Automated Verification of UMLsec Models for Security Requirements

Jan Jürjens* and Pasha Shabalin

Software & Systems Engineering, TU Munich, Germany
http://www4.in.tum.de/~juerjens, http://www4.in.tum.de/~shabalin

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 for usage of this framework, we present verification routines for verifying models of the security extension UMLsec of UML. These plug-ins should not only contribute towards usage of UMLsec in practice by offering automated analysis routines connected to popular CASE tools. The verification framework should also 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. In particular, we focus on an analysis plug-in that utilises the model-checker Spin to verify security properties of UMLsec models which make use of cryptography (such as cryptographic protocols).

1 Introduction

Still only about 4% of software systems in practice are built using modeling techniques of some sort (most of them using UML). There needs to be a convincing added-value to the usage of model-based development techniques before it 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 [Obj02] and allows convenient access to this data and to the human user.

* Supported by the Verisoft Project of the German Ministry for Research (BMBF).

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 analysed 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 automatic to prevent additional running costs in using it.
- It should efficient enough to be effectively and conveniently usable and an ongoing basis.
- It should be possible to use the approach with just a modest training effort.

As an example for such requirements, we focus on security aspects. We demonstrate how to instantiate this framework with analysis plug-ins at the hand of examples for verification routines for constraints associated with the stereotypes of the UML security extension UMLsec [Jür02, Jür04]. In particular, we focus on a plug-in that utilises 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. One 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 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 for the contained security requirements. In Sect. 3, we present one of these plug-ins, a binding with the Spin modelchecker for checking data security requirements, for example of cryptographic protocols. At the hand of a running example, the translation from UMLsec models to Promela code and its execution in Spin is explained. We close with comparisons to related work, a discussion of our work and an outlook on further developments. For background on data security and UMLsec, we refer to [Jür02, Jür04].

2 The UML Verification Framework

We present a framework supporting the construction of automated requirements analysis tools for UML diagrams. The framework is connected to industrial CASE tools using data integration with XMI [Obj02] and allows convenient

access to this data and to the human user. The framework provides three input and output interfaces for the analysis plug-ins: a textual command-line interface, a graphical user interface, and a web-interface. Inputs can be UML diagrams in the form of XMI files, as well as textual parameters. (Alternatively, the diagrams can be input in the .zuml format of the Poseidon tool [Gen03], which also includes graphical information.) As output one can again have UML diagrams as XMI (or .zuml) files and text messages. The tool can access the information in the UML models on the conceptual level of UML model elements through Java Metadata Interface (JMI) methods [Dir02]. The plug-ins can have an internal state which is preserved between different executions of its commands. Any analysis tool written in Java and complying with the input and output interface of the framework can be plugged into the framework. To avoid having to parse XMI files, we use XMI data-binding offered by the Meta Data Repository (MDR) [Mat03], a XMI-specific data-binding library in Java Netbeans. Since MDR allows one to make use of the DTDs for XMI that are officially released, compatibility with the standard is ensured (and MDR is also used in Poseidon). The architecture and basic functionality of the UML verification framework are illustrated in Fig. 1. The framework can be offered as a web application (where the UML models are uploaded to the framework over the web). Additionally, a locally installable version is available. Exemplarily, the figure includes two of the UMLsec analysis plug-ins (a checker for static security properties, and a checker for dynamic properties using a model checker).

The usage of the framework as illustrated in Fig. 1 proceeds as follows. The developer creates a model and stores it in the UML 1.4/XMI 1.2 file format.[1] The file is imported by the UML verification framework into the internal MDR repository. Each plug-in accesses the model through the JMI interfaces generated by the MDR library, they may receive additional textual input, and they may return both a UML model and textual output. The two exemplary analysis plug-ins proceed as follows: The static checker parses the model, verifies its static features, and delivers the results to the error analyzer. The dynamic checker translates the relevant fragments of the UML model into the model-checker input language. The model-checker is spawned by the UML framework as an external process; its results (a counter-example in case a problem was found) are delivered back to the error analyzer. The error analyzer uses the information received from the static checker and dynamic checker to produce a text report for the developer describing the problems found, and a modified UML model, where the errors found are visualized. On any Java-enabled platform, the framework can run in one of three modi:

- as a console application, either interactive or in batch mode
- as a Java Servlet, exposing its functionality over the Internet
- as a GUI application with higher interactivity and presentation capabilities

To achieve a media-independent operation of the tools, input and output are handled by the framework. Each tool command defines a set of required input parameters (currently supported parameter types are Integer, Double, String

[1] This will be updated to UML 2.0 once the corresponding DTD is released.

