International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Jan Jürjens · Pasha Shabalin

Tools for Secure Systems Development with UML

the date of receipt and acceptance should be inserted later

Abstract For model-based development to be a success in practice, it needs to have a convincing added-value associated with its use. Our goal is to provide such added-value by developing tool-support for the analysis of UML models against difficult system requirements. Towards this goal, we describe a UML verification framework supporting the construction of automated requirements analysis tools for UML diagrams. The framework is connected to industrial CASE tools using XMI and allows convenient access to this data and to the human user. As a particular example, we present plugins for verifying models defined using the security extension UMLsec of UML. The verification framework allows advanced users of the UMLsec approach to themselves implement verification routines for the constraints of self-defined stereotypes. In particular, we focus on an analysis plug-in that utilizes the model-checker Spin to verify security properties of cryptography-based systems.

Keywords model-based development · tool-support · UML · security · formal verification

1 Introduction

There needs to be a convincing added-value to the usage of model-based development techniques before they will be widely adopted in industry. Our goal is to provide such added-value by developing tool-support for the analysis of UML models against system requirements which can be formulated at the level of the system model, and which cannot be manually checked in a reliable and efficient way (such as security requirements). Here, we describe a UML verification framework supporting the construction of automated requirements analysis tools for UML diagrams. Its design is influenced by experiences from long-standing efforts at our group regarding the development of the AutoFocus CASE-tool [HMR+98]. The framework is connected to industrial CASE tools using data integration with XMI [XMI02] and allows convenient access to this data and to the human user.

To be of interest in practice, the requirements that can be treated, and the method we propose for handling them, should fulfill the following constraints.

- The properties that can be specified and analyzed should be important and sophisticated enough so that it is necessary to consider them and that it would be difficult to do so manually.
- The analysis should be as automatic as possible to reduce usage costs.
- It should be efficient enough to be effectively and conveniently usable on an ongoing basis.
- It should be possible to use the approach with a modest training effort.

As an example for such requirements, we focus on security aspects. Data security aspects have become an increasingly important issue in developing distributed systems, especially in the electronic business sector. Because security attacks may cause very high damage (e.g., loss of money through fraud), the correctness of such systems is crucial.

Designing security-critical systems correctly is difficult. Today implementation and verification of security aspects of systems is done by experts with limited support from automated tools, and often after the system is implemented. That makes the design often error-prone. Therefore, the consideration of security aspects has to be integrated into general systems development. To facilitate analysis, it should be supported by automated tools. In order to receive broad acceptance, the methodology should employ common modeling techniques used in industry tailored for that purpose.

Jan Jürjens
Software and Systems Engineering,
TU Munich
http://www.jurjens.de/jan

Pasha Shabalin
Internet-based Information Systems
TU Munich
shabalin@in.tum.de

The UML extension UMLsec [Jür02, Jür04] has been proposed for modeling security properties of computer systems. It is based on a formal semantics which allows one to formally reason about security properties of the UMLsec model, and provides background for automated software tools, which can support the design of secure systems by formal verification and by model transformation preserving security-relevant properties.

Using UML as the basis both for the development method and for the new tools has many advantages. Many software developers are trained in UML, which can facilitate adoption of the technology. Existing UML models can be extended with the UMLsec constructs and used to analyze and improve security properties of existing systems. Many existing UML editors support standard UML extensions (like UMLsec) and can be readily used to edit UMLsec models. Therefore, tool support for the new technology does not require creating complex software suites from scratch, but allows one to add necessary functionality using a simple plug-in architecture.

In this paper, we present verification routines for constraints associated with the stereotypes of the UML security extension UMLsec. In particular, we focus on a plug-in that utilizes the model-checker Spin to verify security properties of UMLsec models which make use of cryptography (such as cryptographic protocols). To do so, the analysis routine extracts information from different diagram types (class, deployment, and Statechart diagrams) that may contain additional specific cryptography-related information. With respect to UMLsec, the goal here is thus two-fold. On the one hand, we aim to support the usage of UMLsec in practice by offering analysis routines connected to popular CASE tools which allow the automated verification of the constraints associated with the UMLsec stereotypes. On the other hand, the verification framework should allow advanced users of the UMLsec approach to themselves implement verification routines for the constraints of self-defined stereotypes, in a way that allows them to concentrate on the verification logic (rather than on user interface issues). This verification framework should then be useful beyond UMLsec, as well. For these purposes, the framework is available as open-source.

Sect. 2 recalls the core of the UMLsec extension from [Jür04] and shortly explains how UMLsec models can be analyzed with respect to security requirements. Sect. 3 presents the verification framework supporting the construction of automated requirements analysis tools for UML diagrams. We give a short overview over analysis tool plugins for the framework supporting verification of UMLsec models against security requirements. In Sect. 4, we present one of these plug-ins, which uses the Spin model-checker for checking data security requirements, for example of cryptographic protocols (some of the ideas here were sketched in the conference paper [JS04] and demonstrated in the tool paper [JS05]). Using a running example, we explain the translation from UMLsec models to Promela code and its analysis using Spin. We close with comparisons to related work, a discussion of our work and an outlook on further developments.

2 UML for Security: UMLsec

2.1 Overview

The UMLsec extension [Jür04] aims to support secure system development, in particular through the following goals.

- Given a system model described with UML, it should automatically evaluate it for security-related vulnerabilities in the design.
- The methodology should be available to developers not specialized in security, and allow them to ensure the necessary security properties of the system under design.
- Security properties are often imprecisely defined or misunderstood. Formulating security properties of a system can often be a challenge by itself. Therefore we should enable the user to define easily and unambiguously both security features and security requirements of the system.
- The costs of correcting flaws in a software system grow considerably in the process of development, therefore we would like to consider security from early design phases.
- One should be able to consider security on different levels of abstraction, and in the system context, since security can be violated on different levels.
- Users should be enabled to make use of established rules of prudent security engineering.

In particular, this is achieved by the following notational features.

- Basic security requirements such as *secrecy* and *integrity* are integrated into the language.
- Different threat scenarios are considered depending on the adversary strengths.
- Common security concepts like *tamper-resistant hardware* are available.
- Common security mechanisms like *access control* are included in the notation.
- Cryptographic primitives are defined at an appropriate level of abstraction, since they cannot be feasibly modeled and automatically verified at the bit-sequence level. This relies on the assumption that the cryptographic algorithms used are secure. The goal is to establish that they are *used* securely as well, which is a main source of security problems in practice.
- Physical security of the deployed system is taken into account by the model and its security analysis.
- Security management such as secure workflows has been addressed.

Stereotype	Base Class	Tags	Description
Internet	link		Internet connection
encrypted	link		encrypted connection
LAN	link, node		LAN connection wire
smart card	node		smart card node
secure links	subsystem		enforces secure communication links
secrecy	dependency		assumes secrecy
integrity	dependency		assumes integrity
high	dependency		assumes high sensitivity, both secrecy and integrity
secure dependency	subsystem		structural interaction data security
critical	object, subsystem	secrecy, integrity, high	critical object
no down-flow	subsystem	secret	prevents leak of information

Fig. 1 UMLsec stereotypes

- Domain-specific extensions such as Java, smart cards, or CORBA based systems are supported by the notation.

UMLsec supports these features on the technical level in the following ways:

- It extends the language with new constructs for augmenting a UML model with security-relevant *properties* and *requirements*.
- UMLsec defines a formally precise semantics for security constructs and for the relevant UML fragment using UML Machines, an extended version of the Abstract State Machines [Gur95] with explicit support for UML-style communication and composition. It allows construction of a single formal description for the complete UML model (for the simplified fragment of UML that is used), including information from all diagrams and all abstraction layers.

An important advantage of the UMLsec approach is that it provides the developer with notational elements that represent the most important security requirements which can be directly used in the model, together with their formal definitions. This eliminates the need for the users to themselves formalize these requirements. This is particularly important to enable the approach to be widely used in the industry, also by developers without a formal background.

UMLsec follows the UML specification [UML03] which introduces profiles as a lightweight mechanism for extending the language (as opposed to heavyweight extensions through modifying the UML metamodel). A profile contains definitions for *stereotypes*, *tagged values* and *constraints*. Such a lightweight extension should be ”strictly additive to the standard UML semantics. This means that such extensions must not conflict with or contradict the standard semantics.” [UML03] In particular, this assures that UMLsec can be used to extend any UML model without conflict with existing tools or other UML extension.

Stereotypes define new types of modeling elements extending the semantics of existing types in the UML metamodel. Their notation consists of the name of the stereotype written in double angle brackets « », attached to the extended model element. This model element is then interpreted according to the meaning ascribed to the stereotype.

Tagged values allow explicitly attaching a data value to a model element. They are represented by a `name` = `value` pair in curly brackets associated with model elements. The value can be either a simple datatype value, or a reference to another model element.

Constraints can be attached to a model element to refine its semantics. Attached to a stereotype, constraints must be observed by all model elements marked by that stereotype.

With UMLsec, stereotypes and tagged value are used to define data security requirements on model elements, and to define their security-relevant properties. The constraints, which are automatically derived by our tools using the threat model described in the next subsection, formulate rules which must be met by the design to support the requested security properties. Some examples for UMLsec stereotypes together with associated tags and informal descriptions are listed in Fig. 1.

