A Foundation for Tool-Supported Critical Systems Development with UML

Jan Jürjens*and Pasha Shabalin
Software & Systems Engineering, TU München, Germany
juerjens@in.tum.de, shabalin@in.tum.de

Abstract

High quality development of critical systems poses serious challenges. Formal methods have been proposed to address them, but their use in industry is not as wide-spread as originally hoped. We thus propose to use the Uni£ed Modeling Language (UML), the de-facto industry standard speci£cation language, as a notation together with a formally based tool-support for critical systems development.

We introduce UML Machines, which is a formal notation designed to re¤ect properties of the UML execution semantics relevant to criticality requirements. We use it to de£ne a foundation that puts models for the different diagrams into context and gives a precise meaning to mechanisms such as message-passing between objects or components speci£ed in different diagrams, while offering the possibility to analyze criticality requirements.

We present tool-support for this approach developed at the TU München, which facilitates transfer of the methodology to industrial contexts.

Keywords. *UML, critical systems, formal models of object-oriented design, speci£cation, veri£cation, secure computing, tool support.*

1. Introduction

High quality development of critical systems (be it real-time, security-critical or dependable systems) is dif£cult. 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 con¤ict with cost. Where thorough methods of system design pose high costs through personnel training and use, they are all too often avoided. The Uni£ed Modeling Language (UML) [8] 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 de£ned.

Nevertheless, the UML semantics is given only in prose form [8], leaving room for ambiguities. However, to provide advanced tool-support (for example, automated checking of behavioral properties of a UML speci£cation) 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 [3, 6, 17, 10, 5]; speci£cally, [1] 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 precise mathematical meaning to whole UML speci£cations, 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) [2] 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 ¤exible 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 [4]. Also, we de£ne the concept of UML Machine System

*http://www4.in.tum.de/˜juerjens . Supported within the Verisoft Project of the German Ministry for Education and Research.

(UMS) that allows one to build up speci£cations in a modular way (corresponding to the use of UML subsystems). We use this to de£ne a semantics for a simpli£ed 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 speci£cations 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 [11] but extends to diagram types not treated in [11] (such as sequence diagrams) and to the development of tool-supported for automated veri£cation.

Outline In Sect. 2, we recall the necessary de£nitions and introduce the notion of UML Machine used for our semantics. 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 simpli£ed formal semantics of UML combining different diagrams types at the example of UML Sequence Diagrams. We show results of our work towards tool support for the methodology in Sect. 4. We end with pointers to related work, a conclusion and indication of future work.

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 de£nition see [2]. They are inspired by the *Algebraic State Machines* from [4].

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 £ring the update rule. Thereby, its current state is *updated*; that is, the interpretations of its functions are rede£ned 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 [2]):

$\mathbf{skip}$: causes no change.

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

$\mathbf{if}\ g\ \mathbf{then}\ R\ \mathbf{else}\ S$: If g holds, the rule R is executed, otherwise S.

$\mathbf{do-in-parallel}\ R_1, \ldots, R_k\ \mathbf{enddo}$: 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.

$\mathbf{seq}\ R\ S\ \mathbf{endseq}$: R and S are executed sequentially.

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

$\mathbf{case}\ v\ \mathbf{of}\ \ x_1 : \mathbf{do}\ R_1\ \ \ldots\ \ x_n : \mathbf{do}\ R_n\ \ \mathbf{else}\ S$: execute by case distinction.

Extending ASMs to UML Machines We de£ne 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 $\mathbf{Events}$ which are dispatched from resp. received in multi-set buffers (*output queues* resp. *input queues*).

De£nition 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 de£nition, where a multi-set of input resp. output values represents the input resp. output during a time interval of a given £nite length. Possible non-determinism in the UML Machine rules leads to *sets* of output sequences.