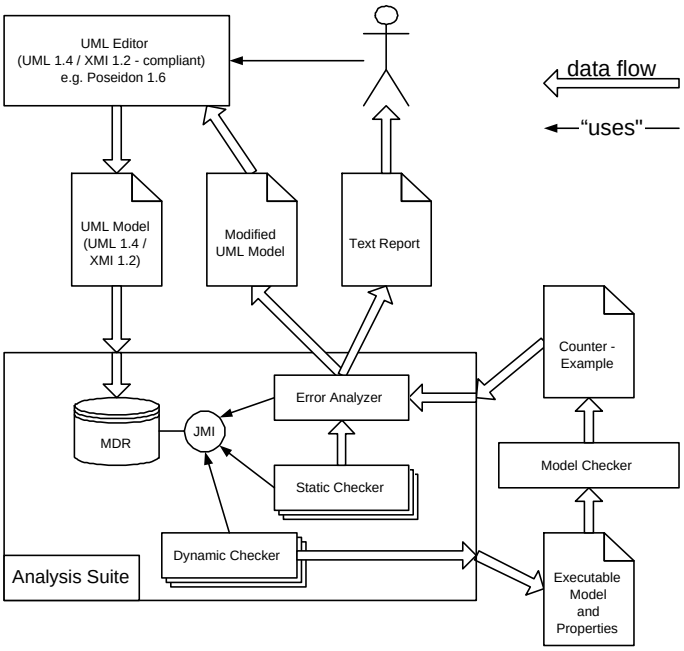

Fig. 1. UML verification framework: usage

and File; others can be easily added). On behalf of the tool, the framework collects the parameters from the user and returns the output to the user using the current input/output media (console, web, or GUI). To enable this, each tool that is integrated into the UML verification framework must implement a common interface (IToolBase), plus three media-dependent interfaces (IToolConsole, IToolWeb and IToolGui). For simplification, the framework provides default implementations for the IToolWeb and IToolGui interfaces. These default wrappers use the interface implemented by the tool IToolConsole and render the text output in HTML or in a text window format respectively. Each tool exposes a set of commands which can be executed through the functions of the corresponding interface (GetConsoleCommands, GetWebCommands and GetGuiCommands). Thus the tool can provide different functionality on different media, adopting to its specifics. The framework uses the IToolBase interface to retrieve general information about the tool, and one of the three media-specific tool interfaces to call a command provided by the tool and receive the output. This output is rendered by the framework on the current media. The tool can execute several commands subsequently; the internal state of the MDR repository and all tools is preserved between command calls. The set of available commands for each tool may vary depending on the execution history and current state. This allows to use the UML framework for complex and interactive operations on the UML model. The source code of the verification framework and the plug-ins is down-

loadable as part of the UMLsec tool from [UML], where they are also offered as a web-interface, including a small user tutorial.

We give a short overview over analysis tool plugins for the UML verification framework which support verification of UMLsec models for the security requirements contained as stereotypes and their constraints (see [Jür02]). One of them will be explained in more detail in the later sections (a binding with the Spin modelchecker for checking the constraints of the « data security » stereotype, for example to verify cryptographic protocols). When given a UMLsec model, the analysis tools automatically produce a semantic model and include a formalization of the security requirements or primitives contained as stereotypes and their constraints. These can be applied by a developer without specialized training in security or formal methods by simply including them in the UML model. The constraints associated with the stereotypes are translated to the formal model, protecting from errors that manual formalization is prone to (see [Jür02] for details about the formal semantics of a simplified fragment of UML we use). Since security requirements are usually defined relative to an adversary, to analyze whether the UML specification fulfills a security requirement, the tools automatically include the adversary model arising from the physical view contained in the UML specification as a deployment diagram. The UMLsec verification plug-ins fall into several different 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 (such as « fair exchange » and « guarded access »).

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 a tool-binding to the model-checker Spin to verify the « data security » constraints in this paper.

External application binding There are also plug-ins analyzing security permissions from configurations for SAP R/3 business applications with respect to security rules and business processes formulated in UML [HJ03].

