Tool-support for Model-Driven Development of Security-Critical Systems with UML

Jan Jürjens*
Pasha Shabalin
Software & Systems Engineering, TU Munich, Germany

1 Introduction

High quality development of critical systems (be it real-time, security-critical or dependable systems) is difficult. Many such systems are developed, deployed, and used that do not satisfy their criticality requirements, sometimes with spectacular failures. However, critical systems on whose correct functioning human life and substantial commercial assets depend on need to be developed especially carefully.

Unfortunately, in critical systems development, correctness is often in conflict with cost. Where thorough methods of system design pose high costs through personnel training and use, they are all too often avoided. The Unified Modeling Language (UML) [Obj03] offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context:

- As the de-facto standard in industrial modeling, a large number of developers is trained in UML.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.

Nevertheless, the UML semantics is given only in prose form [Obj03], leaving room for ambiguities. However, to provide advanced tool-support (for example, automated checking of behavioral properties of a UML specification) to assist application of our approach in industry, we need a mathematically precise semantics for UML.

There has been a substantial amount of work towards providing a formal semantics for UML diagrams (including [BGH$^+$98, FELR98, RACH00, Jür02a]; specifically, [BCR00] gives a Statechart semantics using Abstract State Machines which was a starting point for the current work). However, most work only provides models for single UML diagrams in isolation. When trying to give a

*http://www.jurjens.de/jan – tel +49 89 289 17338, fax +49 89 289 17307 precise mathematical meaning to whole UML specifications, one needs to be able to combine the formal models for the different kinds of diagrams. In this paper, we provide a formal framework to support this using UML Machines.

Our approach is based on *Abstract State Machines* (ASMs) [BS03] where states are represented by algebras. We use ASMs to present our semantics because this notation, essentially a more formal pseudo-code, seems to be relatively accessible. For a given proof tool (for example the model-checker Spin, Prolog, or the automatic theorem prover Setheo), we translate the semantics into the relevant input notation (such as Promela, Horn formulas, or the TPTP notation, resp.). We feel that this approach using an intermediate representation in ASMs may be more flexible and universally usable than directly using a notation closer to a given input notation.

For our purpose, we use an extension of ASMs with UML-type communication mechanisms called UML Machines, inspired by the *Algebraic State Machines* from [BW00]. Also, we define the concept of UML Machine System (UMS) that allows one to build up specifications in a modular way (corresponding to the use of UML subsystems). We use this to define a semantics for a simplified fragment of UML supporting the combined use of different kinds of UML diagrams including actions, activities, and message-passing between different diagrams, and which allows one to easily include different adversary and failure models to analyze specifications for criticality requirements.

Furthermore, we present work by the UMLsec group at TU München aimed at the development of automated tools for analyzing UML models for criticality requirements, to facilitate technology transfer to industry.

The work presented here builds on previous work including [Jür02b] but extends to diagram types not treated in [Jür02b] (such as sequence diagrams) and to the development of tool-supported for automated verification.

Outline After giving some backgroun in Sect. 1.1, we recall the necessary definitions and introduce the notion of UML Machine used for our semantics in Sect. 2. We show how several UML Machines can be composed into a UML Machine System (UMS). In Sect. 3, we sketch how we use our framework for a simplified formal semantics of UML combining different diagrams types at the example of UML Sequence Diagrams. In Sect. 4, we explain how UML, together with an XML-based analysis of UML models, can be used as a basis for a formally based method for critical systems development. In Sect. 5, we present some information on tools for advanced XML-based processing of UML models. We describe a framework that incorporates automated tools for the analysis of UML models against critical requirements. We end with pointers to related work and a conclusion.

1.1 Overview and Background

Traditionally, there exist different methods for ensuring reliability of critical systems.

Break-And-Fix This approach accepts that deployed systems may fail; whenever a problem is noticed and identified, the error is fixed. The Break-And-Fix approach is probably the most obvious one, however it has a lot of drawbacks. It is inherently disruptive - fixing the system often implies distributing patches, which disturbs users, annoys customers and destroys their confidence. What is worse, the method is unsafe and insecure - we can never be sure that the new problem will not disturb critical functionality, or that it will not be spotted at first by a malicious person, who will try to compromise the system further.

Traditional formal methods, on the other hand, offer very good quality of the developed critical systems. There exist a lot of successful research in this direction. For security-critical systems, this includes [MCF87], [BAN89], [Mea91], [Low96], [AG97], [Pau98]. However, formal methods are rarely applied in practice because of the high costs arising from the necessary training for the developers of the system, and from the construction of the formal specification of the system.

The Unified Modeling Language (UML) [UML01], the de facto industry-standard in object-oriented modeling, together with XML-based processing of UML models, offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context.

- As the de facto standard in industrial modeling, a large number of developers is trained in UML, making less training necessary. Also, UML specifications of systems under development may already be available for analysis, which again saves time and cost.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined, opening up the possibility for advanced tool-support to assist the development of safety-critical systems.
- After several years of evolution, a XML/XMI -based standard for UML model representation has evolved, enabling interchange and automated processing of the UML models.

2 UML Machines and UML Machine Systems

UML Machines are based on the *Abstract State Machines* notion. We recall central concepts here, for a formal definition see [BS03]. They are inspired by the *Algebraic State Machines* from [BW00].

Abstract State Machines A *state* A is a non-empty set X containing distinct elements *true*, *false*, and *undef* together with a set $\mathbf{Voc}(A)$ of function names with interpretations in the base set X. An *Abstract State Machine* (ASM) consists of an *initial state* and an *update rule*, where the variable assignment of the initial state sends each variable to the value *undef*. An ASM is executed by iteratively firing the update rule. Thereby, its current state is *updated*; that is, the interpretations of its functions are redefined in terms of the previous interpretations. The syntax and informal semantics of update rules is given inductively as follows (the formal semantics can be found in [BS03]):

skip : causes no change.

f($\bar{s}$):=t : updates f at the tuple $\bar{s}$ to map to the element t.

if g **then** R **else** S : If g holds, the rule R is executed, otherwise S.

do − in − parallel $R_1, \ldots, R_k$ **enddo** : R_i execute simultaneously, if for any two update rules $f(\bar{s}) := t$ and $f(\bar{s}) := t'$, we have $t = t'$; otherwise the execution stops.

seq R S **endseq** : R and S are executed sequentially.

loop v **through list** X $R(v)$: iteratively execute $R(x)$ for all $x \in X$.

case v **of** x_1 : **do** R_1 ... x_n : **do** R_n **else** S : execute by case distinction.

Extending ASMs to UML Machines We define UML Machines, an extension of ASMs with a UML-like communication mechanism that uses buffers. We will use UML Machines to specify components of a system that interact by exchanging messages from a set **Events** which are dispatched from resp. received in multi-set buffers (*output queues* resp. *input queues*).

Definition 1 A *UML Machine* $(A, \mathsf{inQu}_A, \mathsf{outQu}_A)$ is given by an ASM A and two multi-set names $\mathsf{inQu}_A, \mathsf{outQu}_A \in \mathbf{Voc}(A)$ such that the rules in A change inQu_A only by removing and outQu_A only by adding elements.

The set names inQu_A, outQu_A model the input buffer and the output buffer of the UML Machine A. We assume that at the initial state of the UML Machine, they always have the value $\emptyset$.

The behavior of a UML Machine $(A, \mathsf{inQu}_A, \mathsf{outQu}_A)$ is captured in the following definition, where a multi-set of input resp. output values represents the input resp. output during a time interval of a given finite length. Possible non-determinism in the UML Machine rules leads to *sets* of output sequences.