We refer the reader to the book [Jür04] for the complete reference on UMLsec, and describe here only the part which is needed in this article. The Spin plugin, described in Sect. 4, interprets and translates to Promela following UMLsec stereotypes.

The stereotype ⟪secrecy⟫ marks data which the adversary should not get to know in plain-text.

Stereotypes ⟪LAN⟫, ⟪Internet⟫, and ⟪Wireless⟫ describe physical characteristics of communication links, and thus define how the adversary can possibly affect the communication performed over them.

2.2 Relevant Fragment of UMLsec

We shortly recall the fragment of UMLsec needed in this paper. To perform a formal verification of the UMLsec models, we need mathematically precise definitions of the UMLsec models, including their dynamic behavior, which we also explain shortly (again more details can be found in [Jür04]). The foundations are defined in [Jür04] using *UML Machines*, which are inspired by the Abstract State Machines (ASM) [Gur95]. A UML Machine is a transition system with built-in communication mechanisms, which defines the behavior of a system component. It is defined by the *initial state*, *transition rules* which are applied iteratively and define how the machine state changes in time, and two multi-set buffers (*output queue* and *input queue*). A UML Machine communicates with other system parts by adding messages to its output queue and retrieving messages from its input queue. All the messages in the system compose the set **Events**. To build an executable UML specification in a modular way, by combining a set of UML Machines together with communication links connecting them, one can use the notion of a *UML Machine System* (UMS) also defined in [Jür04]. The intuition is that a UMS models a computer system that is divided into components that may communicate by sending messages through communication links and whose execution is scheduled by a specified scheduler.

We now define the fragment of UMLsec used here together with important parts of its execution semantics, which is extended with the adversary model in the next subsection.

Class Diagrams are used to define classes in the model and associations between them. A class contains attributes (local variables), which can be labeled with the tag {secrecy} to specify that the adversary should not get to know the initial value of the variable. Furthermore, classes may be specified to contain *operations*, which define the messages accepted by the class. Each operation can have zero or more parameters. An association has two *association ends*, each connected to a class. Two classes, connected by an association, can reference each other using the opposite association end name. The model developer can define class attributes using any data type available within the UML notation, and can assign *initial values* to them using the UMLsec notation.

Statechart diagrams define dynamic behavior and changes in the internal state of the UML classes in response to incoming messages. *Triggers*, *guards*, and *actions* in a Statechart diagram are formulated using the cryptography language described in Sect. 4.2. Following the UMLsec semantics from [Jür04] and the UML specification, each Statechart diagram is translated into a state machine with message queues for input and output. Its functionality is modeled by a loop which repeatedly dequeues a message from the input queue and executes the transitions that are enabled until another message is required to continue.

```
currentMessage = {}
while (true) {
    while (there are selected transitions) {
        execute one of the selected transitions
        currentMessage = {}
        if (a final state reached) HALT
    }

    currentMessage = Q_in.dequeue

} while true
```

Fig. 2 Executing transitions in a loop

For the behavioral semantics of Statecharts, we follow the one given in [Jür04]. We shortly recall some important features.

A Statechart transition is selected for execution when the following conditions are met:

- The source state is currently active.
- The current input message matches the trigger OR the transition is a completion transition.
- The guard evaluates to true.

Note that several transitions can be selected for execution simultaneously, when they all have the same trigger and source state. Parameters of the current message are copied to class attributes before the corresponding guard is evaluated. The current state of a state machine is iteratively updated following the loop presented in Fig. 2. The idea of the nested loop is that all possible completion transitions are always executed before the next message from the input queue Q_in is consumed. This behavior follows the run-to-completion semantics described in the UML specification ensuring that a message can only be dequeued and dispatched after the previous message has been fully completed. In case the new current message does not cause any transition to be selected, it is dropped and the next message from the input queue is consumed. The execution blocks on an attempt to read from an empty input queue until a new message arrives. After a message causes a transition to be selected and the transition was executed, the message is discarded and cannot be used again.

Deployment diagrams specify the physical structure of the system. *Objects* in a deployment diagram represent instances of classes. Objects are contained in *nodes* which represent physical parts of the system. A *link* represents a physical communication connection between two nodes. Physical properties of the link are defined by attaching for example one of the UMLsec stereotypes ⟪LAN⟫, ⟪Internet⟫, or ⟪Wireless⟫. Together with the *adversary type*, the stereotype defines the *adversary capabilities* regarding the link, as described in Sect. 2.3.

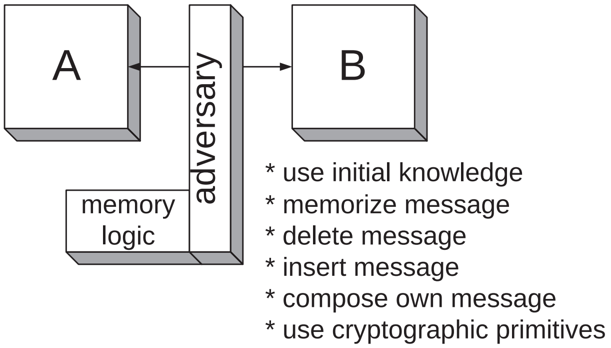

Fig. 3 Simulating adversary behavior

2.3 Security Analysis of UMLsec Models

To apply formal verification methods for analyzing security properties of an open distributed system, it is necessary to ”close” it by modeling the possible interactions between the system and the outside world. This includes behavior of the potential adversary trying to break or compromise the system. For that reason, reliability of the automated verification of security properties depend on the correctness and completeness of the simulated adversary behavior. The adversary is modeled through the basic actions that can be performed to attack the system (see Fig. 3). The complete attack scenarios are then automatically derived from these.

We explain the needed background on the adversary model we use for the security verification. It is built on the adversary model from [Jür04]. More information and examples can be found there. The extent of the possible interference with the system execution depends on the physical properties of the system, and on the adversary abilities. The adversary model is thus defined through certain basic capabilities, depending on the physical properties of the system, and on the strength of the adversary that is considered. The UMLsec methodology provides the possibility to specify these parameters in a *generic* and *modular* way.

By *generic*, we mean that the goal is not to encode all known possible protocol attack strategies on the whole, which would be possibly unsound and hardly practical:

- By construction, the verification procedure will find only attacks which we have (correctly) encoded in the tool. We would however like to detect *any* cryptographically possible attack that is detectable within our given system model and on our level of abstraction.
- The programming task of encoding all known attack strategies is very complex, especially because generally these strategies describe a general attack idea, which must be tailored for an attack on the particular system.

This generic attacker model goes back to an influential approach to security verification proposed in [DY83]. The adversary can use all cryptographic functions defined in Section 4.2 to analyze and compose messages, under the assumption that these functions are themselves secure and correctly implemented (the *perfect cryptography* assumption; see for example [AJ01] for a discussion):

- The decryption key must be known in order to extract the plaintext corresponding to a given ciphertext.
- There is enough redundancy in the cryptosystem that a ciphertext can only be generated using encryption with the appropriate key and message.

For example for an online banking application, a user of the application could read and modify information on the Internet, and access the data on the client node. A malicious employee however could read and manipulate traffic in the internal LAN, and access the internal servers. This is specified in UMLsec using a function mapping an adversary type A and a stereotype s characterizing a physical property of the system in the deployment diagram (such as «Internet») to a set of threats $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$. Each of the threats is implemented by a possible adversary action in the system model.

- **read** gives the adversary the capability to read the information from the communication link and store it in the internal variables.
- **insert** allows the adversary to insert his own messages into the communication link. The message is created from the information known to the adversary, constructed from the initial adversary knowledge and the information learned by the adversary from the previous communication.
- **delete** allows the adversary to remove a message from the communication link.

To define the attack capabilities of an *adversary type A* against a UML model element labeled with the stereotype s, we thus define the function $\mathsf{Threats}_A(s)$ returning a subset of the set $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$ of *abstract threats*. The examples in Fig. 4 and 5 define the threats associated with the *default* and *insider* adversary types for some of the stereotypes; new adversary types can be defined in a similar way.

From the above definitions on the stereotype-level, one can derive how the adversary can interact with system parts represented by concrete model elements: For a link l in a deployment diagram contained in the subsystem S, we define the set $\mathsf{threats}_A^S(l)$ of concrete threats

Stereotype	$\mathsf{Threats}_{default}()$
Internet	$\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$
encrypted	$\{\mathsf{delete}\}$
LAN	$\emptyset$
smart card	$\emptyset$

Fig. 4 Threats from the *default* adversary

Stereotype	$\mathsf{Threats}_{insider}()$
Internet	$\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$
encrypted	$\{\mathsf{delete}\}$
LAN	$\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$
smart card	$\emptyset$

Fig. 5 Threats from the *insider* adversary

to be the smallest set satisfying the following conditions. If each node n that l is contained in[1] carries a stereotype s_n with $\mathsf{access} \in \mathsf{Threats}_A(s_n)$ then:

- If l carries a stereotype s with $\mathsf{delete} \in \mathsf{Threats}_A(s)$ then $\mathsf{delete} \in \mathsf{threats}_A^S(l)$.
- If l carries a stereotype s with $\mathsf{insert} \in \mathsf{Threats}_A(s)$ then $\mathsf{insert} \in \mathsf{threats}_A^S(l)$.
- If l carries a stereotype s with $\mathsf{read} \in \mathsf{Threats}_A(s)$ then $\mathsf{read} \in \mathsf{threats}_A^S(l)$.
- If l is connected to a node that carries a stereotype s with $\mathsf{access} \in \mathsf{Threats}_A(s)$ then $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\} \subseteq \mathsf{threats}_A^S(l)$.