Note that the other stereotypes from [Jür02] not mentioned above do not entail any verification obligation, but just provide some security-relevant information which is used when defining constraints associated with other stereotypes: For example, « Internet » and other stereotypes include information about the physical security level used for example by the « secure links » constraint.

3 Model-checking UMLsec models

As an example for the verification routines implemented in the UMLsec tool, we present a tool-binding with the model-checker Spin [Hol03] for verifying cryptographic protocols following the « data security » requirement from [Jür02]. « 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 » stereotype are supposed to remain secret. 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 is shortly sketched at the end of this section; for details we refer to [Jür02]. In this section, we present work on how to provide tool-support for automatically verifying a UML specification for the « data security » constraint.

Spin supports automatic verification of finite-state reactive systems given in form of a state-transition system against properties expressed in Linear Time Logic (LTL). To check the constraint associated with « data security » attached to a subsystem, we collect information from the following diagrams contained in the subsystem:

Class diagrams In class diagrams, attributes and methods may be tagged for example with {secrecy} to specify that the relevant data should remain secret from an adversary.

Statecharts The behaviour of the instances of each class (for example, with a cryptographic protocol) is defined in a statechart diagram.

Deployment diagrams Deployment diagram are used to specify the physical security of communication links between objects within the system that are distributed, for example over the Internet.

Thus, from the information in the statecharts we construct a formal model of the system, which is augmented 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.

Following [Jür02], we extend the UML notation with cryptography primitives, which can be used in *Guards* and *Effects* in UMLsec Statecharts and to define initial values for variables in the Class diagrams. The BNF representation of the cryptographic expressions is given in Fig. 2. Here, the object identifiers `identifier` are assumed to be given. The functions `SenderOf`, `PublicKeyOf`, `SecretKeyOf`, `SymmetricKeyOf`, and `NonceOf` return the corresponding attributes of the object. Note that the function `NonceOf` was introduced based on the assumptions that for protocols with symmetric session keys, at each iteration of the protocol a new object with a fresh session key is generated; it would alternatively be easily possible to modify the definition to allow

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

Fig. 2. UMLsec Cryptography Language

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). Within a class, the keyword `this` references the class itself, and the other classes are referenced by the corresponding association end identifiers from the class diagram. Expressions can finally 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)

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].

Example We introduce a UML 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

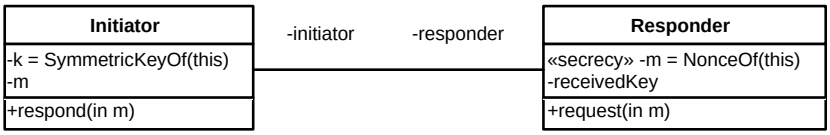

Fig. 3. Example Class diagram

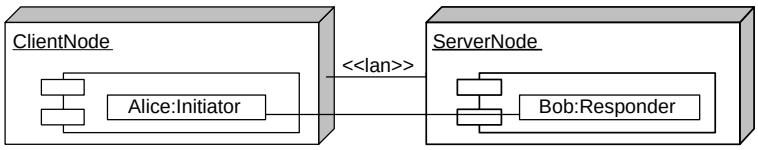

Fig. 4. Example Deployment diagram

secret information from Bob (of class Responder). Alice sends to Bob her key, and Bob returns the secret value encrypted under the key:

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

The UML model of the example is presented in Figures 3 through 6. Fig. 3 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. 4 contains a deployment diagram describing the physical layer underlying the protocol. The communication link is marked with the «lan» stereotype, 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. 5 contains a statechart specifying the behavior of the Initiator in the protocol sketched above, and Fig. 6 a statechart for the Responder.

Translation to Model-Checker We explain some key points in the automatic translation from UMLsec models to Promela code and its execution in Spin at the hand of our running example. We use the Spin model checker since we found it suitable for verification of distributed communicating systems. In particular, Spin's *on-the-fly* model checking allowing to partially verify a model without building the full state space seems suitable for verifying security requirements with their highly non-deterministic adversary models.