Let $\mathsf{toinQu}_A(X) \stackrel{\text{def}}{=} \mathsf{inQu}_A := \mathsf{inQu}_A \uplus X$. Given a UML Machine $(A, \mathsf{inQu}_A, \mathsf{outQu}_A)$, and a sequence $\vec{I}$ of multi-sets, consider the UML Machine $\mathbf{Behav}(A(\vec{I}))$ with the vocabulary $\mathbf{Voc}(\mathbf{Behav}(A(\vec{I}))) \stackrel{\text{def}}{=} \mathbf{Voc}(A) \cup \{\mathsf{outlist}(A)\}$, and the rule $\mathbf{Behav}(A(\vec{I}))$ given in Fig. 1. For any given run $r \in \mathbf{Run}(\mathbf{Behav}(A(\vec{I})))$ of the UML Machine $\mathbf{Behav}(A(\vec{I}))$, after completion of r, $\mathsf{outlist}(A)$ contains a sequence of multi-sets of values $\mathsf{outlist}(A)^r$.

Definition 2 The input/output behavior of a UML Machine $(A, \mathsf{inQu}_A, \mathsf{outQu}_A)$ is a function $[\![A]\!]()$ from finite sequences of multi-sets of values to sets of sequences of multi-sets of values defined by $[\![A]\!](\vec{I}) \overset{\text{def}}{=} \{\mathsf{outlist}(A)^r : r \in \mathbf{Run}(\mathbf{Behav}(A(\vec{I})))$.

Intuitively, given a sequence $\vec{I}$ of multi-sets of input values, the rule $\mathbf{Behav}(A(\vec{I}))$ computes the set of possible sequences of multi-sets of output values by iteratively adding each multi-set in $\vec{I}$ to inQu_A, calling A, and recording the multi-set of output values from outQu_A in $\mathsf{outlist}(A)$.

We would like to build up UML specifications in a modular way, by combining a set of UML Machines together with communication links connecting them to form a new formal specification. To achieve this, we define the notion of a *UML Machine System* (UMS). Our approach allows a rather flexible treatment of the communication since the UMS *main loop* (Fig. 2) can be modified as necessary. For example, our explicit way of modeling the communication links and the messages exchanged over them allows modeling exterior influence on the communication within a system (such as attacks on insecure connections, or quality-of-service aspects of network).

Definition 3 An *UML Machine System (UMS)* $\mathcal{A} = (\mathsf{Name}_{\mathcal{A}}, \mathsf{Comp}_{\mathcal{A}}, \mathsf{Sched}_{\mathcal{A}}, \mathsf{Links}_{\mathcal{A}}, \mathsf{Msgs}_{\mathcal{A}})$ is given by

- a name $\mathsf{Name}_{\mathcal{A}} \in \mathbf{UMNames}$,
- a finite set $\mathsf{Comp}_{\mathcal{A}}$ of UML Machines called *components*
- a UML Machine $\mathsf{Sched}_{\mathcal{A}}$, the *scheduler* that may call the components as subroutines,
- a set $\mathsf{Links}_{\mathcal{A}}$ of two-element sets $l \subseteq \mathsf{Comp}_{\mathcal{A}}$, the *communication links* between them, and
- a set of messages $\mathsf{Msgs}_{\mathcal{A}} \subseteq \mathbf{MsgNm}$ that the UML Machine System is ready to receive.

```
Rule $\mathbf{Behav}(A(\vec{I}))$
    loop $I$ through list $\vec{I}$
        $\mathsf{toinQu}_A(I)$
        $\mathbf{Exec}(A)$
        $\mathsf{outlist}(A) := \mathsf{outlist}(A).\mathsf{outQu}_A$
        $\mathsf{outQu}_A := \emptyset$
```

Figure 1: Behavior of a UML Machine

```
$\mathit{Rule}\ \langle\mathcal{A}\rangle$
seq
    forall $S$ with $S \in \mathsf{Comp}_{\mathcal{A}}$ do
        $\mathsf{inQu}_{\langle S\rangle} := \mathsf{inQu}_{\langle S\rangle} \uplus$
            $\{\!\{\mathbf{tail}(e) : e \in (\mathsf{inQu}_{\langle\mathcal{A}\rangle} \setminus \mathsf{Msgs}_{\mathcal{A}}) \uplus$
            $\biguplus_{l \in links_S} \mathsf{linkQu}_{\langle\mathcal{A}\rangle}(l) \wedge \mathbf{head}(e) = S\}\!\}$
    $\mathsf{inQu}_{\langle\mathcal{A}\rangle} := \emptyset$
    $\langle\mathsf{Sched}_{\mathcal{A}}\rangle$
    forall $l$ with $l \in \mathsf{Links}_{\mathcal{A}}$ do
        $\mathsf{linkQu}_{\langle\mathcal{A}\rangle}(l) := \{\!\{e \in \mathsf{outQu}_{\langle S\rangle} :$
            $S \in \mathsf{Comp}_{\mathcal{A}} \wedge l = \{\mathbf{head}(e), A_i\}\}\!\}$
    $\mathsf{outQu}_{\langle\mathcal{A}\rangle} := \mathsf{outQu}_{\langle\mathcal{A}\rangle} \uplus \biguplus_{S \in \mathsf{Comp}_{\mathcal{A}}} \{\!\{\mathbf{tail}(e) :$
        $e \in \mathsf{outQu}_S \wedge \mathbf{head}(e) = \langle\mathcal{A}\rangle\}\!\}$
    forall $S$ with $S \in \mathsf{Comp}_{\mathcal{A}}$ do
        $\mathsf{outQu}_{\langle S\rangle} := \emptyset$
endseq
```

Figure 2: Main loop of a UML Machine System

The set **MsgNm** consists of finite sequences of names $n_1.n_2.\dots.n_k$ where $n_1,\dots,\ n_{k-2}$ are names of UMSs, n_{k-1} is a name of a UML Machine, and n_k is the local name of the message. We define the set **Events** of events to consist of terms of the form $msg(exp_1,\dots,exp_n)$ where $msg \in$ **MsgNm** is an n-ary message name and $exp_1,\dots,exp_n \in$ **Exp** are expressions, the *parameters* or *arguments* of the event (for a given set of expressions **Exp**).

We recursively define the behavior of any UMS A as a UML Machine $\langle A\rangle$. For any UML Machine A, we define $\langle A\rangle \stackrel{\text{def}}{=} A$. Given a UMS $\mathcal{A}$, the UML Machine $\langle\mathcal{A}\rangle$ models the joint execution of the components of $\mathcal{A}$ and their communication by exchanging messages over the links. The execution rule for$\langle\mathcal{A}\rangle$ is given in Fig. 2 (where $links_S \stackrel{\text{def}}{=} \{\{A,B\} \in \mathsf{Links}_{\mathcal{A}} : A = S\}$ is the set of links connected to S).

3 Formal Semantics for a Fragment of UML

We sketch our approach to defining a formal semantics for UML models on the example of Sequence Diagrams. Further UML Diagrams are formalized similarly [Jür02b] (where also more details about this approach can be found).

In UML, messages can be synchronous (meaning that the sender of the message passes the thread of control to the receiver and receives it back together with the return message) or asynchronous (meaning that the thread of control is split in two, one each for the sender and the receiver). Accordingly, we partition the set of message names **MsgNm** into sets of operations **Op**, signals **Sig**, and return messages **Ret**. Because of the space restrictions, here we only give a formalization of the asynchronous communication.

In our model, every object or subsystem O has associated multi-sets inQu_O and outQu_O (*event queues*). We model sending a message $msg = op(exp_1, \dots, exp_n) \in \mathbf{Events}$ from an object S to an object R as follows:

(1) The object S places the message $R.msg$ into its multi-set outQu_S.

(2) The dispatching component distributes the messages from out-queues to the intended in-queues (while removing the message head); in particular, $R.msg$ is removed from outQu_S and msg added to inQu_R.

(3) The object R removes msg from its in-queue and processes its content.

This way of modeling communication allows for a very flexible treatment; for example, we can modify the UMS main loop (Fig. 2) to take account of knowledge on the underlying communication layer (such as security or performance issues).

Objects may execute *actions*. We write **Action** for the set of actions which are expressions of the following forms:

Send action: $\mathsf{send}(sig(a_1, \dots, a_n))$ for an n-ary signal $sig \in \mathbf{Sig}$ and argument $a_i \in \mathbf{Exp}$.

