Property Specification and Static Verification of UML Models

Igor Siveroni, Andrea Zisman and George Spanoudakis
Department of Computing, City University *
London EC1V 0HB, United Kingdom
{siveroni,a.zisman,gespan}@soi.city.ac.uk

Abstract

We present a Static Verification Tool (SVT), a system that performs static verification on UML models composed of UML class and state machine diagrams. Additionally, the SVT allows the user to add extra behavior specification in the form of guards and effects by defining a small action language. UML models are checked against properties written in a special-purpose property language that allows the user to specify linear temporal logic formulas that explicitly reason about UML components. Thus, the SVT provides a strong foundation for the design of reliable systems and a step towards model-driven security

1. Introduction

Secure software engineering is a new research area concerned with the integration of security and software engineering. Among other things, it proposes that security and system reliability requirements should be considered from the early stages of the software development cycle, that is, from the design and modeling stage. The Unified Modeling language (UML) has become the standard notation for the analysis and design of object-oriented software and systems [10], and, moreover, several extensions have been proposed to include application specific notions and deal with different type of requirements, both quantitative and qualitative. Furthermore, the use of UML models, given their simplicity and high level of abstraction, provides a great opportunity for the application of formal methods and automated verification techniques. The combination of UML models and formal verification techniques is , therefore, extremely attractive.

The work presented in this paper proposes the specification of security and system reliability aspects, and the use of formal verification techniques, from the early stages of the software development. We present a Static Verification Tool (SVT), a system that provides a strong foundation for model-driven design of secure and reliable systems.

The SVT uses UML models composed of UML class diagrams and UML state machine diagrams. Additionally, the SVT allows the user to add extra behavior specification in the form of guards and effects - the SVT defines and implements a small language for this purpose. UML models are checked against security and general application properties written in a special-purpose property language that allows the user to specify linear temporal logic formulas that reason about UML machine states and transitions as well as field values and message passing. The results of simulation runs and the static verification process are sent back to the user in the form of execution traces of the model's UML state-machines.

Our main contributions are (1) the translation of UML models - including the special guard/effect language - into Spin/Promela models that are amenable for model checking and (2) the definition and translation of an expressive property language designed to reason about temporal and general properties of UML state machines. Future work includes extensions in the property language to handle notions of secrecy and access control policies, together with the implementation of attacker models.

2. Framework

The Static Verification Tool (SVT) is based on the static verification framework defined in [18]. The main aim of this framework is to support the design and analysis of abstract behavioral system specification. The SVT implements a subset of the framework components, namely, the (a) Design Model Constructor, (b) Property Editor, (c) Verifiers, (d) Results Visualization and (e) Translators components.

The *Design Model Constructor* component is responsible for the construction of abstract design models of the system. We use UML models [10] for the specification of the structural and behavioral elements of the systems to be verified. We integrate an existing UML case tool to assist with

*This work was supported by the European Commission under the Information Society Technologies Programme as part of the project PEPERS (contract IST-26901)

the construction of such design models. The *Property Editor* allows the user to build the properties to the verified by the SVT. These properties are specified using an extended and user-friendly version of linear temporal logic (LTL) [2] tailored to reason about objects and state machines. We have chosen to use model checking as our main verification technique and, in particular, the *Verifiers* component uses SPIN [5], a generic model checking tool that has been applied to the verification of several control and software systems. SPINs specification language, Promela, is similar to C and supports message passing channels, essential for the modeling of distributed systems. Furthermore, SPIN uses an on-the-fly model checker thus avoiding the need to construct a global state graph. The mix of flexibility and efficiency, together with its ability to specify properties as LTL formulas and automata, makes SPIN an ideal target system.

In order to present the results of the verification process, the framework contains a *Result Visualization Component* that shows the parts of the design models involved in a property violation. This is achieved by displaying a step-by-step execution trace of simulations and error trails. Finally, the framework contains *Translators* to support the mappings from UML design models into SPIN/Promela models, and from properties expressed in the extended LTL property language into the specification language used by the verification tools.