Let $\mathsf{toinQu}_A(X) \stackrel{\mathrm{def}}{=} \mathsf{inQu}_A := \mathsf{inQu}_A \uplus X$. 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 $\mathbf{Voc}(\mathbf{Behav}(A(\vec{I}))) \stackrel{\mathrm{def}}{=} \mathbf{Voc}(A) \cup \{\mathsf{outlist}(A)\}$, and the rule $\mathbf{Behav}(A(\vec{I}))$ given in Fig. 1.

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

Figure 1. Behavior of a UML Machine

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 $\mathsf{outlist}(A)^r$.

De£nition 2 The input/output behavior of a UML Machine $(A, \mathsf{inQu}_A, \mathsf{outQu}_A)$ is a function $[\![A]\!]()$ from £nite sequences of multi-sets of values to sets of sequences of multi-sets of values de£ned by $[\![A]\!](\vec{I}) \stackrel{\mathrm{def}}{=} \{\mathsf{outlist}(A)^r : r \in \mathbf{Run}(\mathbf{Behav}(A(\vec{I})))$.

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 speci£cations in a modular way, by combining a set of UML Machines together with communication links connecting them to form a new formal speci£cation. To achieve this, we de£ne the notion of a *UML Machine System* (UMS). Our approach allows a rather ¤exible treatment of the communication since the UMS *main loop* (Fig. 2) can be modi£ed as necessary. For example, our explicit way of modeling the communication links and the messages exchanged over them allows modeling exterior in¤uence on the communication within a system (such as attacks on insecure connections, or quality-of-service aspects of network).

De£nition 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 £nite 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.

The set $\mathbf{MsgNm}$ consists of £nite sequences of names $n_1.n_2.\ldots.n_k$ where $n_1,\ldots,\ n_{k-2}$ 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 de£ne the set $\mathbf{Events}$ of events to consist of terms of the form $msg(exp_1, \ldots, exp_n)$ where $msg \in \mathbf{MsgNm}$ is an n-ary message name and $exp_1, \ldots, exp_n \in \mathbf{Exp}$ are expressions, the *parameters* or *arguments* of the event (for a given set of expressions $\mathbf{Exp}$).

We recursively de£ne the behavior of any UMS A as a UML Machine $\langle A \rangle$. For any UML Machine A, we de£ne $\langle A \rangle \stackrel{\mathrm{def}}{=} A$. Given a UMS $\mathcal{A}$, the UML Machine

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

$\langle \mathcal{A} \rangle$ 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 $links_S \stackrel{\mathrm{def}}{=} \{\{A, B\} \in \mathsf{Links}_{\mathcal{A}} : A = S\}$ is the set of links connected to S).

3. Formal Semantics for a Fragment of UML

We sketch our approach to de£ning a formal semantics for UML models on the example of Sequence Diagrams. Further UML Diagrams are formalized similarly [11] (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 $\mathbf{MsgNm}$ into sets of operations $\mathbf{Op}$, signals $\mathbf{Sig}$, and return messages $\mathbf{Ret}$. 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 $msg = op(exp_1, \ldots, exp_n) \in \mathbf{Events}$ 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 re-

moving 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 ¤exible 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 $\mathbf{Action}$ 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 de£ne the expression $\mathbf{ActionRule}(a)$ (where $\mathcal{A}$ is the UML machine in which $\mathbf{ActionRule}(a)$ is executed).

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

The set of *Boolean expressions* $\mathbf{BoolExp}$ is the set of £rst-order logical formulae with equality statements between elements of $\mathbf{Exp}$ as atomic formulae.