Void action: *nil*

For any action a, we define the expression $\mathbf{ActionRule}(a)$ (where $\mathcal{A}$ is the UML machine in which $\mathbf{ActionRule}(a)$ is executed).

$$\mathbf{ActionRule}(\mathsf{send}(e)) \equiv \Big(\mathsf{outQu}_{\mathcal{A}} := \mathsf{outQu}_{\mathcal{A}} \uplus \{\!\{ e \}\!\}\Big)$$
$$\mathbf{ActionRule}(nil) \equiv \mathbf{skip}$$

The set of *Boolean expressions* **BoolExp** is the set of first-order logical formulae with equality statements between elements of **Exp** as atomic formulae.

3.1 Sequence diagrams

To demonstrate how behavioral diagrams can be included in our framework for defining a formal semantics for UML, we exemplarily consider a (again simplified and restricted) fragment of sequence diagrams.

For readability, the prefix *obj* on the messages sent to an object *obj* which is contained in a sequence diagram may be omitted in that diagram (since it is implicit).

Abstract syntax of sequence diagrams A sequence diagram $D = (\mathsf{Obj}(D), \mathsf{Links}(D))$ is given by

- a set $\mathsf{Obj}(D)$ of pairs (O, C) where O is an object of class C whose interaction with other objects is described in D and

- a sequence $\mathsf{Links}(D)$ consisting of elements of the form $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ where
  - $\mathsf{source}(l) \in \mathsf{Obj}(D)$ is the source object of the link,
  - $\mathsf{guard}(l) \in \mathbf{BoolExp}$ is a Boolean expression (the guard of the link),
  - $\mathsf{msg}(l) \in \mathbf{Events}$ is the message of the link, and
  - $\mathsf{target}(l) \in \mathsf{Obj}(D)$ is the target object of the link.

Behavioral semantics We fix a sequence diagram $\mathcal{S}$modeling the objects in $\mathsf{Obj}(\mathcal{S}) \overset{\text{def}}{=} \bigcup_{D \in \mathcal{S}} \mathsf{Obj}(D)$ and an object $O \in \mathsf{Obj}(\mathcal{S})$. Further we assume that the set **Var** contains elements $arg_{O,l,n}$ for each $O \in \mathsf{Obj}(\mathcal{S})$ and numbers l and n, representing the nth argument of the operation that is supposed to be the lth operation received by O according to the set of sequence diagrams $\mathcal{S}$, and define $args_{O,l} = [arg_{O,l,1}, \ldots, arg_{O,l,k}]$ (where the operation is assumed to have k arguments). Then we give the behavior of O as defined in $\mathcal{S}$ as a UML machine $(\llbracket \mathcal{S}.O \rrbracket^{SD}, \{\mathsf{inQu}_{\llbracket \mathcal{S}.O \rrbracket^{SD}}\}, \{\mathsf{outQu}_{\llbracket \mathcal{S}.O \rrbracket^{SD}},$
$\mathsf{finished}_{\llbracket \mathcal{S}.O \rrbracket^{SD}}\})$. The rule of the UML machine $\llbracket D.O \rrbracket^{SD}$ is given in Fig. 3.

```
Rule Exec($D.O$)
if $\mathsf{cncts} = [\,]$ then $\mathsf{finished}_{D.O} := true$
else
    if $\mathsf{source}(\mathbf{head}(\mathsf{cncts})) = O \wedge \mathsf{guard}(\mathbf{head}(\mathsf{cncts}))$
        then
            ActionRuleSD($\mathsf{msg}(\mathbf{head}(\mathsf{cncts}))$);
            if $\mathsf{target}(\mathbf{head}(\mathsf{cncts})) \neq O$ then
                $\mathsf{cncts} := \mathbf{tail}(\mathsf{cncts})$;
    if $\mathsf{target}(\mathbf{head}(\mathsf{cncts})) = O$ then
        choose $e$ with $e \in \mathsf{inQu}_O \wedge$
        $\mathbf{msgnm}(\mathsf{msg}(\mathbf{head}(\mathsf{cncts}))) = \mathbf{msgnm}(e)$ do
            $\mathsf{inQu}_O := \mathsf{inQu}_O \setminus \{\!\{ e \}\!\}$ ;
            $args_{D,\mathsf{lnum}} := \mathbf{Args}(e)$;
            $\mathsf{lnum} := \mathsf{lnum} + 1$;
            if $\mathbf{msgnm}(e) \in \mathbf{Op}$ then
                $\mathsf{sender}(\mathbf{msgnm}(e)) :=$
                    $\mathbf{sndr}(e).\mathsf{sender}(\mathbf{msgnm}(e))$;
            $\mathsf{cncts} := \mathbf{tail}(\mathsf{cncts})$
```

Figure 3: UML machine for sequence diagram

Given a sequence $\vec{l}$ of links and an object O, define $\vec{l}\!\upharpoonright_O$ to be the subsequence $\vec{l}$ of those elements l with $\mathsf{source}(l) = O$ or $\mathsf{target}(l) = O$.

3.2 Reasoning about model properties

The UML Machines framework allows formally inspecting the UML Model for certain properties. In case of a security-critical system, these can be for example "data security" (indicating that certain data item shall not leak out of a system component). The desirable security properties can be introduced in the model using UML extension (see [Jür03] for details) and further the whole model can be checked for consistency, whether the required properties are met by the design.

To investigate security properties of a system, it is extended with a subsystem modeling behavior of a potential adversary. The notion of UMS allows its natural modeling. We can create specific types of adversaries that attack different parts of the system in a specified way. For example, an attacker of type *insider* may be able to intercept the communication links in a company-wide local area network. We model the behavior of the adversary by defining a class of UML Machines that can access the communication links of the system in a specified way. To evaluate the security of the system with respect to the given type of adversary, we consider the joint execution of the system with any UML Machine in this class.

Security evaluation of specifications is done with respect to a given type A of adversary. For this, in particular, one has to specify a set $\mathcal{K}^p_A \subseteq \mathbf{Exp}$ of previous knowledge of the adversary type A. Also, $\mathcal{K}^a_A \subseteq \mathbf{Exp}$ contains knowledge that may arise from accessing components (see below). We define $\mathcal{K}^0_A = \mathcal{K}^a_A \cup \mathcal{K}^p_A$ to be the initial knowledge of any adversary of type A.

Given a UMS $\mathcal{A}$ we define the set $int_{\mathcal{A}}$ of (recursively) contained components:

- for an UML Machine A, $int_A := \{A\}$ and
- for a UMS $\mathcal{A}$, $int_{\mathcal{A}} := \bigcup_{\mathcal{B} \in \mathsf{Comp}_{\mathcal{A}}} int_{\mathcal{B}}$.

Similarly, for a UMS $\mathcal{A}$ we define the set $lks_{\mathcal{A}}$ of (recursively) contained links:

- for an UML Machine A, $lks_A := \emptyset$ and
- for a UMS $\mathcal{A}$, $lks_{\mathcal{A}} := \mathsf{Links}_{\mathcal{A}} \cup \bigcup_{\mathcal{B} \in \mathsf{Comp}_{\mathcal{A}}} lks_{\mathcal{B}}$.

To capture the capabilities of a possible attacker, we assume that, given a UMS $\mathcal{A}$, we have a function $\mathsf{threats}^{\mathcal{A}}_A(x)$ that takes a component or link $x \in int_{\mathcal{A}} \cup lks_{\mathcal{A}}$ and a type of adversary A and returns a set of strings $\mathsf{threats}^{\mathcal{A}}_A(x) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$ under the following conditions:

- for $x \in int_{\mathcal{A}}$, we have $\mathsf{threats}^{\mathcal{A}}_A(x) \subseteq \{\mathsf{access}\}$,
- for $x \in lks_{\mathcal{A}}$, we have $\mathsf{threats}^{\mathcal{A}}_A(x) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$, and
- for $l \in lks_{\mathcal{A}}$ with $i \in l$ and $\mathsf{threats}^{\mathcal{A}}_A(i) = \{\mathsf{access}\}$, the equation $\mathsf{threats}^{\mathcal{A}}_A(l) = \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$ holds.