Parameters and Data Types In a UML model, developers can use a broad range of predefined data types, and can also define their own data types. In

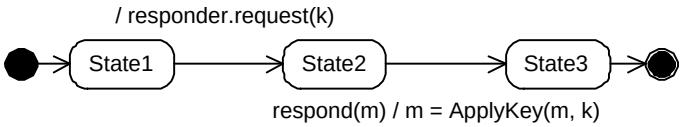

Fig. 5. Example Initiator Statechart

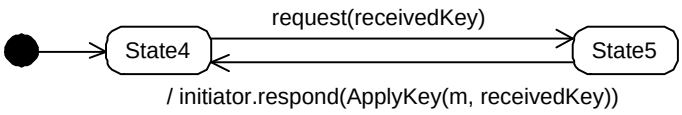

Fig. 6. Example Responder Statechart

contrast, model checker notations usually support only a very limited set of data types (in the case of Promela: Boolean, Integer and enumeration [Hol03]). For a given UML model, we thus need to define a mapping of 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 sufficient 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 constructs received from applying operations (the *data transformation functions*) to atomic values, such as $\{v\}_k$ (x encrypted under the key k).

We consider a model with a base data type (say, Integer) supposed to include the atomic values to discuss three possibilities of representing and processing complex values:

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 the all possible complex data values, but it may be possible to be extended in this way. However, taking the discussed drawbacks into account, we consider other possibilities.

Fixed Types The UML explicitly defines the data types of all variables, parameters and return values in the model. For every data type, it is then possible to enumerate all its values and define data transformation functions hardcoded in the translated Model Checker code according to the data type they process. We are not aware of any existing tools implementing this approach. Comparing to the first solution, it will result in a better structured code, and the translation logic will be cleaner and easier to understand. However, we then have to limit

the developer to use only those data types known to our translator. Alternatively we can request the developer to provide mapping rules describing how the data types in the UML model relate to the translator data types. Both solutions mean significant additional effort which might prevent developers from using the technology. To handle these problems, we introduce dynamic handling of data types, where the message itself carries information about its type.

Dynamic Types The complex data type is defined during the translation process and holds any complex 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 applying all expressions which are met in the model (namely, initial values, transition effects, and transition guards), the tool creates new vertexes in the graph as necessary. The data transformation functions are presented by edges, and the complex data types incarnations are presented by nodes.

The data graph for our example is presented in Fig. 7. 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 complex data type, for each edge in the graph. The translation result for our example in Promela language is presented in Fig. 8. The MSG structure is then used in the translated code to represent complex data type. The *messageType* field defines which vertex the structure represents. The *param*1 field stores the message v, and param2 stores the key k only when the structure represents vertex $\{v\}_k$. 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.

UML Semantics We sketch how the analysis plug-in translates the UML model into the Promela notation, following the simplified UML semantics in [Jür02, Jür04]. The resulting model consists of a network of communicating objects, based on the deployment diagram. Each object has an input queue and an output queue for exchanging messages with other parts of the system. Each object has a separate thread of execution within the model. This is achieved by creating a Promela *proctype* definition for every UML class, and by instantiat-

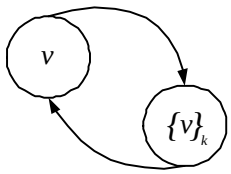

Fig. 7. Example Data Graph

```
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. 8. Translated Example - fragment

ing it for every corresponding object in the deployment diagram. Each object in the resulted code receives a unique ID. From the class diagram, the tool collects information about the attributes 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 enabled actions are executed in a loop, without consuming external events. If more than one action is enabled, the selection of the executed action is non-deterministic.
- 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. Some of the other UML constructs can be reduced as usual to the subset the tool supports. In particular, composite and history states are not allowed, events cannot be deferred, and only asynchronous communication is supported at present.

Adversary To apply formal verification methods for verifying security properties of an open, distributed system, it is necessary to model all possible interactions between the system and the outside world. This includes behavior of an adversary trying to break or compromise the system. The adversary model is defined through certain basic capabilities, depending on the physical properties of the system, and on the strength of the adversary that is considered. 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 $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$ (see [Jür02]). Each of the threats is implemented by a possible adversary action in the system model. The behavior of the adversary is modeled by a separate Promela *proctype* process definition and instantiated with a separate execution thread.

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

Fig. 9. Never Claim in Promela

- **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.

In our example, the adversary capabilities are limited to the subset {*read*, *insert*}, which results into the loop given in Fig. 10 in pseudocode. Note that in this case, the adversary cannot drop the message which is received, but always forwards it to the intended receiver, according to the missing *delete* capability.

Security requirement 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. 9.

Verification results For space restrictions we cannot include the full Promela code for our example. It can however be downloaded from [UML]. 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. 11, the complete verification