The SVT is implemented in Java and is packaged as an Eclipse plug-in that runs along the Papyrus UML [13] graphical modeler. The plug-in will be made available at the PEPERS project site (www.pepers.org).

3 The Static Verification Tool (SVT)

3.1 UML Diagrams

The UML models used by the SVT are defined by two types of UML diagrams: class diagrams, which define the structure of the model, and state chart diagrams, which specify the behavior of each of the defined classes. A valid UML model, made of a single class diagram and a single state chart diagram per class, must be a correctly typed element of Model as defined in Figure 1 and Figure 1.

The structure of the UML models used by the SVT is defined in Figure 1. A model is made of a set of class and object declarations. Class declarations (C) correspond to the classes defined by the class diagram while object declarations bind object names (o) to class names (c). Figure 2 shows the class diagram of a UML model (our chosen example) that declares two classes: Sender and Receiver. The current implementation automatically generates one object declaration per class e.g. it declares objects oSender and oReceiver from the sample class diagram.

$$
\begin{array}{rcl}
Model & = & (o:c)^{+}\ \ C^{+}\ ,\ o \in OName \\
C \in Class & = & c\ \ SF^{*}\ F^{*}\ M^{*}\ SM \\
F \in Field & = & f\ :\ \tau \\
SF \in SField & = & sf : \tau\ [v]\ ,\ v \in BValue \\
M \in Operation & = & [\tau_r]\ m\ (x:\tau)^{*}\ ,\ x \in Par \\
\tau \in Type & = & Basic \mid c \\
 & & \\
SM \in Machine & = & s_0\ s_f\ s^{+}\ T^{*}\ ,\ s \in State \\
T \in Transition & = & t\ s_1\ s_2\ [m]\ [g]\ ac^{*} \\
g \in Guard & = & BExp \\
ac \in Action & := & v = e;\ \mid \textsf{call}\ v.m(e^{*}); \\
 & & \textsf{if}\ b\ \textsf{then}\ ac^{*}\ \textsf{else}\ ac^{*}\ \textsf{fi}; \\
v & ::= & c.sf \ \mid\ o.f \ \mid\ f \ \mid\ x \\
e \in Exp & := & a \ \mid\ b\ ,\ aOp \in \{+,-,*,/\} \\
a \in AExp & := & a\ aOp\ a \ \mid\ (a) \ \mid\ v \ \mid\ Int \\
b \in AExp & := & \textsf{not}\ b \ \mid\ b\ bOp\ b \ \mid\ v \ \mid\ (b) \\
 & & a\ cOp\ a \ \mid\ \textsf{true} \ \mid\ \textsf{false} \\
\end{array}
$$

$$bOp \in \{\textsf{and}, \textsf{or}, \textsf{implies}\},\ cOp \in \{<, \leq, >, \geq\}$$

Figure 1. UML Model Definition

A class is made of zero or more field (F) and operation (M) declarations, plus a Machine component. Fields are of basic UML type (result field in Sender) or reference type (peer field of type Sender in Receiver). In addition, static fields (SF) can be assigned default values e.g. number field in Receiver has default value 2. An operation (M) is defined by its return value, name (m) and list of argument declarations. Our sample model declares two operations, receiveValue, with a single argument of type Integer (not shown), and getValue in class Sender.

Machine (Figure 1) denotes the state machine defined by the state chart diagram associated to it. The SVT deals with flat, non-hierarchical state machines, that is, state machines with a single region. Therefore, state machines are composed solely by an initial state, a final state, and two finite sets of states and transitions. A state (s) is defined by its name. A transition, denoted by $s_1 \rightarrow s_2$, is composed by its name (t), source state s_1 and target state s_2. Additionally, a transition may carry annotations of the form $m[f]/ac^*$ to indicate the presence of trigger, guard and effect elements. A trigger defines the event (operation m) that triggers the execution of the transition. The guard defines the condition that must be satisfied before the transition is executed, and

Sender
- result: Integer + peer: Receiver
+ receiveValue(inValue)

Receiver
+ value: Integer + peer: Sender + number: Integer = 2
+ getValue()