The idea is that $\mathsf{threats}_A^{\mathcal{A}}(x)$ specifies the *threat scenario* against a component or link x in the UML Machine System $\mathcal{A}$ 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 is permitted by the threat scenario to apply to the concerned links. Thus each function $\mathsf{threats}()$ gives rise to the set of accessed data $\mathcal{K}_A^a$ mentioned above and a set of permitted actions $perm_A$:

- $\mathcal{K}_A^a$ consists of all expressions appearing in the specification for any $i \in int_{\mathcal{A}}$ with $\mathsf{access} \in \mathsf{threats}_A^{\mathcal{A}}(i)$.
- $perm_A$ consists of
  - all actions $\mathsf{delete}_l \equiv \mathsf{linkQu}_{\mathcal{A}}(l) := \emptyset$ for any $l \in lks_{\mathcal{A}}$ with $\mathsf{delete} \in \mathsf{threats}_A^{\mathcal{A}}(l)$ (deletes all elements from $\mathsf{linkQu}_{\mathcal{A}}(l)$),
  - all actions $\mathsf{read}_l(m) \equiv m := \mathsf{linkQu}_{\mathcal{A}}(l)$ for any $l \in lks_{\mathcal{A}}$ with $\mathsf{read} \in \mathsf{threats}_A^{\mathcal{A}}(l)$ and any variable name m (copies the content of $\mathsf{linkQu}_{\mathcal{A}}(l)$ to the variable m), and
  - all actions $\mathsf{insert}_l(e) \equiv \mathsf{linkQu}_{\mathcal{A}}(l) := \mathsf{linkQu}_{\mathcal{A}}(l) \uplus \{\!\{e\}\!\}$ for any $l \in lks_{\mathcal{A}}$ with $\mathsf{insert} \in \mathsf{threats}_A^{\mathcal{A}}(l)$ and any $e \in \mathcal{K}_A^0$ (adds an element e to $\mathsf{linkQu}_{\mathcal{A}}(l)$).

Intuitively, $perm_A$ consists of those actions that an adversary of type A is capable of doing with respect to the multi-set $\mathsf{linkQu}_{\mathcal{A}}(l)$ for any link l.

4 UML and XML-based Analysis for Critical Systems Development

We will now explain how UML, together with an XML-based analysis of UML models, can be used as a basis for a formally based method for critical systems development. We will first analyze our requirements on the proposed method, and demonstrate how the UML-based solution meets them. To keep the presentation concise and intelligible, we will restrict ourself to security-critical systems. However our approach is generic, and can be applied to other criticality requirements like safety, quality-of-service, etc.

The UML-based formal methodology for development of security-critical system should meet the following requirement.

- Given a system model described with UML, it should *automatically* evaluate it for security-related vulnerabilities in design.
- The methodology should be available to developers *not specialized in security*, and still 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 latter step is often considered as granted, however for many security properties it can be very difficult [RSG+01] and normally requires the developer to have special qualifications in cryptography.

- Costs of correcting flaws in a software system grow dramatically in the process of development, therefore we would like to consider security from *early design phases*.

- Consider security on *different levels of abstraction*, and *in system context*. Security of a complex distributed computer system can be violated on different levels. Even worse, security properties are generally not preserved by the composition [Jür01] and therefore blindly combining even proven security mechanisms may result in a faulty system. The method should detect these kinds of errors.

- Make use of the powerful pattern concept and *encapsulate established rules of prudent security engineering*.

- For certain security-critical software products, like for example firewalls, the acceptance procedure is comparable to the development itself in laboriousness. Thus we want to make *certification cost-effective*.

Now we will look closer at some of the requirements listed. It is obvious that today any software development methodology, which aims broad acceptance, needs to provide the end user with software tools supporting it. We were facing two challenges in this regard.

First, we need a uniform and standardized way of acquiring and processing UML models. Until recently there were no standards on storing UML models, and different UML editing tools were producing files in proprietary format. The development and spreading of XML as a universal data representation language motivated the development of the XML Metadata Interchange (XMI) [Obj02b] language for storing UML models into a file.

For developing critical systems using UML and XML-based analysis, one needs a precise semantics of the used notation. The UML is relatively precisely defined, however its semantics is given partially in prose, leaving room for ambiguities [UML01]. We have refined the semantics by giving mathematically precise meaning of UML constructs, as shown in the next chapter.

Now we will take a close look at *UMLsec*, the UML-based language for secure-critical system development.

4.1 Creating and Using UMLsec

There is a set of requirements to meet before UML can be used for secure system development. The language must be extended with the necessary security-related constructs. Correct application of the new language in the application domain must be described and enforced. The conflict between flexibility and non-ambiguity of the notation must be solved. Last but not least, tools for working with UMLsec models must be created. In more details, on the notational level we want the language to have the following properties.

Basic security requirements such as *secrecy* and *integrity* should be integrated into the language.

Different Threat scenarios should be considered automatically depending on the adversary strengths.

Common security concepts like for example *tamper-resistant hardware* should be readily available.

Common security mechanisms like for example *access control* should be included in the language.

Cryptographic primitives (e.g. *(a)symmetric encryption*) should be defined and correctly handled. Data security properties of a computer system cannot be feasibly modelled at the abstraction level of the data values and cryptographic key values being processed. In fact a possibility for precise modeling of a cryptographic algorithm at the data value level means finding a feasible way of breaking the algorithm, and therefore renders it useless. Therefore the modeling with UMLsec is normally done on the protocol level using the supplied Cryptographic primitives.

Physical security of the deployed system need to be described by using the language.

Security management (e.g. *secure workflow*) should be addressed.

Domain-specific extensions (Java, smart cards, CORBA, ...) shall be considered during language design.

4.2 UMLsec: Extending UML

The UML specification [UML01] 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*. Important feature of a lightweight extension is that it should be "strictly additive to the standard UML semantics. This means that such extensions must not conflict with or contradict the standard semantics." [UML01] In particular, adhering to this requirement assures that UMLsec can be used to extend any UML model without conflicting with existing tools or other UML extension, potentially UML profiles for other criticality requirements.

Stereotypes define new types of modelling 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 property 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, given using the formal semantics described below, formulate rules which must be met by the design to support the requested security properties. The most commonly used UMLsec stereotypes together with associated tags are listed in the Fig. 4.

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, fresh	critical object
no down-flow	subsystem	secret	prevents leak of information
fair exchange	subsystem	start, stop	after start eventually reaches stop
guarded access	subsystem		access control using guard objects
guarded	object	guard	guarded object

Figure 4: UMLsec stereotypes

To enable automatic reasoning about UMLsec model properties, formal semantics are introduced, which give mathematically precise meaning to the UML subset we are using, and to UMLsec constructs in its content. The following subset of UML diagrams is considered.

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

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

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

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

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

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

4.3 Security Analysis of UMLsec Models

To apply formal verification methods for testing security properties of a distributed - which means open - system, it is necessary to "close" it by modeling all 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, efficiency and reliability of the automated verification of security properties depend on the correctness and completeness of the simulated adversary behavior. The issue is addressed in the literature, for example [Low99] and [FJ00]. To create the adversary model we do not have to foresee any possibly attack scenario; we can define an intruder which can do everything possible, as shown on Fig. 5, and further restrict his behavior using additional information about the system.

The extent of possible intervention into the system functionality depends on the physical properties of the system, and on the adversary abilities. The UMLsec methodology provides possibility to define these parameters, addressing two issues. Firstly, obviously impossible attack scenario are not included in the analysis report. Secondly, the whole analysis process can be clustered into several steps, avoiding the state explosion problem [EMCGP00].