The idea is that $\mathsf{threats}_A^S(x)$ specifies the threat scenario against a component or link x in the subsystem S that is associated with an adversary type A. On the one hand, the threat scenario determines which data the adversary can obtain by accessing components. On the other hand, it determines which actions the adversary can apply to the links according to the threat scenario. delete means that the adversary may delete the messages on the corresponding link, read allows him to read the messages on the link, and insert allows him to insert messages into the link. The new messages can be created by applying cryptographic primitives to the initial knowledge of the adversary and the data that was previously read.

To investigate the security of the system with respect to the chosen *adversary type*, we build an executable specification of the system combined with the adversary model, and verify its security properties.

3 Framework for UMLsec Processing

3.1 Representing UML in XML

The XML Metadata Interchange (XMI) language [XMI02] issued by the Object Management Group (OMG) is a standard for storing UML models into a file used by many UML tools. It allows one to easily develop lightweight software extending the functionality of existing UML tools. The XMI format is compliant with the Meta Object Facility (MOF) framework [MOF02] for specifying meta-information (also called metamodels). Applications of the MOF framework include the definition of modeling languages such as for example UML and Common Warehouse Model (CWM). The framework is defined using a four-level data abstraction model, shown in Fig. 6. The lowest level M0 deals with data instances in the program during execution, for example: *Mr. Smith, 35 years old, lives in New York*. The level M1 describes data models, for example a UML model of the application. To continue with our example, this could be a class model defining the class: *Person* with attributes: *Name, Age, Address*. The next abstraction level M2 corresponds to the modeling language itself, in our case the UML notation. The last abstraction level M3 is where these modeling languages are defined using the MOF framework. It uses three main elements: *MOF Object*, *MOF Association* and *MOF Package*, and four secondary elements: *Data Types*, *Constants*, *Exceptions* and *Constraints*.

The XML Metadata Interchange then defines a mapping from MOF to XML. It can be used to automatically produce an XML interchange format for any language described with MOF. For example, to produce a standardized UML interchange format, we need to define the UML language using MOF, and use XMI mapping rules to derive DTDs and XML Schemas for UML serialization. MOF itself is defined using the MOF framework, and therefore XMI can be applied not only for metamodel instances but for metamodels themselves (as they are also instances of a metamodel which is MOF).

Also relevant is the Java Metadata Interface (JMI) which defines a MOF-to-Java mapping (similarly to the MOF-to-XML mapping provided by XMI). It is used for deriving Java interfaces tailored for accessing instances of a particular metamodel. As MOF itself is MOF-compliant, it can be used to access metamodels as well. The standard also defines a set of *reflective* interfaces which can be used similarly to the metamodel-specific API without prior knowledge of the metamodel.

There are several possibilites for working with XMI files:

- XML parsing and transformation languages coupled with the XML standard (such as XPath, XSLT).
- Any high-level language with appropriate libraries (such as Java, C++, Perl).
- Data-binding.

The first two methods, although quite flexible, require more development effort. However, for UML processing, we are concerned about the data contained in documents, rather than the document itself and its structure. For this purpose, data binding offers an approach to working with XML data which is more specifically tailored to our goals.

There exist several libraries supporting data-binding for XML. The widely used Castor library [Cas03] offers the developer a relatively generic representation of the UML model, on the level of MOF constructs. However, there are also data-binding libraries which provide representation of a UML model as an XMI file on the abstraction level of a UML model. This allows the

[1] Note that nodes and subsystems may be mutually nested in each other.

M3	Meta-Metamodel	MetaClass, MetaAssociation - MOF Model
M2	Metamodel	Class, Attribute, Dependency - UML (as a language), CWM
M1	Model	Person, Age, Address - UML Model
M0	Data	Mr. Smith, 35 years, New York - Running program

Fig. 6 MOF framework

developer to work directly with UML concepts (such as classes, Statecharts, stereotypes, etc.). We use the MDR (Meta Data Repository) library which is part of the Netbeans project [Net03] and also used by the freely available UML modeling tool Poseidon 1.6 Community Edition [Gen03]. Another such library is the Novosoft NSUML project [NSU03]. The MDR library implements a MOF-compliant repository with support for XMI and JMI standards. Fig. 7 illustrates how the repository is used for working with UML models.

The XMI description of the modeling language is used to customize the MDR for working with the particular model type, UML in this case (step 1). The XMI description of UML is published by the Object Management Group (OMG). A storage (*extent* in the MDR terminology) customized for the given model type is created (step 2). Additionally, based on the XMI specification of the modeling language, MDR creates JMI (Java Metadata Interface) definitions for accessing the model (step 3). The interfaces are used from the user application to manipulate the UML model on the conceptual level of UML. A UML model is loaded into the repository (step 4). Now it can be accessed through the JMI interfaces from a Java application. The model can be read, modified, and later saved into an XMI file again.

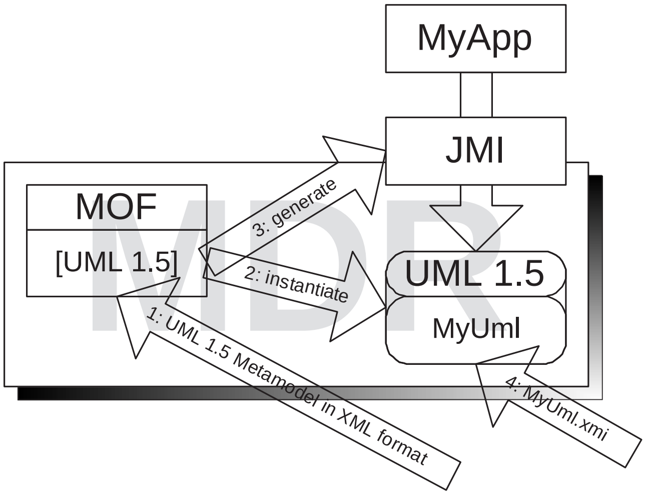

Fig. 7 Using the MDR library

Because it allows one to use the XMI DTD files officially released by the OMG, using MDR in a UML processing tool promises a high standard compatibility and is hoped to facilitate upgrading to upcoming UML versions, by replacing the UML Metamodel file loaded in the first step with a metamodel of the next UML version (and possibly making further adjustments).

3.2 Framework Architecture

The UMLsec framework was designed to support a development and execution environment for various UML model processing tools, including but not restricted to UMLsec analysis plugins. The following considerations have influenced its design.

- A tool developer should be able to concentrate on the verification logic rather than on the interface or UML processing issues. The latter two aspects should be handled by the framework and be clearly separated from the implementation of the independent tools.
- The framework is intended for use by different developers providing extensions to it, who may work independently. Therefore, the design must be kept simple and straightforward and easy to use and maintain.
- It should be possible to use tools through different input/output interfaces. In particular, a traditional console mode and a web interface mode are important and a graphical interface would be desirable. On the other hand, the support for different media should not significantly increase the development effort for the tool plugins.

The architecture and basic functionality of the UMLsec framework are presented in Fig. 8. As the first step, the developer creates a UMLsec model and stores it in the UML 1.5 / XMI 1.2 file format. The file is imported by the framework into the internal MDR repository. Separate tools, such as the static checkers and dynamic checkers in Fig. 8, access the model through JMI interfaces, generated by the MDR library. A tool can optionally request additional parameters from the user. A static checker parses the model, verifies its static features, and delivers verification results to the error analyzer. A dynamic checker translates the relevant fragment of the UMLsec model into an intermediate language for processing by an external tool. Results of its ex-

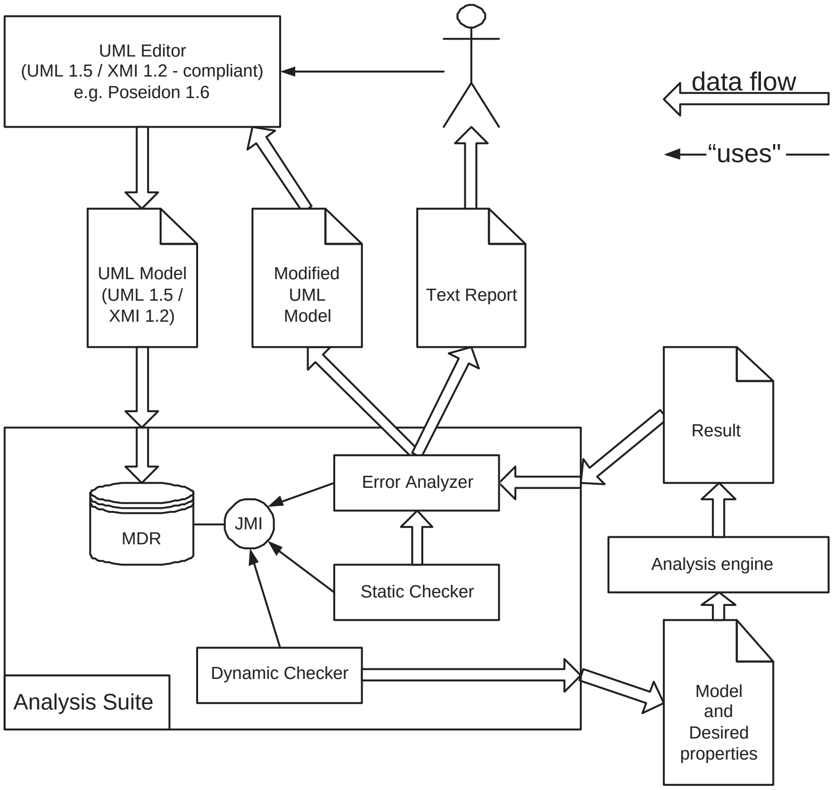

Fig. 8 UML tools suite

ecution are delivered back to the error analyzer. The error analyzer uses the information that is received to produce a text report for the developer describing the problems that were found, and a modified UMLsec model, where the problems are visualized and possibly corrected.