Figure 2. Example: UML Class Diagram

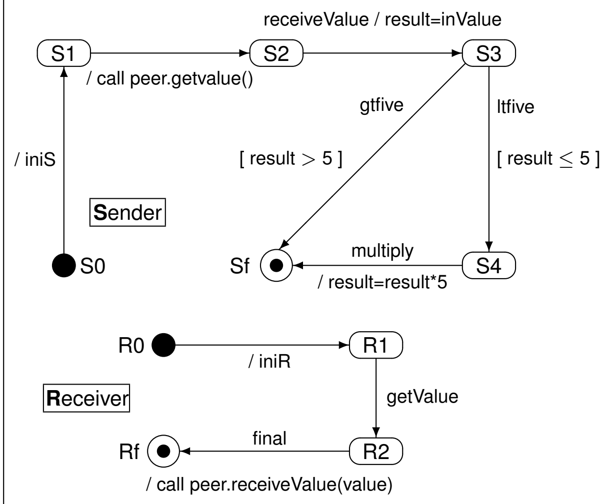

Figure 3. Sender/Receiver State Machines

the effect lists the set of actions that are executed together with the change of state. Figure 3 shows the state chart diagrams associated to the Sender and Receiver classes of our example.

UML leaves the syntax and semantics of guards and effects unspecified. We introduce a small action language, defined by the syntactic class $Action$ in Figure 1, in order to model object state changes and message passing.

An effect is a sequence of actions executed atomically. Guards are boolean expressions that deal with variables denoting static fields ($c.f$), fields of external objects ($o.f$), local fields (f) and argument variables (x). Variables values are modified by assignment actions $v = e$; while guards and effects can be combined conditional actions (if-then-else). Going back to our example, effects iniS and iniR, labeling transitions S0→S1 and R0→R1, respectively, are defined as follows:

result = 2*Receiver.number; peer = oReceiver;
value = Receiver.number; peer = oSender

Objects communicate via messages, sent by call actions and received as triggers. Messages are sent by call actions and placed in the receiver's queue of incoming events. There are two types of transitions: completion (no trigger) and triggered transitions. Completion transitions are checked (for guards) and executed first. If there are no completion transitions available, and the current state is the source of, at least, one triggered transition, the state machine removes an event from the incoming queue and looks for a matching triggered transition (valid guard included) to execute. If there is no matching transition, the event/message is dropped. If there is more than one executable transition, the machine picks one nondeterministically. If the input queue is empty, the state machine blocks.

$$
\begin{array}{rcl}
P & ::= & ultl\ P \mid P\ bltl P \mid (P) \mid o.\text{dropped} \\
 & & uml[.\{\ b\ \}] \mid v^q \mid a^q \mid \text{true} \mid \text{false} \\
ultl & ::= & \text{always} \mid \text{eventually} \mid \text{next} \mid \text{not} \\
bltl & ::= & \text{until} \mid bOp \\
uml & ::= & \text{call}(v, v, m \mid *) \mid \text{state}(o, s) \mid \text{trans}(o, t) \\
v^q & ::= & c.sf \mid o.f \\
a^q & := & a^q\ aOp\ a^q \mid (a^q) \mid v^q \mid Int
\end{array}
$$

Figure 4. Property Language Syntax

Going back to our example, transition R2→Rf in class Receiver is executed together with effect call peer.receiveValue(value); which sends message receiveValue, with argument value, to the object pointed by peer i.e. oSender. On the Sender's side, receiveValue is first removed from the queue of incoming messages and then matched against the transitions that start from the current state. Only a transition whose trigger matches receiveValue will be executed. In our case, the current state should be S2 and, therefore, S2→S3 is executed. The semantics of guards and effects is implemented by the translation described in Section 3.4.

3.2 Property Specification

LTL is a popular formalism well suited not only for the verification of general system requirements, but also for the specification of security and reliability properties. However, in order to be useful in the context of UML models, LTL has to be able to explicitli reason about transition execution, states, class values and messages. In this section we define an extension of LTL that tackles these problems.