```
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. 10. Example Adversary

```
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. 11. Fragment of the Spin output

result also can be found at [UML]. 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.

Related Work There are several tools for automated verification of UML models. 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 freeness and liveliness, and find problems like entering a forbidden state or sending a message to a terminated object. 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. Work on how to use XMI to provide tool support for UML includes [Ste03] (including an example using the Edinburgh Concurrency Workbench for analyzing UML models). [CBC+01] applies model checking to the formal verification of concurrent object-oriented systems, using the model checker SPIN. It uses an extension of the SPIN notation Promela with additional primitives needed to model concurrent object-oriented systems, such as class definition, object instantiation, message send, and synchronization. [EKHL03] presents automated verification of UML models using the model-checker FDR. [CRS04] presents a simulation framework for UML models based on a UML semantics using Abstract State Machines (ASMs).

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.

There is an increasing interest in using UML for the development of security-critical systems. For example, [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. [BP04] 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. [HW04] defines a Security assessment Object Language to specify security requirements in UML.

4 Conclusion and Future Work

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 utilises the model-checker Spin to verify systems which may use cryptographic algorithms.

The tools we presented are used in industrial projects involving 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 and expressive to support analysis plug-ins for a variety of checks. With respect to the tool-binding to the Spin model-checker presented here the Promela code that is generated consists to a large extent of general definitions and security analysis machinery which are always present. Thus the size and complexity of the code scale sufficiently well with increasing size of models. Although we focused on a core definition of the diagrams we used, the tools can be extended to support more complex features. The tools presented here can be downloaded from [UML] as open-source. A support mailinglist for users and developers is available.

In future work, our usage of model-checking can be further optimized in performance to deal with the state explosion problem inherent in model-checking; additionally, we work on the usage of automated theorem provers. We aim to to include feedback from the model checker back into the UML model. Also, we aim to further support extensibility of the approach by allowing advanced users to define stereotypes, tags, and first-order logic constraints which are then automatically translated for verification on a given UML model.

Acknowledgments Fruitful collaborations with about 25 students performing Masters and Bachelors theses and study projects on the construction and use of the UMLsec tools are very gratefully acknowledged; see [UML] for details. Special thanks go to Alexander Knapp for very helpful explanations on his work.

References

[BP04] R. Breu and G. Popp. Actor-centric modeling of user rights. In Wermelinger and Margaria [WM04], pages 165–179.

[CBC+01] Seung Mo Cho, Doo-Hwan Bae, Sung Deok Cha, Young Gon Kim, Byung Kyu Yoo, and Sang Taek Kim. Applying model checking to concurrent object-oriented software. In *ISADS 1999*, pages 380–383. IEEE Computer Society, 2001.

[CRS04] A. Cavarra, E. Riccobene, and P. Scandurra. A framework to simulate UML models: moving from a semi-formal to a formal environment. In *SAC*, pages 1519–1523. ACM, 2004.

[Dir02] R. Dirckze. Java Metadata Interface (JMI) API 1.0 Specification. Available at http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html, June 2002.

[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.

[Gen03] Gentleware. http://www.gentleware.com (February 2004), 2003.

[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)*, Lausanne, Switzerland, November 13–14, 2003. IFIP, Kluwer.

[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.

[HW04] S. Houmb and R. Winther. Security assessment object language (SOL). *Software and Systems Modeling*, 2004. Special issue on the CSDUML workshop, to be published.

[Jür02] J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.

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

[KRFL04] D. Kim, I. Ray, R. France, and Na Li. Modeling role-based access control using parameterized UML models. In Wermelinger and Margaria [WM04], pages 180 – 193.

[LP99] J. Lilius and I. Porres. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99*, volume 1723 of *LNCS*, pages 430–445. Springer, 1999.

[Mat03] M. Matula. Netbeans Metadata Repository (MDR). Available from http://mdr.netbeans.org, 2003.

[Obj02] Object Management Group. OMG XML Metadata Interchange (XMI) Specification. Available at http://www.omg.org/cgi-bin/doc?formal/2002-01-01 (February 2004), January 2002.

[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.

[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 of *ENTCS*. Elsevier, 2001.

[Ste03] P. Stevens. Small-scale XMI programming; a revolution in UML tool use? *Journal of Automated Software Engineering*, 10(1):7–21, 2003. Kluwer.

[UML] http://www4.in.tum.de/~umlsec.

[WM04] M. Wermelinger and T. Margaria, editors. *7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *LNCS*. Springer, 2004.