For creating and editing UMLsec models we are using the UML editor Poseidon 1.6 Community Edition, freely available from Gentleware [Gen03]. However, it should be possible to use any other UML editor, capable of storing the model in the UML 1.5 / XMI 1.2 format.

To improve usability, the user interface supports different input/output media. To still keep tool extension development simple and avoid redundant code, the framework allows one to program plugins using the Console, GUI and Web user interfaces without additional implementation effort. To achieve this, input/output operations are handled by the framework transparently for the tools. Tools expose their functionality to the framework through four interfaces, as presented in Fig. 9.

– The IToolBase interface defines the common functionality for all media types - Console, Gui and Web.
– IToolConsole, IToolGui and IToolWeb interfaces define additional functionality required for each particular medium.

Each tool must implement at least the IToolBase and IToolConsole interfaces. The framework provides default wrappers for the GUI and Web mode, implemented in the classes DefaultGuiWrapper and DefaultWebWrapper, as presented in Fig. 10. Wrappers are used by the framework automatically when the tool returns *null* from getGui or getWeb functions of the IToolBase interface respectively. The wrappers render the textual output of the tool, obtained through the IToolConsole interface, to a TextBox control in the GUI mode, or to an HTML page in the web mode. However, each tool can implement IToolGui and/or IToolWeb interfaces itself to fully exploit the functionality of the corresponding media, for example to use GUI mode capabilities for displaying graphical information.

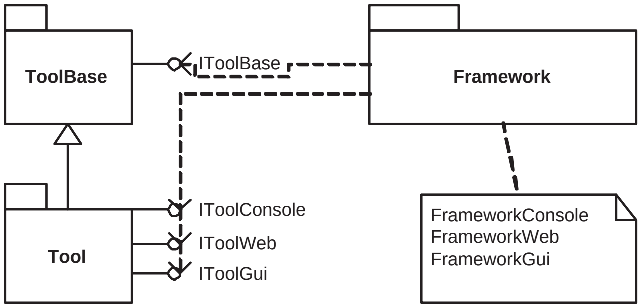

Fig. 9 Tool interfaces

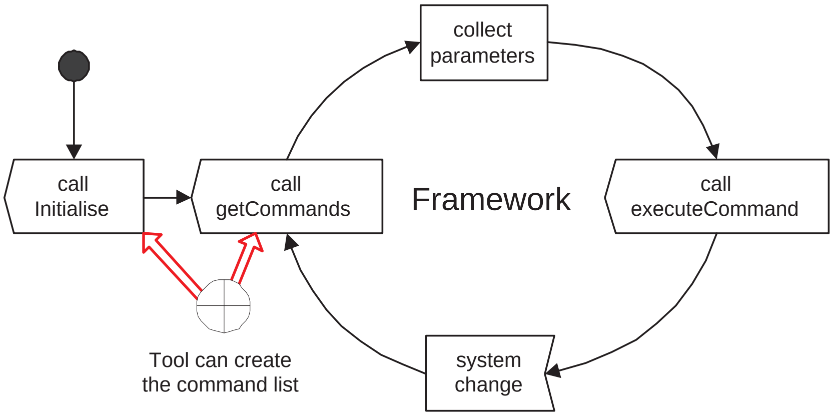

Fig. 11 Framework life cycle

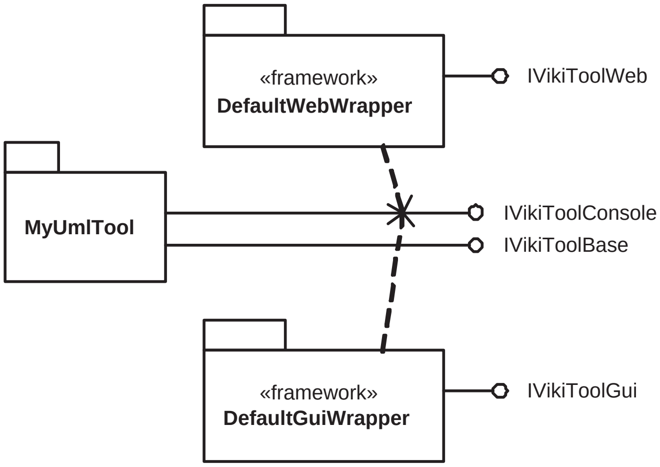

Fig. 10 Framework wrappers

To support media-independent functionality, not only the tool output, but also the input of necessary parameters is handled in a generic way independently of the current input/output medium. To support this concept, the implementation of the IToolConsole interface for each tool conforms with following rules:

– A tool exposes a set of commands which it can execute.
– Every single command is not interactive. It receives parameters, executes, and delivers feedback.
– The tool has its internal state which is preserved between commands. This allows one to perform complex and interactive operations on the UML model.
– The tool is given the UML model that it operates on. It may load further models if necessary.

As presented in Fig. 11, the appropriate getCommands function is called every time the list of commands is presented to the tool user. The tool developer can create the list of commands once during the tool initialization, save it in a tool class variable, and return the list every time the getCommands function is called. Alternatively, the list of commands can be generated anew on every function call, based on the current tool state. The list of commands exposed by the tool can be different for different input/output media, enabling the tool to provide different functionality through the different media.

3.3 UMLsec Verification Tools

Using the framework described in the previous section, a number of plug-ins for UML (and in particular UMLsec) model verification have been implemented. They fall into several categories.

Static features: For each of the static security requirements in UMLsec (such as «secure links» and «secure dependency»), we have implemented an analysis plugin which directly checks the relevant conditions in a Java routine.

Simple dynamic features: For dynamic properties, we need a mapping from UMLsec models to a representation of their behavioral semantics as event histories. This is done for Statecharts, activity diagrams, sequence diagrams, and for subsystems containing the above diagram types in four other plugins. The semantics is analyzed to verify basic security requirement, defined on the behavioral level.

Complex dynamic features: For complex dynamic properties, the UMLsec model is translated into the input language of a suitable analysis tool. As an example, we describe an integration with the model-checker Spin to verify the «data security» constraints in the next subsection.

Runtime analysis: There are also plug-ins analyzing UMLsec models with respect to security-critical runtime information. An example is the analysis of security permissions from configurations for SAP R/3 business applications with respect to security rules and business processes formulated in UML [HJ03].

4 Model-Checking UMLsec Specifications

4.1 Overview

As an example for the verification routines implemented in the UMLsec tool, we present an integration with the model-checker Spin [Hol03] for verifying crypto-based software as defined by the «data security» requirement from [Jür04]. «data security» is a UMLsec stereotype for subsystems which one can use to specify that certain attributes in the subsystem that are marked using the {secrecy} tag are supposed to remain secret given the behavior specified in the UML model. These UML subsystems, such as cryptographic protocols, can be specified to make use of cryptographic algorithms. That the secrecy requirement is actually fulfilled (as far as one can determine from the model), is formalized using the constraint associated with «data security». This is done with respect to a formal semantics of (a restricted version of) the subsystem and its subdiagrams, and using an adversary model arising from the physical security specification given in the deployment diagram contained in the subsystem. This was shortly sketched in Sect. 2.

Many model-checkers support higher level languages like Promela for the Spin model-checker [Hol03] and the SMV notation [SMV]. These languages may describe the system as an interleaving composition of several automata running in parallel. Different processes may communicate by message passing or through shared variables. The second input to the model-checker is a property which is checked against the given transition system. To specify complex properties, one can make use of temporal logic. On successful termination, the model-checker delivers one of the following two results:

- **System meets the requirement:** By evaluating all reachable states of the transition system, the model-checker establishes that it always fulfills the desired property.
- **System violates the requirement:** The model checker finds a reachable system state where the specified property does not hold. It can also provide the error trace which illustrates how the system entered the undesired state.