The SVT performs verification of UML models against properties specified in a property language based on linear temporal logic (LTL) [2]. The SVT property language (Figure 4) is made of predicate logic terms (including UML specific predicates and variables) mixed with temporal logic operators. Examples of non-temporal formulas follow:

(oSender.result $<$ 6) and (oSender.result $>$ 0)
(5 $<$ oReceiver.value) implies (oSender.result = 5)

All variables (fields and arguments) must be qualified by an object or class (static fields) name. For example, syntax-oSender.result refers to the field result of class Sender.

Linear temporal logic reasons about the validity of predicates over all execution traces of the model. The LTL operators used by the SVF are always, eventually and until. For example, the formula always oSender.result $<$ 100 is true if the field result belonging to the oSender object is less than 100 on all execution states, and eventually (oSender.return = 7) is true if return becomes 7 at least once in all possible executions.

LTL and predicate logic can be mixed with the special predicates call, state and trans specific to UML state ma-

chines: call(o1,o2,m) checks if object o1 has made a call to method m in object o2, state(o,s) checks if the current state of object o is s and state(o,t) checks if transition t is executed in object o. For example, given the following formulas:

P1 eventually state(oSender,S2)

P2 always (call(oSender,oReceiver,getValue) implies (eventually call(oReceiver, oSender, receiveValue))

P3 always (state(oReceiver,R1) implies (oSender.result > 100)

P4 always state(oSender,initial) implies eventually trans(oReceiver,R2)

P1 checks if the state machine of oSender eventually reaches state S2, P2 checks that all calls to getValue are matched by a call to receiveValue, P3 is true if the value of result in object oSender is always greater than 100 every time oReceiver reaches state R1, and P4 will check that transition R2 in oReceiver is always, at some point, executed after oSender reaches state initial.

Of particular interest is the special (optional) scope construct added to the call and state predicates. By writing call(o1,o2,m).{ x < 2 } the user can reason about the arguments of operations. Assuming x is declared as argument of m, the predicate above will be true if there is a call on m from o1 to o2 and the value of x is less than 2. Similarly, there is no need to qualify local field f belonging to object o in state(o,s){ f < 5 and f >= 0 }

Security properties, as defined by [15], can be specified using our property language. For example, the property stating that the first call from o1 to o2 must be Read followed by no calls to Send can be specified by writing:

call(o1,o2,*) .{method != Read} until (call(o1,o2.Read) and always call(o1,o2,*).{method != Send})

3.3 Translating UML models into SPIN

The SVT translates the structural and behavioral aspects of all loaded UML models into Promela, the modeling language used by Spin. This is a rather complex process that, together with the Property Editor, constitutes one of the main contributions of the SVT.

The translation of UML models into Promela must not only implement the semantics of UML state machines; it must also provide the infrastructure to facilitate the verification of properties that reason about state machines, including the provision of variables to keep track of UML elements such as states, transitions and messages.

Promela models are constructed from three basic types of objects: processes, data objects and message channels. Processes, instantiations of proctype declarations, are used to define behavior. Given a UML model, the translator generates a proctype declaration per class and instantiates one process per object. Class fields and operation parameters are implemented as data objects; the transformation declares static fields as global variables while non-static fields and method arguments are declared as local variables inside the body of the process type declaration of the owning class. Promela message channels are used to model the exchange of data between processes. We use channels to model the input and output queues of state machines, essential parts in the implementation of triggers and method invocation.

A Promela model generated by the SVT has the following structure:

<GlobalDeclarations> <ObjectDeclaration>
<StaticFieldsDeclaration> <ClassDeclaration>+
<CommProcessDeclaration> <InitProcess>

The <GlobalDeclarations> section declares the channels used for communication, special datatypes and special global variables. <ObjectDeclaration> declares the constants used to index model objects and <StaticFieldsDeclaration> lists the static fields from all classes they are all implemented as global variables. Processes are declared next. The translator generates a <ClassDeclaration> per class and a special proctype declaration for Comm, the process in charge of message traffic between objects. Finally, the code generated for <initProcess> instantiates all the objects that take part of the execution of the system.