3.1. Sequence diagrams

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

For readability, the pre£x *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 £x a sequence diagram $\mathcal{S}$modeling the objects in $\mathsf{Obj}(\mathcal{S}) \stackrel{\text{def}}{=} \bigcup_{D \in \mathcal{S}} \mathsf{Obj}(D)$ and an object $O \in \mathsf{Obj}(\mathcal{S})$. Further we assume that the set $\mathbf{Var}$ 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 de£ne $args_{O,l} = [arg_{O,l,1}, \dots, arg_{O,l,k}]$ (where the operation is assumed to have k arguments). Then we give the behavior of O as de£ned in $\mathcal{S}$ as a UML machine $([\![\mathcal{S}.O]\!]^{SD}, \{\mathsf{inQu}_{[\![\mathcal{S}.O]\!]^{SD}}\}, \{\mathsf{outQu}_{[\![\mathcal{S}.O]\!]^{SD}}, \mathsf{finished}_{[\![\mathcal{S}.O]\!]^{SD}}\})$. The rule of the UML machine $[\![D.O]\!]^{SD}$ is given in Fig. 3.

$Rule$ $\mathbf{Exec}(D.O)$
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
      $\mathbf{ActionRuleSD}(\mathsf{msg}(\mathbf{head}(\mathsf{cncts})))$;
      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, de£ne $\vec{l}\upharpoonright_O$ 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 [12] 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 speci£c types of adversaries that attack different parts of the system in a speci£ed 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 de£ning a class of UML Machines that can access the communication links of the system in a speci£ed 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 speci£cations 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 de£ne $\mathcal{K}^0_A = \mathcal{K}^a_A \cup \mathcal{K}^p_A$ to be the initial knowledge of any adversary of type A.

Given a UMS $\mathcal{A}$ we de£ne 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 de£ne 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 $\mathsf{threats}^\mathcal{A}_A(x)$ speci£es 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 speci£cation for any $i \in int_\mathcal{A}$ with $\mathsf{access} \in \mathsf{threats}^\mathcal{A}_A(i)$.
- $perm_A$ consists of
  - all actions $\mathsf{delete}_l \equiv \mathsf{linkQu}_\mathcal{A}(l) := \emptyset$ for any $l \in lks_\mathcal{A}$ with $\mathsf{delete} \in \mathsf{threats}^\mathcal{A}_A(l)$ (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 $\mathsf{read} \in \mathsf{threats}^\mathcal{A}_A(l)$ 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 $\mathsf{insert} \in \mathsf{threats}^\mathcal{A}_A(l)$ and any $e \in \mathcal{K}^0_A$ (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 multiset $\mathsf{linkQu}_\mathcal{A}(l)$ for any link l.

4. Tool support

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 veri£ers currently developed at the TU München.

Functionality We can group all the UML model features, which can be veri£ed, 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.* Veri£cation of these properties requires interfacing with a Model Checker. The relevant elements of the UML speci£cation are translated into the model-checker input language; the required model properties are presented by Temporal Logic formulae.

At present there exist a veri£cation tool for the £rst group of features, and implementation of the support for the Spin

Model Checker is in progress. The implemented functionality is publicly available through a webbased interface (see http://www4.in.tum.de/csduml/interface/interface.html).

Accessing UML models On the technical level, the central question is how to acquire and process UML models. Most existing UML editing tools can store the model in a XMI 1.2 (XML Metadata Interchange). However, processing of a UML model directly on the XMI level leaves the developer with a very abstract representation of the model. Therefore libraries have been developed which provide representation of a UML XMI £le on the abstraction level of a UML model, and thus allow the developer to directly operate with UML concepts (such as classes, statecharts, stereotypes, etc.). We use the MDR (MetaData Repository) project which is part of the Netbeans project [14], also used by the freely available UML modeling tool Poseidon 1.6 Community Edition [7].

The MDR library implements a repository for any model described by a MOF-compliant modeling language. The Fig. 4 illustrates how the repository is used for working with UML models.