Due to the notorious state explosion problem, the third possible outcome of the model-checking (no successful termination within acceptable time limits) is not uncommon for real life systems. In particular, components like the adversary in security analysis, increase the complexity of the model significantly. In our approach, we handle the complexity issue by using the bounded model-checking approach which allows one to obtain partial verification results without building the complete state space of the modeled system. More concretely, we use the model-checker Spin [Hol03] that supports automatic verification of finite-state reactive systems given in form of a state-transition system against properties expressed in Linear Time Logic (LTL).

The tool verifies a subset of dynamic security requirements, which can be expressed with the UMLsec profile, by translating the model into the Promela language (following the UML formal semantics) for further processing by the Spin model-checker. Specifically, it checks whether the initial value of a class attribute, marked with the ⟨⟨secrecy⟩⟩ stereotype, can be obtained in plain text by the adversary. To check this constraint associated with «data security» attached to a subsystem, we construct a formal model of the system behavior. This model is extended with an adversary model derived from the threat information in the deployment diagram. This formal model is then verified with respect to the security requirement contained in the class diagram. If this requirement is violated, an attack trace is produced.

4.2 Cryptographic Notation

The BNF representation of the cryptographic expressions is given in Fig. 12. There are three top-level constructs. A *guard* is used to define conditional transitions in UML Statechart diagrams. An *effect* is used to define UML actions, including assignment of new values to class attributes and the invocation of methods. An expression is used in class diagrams to specify initial values for class attributes, and as a part of *guards* and *effects*. Valid identifiers are names of class attributes, other classes using named associations, various constants, and the keyword: `this`. The functions `SenderOf`, `PublicKeyOf`, `SecretKeyOf`, `SymmetricKeyOf`, and `NonceOf` return the corresponding attributes of the object. Note that the function `NonceOf` relies on the assumption that for protocols with symmetric session keys, at each iteration of the protocol a new object with a fresh session key is generated. One can modify the definition to allow each object to have several symmetric keys.

The function `ApplyKey` performs the cryptographic operations of encryption, decryption, signing, and extraction from signatures (as formalized below). Expressions can also include the concatenation and indexing operators `::` and `[]` (where a concatenation of n expressions followed by `[`m`]` evaluates to the mth of these expressions if $m \leq n$). Furthermore, one can use the Boolean comparison operators `==` (equal) and `!=` (not equal) between expressions, the assignment `=` of expressions to attributes, as well as events (which specify incoming method calls at Statechart transitions).

For any symmetric key k, any asymmetric key pair consisting of a secret key sk and a public key pk, and any message m, the following rules apply:

- `ApplyKey`(`ApplyKey`$(m, k), k) = m$
  (symmetric encryption)
- `ApplyKey`(`ApplyKey`$(m, pk), sk) = m$
  (asymmetric encryption)
- `ApplyKey`(`ApplyKey`$(m, sk), pk) = m$
  (digital signature)

```
<toplevel>     ::= <guard> |
                   <effect> |
                   <expression>

<effect>       ::= <functioncall> |
                   <assignment> |
                   <effect> "," <effect>

<guard>        ::= <expression> <compareop>
                                     <expression> |
                   else

<compareop>    ::= "==" |
                   "!="

<functioncall> ::= <expression>"."<identifier>
                                     "("paramlist")"

<identifier>   ::= letter { letter | digit }

<paramlist>    ::=                   |
                   <expression>      |
                   <expression> "," <paramlist>

<assignment>   ::= <identifier> "=" <expression>

<expression>   ::= "(" <expression> ")" |
                 <expression> "::" <expression> |
                 <expression> "[" integer "]" |
                 <identifier> |
                 "this" |
                 "ApplyKey" "(" <expression> ","
                                     <expression> ")" |
                 "SenderOf" "(" <identifier> ")" |
                 "PublicKeyOf" "(" <identifier> ")" |
                 "SecretKeyOf" "(" <identifier> ")" |
                 "SymmetricKeyOf" "(" <identifier> ")"
                 | "NonceOf" "(" <identifier> ")"
```

Fig. 12 UMLsec Cryptography Language

The first rule axiomatizes the functional properties of any symmetric encryption algorithm, the latter two rules the properties of the RSA asymmetric encryption algorithm [RSA78].

The SenderOf function is given as the input argument a message received from the communication channel and returns the object identifier of the message sender. For example, the following expression returns the public key of the sender of the message *msg*:

```
PublicKeyOf(SenderOf(msg))
```

Note that the adversary may be able to forge the message sender information.

Sending a message to an object is encoded in UML actions in Statechart diagrams using an expression of the form:

```
ObjectId.FunctionName(param1, param2, ...)
```

where the ObjectId can be

- either hard-coded using a name of an association end or the keyword: this,
- or obtained at runtime from a class attribute carrying the value of an object identifier,
- or evaluated from a SenderOf expression.

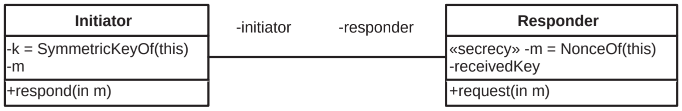

Fig. 13 Example class diagram

If a message is sent to an object which does not support it, or an invalid runtime ObjectId is used, the message is omitted by the system.

Example We introduce a UMLsec specification of a simple cryptographic protocol, which we use in the remainder of this paper as a running example. In this simple (and obviously insecure) protocol, Alice (of class Initiator) wants to receive some secret information from Bob (of class Responder). Alice sends to Bob her key, and Bob returns the secret value encrypted under the key:

$$\begin{array}{llll} \text{Alice} & \rightarrow \text{Bob} & : & k \\ \text{Bob} & \rightarrow \text{Alice} & : & \{x\}_k \end{array}$$

The UML model of the example is presented in Figures 13 through 16. Fig. 13 contains a class diagram defining the data structure of the system consisting of the Initiator and the Responder. Note that the attribute m of the Responder class is marked with the «secrecy» stereotype, which expresses the requirement that the content of this attribute is never leaked to the adversary. Fig. 14 contains a deployment diagram describing the physical layer underlying the protocol. The communication link is marked with the stereotype «LAN», meaning that the communication link is supposed to be a connection in a local area network, which implies that the (internal) adversary we consider in this example is capable of reading and writing on the link. Fig. 15 contains a Statechart specifying the behavior of the Initiator in the protocol sketched above, and Fig. 16 a Statechart for the Responder.

4.3 Translation to Promela: Types

We explain some key points in the automatic translation of UMLsec models to Promela code and its analysis using the Spin model-checker at the hand of our running example. We use the Spin model-checker since we found it particularly suitable for the verification of communicating distributed systems. Also, Spin's *on-the-fly* model-checking (which allows one to partially verify a model

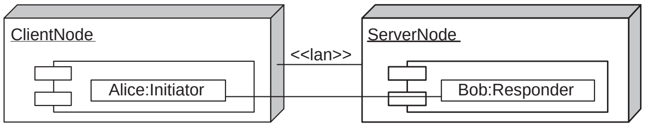

Fig. 14 Example Deployment Diagram

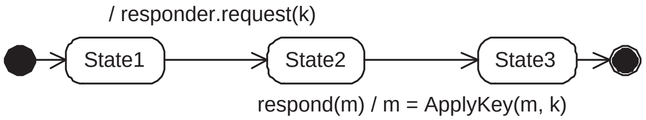

Fig. 15 Example Initiator Statechart

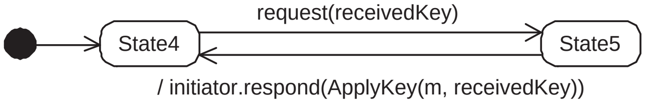

Fig. 16 Example Responder Statechart

without building the full state space) seems suitable for verifying security requirements against the highly non-deterministic adversary models.

Parameters and Data Types In a UML model, developers can use a range of predefined data types, and can also define their own data types. In contrast, model-checker notations usually support only a very limited set of data types (in the case of Promela: Boolean, Integer, and enumerated types [Hol03]). For a given UML model, we thus need to define a mapping of the complex UML data types onto the limited set of data types supported by the model-checker. We discuss two obvious approaches for constructing such a mapping and explain why they are not suitable for our needs, which motivates the solution we then propose. We use the term *atomic values* for values like an encryption key k, or a message v, which are considered to be unique and cannot be derived from other atomic values. We use the term *complex values* for data expressions constructed by applying operations (the *data transformation functions*) to atomic values. An example is the expression $\{v\}_k$, which is the value v encrypted under the key k.

We consider a model with a base data type (say, Integer) which we assume to include the atomic values and discuss three possibilities of representing and processing complex values: simple enumeration, fixed types, and dynamic types.

Simple Enumeration We use the Integer data type to enumerate all possible data values. For the $\{v\}_k$ expression we assign a new integer value to every combination of atomic values v and k. The data transformation functions are represented by a simple mapping function. In this approach, it is difficult to decide which combinations of values are possible and need to be enumerated. The translation process and the resulting code are complex. The internal logic of different processes in the translated code becomes mutually dependent. Detecting and enumerating all possible combinations is in fact the task of the model-checker. Implementing the same logic in the translator complicates the translation process and the resulting model. The HUGO UML to Promela translator [SKM01], for example, allows using native model-checker data types in the function parameters in the UML Model. It does not explicitly implement enumerating of all the possible complex data values, but it may be possible to be extended in this way. However, taking the drawbacks discussed above into account, we consider other possibilities.