We suggest that there can be adversaries with different capabilities regarding intervention with the system. Considering for example an online banking application, a simple user could read and modify information on the Internet, and

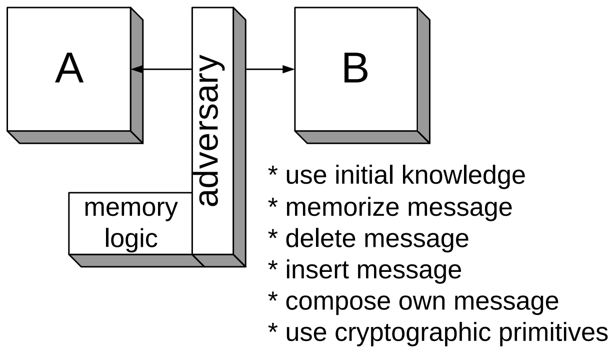

Figure 5: Simulating Adversary Behavior

access the client node. A malicious employee however could read and alter traffic in the internal LAN, and access internal servers. To define capabilities of an *adversary type* A against an object stereotyped with s we introduce a function $\mathsf{Threats}_A(s)$ returning a subset of *abstract threats* $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$. Examples on Fig. 6 and 7 define the function for two adversary types; new adversary types can be freely formulated.

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

Figure 6: 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	$\emptyset$

Figure 7: Threats from the *insider* adversary

Basing on these model-wide definitions, we define how the adversary can interact with each model element. For a link l in deployment diagram contained in the subsystem S we define the set $\mathsf{threats}_A^S(l)$ of concrete threats to be the smallest set satisfying the following conditions. If each node n, that l is contained in (Note that nodes and subsystems may be nested one in another.), 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 is permitted by the threat scenario to apply to the concerned links. 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 in the link. The new messages can be created by applying cryptographic primitives to the previously read data and initial knowledge of the adversary.

To investigate 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 properties.

This requires mathematically precise definitions of the UMLsec models, including their dynamic behavior. This is defined in [Jür03] using *UML Machines*, which are inspired by the Abstract State Machines (ASM) [Gur95] [BS03]. A UML Machine is a transition system the states of which are algebraic structures, with built-in communication mechanisms; it defines behavior of a system component. A UML Machine is defined by the *initial state*, *transitions rules* which is applied iteratively and defines 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 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 [Jür03]. 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.

The definition of the UML semantics has to be omitted here and can be found in [Jür03].

4.4 Example Application

We demonstrate usability of the UMLsec methodology on a variant of the Internet security protocol TLS (the successor of SSL) as proposed in [APS99]. The example in Fig. 8 shows the UMLsec specification.

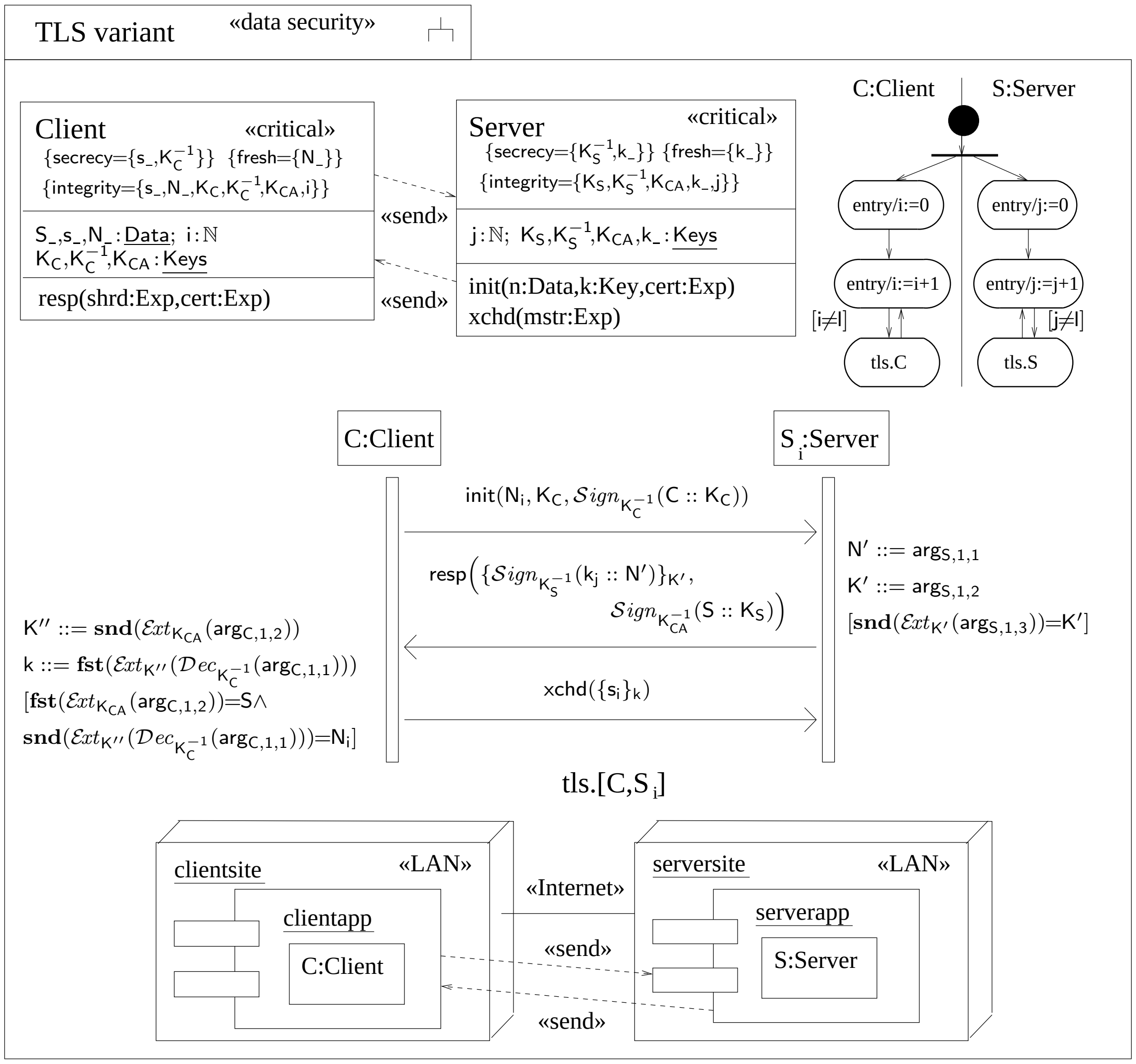
TLS variant
«data security»
Client
«critical»
{secrecy={s_,K_C^{-1}}} {fresh={N_}}
{integrity={s_,N_,K_C,K_C^{-1},K_CA,i}}
S_,s_,N_:Data; i:ℕ
K_C,K_C^{-1},K_CA:Keys
resp(shrd:Exp,cert:Exp)
«send»
«send»
Server
«critical»
{secrecy={K_S^{-1},k_}} {fresh={k_}}
{integrity={K_S,K_S^{-1},K_CA,k_,j}}
j:ℕ; K_S,K_S^{-1},K_CA,k_:Keys
init(n:Data,k:Key,cert:Exp)
xchd(mstr:Exp)
C:Client
S:Server
entry/i:=0
entry/j:=0
entry/i:=i+1
entry/j:=j+1
[i≠l]
[j≠l]
tls.C
tls.S
C:Client
S_i:Server
init(N_i, K_C, Sign_{K_C^{-1}}(C :: K_C))
N′ ::= arg_{S,1,1}
K′ ::= arg_{S,1,2}
[snd(Ext_{K′}(arg_{S,1,3}))=K′]
resp({Sign_{K_S^{-1}}(k_j :: N′)}_{K′}, Sign_{K_CA^{-1}}(S :: K_S))
K″ ::= snd(Ext_{K_CA}(arg_{C,1,2}))
k ::= fst(Ext_{K″}(Dec_{K_C^{-1}}(arg_{C,1,1})))
[fst(Ext_{K_CA}(arg_{C,1,2}))=S∧
snd(Ext_{K″}(Dec_{K_C^{-1}}(arg_{C,1,1})))=N_i]
xchd({s_i}_k)
tls.[C,S_i]
clientsite
«LAN»
clientapp
C:Client
«Internet»
«send»
«send»
serversite
«LAN»
serverapp
S:Server

Figure 8: Variant of the TLS handshake protocol

The goal is to let a client C send a secret m over an untrusted communication link to a server S in a way that provides confidentiality and server authentication, by using a symmetric key K_{CS} to be exchanged.