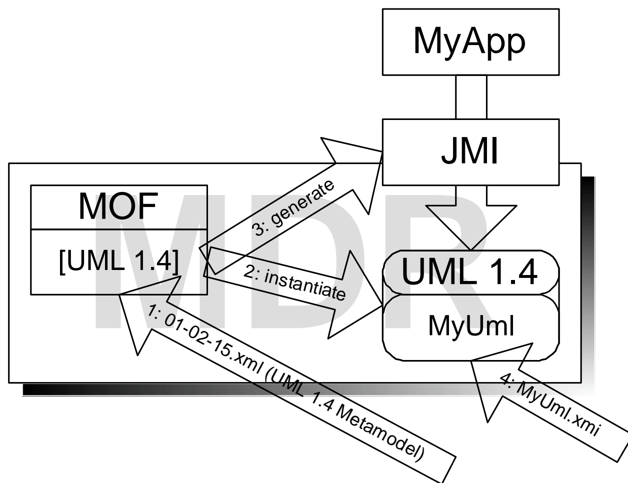

Figure 4. Using the MDR Library

Initially 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 de£nition of the UML 1.4 is published by the Object Management Group (OMG) [16]. A storage customized for the given model type is created (step 2). Additionally, based on the XMI speci£cation 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, modi£ed, and later saved into an XMI £le again.

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

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

The Fig. 5 illustrates the architecture of the UML tool framework which meets the listed requirements. We brie¤y describe its functionality. The developer creates a model and stores it in the UML 1.5 / XMI 1.2 £le format. The £le is imported by the tool into the internal MDR repository. The tool accesses the model through the JMI interfaces generated by the MDR library. The checker parses the model and checks the constraints associated with the stereotype. The results are delivered as a text report for the developer describing found problems, and a modi£ed UML model, where the stereotypes whose constraints are violated are highlighted.

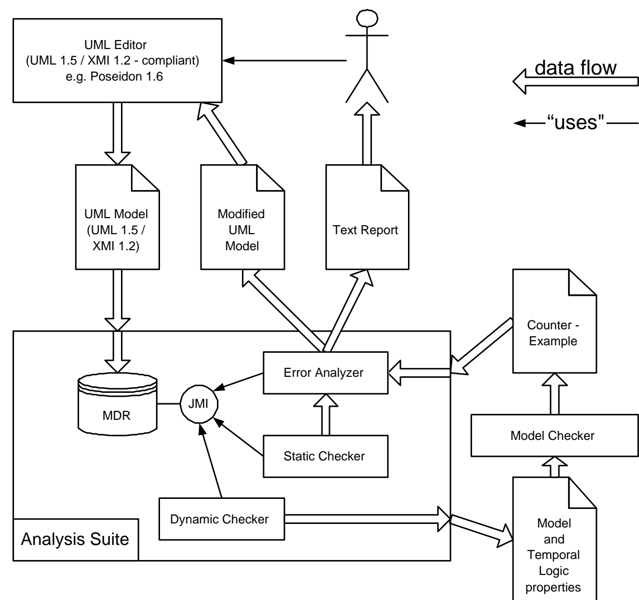

Figure 5. UML tools suite

A further feature implemented in the UML framework, which should facilitate practical application, is its availabil-

ity in different input/output environments. On any Java-enabled platform, it 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 to this requirement each integrated in the UML framework tool must implement a common interface IToolBase plus three media-dependent interfaces IToolConsole, IToolWeb and IToolGui as illustrated on the Fig. 6.

However the requirement to implement all three media-dependent interfaces for a tool would mean a serious overhead for the tool developer. To assist the developer in this regard, the framework provides default implementations for the IToolWeb and IToolGui interfaces, as illustrated on the Fig. 7. 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-speci£c interfaces to call command provided by the tool and receive the output. The output is further rendered by the framework on the current media.

