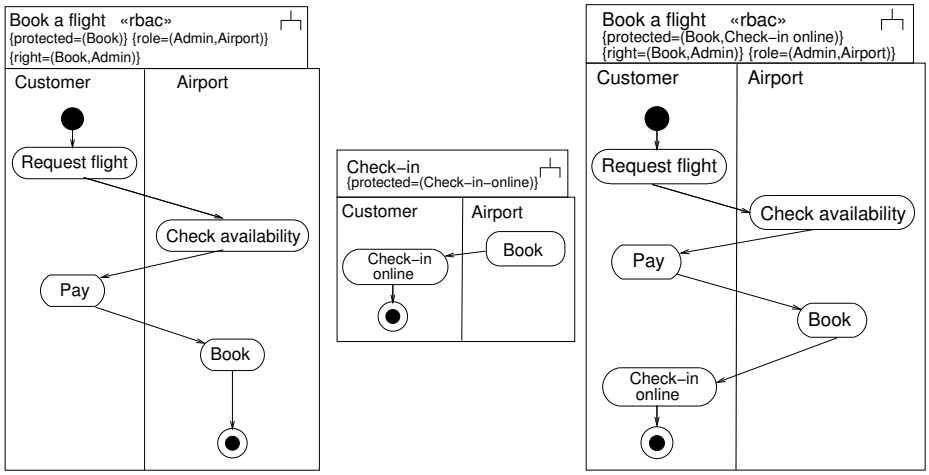

Fig. 8. Activity Diagram Annotated with ≪rbac≫ Before Evolution (left-hand side), Added Model Elements (middle), and After Evolution (right-hand side)

lution is security preserving in both cases. For more details about the verification technique see [27].

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

Example In Fig. 8 (left-hand side), we show an activity diagram to which we want to add a new set of activities, introducing a web-check-in functionality to a flight booking system. The new activity "Check-in online" (middle of Fig. 8) is protected, but we do not add a proper role/right association to this activity, thus resulting in a security violating diagram (right-hand side Fig. 8).

4.2 Tool Support

The UMLsec Tool Suite provides mechanical tool support for analyzing UML specifications for security requirements using model-checkers and automated theorem provers for first-order logic . The tool support is based on an XML dialect called XMI which allows interchange of UML models. For this, the developer creates a model using a UML drawing tool capable of XMI export and stores it as an XMI file. The file is imported by the UMLsec analysis tool (for example, through its web interface) which analyses the UMLsec model with respect to the security requirements that are included. The results of the analysis are

given back to the developer, together with a modified UML model, where the weaknesses that were found are highlighted.

We also have a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes. In particular, the framework includes the UMLsec tool web interface, so that new routines are also accessible over this interface. The idea behind the framework is to provide a common programming framework for the developers of different verification modules. A tool developer should be able to concentrate on the implementation of the verification logic and not be required to implement the user interface.

As mentioned in Sect. 3, tool support for UMLseCh would be beneficial in at least two ways:

- Supporting the user in modelling expected evolutions explicitly in a graphical way, without using a particular grammar or textual abstract syntax, and supporting the specification of non-elementary changes.
- Supporting the decision of including a change based on verification techniques for model consistency preservation after change.

but more importantly:

- Supporting the decision of including a change based on verification techniques for security preservation after change

First steps in this direction have been done in the context of the EU project SecureChange for statical security properties. For more details refer to [27].

5 Conclusions and Future Work

For system evolution to be reliable, it must be carried out in a controlled manner. Such control must include both functional and quality perspectives, such as security, of a system. Control can only be achieved under structured and formal identification and analysis of change implications up front, i.e. a priori. In this chapter, we presented a step-by-step controlled change process with emphasis on preservation of security properties through and throughout change, where the first is characterized as a security-driven change and the second a functionality-driven change. As there are many reasons for evolution to come about, security may drive the change process or be directly or indirectly influenced by it. Our approach covers both. In particular, the chapter introduces the change notation UMLseCh that allows for formally expressing, tracing, and analysing for security property preservation. UMLseCh can be viewed as an extension of UMLsec in the context of secure systems evolution. We showed how can one use the notation to model change for different UMLsec diagrams, and how this approach could be useful for tool-aided security verification. Consequently, this work can be extended in different directions. First of all, compositional and incremental

techniques to reason about the security properties covered by UMLsec are necessary to take advantage of the precise model difference specification offered by UMLseCh. On the other hand, comprehensive tool support for both modelling and verification is key for a successful application of UMLseCh in practical contexts.