The top level structure of the Promela model generated for the Sender/Receiver example is shown in Figure 5. The top part of the code corresponds to the global declarations used by the model. The special Promela datatype mtype can be used by the programmer to define constants in a similar way as its done with an enumeration type. We use mtype to assign constants (tokens) to operations which, in turn, denote the triggers accepted by the transitions of the state machines. In our example, tokens A0 and A1 denote operations receiveValue and getValue, respectively. Global variables numStarted and checkFlag are internal variables used for object synchronization and model checking.

Communication between processes is performed using channels. The communication model used by the SVT assigns two channels per object, one for incoming messages and another for outgoing messages. Object channels are stored in arrays inQ and outQ, of type chan, and size NUMCHAN (number of objects). All messages have the same structure, <operation, sender, receiver, p0,,pn>, with p1,,pn carrying the values passed as arguments to the method call. The maximum number of parameters in our example is one and, thus, messages are of type mtype, byte, byte, byte. Method calls are implemented by writing into the sending object's (we use variable pNum to keep track of the objects process number) output queue while incoming messages (triggers) are read from the receiving object's incoming queue. The SPIN code generated is:

```
outQ[pNum]!trigger(sender,receiver,prm0);
inQ[pNum]?trigger(sender,receiver,prm0);
```

```
/**** Declarations ****/
mtype = { A0, A1 };
#define QSIZE 2
#define NUMCHAN 2
chan inQ[NUMCHAN] = [QSIZE] of <message>;
chan inQ[NUMCHAN] = [QSIZE] of <message>;
chan outQ[NUMCHAN] = [QSIZE] of <message>;
chan inCom = [QSIZE] of <message>;
chan outCom = [QSIZE] of <message>;
byte numStarted=0;
bool checkFlag=true;

/* Object declarations */
#define oSender 0
#define oReceiver 1

/* Static fields Class Receiver */
byte Receiver_static=2;

/**** Class Declarations ****/
/* Class number 0 */
proctype Sender(byte pNum) { <body> }

/* Class number 1 */
proctype Receiver(byte pNum) { <body> }

proctype Comm(byte pNum) {
byte trigger,receiver,sender;
byte prm0;

end2: atomic {
if
:: outQ[0]?trigger(sender,receiver,prm0) ->
inQ[receiver]!trigger(sender,receiver,prm0);
:: outQ[1]?trigger(sender,receiver,prm0) ->
inQ[receiver]!trigger(sender,receiver,prm0);
fi;
goto end2; }}

init {
atomic {
  run Sender(oSender);
  run Receiver(oReceiver);
  run Comm(2)
}}
where <message> = {mtype,byte,byte,byte};
```

Figure 5. Sender/Receiver Promela Model

We have chosen to use a separate process, Comm,to handle the traffic of messages between objects. This is an approach similar to the one used in [7]; like them, our intention is to use Comm to implement attacker models (future work). The current implementation of Comm (Figure 5) defines the process as an infinite loop that, at each iteration, removes a message from the output queue of one of the objects of the model and places it, untouched, in the input queue of the matching receiver. If all output queues are empty, the communication process blocks. If more than one input queue has a pending message, Comm picks one of them non-deterministically. By following this approach, several kinds of attacks can be modeled with very little effort e.g. messages can be dropped, modified or replicated by changing a few lines.

UML classes are translated into Promela processes that implement the (non-static) structure specified by the class diagram and the behavior defined by its state machine. We describe class declarations by showing the details of the code generated for class Receiver of our example (Figure 6

Objects are implemented as processes and are instantiated by using the Promela operator run. Instantiation requires one argument, pNum, the process number assigned by the translator - see object declarations in Figure 5 - and used as index to access the processs global data structures. All declared processes are instantiated from inside the init process, as listed in the bottom of Figure 5. For example, object oReceiver of class Receiver is instantiated by executing **run Receiver(oReceiver);**.

Local declarations include non-static fields (value, peer), parameters (prm0 denotes the single parameter used by the class) and special variables to keep track of states, transitions (current and transition, respectively), and message passing - all this variables are made available to the verification process and can be accessed by the property language. The translator maps state, transition and operation names to constants. For example, the initial state, assigned to variable state before the execution of the classs body, has value 1. SVT traces are generated by print statements of the form printf(<t start ...) described in more detail in Section 3.5. All state machines must start together from their corresponding initial states. We enforce this by blocking each process until all objects have executed the Synchronise section.