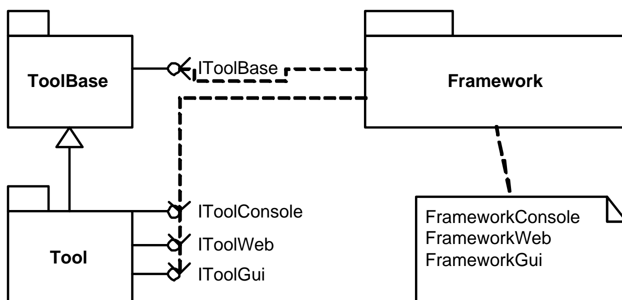

Figure 6. Tool interfaces

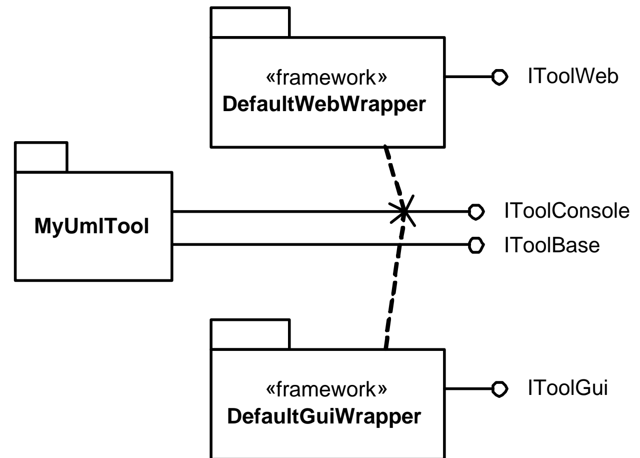

Figure 7. Default interface wrappers

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 speci£cs.

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 de£nes 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.

5. 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. [6] discusses some fundamental issues concerning a formal foundation for UML. [17] gives an approach using algebraic speci£cation. [3] uses a framework based on stream-processing functions. [1] uses ASMs for UML statecharts which was a starting point for the current work. [5] uses the p-calculus to formalize UML Activity Diagrams.

There are several existing tools for automatic veri£cation of the UML models. The HUGO Project [19] checks behavior described by a UML Collaboration diagram against a transitional system comprising several communicating objects. The vUML Tool [13] analysis the behavior of a set of interacting object, de£ned in the similar way. A related approach to ours is also given by the CASE tool AUTOFOCUS [9] which uses a UML-like notation. However neither tool can be directly extended for our purpose. Firstly, our approach allows formalisation and veri£cation of different UML diagrams in combination. Secondly, our implementation explicitly models data types, which is necessary for handling, for example, encryption primitives.

6. Conclusion and Future Work

We introduced a framework 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 speci£cations in a modular way. In particular, we explained how to use the semantics to analyze UML speci£cations for criticality requirements, by including an adversary or failure model. Our model-based approach to critical systems development should be seen as complementary to existing approaches, which are often based on testing methodology (see for example [18] in the context of UML).

We have demonstrated a framework for automated processing of UML models, which facilitates technology transfer to industry. In ongoing work, this framework is used to connect various analysis engines (Model Checker, Prolog, Theorem Prover) to support the automated analysis of criticality requirements (in particular, behavioral properties).

Acknowledgements Collaboration with the other members of the UMLsec group on creating tool-support for the approach presented here is gratefully acknowledged.

References

[1] 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.

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

[3] 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 Uni£ed Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, 1998.

[4] 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.

[5] Y. Dong and Z. ShenSheng. Using p - calculus to formalize UML activity diagrams. In *10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003)*, pages 47–54, Huntsville, AL, USA, 7-10 April 2003. IEEE Computer Society.

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

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

[8] O. M. Group. OMG Uni£ed Modeling Language Speci£cation v1.5: Revisions and recommendations, Mar. 2003. Version 1.5. OMG Document formal/03-03-01.

[9] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus: A tool for distributed systems speci£cation. 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.

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

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

[12] J. Jürjens. *Secure Systems Development with UML*. Springer, Mar. 2004. To be published.

[13] 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.

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

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

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

[17] 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.

[18] M. Riebisch, I. Philippow, and M. Götze. UML-based statistical test case generation. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World (NetObjectDays 2002)*, volume 2591 of *Lecture Notes in Computer Science*, pages 394–411, Erfurt, Germany, October 7-10 2003. Springer.

[19] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In S. Stoller and W. Visser, editors, *Workshop on Software Model Checking*, volume 55 of *ENTCS*. Elsevier, 2001.

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