We write $\{E\}_K$ for the encryption of the expression E under the key K and $\mathcal{D}ec_K(E)$ for the decryption of E with K. The protocol uses both RSA encryption and signing. Thus we assume the equations $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ and $\{\mathcal{D}ec_{K^{-1}}(E)\}_K = E$ to hold (where K^{-1} is the private key belonging to the public key K).

The protocol assumes that there is a secure (with respect to integrity) way for C to obtain the public key K_{CA} of the certification authority, and for S to obtain a certificate $\mathcal{D}ec_{K_{CA}^{-1}}(S :: K_S)$ signed by the certification authority that contains its name and public key. The adversary may also have access to K_{CA}, $\mathcal{D}ec_{K_{CA}^{-1}}(S :: K_S)$ and $\mathcal{D}ec_{K_{CA}^{-1}}(Z :: K_Z)$ for an arbitrary $Z \in \mathbf{Data} \setminus \{S\}$. The complete specification can be found in [Jür03].

One can now demonstrate that this proposed variant of TLS contains a security flaw. The message flow diagram corresponding to the man-in-the-middle attack is the following.

$$\begin{tikzcd}
C \arrow[r, "N_i::K_C::\mathcal{S}ign_{K_C^{-1}}(C::K_C)"] & A \arrow[r, "N_i::K_A::\mathcal{S}ign_{K_A^{-1}}(C::K_A)"] & S \\
C & A \arrow[l, "\{\mathcal{S}ign_{K_S^{-1}}(K_j::N_i)\}_{K_C}::\mathcal{S}ign_{K_{CA}^{-1}}(S::K_S)"'] & S \arrow[l, "\{\mathcal{S}ign_{K_S^{-1}}(K_j::N_i)\}_{K_A}::\mathcal{S}ign_{K_{CA}^{-1}}(S::K_S)"'] \\
C \arrow[r, "\{s\}_{K_j}"] & A \arrow[r, "\{s\}_{K_j}"] & S
\end{tikzcd}$$

Here the adversary has access to the communication link between client and server modelled by $\mathsf{linkQu}_{\mathcal{T}}(l_{CS})$ since it is supposed to be an Internet link. More details are given in [Jür03] (as well as a correction which is proved secure using the formal semantics).

5 Tools for Advanced XML-based Processing of UML models

Using the UML notation has two important aspects:

- *Standardized notation* helpes to capture, store and exchange knowledge about the system under design.
- *Semantics*, although semi-formal, assures that different developers understood the common meaning of UML diagrams.

Initially, different UML tools implemented proprietary UML storage formats which made exchange and reuse of the models impossible. Having chosen a UML tool, the developer was tied to using it through the whole project. Applying emerging technologies to the UML modeling on the industrial level was virtually impossible. To suggest any custom UML processing, one would have to develop a complete UML editor and persuade the auditorium to use it.

Development of the XML as universal data storage format changed this situation dramatically. In the year 2000 The Object Management Group (OMG) [OMG03] issued the first specification for the XML Metadata Interchange (XMI) language [Obj02b] which – among other applications – became a standard for serializing UML models into a file.

The XMI language is compliant with MOF (Meta Object Facility, [Obj02a]), which is a framework for specifying meta-information (also called metamodels). Initially it was developed to define CORBA-based services for managing meta-information. Currently its applications include definition of modeling languages such as UML and CWM (Common Warehouse Model). The framework operates on a four-level data abstraction model, shown on the Fig. 9.

M3	Meta-Metamodel	MetaClass, MetaAssociation - MOF Model
M2	Metamodel	Class, Attribute, Dependency - UML (as a language), CWM
M1	Model	Person, City, Book - UML Model
M0	Data	Bob Marley, Bonn - Running programm

Figure 9: MOF Framework

We consider the abstraction levels from bottom up. The lowest level M0 deals with the data instances, for example *Mr. Smith, 35 years old, lives in New York*. The level M1 describes data models, in software development this corresponds to the UML model of the application. An example for this layer is a *Person* with attributes *Name, Age, Address*. The next abstraction level M2 is the modeling language itself. There exist different modeling languages for different application domains, and the last abstraction level M3 is the common environment for defining these modeling languages, standardized by the MOF. It operates with three elements:

MOF Object defines object types for the target model. It includes a *name*; a set of *attributes*, both pre-defined and custom; a set of *operations*; a set of *association references*; a set of *supertypes* it inherits, and some other information. The MOF object is a *container* for its component features; i.e. any *attributes*, *operations* and *association references*. It may also contain MOF definitions of *data types* and *exceptions*.

MOF Association defines a link between two MOF objects. The MOF links are always binary and directed. A link is a *container* for two *association ends*, each representing one object the link is connected to.

MOF Package groups related MOF elements for reuse and modularization. It is defined by a *name*; a list of *imports* which defines a set of other MOF Packages whose components may be re-used by components defined within the Package; a list of *supertypes* which defines a set of other MOF Packages whose components form a part of the Package; and a set of contained elements including other Objects, Associations and Packages.

The MOF also defines the following secondary elements:

Data Types can be used to define *constructed* and *reference* data types.

Constants define compile-time constant expressions.

Exceptions can be raised by Object operations.

Constraints can be attached to other MOF Elements. Constraint semantics and verification are not part of the MOF specification, and therefore they can be defined with any language.

The MOF is related to two other standards.

XML Metadata Interchange (XMI) is 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 the XMI mapping rules to derive DTDs and XML Schemas for UML serialization. MOF itself is defined using MOF itself, 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).

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

After the standards were introduced, major producers of UML editors eventually have picked it up, and currently support model interchange in the XMI format. Together with the wide support for the XML language in the industry, including broad range of libraries, editors and accompanying technologies, this enables development of the lightweight UML processing tools, tailed to carry one particular task.

The whole story is applicable to the formalized critical system development with UML. To facilitate acceptance of the formalized UML-based software development, automated processing of UML models was highly required. Prototype tools supporting this functionality have been developed at the TU Munich; some results of these projects are presented below. Especially we hope that the publicly available web-based interface will provide a simple and accessible entry into the methodology.

5.1 XML-based Data-Binding with MDR

Technically the central question was how to work with UML/XMI files. There exist three possible approaches.

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

The first two methods although more flexible, require more development effort. However for UML processing we are concerned about the data contained in documents rather about the document itself and its structure. For this purpose, data binding offers a much simpler approach to working with XML data.

There exist several libraries supporting data-binding for XML. It was important to use one with appropriate data abstraction level. For example the widely used Castor library [Cas03] would leave the developer with a very abstract representation of the UML model, on the level of MOF constructs. However there exist data-binding libraries which provide representation of a UML XMI file on the abstraction level of a UML model. This allows the developer to operate directly with UML concepts (such as classes, statecharts, stereotypes, etc.). We use the MDR (MetaData Repository) library which is part of the Netbeans project [Net03], also used by the freely available UML modeling tool Poseidon 1.6 Community Edition [Gen03]. Another such library is the Novosoft NSUML project [Nov03].

The MDR library implements a MOF repository with support for XMI and JMI standards. The Fig. 10 illustrates how the repository is used for working with UML models.