Fixed Types The UML fragment that is used explicitly defines the data types of all variables, parameters and return values that can be used in the model. For every data type, it is then possible to enumerate all its values and define data transformation functions hardcoded in the resulting model-checker code according to the data type they process. We are not aware of any existing tools implementing this approach. Compared to the first solution, it would result in a better structured code, and the translation logic would be cleaner and easier to understand. However, we would have to limit the developer to use only those data types known to our translator. Alternatively, one could request the developer to provide mapping rules describing how the data types in the UML model relate to the data types defined in the UML fragment. Both alternatives would mean significant additional effort which might prevent developers from using the technology.

Dynamic Types The solution we propose is the dynamic handling of data types, where the message itself carries information about its type. The complex data type is defined during the translation process and holds any value which may appear in the system during its execution. For this, the tool builds a *type graph*. Starting from the root node of type *atomic*, and by using all expressions which are met in the model (namely, initial values, transition effects, and transition guards), the tool creates new vertices in the graph as necessary. The data transformation functions are represented as edges in the graph, and the data types as nodes.

The data graph for our example is presented in Fig. 17. It contains two vertices representing a simple variable (root) and an encrypted variable, and two edges representing encryption and decryption. Based on the data graph, the complex data type is encoded by a structure which holds as many atomic values as necessary to represent the most complex vertex plus a variable to encode the actual value type. Then the tool defines a set of data transformation functions which perform operations on the data types, for each edge in the graph. The translation result for our example in the Promela notation is

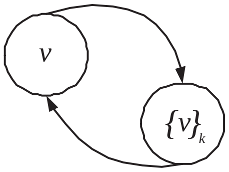

Fig. 17 Example data graph

```
#define MT_GARBAGE 1
#define MT_v 2
#define MT_LvRk 3

typedef MSG {
    TYPE_MSGTYPE messageType;
    TYPE_DATAVAL param1;
    TYPE_DATAVAL param2;
}
inline ApplyKey(message, key) {
if
    :: message.messageType == MT_GARBAGE -> ;
    :: message.messageType == MT_v ->
        message.messageType = MT_LvRk;
        message.param2 = key;
   :: message.messageType == MT_LvRk ->
    if
        :: message.param2 == InverseKey(key) ->
           message.messageType = MT_v;
        :: else -> message.messageType = MT_GARBAGE;
      fi;
fi;
}
```

Fig. 18 Translated example - fragment

given in Fig. 18. The MSG structure is used to represent the complex data type. Its *messageType* field has values of the form MT_xxx and defines which vertex in the data graph the structure currently represents. The $param1$ and $param2$ fields store the needed atomic values. For example, to encode the value $\{v\}_k$, the $param1$ field stores the value v, $param2$ stores the key k, and *messageType* stores the value MT_LvRk. The *ApplyKey* function defines a transformation rule for the graph: encryption on a v-type vertex produces a $\{v\}_k$ vertex; decryption with a valid key on a $\{v\}_k$ vertex produces a v vertex.

4.4 Translation to Promela: UMLsec Semantics

We sketch how the analysis plug-in translates the UML model into the Promela notation, following the simplified UML semantics in [Jür04]. The resulting model consists of a network of communicating objects whose structure is based on the deployment diagram. Each object has an input queue and an output queue for exchanging messages with other parts of the system. To give each object a separate thread of execution within the model, we create a Promela *proctype* definition for every UML class, and instantiate it for every corresponding object in the deployment diagram. Each object in the resulting code receives a unique ID. From the class diagram, the tool collects information about the attributes of the object and its associations with other classes; each association is resolved to an object ID based on the deployment diagram. The behavior of each class is encoded in a loop following the UML *run-to-completion* semantics by repeatedly executing the following two steps:

- If not in the end state, all actions that are enabled are executed in a loop, without consuming external events. If more than one action is enabled, the action that is executed is selected non-deterministically.
- A single event is read from the input communication channel, and the corresponding action is executed. The execution of the object is blocked if the channel is empty. The events which do not trigger any actions in the current object state are lost.

The tool uses a simplified UML semantics. In particular, composite and history states are not allowed, events cannot be deferred, and only asynchronous communication is supported at present. Some of the other UML constructs can be reduced as usual to the subset the tool supports (see [Jür04]).

Adversary In consideration of memory complexity, we have separated the adversary knowledge into two different classes:

- A message from the communication channel can be captured and stored without understanding it. It can be analyzed by the adversary later when he receives the necessary key, or replayed to any protocol party without changes. This complex knowledge is stored in variables of the type MSG.
- The simple knowledge of the adversary is formed by a set of boolean variables, one for each atomic value not known to the adversary at the system initialisation, named known_"atomic value name". They are initialized to false and set by the adversary procedure to true when he can derive the corresponding plaintext value from the data learnt by eavesdropping.

The knowledge of both categories can be used by the adversary to compose new messages. In the Promela code we implement this by starting the message composition either from one of the known atomic values, or from a stored, composed message. Then the adversary can apply any cryptographic operation up to $n = TypeGraphDiameter$ times.

By default all public keys in the system and all object identifiers are known to the adversary at the system initialization. The model developer can extend the adversary model's initial knowledge: it contains initial values of all class attributes marked with the stereotype «public».

The size of the data type graph and the adversary memory capacity can be determined by the user of the tool. We plan to investigate formal results on bounds that yield a sound analysis, based on results such as [Sto02]. The data type graph size is defined by most complex operations performed by legitimate parties. The rationale is that for an adversary it makes sense to produce expressions only as complex as actually processed by the protocol parties.

The adversary behavior is modelled by a separate Promela proctype definition and instantiated with a separate execution thread. The adversary procedure accepts as parameters input and output channels of all other

```
loop { do this { read message from Bob
                 send it to Alice
                 analyze and save the message }
       or this { generate a message from knowledge
                 send it to Bob }
       or this { read message from Alice
                 send it to Bob
                 analyze and save the message }
       or this { generate a message from knowledge
                 send it to Alice } }
```

Fig. 19 Example adversary

objects in the system. It executes an infinite loop, nondeterministically selecting and executing one of the possible actions on one of the communication channels. In our example, the adversary capabilities are limited to the subset $\{read, insert\}$, which results into the loop given in Fig. 19 in pseudocode. Note that in this case, the adversary cannot delete messages, but always forwards them to the intended receiver, according to the missing *delete* capability.

The Promela code of the analyse message functionality for our example is shown in Fig. 20. The procedure TweakMessage applies one of the possible data transformation functions to the message (or leaves the message unchanged). The procedure is repeated *TypeGraphDiameter* times (four in this case). The adversary then learns the resulting value.

Generation of a new message by the adversary is pictured in Fig. 21 for our example. Firstly, the message can be started either by using one of the stored messages, or an intially known value. The message may be modified up to *TypeGraphDiameter* times (again four in this case). Finally the message is sent over the specified channel (message header including sender, recipient and *MethodId* are already sent before this code fragment is executed).

Encoding the security requirement We use the Promela *never claim* construct to formalize the security requirement on the model. In particular, we capture the secrecy requirement by formulating the condition that the initial value of a class attribute that is marked with the stereotype «secrecy» is never recovered by the adversary in plain text. Technically, a Promela model defines an asynchronous interleaving product of concurrent behaviours of all its processes. The never claim is then used to specify which kinds of traces should not be produced when executing this product. The Spin model-checker verifies this by monitoring the statement in the never claim when calculating the possible system executions.

```
TweakMessage(message);
TweakMessage(message);
TweakMessage(message);
TweakMessage(message);
if
    :: message.messageType == MT_v ->
       LearnValue(message.param1);
    :: else -> ;
fi;
```

Fig. 20 Adversary code for analyzing a message

```
if
:: rv_tmpMessage[0].messageType != MT_GARBAGE ->
   rv_lastReadMessage.messageType=rv_tmpMessage[0].messageType;
   rv_lastReadMessage.param1 = rv_tmpMessage[0].param1;
   rv_lastReadMessage.param2 = rv_tmpMessage[0].param2;
   rv_lastReadMessage.param3 = rv_tmpMessage[0].param3;
   rv_lastReadMessage.param4 = rv_tmpMessage[0].param4;
   rv_lastReadMessage.param5 = rv_tmpMessage[0].param5;
:: rv_lastReadMessage.messageType = MT_v;
   GetKnownValue(rv_lastReadMessage.param1);
fi;

TweakMessage(rv_lastReadMessage);
TweakMessage(rv_lastReadMessage);
TweakMessage(rv_lastReadMessage);
TweakMessage(rv_lastReadMessage);

SendMessage(cho, rv_lastReadMessage);
```

Fig. 21 Adversary code for generating a message

The security requirement from the UML model, expressed in our example by the stereotype «secrecy» on the variable m of the *Responder* class, is translated into the *never claim* construct in the Promela code, saying that the adversary should never get to know the secret values. It defines a process which runs in parallel with the rest of the system and monitors this property. The *never claim* for our example is presented in Fig. 22. It defines a loop which does not terminate as long as the variable known_DV_Bob_nonce, corresponding to the initial value of the attribute m of the class *Bob*, has the value $false$. This reflects the requirement that the value should not be known to the adversary. Any possible system execution scenario that leads to termination of the never claim construct is treated by Spin as a failed verification and reported to the developer.