The remaining of the code implements the semantics of state machines. The first part, <CompletionLoop> in Figure 6, implements the execution of transitions that do not contain triggers i.e. completion transitions. Both completion transitions in Receiver are unguarded, with conditional (current==stateNumber) used to select the completion transitions leaving from source state stateNumber. Non-deterministic choice is applied if more then one transition is available. Note that the translation of actions

```
/* Class number 1 */
proctype Receiver(byte pNum) { atomic {
byte value,peer;
byte current = 1;
byte trigger,receiver,sender;
bool msgUnread=false;
byte transition=0;
byte prm0;

printf("<t start ... > \n",pNum,current);
/* Synchronise */
numStarted++;
(numStarted >= 2); // blocking
checkFlag=false;
} // end atomic
LMAIN1:
atomic {
msgUnread=false;
<CompletionLoop>
goto LMAIN1;

LCOMPLETED1:
<ReadMessage>
<ExecuteTriggered>
goto LMAIN1;
LFINAL1: skip;
} }
```

```
<CompletionLoop>
if
:: (current == 1) ->
  if
  :: (true) ->
    peer = oSender;
    value = Receiver_number;
    current=2;
    transition=8;
  fi
  :: (current == 3) ->
  if
  :: (true) ->   /* call peer.receiveValue(value); */
    trigger = A0;   /* receiveValue symbol A0 */
    sender = pNum;
    receiver = peer;
    prm0 = value;
    outQ[pNum]!trigger(sender,receiver,prm0);
    printf("<t send ... );
    current=0;
    transition=10;
  fi
:: else -> goto LCOMPLETED1;
fi;
printf("<t trans ... );
```

Figure 6. Receiver class Promela code

is almost straightforward with the exception of the call statement that uses the special syntax of channel writing. <ReadMessage> implements message reading and parameter assignment. In our example, the code generated is:

```
if
:: (current == 0) -> goto LFINAL1;
:: else -> skip;
fi;
inQ[pNum]?trigger(sender,receiver,prm0);
if
:: (trigger == A1) -> { /* No arguments */
  printf("<t recv ... > \n",sender,receiver);
...
fi;
```

The code first checks if the final state has been reached. If not, it reads on its input channel and blocks until a message arrives. When the message arrives, the process unblocks and assigns, depending on the value of the trigger (operation), the actual parameters to the corresponding formals parameters. <ExecuteTriggered> implements the execution of triggered transitions. It's implemented as a conditional which branches depending on the value of the current state and trigger. The code generated for Receiver is:

```
if
:: (current == 2) ->
  if
  :: (trigger == A1) ->
    current=3;
    transition=9;
  :: else -> msgUnread=true;
    transition=1;
  fi
fi;
```

The code above corresponds to the execution of transition R1→R2. The else branch flags the msgUnread variable, indicating that the message was dropped, and executes the special transition designated to indicate such cases.

Transition execution, effects included, must be performed atomically. This is achieved by inserting the special Promela atomic statement around the loops that implement the state machine. This is of particular importance because it defines the places where verification takes place; model checking is performed (never automata) after the execution of every atomic statement. Atomicity is broken when a process blocks (message waiting) or when the code jumps out of the scope of the atomic region. We use this fact and introduce back jumps to LMAIN1 outside the atomic area in order to make sure that checks are performed by the verifier after the execution of each transition.