Acknowledgements

This research was partially supported by the EU project "Security Engineering for Lifelong Evolvable Systems" (*Secure Change*, ICT-FET-231101).

References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.
2. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 440–453. Springer, 2006.
3. S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software, 2004.
4. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Annual Symposium on Logic in Computer Science (LICS)*, pages 353–362, June 1989.
5. 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.
6. J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and verification of layered security protocols: A bank application. In S. Anderson, M. Felici, and B. Littlewood, editors, *SAFECOMP*, volume 2788 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 2003.
7. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS) - Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.
8. S. Höhn and J. Jürjens. Rubacon: automated support for model-based compliance engineering. In Robby [26], pages 875–878.
9. J. Jürjens. Formal Semantics for Interacting UML subsystems. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 29–44. International Federation for Information Processing (IFIP), Kluwer Academic Publishers, 2002.
10. J. Jürjens. *Principles for Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, 2002.
11. J. Jürjens. Model-based security engineering with UML. In A. Aldini, R. Gorrieri, and F. Martinelli, editors, *FOSAD*, volume 3655 of *Lecture Notes in Computer Science*, pages 42–77. Springer, 2004.

12. J. Jürjens. Code security analysis of a biometric authentication system using automated theorem provers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 138–149. IEEE Computer Society, 2005.
13. J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.
14. J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 322–331. ACM Press, 2005.
15. J. Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *MEMOCODE*, pages 89–98. IEEE, 2005.
16. J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–176. IEEE Computer Society, 2006.
17. J. Jürjens, J. Schreck, and P. Bartmann. Model-based security analysis for mobile communications. In Robby [26], pages 683–692.
18. J. Jürjens and P. Shabalin. Tools for secure systems development with UML. *Intern. Journal on Software Tools for Technology Transfer*, 9(5–6):527–544, Oct. 2007. Invited submission to the special issue for FASE 2004/05.
19. J. Jürjens and G. Wimmel. Formally testing fail-safety of electronic purse protocols. In *16th International Conference on Automated Software Engineering (ASE 2001)*, pages 408–411. IEEE Computer Society, 2001.
20. D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 6(9):53–69, 2007.
21. M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, 1998.
22. H. Lipson. Evolutionary systems design: Recognizing changes in security and survivability risks. Technical Report CMU/SEI-2006-TN-027, Carnegie Mellon Software Engineering Institute, September 2006.
23. H. Mantel. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 88–101, Oakland, CA, USA, 2002. IEEE Computer Society.
24. D. C. Petriu, C. M. Woodside, D. B. Petriu, J. Xu, T. Israr, G. Georg, R. B. France, J. M. Bieman, S. H. Houmb, and J. Jürjens. Performance analysis of security aspects in UML models. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 91–102. ACM, 2007.
25. A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proceedings of the International Conference in Graph Transformation (ICGT)*, pages 226–241. Springer, 2004.
26. Robby, editor. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008.
27. Secure Change Project. Deliverable 4.2. Available as `http://www-jj.cs.tu-dortmund.de/jj/deliverable_4_2.pdf`.
28. UML Revision Task Force. *OMG Unified Modeling Language: Specification*. Object Management Group (OMG), September 2001. `http://www.omg.org/spec/UML/1.4/PDF/index.htm`.
29. UMLsec group. UMLsec Tool Suite, 2001-2011. `http://www.umlsec.de`.
30. B. Watson. Non-functional analysis for UML models. In *Real-Time and Embedded Distributed Object Computing Workshop*. Object Management Group (OMG), July 15–18, 2002.

31. C. M. Woodside, D. C. Petriu, D. B. Petriu, J. Xu, T. A. Israr, G. Georg, R. B. France, J. M. Bieman, S. H. Houmb, and J. Jürjens. Performance analysis of security aspects by weaving scenarios extracted from UML models. *Journal of Systems and Software*, 82(1):56–74, 2009.