Collecting objects into system For each object in the deployment diagram a separate Promela process is instantiated using the run keyword. A separate process is started for the adversary. All processes will run asynchronously and interact by message passing over communication channels which connect them according to the deployment diagram specification. The Spin model-

```
never {
T0_init:
if
    :: known_DV_Bob_nonce == true ->
       goto accept_all;
    :: (1) -> goto T0_init;
fi;
accept_all:
skip
}
```

Fig. 22 Never claim in Promela

```
Depth=      80  States= 122065 Transitions=   165114 Memory= 22.422
pan: claim violated! (at depth 82)
pan: wrote pan_in.trail (Spin Version 4.1.1 -- 2 January 2004)
```

Fig. 24 Fragment of the Spin output

```
init {
    chan Server_Bob_Input = [1] of {byte};
    chan Server_Bob_Output = [1] of {byte};
    chan Client_Alice_Input = [1] of {byte};
    chan Client_Alice_Output = [1] of {byte};

    run ClassServer(DV_Bob_id,
            Server_Bob_Input, Server_Bob_Output,
            DV_Alice_id);
    run ClassClient(DV_Alice_id,
            Client_Alice_Input, Client_Alice_Output,
            DV_Bob_id);

    run Intruder(DV_Intruder_id,
            Server_Bob_Input, Server_Bob_Output,
            Client_Alice_Input, Client_Alice_Output);
}
```

Fig. 23 The main procedure in Promela

checker is used to verify whether the system meets the desired properties.

The entry point procedure of the Promela program for the running example is presented in Fig. 23. At the beginning of the procedure, four communication channels are defined, one input and one output for every object in the deployment diagram. Further objects are instantiated using the Promela keyword run. Each object is a process with its own execution thread which runs code in the proctype declaration of the correspondent class. Parameters passed to each procedure define the unique ob ject identifier, the communication channels which the ob ject uses and identifiers of other objects, connected to it by associations, as described before. A separate process is started for the adversary.

Verification results For space restrictions we cannot include the full Promela code for our running example. It can however be downloaded from [Jür06]. Spin completes verification of this simple example within a minute after detecting a flaw in the protocol. Part of the Spin output is shown in Fig. 24, the complete verification result also can be found at [Jür06]. In the attack found in this simple example, the adversary sends his own key to Bob, pretending to be a legitime protocol participant, and receives back the secret value, encrypted under the key. The adversary can easily decrypt the message and obtain the plain text secret value. If we restrict the adversary from writing messages to the communication link, another attack is still found: the adversary records the key passed from Alice to Bob in the first protocol step, and uses it to decipher the message in the second.

As part of the verification process, Spin produces a trail file, which records the sequence of actions of the potential attack. This information can be used by the system developer to improve the protocol.

Note that as usual in bounded model checking, the user needs to specify the search depth. This means that verification results are only valid up to this restriction. There are however results which establish bounds on the execution of the system that needs to be verified in order to be complete, see for example [Sto02].

Evaluation From our experiments, we know that the state space of the produced model quickly increases with increasing size of the model. However, there are several approaches to deal with this.

- The complexity of the model can be reduced by abstracting from further details in the system description.
- The largest factor that contributes to the complexity of the Spin model is the highly non-deterministic adversary behavior. We aim to investigate ways to reduce the adversary state space by defining heuristics which exclude system traces that obviously cannot result in a successful attack, again based on results such as [Sto02].

Note that this means in particular, that our approach does not currently aim to be completely automatic (because of the manual abstraction that is currently still necessary), but to provide as much automation as possible given the inherent complexity of the verification problem and the limitations of current computing technology to solve it.

In particular, the aim of the current work is not to push the current limits in verification technology, but to demonstrate how they can be put to use in the context of a CASE tool environment.

In subsequent work [Jür05, JF06] we have been able to improve on the above verification run-times in orders of magnitude, by using a more abstract approach to verification using automated theorem provers for first-order logic, instead of using a very concrete adversary execution model and model-checking verification. This has been implemented as another plugin of the verification framework that is presented in this paper.

However, the current work still has the advantage over the newer approach that it enables the user to actually produce a concrete attack trace in case the system under consideration is insecure. This is currently not possible with the automated theorem proving approach cited above.

The two tools are therefore complementing each other in a beneficial way as follows:

– Firstly, one can use the automated theorem proving plugin to determine whether the system is indeed secure (against the given adversary model) or not. This is done very efficiently for industrial size systems (see e.g. [BJN07]).
– If the automated theorem plugin determines that there may be an attack, this still leaves the possibility of a false positive since the automated theorem prover approximates the verification problem on the "safe" side for efficiency reasons. Therefore, one can use the model-checker plugin presented in the current paper to produce the actual attack sequence (and determine whether it really is a realistic attack). Although the model-checker plugin is less performant than the automated theorem prover plugin, this is not a problem, since the state space now does not have to be fully traversed (which would be the case if the goal was to determine the *absence* of an attack), but only until the attack sequence is found (which we already know exists after using the automated theorem proving plugin).

5 Related Work

There are several existing tools for automatic verification of UML models described in the literature. The HUGO Project [SKM01] checks the behavior described by a UML Collaboration diagram against a transitional system comprising several communicating objects. The functionality of each object is specified by a UML Statechart diagram. The vUML Tool [LP99] analyzes the behavior of a set of interacting objects, defined in a similar way. The tool can verify various properties of the system, including deadlock freedom and liveliness, and find problems like entering a forbidden state or sending a message to a terminated object. Both tools do not have any special features for describing the security features of the system being modeled. [CCMS02] describes automated structural and behavioral analyzes for UML diagrams built on a previously developed formalization framework. The paper explains how consistency checks, simulation, and model-checking (specifically, using Promela) can be used together for behavioral analysis of UML diagrams. [EKHL03] presents automated verification of UML models using the model-checker FDR. [SV03] presents a tool for model-checking dynamic consistency properties in visual models such as UML using the model-checker SPIN. [OGO04] presents a tool for model-checking behavioral UML models based on a semantics in communicating extended timed automata and using model-checking and simulation tools. The combination of different aspects and diagrams is supported as well as a particular semantic profile for communication, concurrency, and timing. UML models are imported via an XMI repository. Feedback is given to the user in terms of the original UML model. All of this work is relevant to ours in that it gives alternative approaches to UML model analysis tools. However, although there is an increasing amount of research on advanced tool support for UML, it seems that little work has been done to provide advanced tool support, such as model-checkers or automated theorem provers, for verifying particular properties included as stereotypes in application-specific UML extensions. In particular, to our knowledge, none of the existing bindings of UML to model-checkers can be easily extended to analyze UMLsec models. The first reason is the support for security constructs. The second issue is the translation of complex data types, which is necessary for supporting the cryptography extension.

With respect to tool frameworks in general, related work includes [MNS05], which offers a tool framework which goes beyond our framework presented here in that it is open to non-UML-based tools. This work is relevant to ours in so far as it would be interesting to see whether it can be used to transfer our UML-based tools presented in this paper to non-UML-based environments.

Compared to research done using formal methods, less work has been done more generally using software engineering techniques for computer security. For an overview of the topic see [DS00].

There is an increasing interest in using UML for the development of security-critical systems. For example, [FH97] defines role-based access control rights from object-oriented use cases. [HdBLS02] uses UML for the risk assessment of an e-commerce system within the CORAS framework for model-based risk assessment. [FMMMP02] uses UML for the design of secure databases. It proposes an extension of the use case and class models of UML using their standard extension mechanisms designing secure databases. [KPP02] demonstrate how to deal with access control policies in UML. The specification of access control policies is integrated into UML. [LBD02] show 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 authorization constraints. [KRFL04] describes an approach for specificating role-based access control policies in UML design models. It allows developers to specify patterns of violations against the policies. [BPA06] presents an approach for the specification of user rights using UML. The approach is based on a first-order logic with a built-in notion of objects and classes with an algebraic semantics and can be realized in OCL. These approaches are relevant to ours in that they constitute alternative approaches to treating security requirements using UML, although tool-support similar to the one presented here does not seem to exist for these approaches yet. It would thus be interesting to see to what extend the tools from the UMLsec tool suite could be used for these other extensions, with suitable adaptations.

In other approaches to automated software engineering for security, [KAH99] uses the Software Cost Reduction method (SCR) to analyze a cryptographic system called for various security properties.

There is too much work on verifying cryptographic protocols to give a complete overview. Overviews of applications of formal methods to security protocols can be found for example in [Aba00,Mea00]. Another approaches to using first-order logic ATPs for cryptoprotocol analysis include the following: [Sch97] formalizes the well-known BAN logic in first-order logic and uses the atp SETHEO to proof statements in the BAN logic. It is different from our approach which is based on the knowledge of the adversary, instead of the beliefs of the protocol participants. [Coh03] uses first-order invariants to verify cryptographic protocols against safety properties. The approach is supported by the atp TAPS. Compared to our approach, the method does not generate counterexamples (that is, attacks) in case a protocol is found to be insecure.