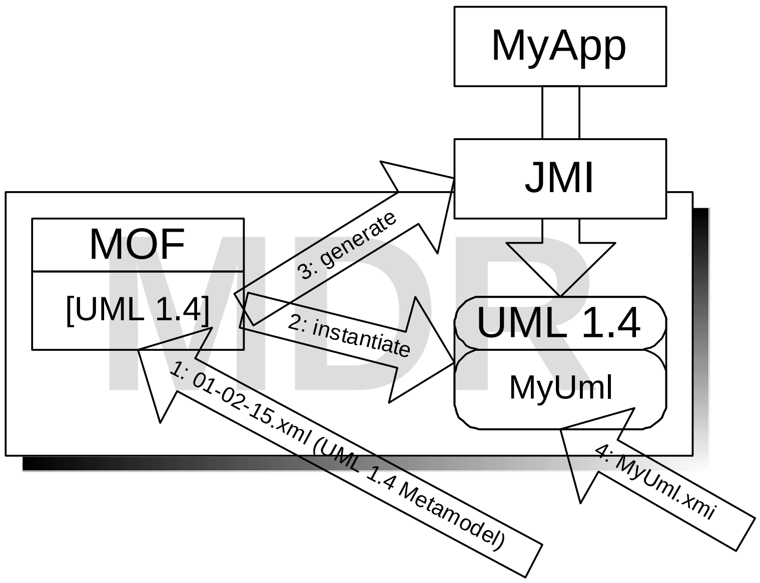

Figure 10: Using the MDR Library

The XMI description of the modeling language is used to customize the MDR for working with a particular model type, UML in this case (step 1). The XMI description of the UML 1.4 is published by the Object Management Group (OMG) [OMG03]. A storage customized for the given model type is created (step 2). Additionally, based on the XMI specification of the modeling language, the MDR library creates the JMI (Java Metadata Interface) implementation for accessing the model (step 3). This allows the application to manipulate the model directly on the conceptual level of UML. The UML model is loaded into the repository (step 4). Now it can be accessed through the supplied JMI interfaces from a Java application. The model can be read, modified, and later saved into an XMI file again.

Because of the additional abstraction level implemented by the MDR library, using it in the UML suite should facilitate upgrading to upcoming UML versions, and promises the highest available standard compatibility.

5.2 XML-based UML Tool Suite

To facilitate the application of our approach in industry, automated tools for the analysis of UML models using the suggested semantics are required. We describe a framework that incorporates several such verifiers currently developed at the TU München.

Functionality We can group all the UML model features, which can be verified, into two major categories.

- *Static features.* Checkers for static features (for example, a type-checking like enforcement of security levels in class and deployment diagrams) can be implemented directly.
- *Dynamic features.* Verification of these properties requires interfacing with a Model Checker. The relevant elements of the UML specification are translated into the model-checker input language; the required model properties are presented by Temporal Logic formulae.

We present the architecture of the UML tool suite developed at the TU Munich, providing verification tools for these features. Its architecture and basic functionality are illustrated on the Fig. 11. The implemented functionality is publicly available through a webbased interface (see http://www4.in.tum.de/csduml/interface/interface.html).

The developer creates a model and stores it in the UML 1.5 / XMI 1.2 file format. The file is imported by the UML into the internal MDR repository. Other components of the UML suite access the model through the JMI interfaces, generated by the MDR library. 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 input language of the relevant analysis engine (currently, the Promela language

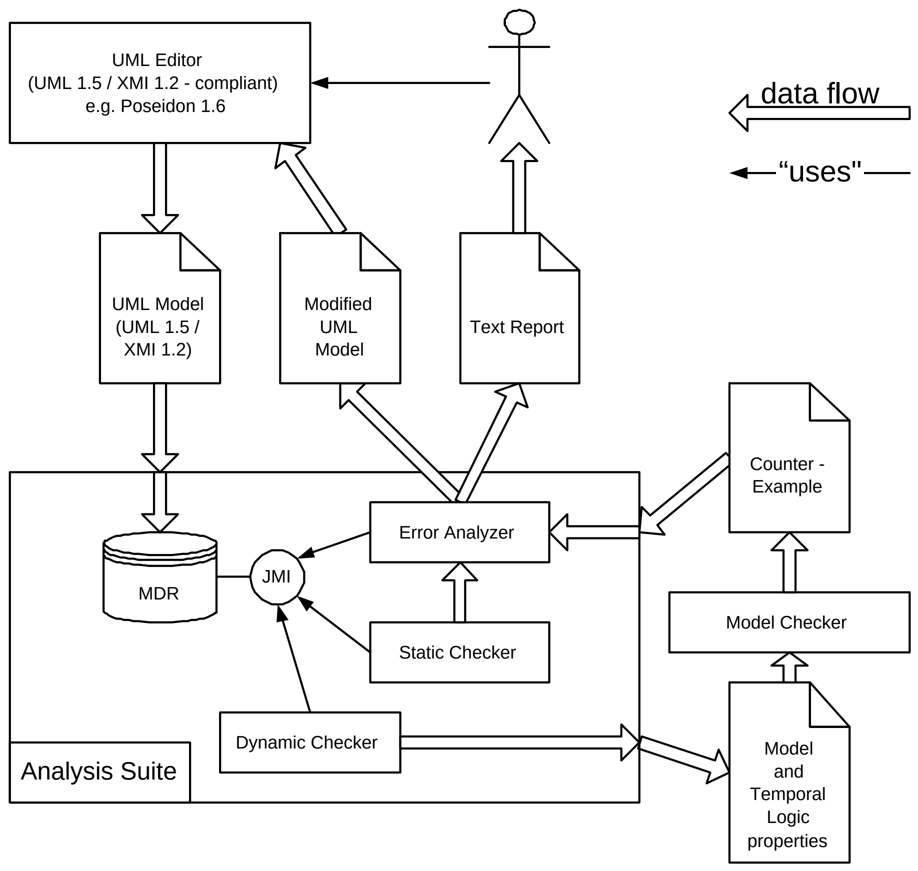

Figure 11: UML tools suite

for the Spin model checker and the TPTP format for the automated theorem prover Setheo). The analysis engine is spawned by the UML suite as an external process; its results (and a counterexample in case there was a problem found) are delivered back to the Error Analyzer. The Error Analyzer uses the information received from both Static Checker and Dynamic Checker to produce a Text Report for the developer describing found problems, and a Modified UML Model, where the found errors are visualized and/or corrected.

The idea behind the tools suite is to provide a common programming framework for the developers of different verification modules (*tools*). Thus a tool developer should concentrate on the verification logic and not on the handling Input/Output. Different tools, implementing verification logic modules (*Static Checkers* or *Dynamic Checkers* on the Fig. 11) can be independently developed and integrated. Currently there exist *Static Checkers* for most static UMLsec properties, and *Dynamic Checkers* for some dynamic properties.

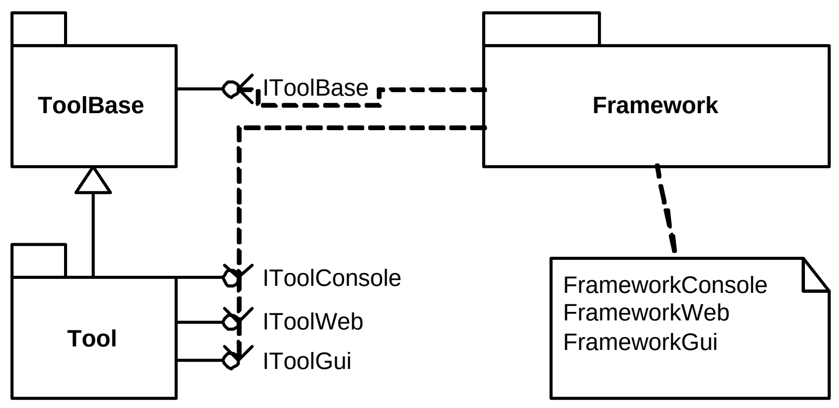

Figure 12: Tool interfaces

The tool implementation follows the following simple concepts.

- It is given a default UML model it operates on. It may load further models if necessary.
- The 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 can have its internal state which is preserved between commands.

Architecture By its design the UML framework provides a common programming environment for the developers of different verification modules (*tools*). Thus a tool developer concentrates on the verification logic and not on the auxiliary tasks like handling input/output. An additional requirement is the independent implementation of different pieces of UML model verification logic by different developers.

On any Java-enabled platform, the framework can run in one of the three moduses:

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

Accordingly, each tool that is integrated in the UML framework must implement a common interface IToolBase plus three media-dependent interfaces IToolConsole, IToolWeb and IToolGui as illustrated in Fig. 12.