3.4 Translating properties into Spin LTL

Properties written in the property language defined in Section B must be translated into the LTL version used by Spin. The property language is more than just a syntax-

friendly version of LTL; it introduces new predicates and variables that deal with UML elements and, furthermore, addresses these elements by the names defined in the class and state chart diagrams. Model checking is performed by Spin on generated Promela models that work at a much lower level of abstraction which, in turn, are checked against never automata the latter can be generated from Spin LTL formulas. The translation must bridge the gap between both representations. For example, given the formula

```
always state(oReceiver,R1) implies
  eventually trans(oReceiver,final)
```

the SVT resolves all internal references, e.g. oReceiver and R1 are mapped to the corresponding object and state, and generates two outputs: a set of definitions and the Spin LTL version of the formula. We get:

```
#define pp0 (Receiver:current==2)
#define pp1 (Receiver:transition==10)
!([]((pp0) -> (<>(pp1))))   // Spin LTL
```

Spin LTL formulas can only deal with boolean variables e.g. predicates like (x > 2) are not valid. Therefore, the transformation has to generate special #define declarations that name all boolean expressions and plug the new variables inside the generated Spin formula. In the example above, pp0 and pp1 encode predicates state and trans, respectively. Note that operators are translated into their LTL counterparts e.g. [], <>, U; and formulas are written using the variables and values used by the generated Promela model e.g. state R1 is denoted by 2. Note that we actually get the negation of the initial formula. This is because the verifier looks for places where the property does not hold and, therefore, requires the negated condition as input. Similarly, the translator takes as input the property

```
(always (state(oReceiver,R1)
  implies (oSender.result > 100))
```

and generates:

```
#define pp0 (Receiver:current==2)
#define pp1 (Sender:result > 100)
!([]((pp0) -> (pp1)))   // Spin LTL
```

The generated formula is then transformed into a Büchi automata (never clause) using one of SPIN utilities and, both definitions and never clause, are appended to the model to form the never file.

3.5 Result Visualisation

The output generated by the Spin model checker is of little use for the user since it is far removed from the level of abstraction used to specify the system: UML models. Even the simulation runs, displayed as a list of executed source code lines, are difficult to decipher. This is expected; the Spin model is the result of a translation from UML models to Promela and, consequently, the results need to be translated back to UML notation.

We tackle this problem by generating SVT specific messages interweaved with Spins usual. The translated Promela model contains a series of printf statements (see Section 3.4) that, when executed, generate a separate trace specifically designed to show the execution of state machine transitions and message passing. The output of these statements is mixed with other Spin messages that have to be filtered out. Once this is done we get a sequence of tagged messages that are then processed by the SVT. A simulation run of our example will generate, after filtering out unrelated Spin output, the following text sequence:

```
<t start c=Receiver o=1 s=1 >
<t start c=Sender o=0 s=0 >
<t trans c=Sender o=0 s=1 >
<t send s=0 r=1 m="getValue()" >
<t trans c=Sender o=0 s=2 > ...
```

Clearly, most of the information shown above relates to the internal representation of the UML model used by the Promela model. A second phase is required, namely, all Promela constants have to be translated back into UML names. For example, state 2 in Sender must be transformed back into R1. The resulting trace, with all constants mapped back into UML names, uses the following format:

```
START o:Class state=S     TRANS o:Class S1->S2
OUT: o1->o2 m(args)       IN: o2<-o1 m(args)
```

START indicates that execution of object o has started in state S. TRANS indicates that the transition that goes from state S1 to state S2 in object o has been executed. OUT and IN indicate that message m(args) has been sent from object o1 to object o2, and received by o2, respectively. A simulation trace for our example is shown in Figure 7-1.

Verification counterexamples are also displayed as execution traces. The result of checking

```
always (state(oReceiver,R1)
  implies (oSender.result > 100)
```

against our sample model is shown in Figure 7-2. The output basically says that verification stopped right after transition R0->R1 was executed on oReceiver. This is the first time oReceiver gets to state R1 and the first time the comparison is made. The check fails and the verification process stops.