In this paper, we deal with security on the design level, since there are many security problems apparent at that level in practical systems. However, we find it equally important to try to consider security already during the requirements elicitation phase. To that aim, [CILN02] formulates a vision for the requirements engineering community towards providing a "bridge between the well-ordered world of the software project informed by conventional requirements and the unexpected world of anti-requirements associated with the malicious user". [GMM03] proposes an extension of the i*/Tropos requirements engineering framework to deal with security requirements. [SO05] presents an approach to eliciting security requirements using use cases which extends traditional use cases to also cover misuse. [MJF06] uses a combination of UMLsec and Tropos to get a transition from the security requirements to the design phase. This work is relevant to the one since the eventual goal is to provide an integrated security engineering approach.

6 Conclusion

We presented work to support model-based development using UML by providing tool-support for the analysis of UML models against difficult system requirements. We described a UML verification framework supporting the construction of automated requirements analysis tools for UML diagrams which is connected to industrial CASE tools using XMI. As an example for its usage, we presented verification routines for verifying UMLsec models. Their aim was firstly to contribute towards usage of UMLsec in practice. Secondly, the verification framework should allow advanced users of the UMLsec approach to themselves implement verification routines for the constraints of self-defined stereotypes. We focussed on an analysis plug-in that utilizes the model-checker Spin to verify systems which may use cryptographic algorithms.

The tools we presented are used in industrial projects involving for example a car manufacturer, a bank, and a telecommunications company. Several security design weaknesses could be demonstrated which have lead to changes in the designs of the systems that are being developed.

The verification framework has proven to be sufficiently flexible to support analysis plug-ins for a variety of checks.

In particular, the model-checking plugin presented in this paper has proven to be very useful in producing attack sequences for insecure systems specifications. It can be used in particular in conjunction with more abstract verification approaches such that those making use of automated theorem proving [Jür05,JF06,BJN07], as explained above. The tools presented here can be downloaded from [Jür06] as open-source.

Acknowledgements Fruitful collaborations with the members of the UMLsec group at TU Munich are gratefully acknowledged, as well as constructive comments by the anonymous referees which lead to significant improvements in the presentation of this paper.

References

[Aba00] M. Abadi. Security protocols and their properties. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000. 20th International Summer School, Marktoberdorf, Germany.

[AJ01] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software (4th International Symposium, TACS 2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 82–94. Springer-Verlag, 2001.

[BJN07] B. Best, J. Jürjens, and B. Nuseibeh. Model-based security engineering of distributed information systems using UMLsec. In *ICSE*. ACM, 2007.

[BPA06] R. Breu, G. Popp, and M. Alam. Model based development of access policies. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006. Contained in this issue.

[Cas03] Castor library. Available at http://castor.exolab.org, June 2003.

[CCMS02] L. Campbell, B. Cheng, W. McUmber, and K. Stirewalt. Automatically detecting and visualising errors in UML diagrams. *Requir. Eng.*, 7(4):264–287, 2002.

[CILN02] R. Crook, D.C. Ince, L. Lin, and B. Nuseibeh. Security requirements engineering: When anti-requirements hit the fan. In *RE*, pages 203–205. IEEE Computer Society, 2002.

[Coh03] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[DS00] P. Devanbu and S. Stubblebine. Software engineering for security: A roadmap. In *22nd International Conference on on Software Engineering (ICSE 2000): Future of Software Engineering Track*, pages 227–239. ACM, 2000.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[EKHL03] G. Engels, J. Küster, R. Heckel, and M. Lohmann. Model-based verification and validation of properties. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[FH97] E. B. Fernandez and J. C. Hawkins. Determining role rights from use cases. In *Workshop on Role-Based Access Control*, pages 121–125. ACM, 1997.

[FMMMP02] E. Fernández-Medina, A. Martínez, C. Medina, and M. Piattini. UML for the design of secure databases: Integrating security levels, user roles, and constraints in the database design process. In Jürjens et al. [JCF$^+$02], pages 93–106.

[Gen03] Gentleware. http://www.gentleware.com, 2003.

[GMM03] P. Giorgini, F. Massacci, and J. Mylopoulos. Requirement engineering meets security: A case study on modelling secure electronic transactions by VISA and Mastercard. In I.-Y. Song, S. W. Liddle, T. W. Ling, and P. Scheuermann, editors, *22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 263–276. Springer-Verlag, 2003.

[Gur95] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[HdBLS02] S. H. Houmb, F. den Braber, M. S. Lund, and K. Stølen. Towards a UML profile for model-based risk assessment. In Jürjens et al. [JCF$^+$02], pages 79–92.

[HJ03] S. Höhn and J. Jürjens. Automated checking of SAP security permissions. In *6th Working Conference on Integrity and Internal Control in Information Systems (IICIS)*. International Federation for Information Processing (IFIP), Kluwer Academic Publishers, 2003.

[HMR$^+$98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

[Hol03] G. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[JCF$^+$02] J. Jürjens, V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors. *Critical Systems Development with UML (CSDUML 2002)*, TU München Technical Report TUM-I0208, 2002. UML 2002 satellite workshop proceedings.

[JF06] J. Jürjens and J. Fox. Tools for model-based security engineering. In *28th International Conference on Software Engineering (ICSE 2006)*. ACM, 2006.

[JHC02] J.-M. Jézéquel, H. Hußmann, and S. Cook, editors. *5th International Conference on the Unified Modeling Language (UML 2002)*, volume 2460 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[JS04] J. Jürjens and P. Shabalin. Automated verification of UMLsec models for security requirements. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2004 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425. Springer-Verlag, 2004.

[JS05] J. Jürjens and P. Shabalin. Tools for secure systems development with UML. In *FASE 2005*, Lecture Notes in Computer Science, Edinburgh, 2-10 April 2005. Springer-Verlag.

[Jür06] J. Jürjens. UMLsec webpage, 2002-06. Accessible at http://www.umlsec.org.

[Jür02] J. Jürjens. UMLsec: Extending UML for secure systems development. In Jézéquel et al. [JHC02], pages 412–425.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[KAH99] J. Kirby, M. Archer, and C. Heitmeyer. Applying formal methods to an information security device: An experience report. In *4th IEEE International Symposium on High Assurance Systems Engineering (HASE 1999)*, pages 81–88. IEEE Computer Society, 1999.

[KPP02] M. Koch and F. Parisi-Presicce. Access control policy specification in UML. In Jürjens et al. [JCF$^+$02], pages 63–78.

[KRFL04] D.-K. Kim, I. Ray, R. B. France, and Na Li. Modeling role-based access control using parameterized UML models. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 180–193. Springer-Verlag, 2004.

[LBD02] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In Jézéquel et al. [JHC02], pages 426–441.

[LP99] J. Lilius and I. Porres. Formalising UML state machines for model checking. In R. B. France and B. Rumpe, editors, *The Unified Modeling Language (UML 1999)*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer-Verlag, 1999.

[Mea00] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 237–250. IEEE Computer Society, 2000.

[MJF06] H. Mouratidis, J. Jürjens, and J. Fox. Towards a comprehensive framework for secure systems development. In *18th International Conference on Advanced Information Systems Engineering (CAiSE 2006)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[MNS05] T. Margaria, R. Nagel, and B. Steffen. jETI: A tool for remote tool integration. In N. Halbwachs and L.D. Zuck, editors, *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 557–562. Springer-Verlag, 2005.

[MOF02] Object Management Group. *MOF 1.4 Specification*, April 2002. Available at http://www.omg.org/technology/documents/formal/mof.htm.

[Net03] Netbeans project. Open source. Available from http://mdr.netbeans.org, 2003.

[NSU03] Novosoft NSUML project. Available from http://nsuml.sourceforge.net/, 2003.

[OGO04] Iu. Ober, S. Graf, and Il. Ober. Validation of UML models via a mapping to communicating extended timed automata. In *SPIN 2004*, pages 127–145, 2004.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Sch97] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In W. McCune, editor, *14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Computer Science*, pages 87–100. Springer-Verlag, 1997.

[SKM01] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In S. D. Stoller and W. Visser, editors, *Workshop on Software Model Checking*, volume 55(3) of *Electronical Notes in Theoretical Computer Science*. Elsevier, 2001. Satellite event of the 13th International Conference on Computer-Aided Verification (CAV 2001)

[SMV] The SMV system. Available from http://www-2.cs.cmu.edu/˜modelcheck/smv.html.

[SO05] G. Sindre and A.L. Opdahl. Eliciting security requirements with misuse cases. *Requir. Eng.*, 10(1):34–44, 2005.

[Sto02] Scott D. Stoller. A bound on attacks on authentication protocols. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *IFIP TCS*, volume 223 of *IFIP Conference Proceedings*, pages 588–600. Kluwer, 2002.

[SV03] Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In P. Stevens, editor, *The Unified Modeling Language (UML 2003)*, volume 2863 of *Lecture Notes in Computer Science*, pages 92–95. Springer-Verlag, 2003. 6th International Conference.

[UML03] Object Management Group. *OMG Unified Modeling Language Specification v1.5*, March 2003. Version 1.5. OMG Document formal/03-03-01.

[XMI02] Object Management Group. *OMG XML Metadata Interchange (XMI) Specification*, January 2002.