However the requirement to implement all three media-dependent interfaces for a tool would mean a serious overhead for the tool developer. To assist

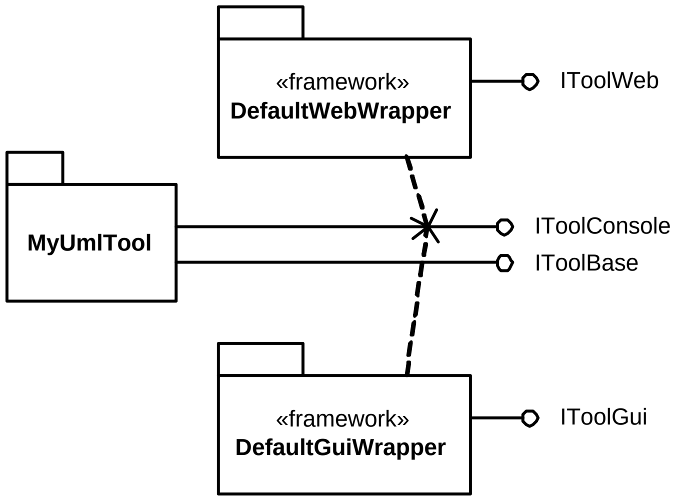

Figure 13: Default interface wrappers

the developer in this regard, the framework provides default implementations for the IToolWeb and IToolGui interfaces, as illustrated on the Fig. 13. These default wrappers use the implemented by the tool IToolConsole interface and render the provided text output in the HTML or scrolling text window format respectively. Thus each plugged into the framework tool must implement at least IToolBase and IToolConsole interfaces. If the tool developers wants to exploit all capabilities of the Web or GUI media, he has to implement the IToolWeb and/or IToolGui interfaces, which give him more control over the tool input and output. In the Gui mode the developer is then requested to provide an instance of the JPane - derived class which hosts the complete UI of the tool and has ability to customize menu and toolbar of the framework. In the Web mode the developer can fully control the rendered HTML document.

The UML framework uses the IToolBase interface to retrieve general information about the tool, and one of the three tool media-specific interfaces to call command provided by the tool and receive the output. The output is further rendered by the framework on the current media.

Each tool exposes a set of commands which can be executed through the function GetConsoleCommands, GetWebCommands and GetGuiCommands of the corresponding interface. Thus the tool can provide different functionality on different media, adopting to its specifics.

The tool can execute several commands consequently; 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.

To achieve the media-independent operation of the tools their parameter input, as well as their output, is handled by the framework and not by the tools themselves. Each single command during its execution defines the set of required input parameters, receives the them from the framework. On behalf of the tool, the UML framework collects the parameters from the user using the current input/output media (console, web, or GUI). Currently supported parameter types are Integer, Double, String and File. Further types can be easily integrated into the framework as necessary.

6 Related Work

There has been a considerable amount of work towards a formal semantics for various parts of UML; a complete overview has to be omitted. [FELR98] discusses some fundamental issues concerning a formal foundation for UML. [RACH00] gives an approach using algebraic specification. [BGH$^+$98] uses a framework based on stream-processing functions. [BCR00] uses ASMs for UML statecharts which was a starting point for the current work.

There are several existing tools for automatic verification of the UML models. The HUGO Project [SKM01] checks behavior described by a UML Collaboration diagram against a transitional system comprising several communicating objects. The vUML Tool [LP99] analysis the behavior of a set of interacting object, defined in the similar way. A related approach to ours is also given by the CASE tool AutoFocus [HSSS96] which uses a UML-like notation. However neither tool can be directly extended for our purpose. Firstly, our approach allows formalisation and verification of different UML diagrams in combination. Secondly, our implementation explicitly models data types, which is necessary for handling, for example, encryption primitives.

7 Conclusion

The development and spreading of the UML language within the last years, especially its standardization and introduction of supporting technologies, is changing its role in the software development from a notational aid to a powerful framework with support for automation of many development tasks.

We presented work for formal critical systems development using UML. We provided a formal semantics for a fragment of UML using UML Machines which puts diagrams into context. The semantics is particularly useful to analyze the interaction between a system and its environment and to analyze UML specifications in a modular way. In particular, we explained how to use the semantics to analyze UML specifications for criticality requirements, by including an adversary model.

We have demonstrated a framework for automated processing of UML models, which facilitates technology transfer to industry. This framework has been used to connect various analysis engines (the model checker Spin, the automated theorem provers Setheo and SPASS, and a Prolog verification engine) to support the automated analysis of criticality requirements (in particular, behavioral properties).

The proposed method of XML-based Analysis of UML Models has been successfully tried out in several industrial projects.

We believe that the suggested approach to critical system development using UML and XML-based processing of UML models will find widespread acceptance in the modern software development industry for the following reasons.

- It is based on UML, which is the de-facto standard in the software development, which facilitates acceptance in industrial software development teams.
- The application of the methodology requires no special training in security (in case of UMLsec) or the other criticality domains.
- The suggested formal semantics for a simplified fragment of UML lays a foundation for advanced XML-based tool support for the methodology, making automatic verification of the criticality features possible.

References

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[APS99] V. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost ? In *Conference on Computer Communications (IEEE Infocom)*, New York, March 1999.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[BCR00] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines. Theory and Applications*, volume 1912 of *LNCS*, pages 223–241. Springer, 2000.

[BGH$^+$98] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, views and models of UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, 1998.

[BS03] E. Börger and R. Stärk. *Abstract State Machines*. Springer, 2003.

[BW00] M. Broy and M. Wirsing. Algebraic state machines. In T. Rus, editor, *8th International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *LNCS*. Springer, 2000.

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

[EMCGP00] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. 2000.

[FELR98] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19:325–334, 1998.

[FJ00] Laurent Vigneron Florent Jacquemard, Michael Rusinowitch. Compiling and verifying security protocols. Technical report, LORIA Universite, France, 2000.

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

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

[HSSS96] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus: A tool for distributed systems specification. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96*, volume 1135 of *LNCS*, pages 467–470, Uppsala, Sweden, Sept. 9–13 1996. Springer.

[Jür01] J. Jürjens. Composability of secrecy. In V. Gorodetski, V. Skormin, and L. Popyack, editors, *International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS 2001)*, volume 2052 of *LNCS*, pages 28–38. Springer, 2001.

[Jür02a] J. Jürjens. A UML statecharts semantics with message-passing. In *Symposium of Applied Computing 2002*, pages 1009–1013, Madrid, March 11–14 2002. ACM.

[Jür02b] J. Jürjens. Formal Semantics for Interacting UML subsystems. In *FMOODS 2002*, pages 29–44. IFIP, Kluwer, 2002.

[Jür03] J. Jürjens. *Secure Systems Development with UML*. Springer, 2003. In preparation.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.

[Low99] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(2, 3):89–146, 1999.

[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.

[MCF87] J.K. Millen, S.C. Clark, and S.B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, February 1987.

[Mea91] C. Meadows. A system for the specification and analysis of key management protocols. In *IEEE Symposium on Security and Privacy*, pages 182–195, 1991.

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

[Nov03] Novosoft NSUML project. Available from http://nsuml.sourceforge.net/, 2003.

[Obj02a] Object Management Group. Mof 1.4 specification. Available at http://www.omg.org/technology/documents/formal/mof.htm, April 2002.

[Obj02b] Object Management Group. OMG XML Metadata Interchange (XMI) Specification. Available at http://www.omg.org/cgi-bin/doc?formal/2002-01-01, January 2002.

[Obj03] Object Management Group. OMG Unified Modeling Language Specification v1.5: Revisions and recommendations, March 2003. Version 1.5. OMG Document formal/03-03-01.

[OMG03] Object Management Group. http://www.omg.org/, 2003.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hußmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE2000)*, volume 1783 of *LNCS*, pages 127–146. Springer, 2000.

[RSG$^+$01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[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.

[UML01] UML Revision Task Force. OMG UML Specification v. 1.4. OMG Document ad/01-09-67. Available at http://www.omg.org/uml, 2001.