4 Related and Future Work

The need to develop a more precise specification of UML has been a concern [3] since its inception and adoption as standard notation for object-oriented analysis and design by the Object Management Group (OMG). As a result, several formalizations have been proposed to model the behavioral part of UML and, in particular, the formal specification of the semantics of statechart diagrams [11, 17, 7]. Most of the work on formalization of UML state machines has been in the context of automated formal verification of systems and,

```
Trace 1:
START oSender:Sender state=S0
 START oReceiver:Receiver state=R0
TRANS oSender:Sender S0->S1
 TRANS oReceiver:Receiver R0->R1
OUT:oSender->oReceiver "getValue()"
TRANS oSender:Sender S1->S2
 IN:oReceiver¡-oSender "getValue()"
 TRANS oReceiver:Receiver R1->R2
 OUT:oReceiver->oSender "receiveValue(2)"
 TRANS oReceiver:Receiver R2->Rf
IN:oSender<-oReceiver "receiveValue(2)"
TRANS oSender:Sender S2->S3
TRANS oSender:Sender S3->S4
TRANS oSender:Sender S4->Sf
-------------------------------------------
Trace 2:
 START oReceiver:Receiver state=R0
START oSender:Sender state=S0
 TRANS oReceiver:Receiver R0->R1
```

Figure 7. Sample Traces

in particular, model checking [8, 12, 14, 9]. A good number of these specifications have been used as input for Spin translations. The automatic verification of UMLsec models, described in [6, 7], is the most thorough work on model checking security requirements of UML models using Spin. The Spin translation used in this paper resembles the one defined in [6]; we both deal with non-hierarchical state machines, completion and triggered transitions are treated separately (in the main loop), and we both define a separate process for message exchange. The latter is used to model intruders in conjunction with a cryptographic action language. Our main contribution with respect to the UMLsec translation is that ours is fine-tuned for model checking of properties that reason about UML elements. Jussila et.al. [8], like us, model UML classes as processes and define a separate action language but very little effort is put on the verification of properties besides the basic checks performed by Spin. The Hugo project [14] uses a Spin translation to verify collaboration diagrams against UML state machines but unfortunately no further work is done with other types of specification, such as LTL. Gnesi et.al. [4] define a logic based on μ-ACTL, a temporal logic similar in essence to our property language. However, our language captures more UML features and presents them to the system developer an a more user-friendly way. The work presented in this paper sets up the basis for future work in exciting research areas. Our main goals include the extension of UML models to include features such as synchronous message passing and hierarchical state machines, and to further develop the property language in conjunction with alternative specification techniques. In particular, we are interested in the development of a graphical property editor, the application of templates to property specification [1] and the inclusion of UML stereotypes for the specification of security requirements using the UMLsec [6] profile, and SecureUML [16] for access control policies. The latter includes the modeling of intruders and role-based access.

References

[1] M. Dwyer, G. Avrunim, and J. Corbett. Patterns in property specifications for finite state verification. In *International Conference on Software Engineering (ICSE'99)*, 1999.

[2] E. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. MIT Press, 1990.

[3] A. Evans et al. Making UML precise. In *Formalizing UML. Why? How, OOPSLA'98 Workshop Proceedings*, 1998.

[4] S. Gnesi and F. Mazzanti. On the fly model checking of communicating UML state machines. In *ACIS*. IEEE, 2004.

[5] G. Holzman. *The Spin model checker: primer and reference manual*. Addison-Wesley, 2003.

[6] J. Jürgens. *Secure Systems Development with UML*. Springer, 2004.

[7] J. Jürgens and P. Shabalin. Automatic verification of umlsec models for security requirements. In *SAC'02*. ACM, 2002.

[8] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres. Model checking dynamic and hierarchical uml state machines. In *MoDeV^{2}a*, 2006.

[9] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computation*, 11(6), 1999.

[10] Object Management Group. http://www.uml.org.

[11] I. Paltor and J. Lilius. Formalising UML state machines for model checking. In *UML 1999*. LNCS 1723, Springer, 1999.

[12] I. Paltor and J. Lilius. vUML: A tool for verifying UML models. In *ASE'99*. IEEE, 1999.

[13] Papyrus UML. http://www.papyrusuml.org.

[14] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In *Workshop on Software Model Checking*. ENTCS, 55(3), 2001.

[15] F. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1), 2000.

[16] L. Torsten, D. Basin, and J. Doser. SecureUML: A UML-based modellng language for model-driven security. In *UML'02*. Springer-Verlag, 2002.

[17] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and System Modeling*, 1(2), 2002.

[18] A. Zisman. A static verification framework for secure peer-to-peer applications. In *ICIW07*. IEEE, 2007.