STUDIA INFORMATICA
Nr 1(8) Systemy i technologie informacyjne 2007

UML Verification with Verics

Artur Niewiadomski[1*], Wojciech Penczek[1,2*]
[1] Institute of Computer Science,
University of Podlasie, ul. Sienkiewicza 51,
08-110 Siedlce, Poland
[2] Institute of Computer Science,
Polish Academy of Science, ul. Ordona 21,
01-237 Warsaw, Poland

Abstract. We show how to verify UML specifications against properties expressed by CTL-like formulas using the symbolic model checker Verics. Our method is illustrated with an example showing a verification of Alternating Bit Protocol.

Keywords: Model checking, UML, Verics.

1 Introduction

Object-oriented techniques are commonly used during a software design process and programming. Today it would be hard to imagine how to develop large and complex systems without these techniques. Unified Modeling Language (*UML*) [1] is one of the most popular object-oriented specification languages. UML has become the common language of almost all IT projects as it improves the communication between the developer team members.

UML allows to present the system designed from different points of view at any level of abstraction: from a high level concept to a detailed description of the actions together with their parameters and types. On the other hand an application of formal techniques allowing for verification of crucial properties of the system at early design stages can give great benefits. It can save a lot of time, money, and hard work wasted on searching and correcting errors occurring during an implementation and testing of the system.

In order to automatically verify an UML specification we need a verification tool that accepts UML as an input. However, most of the verification tools accept low level specifications, e.g. in a form of labelled transition systems. So,

[*] The authors acknowledge support from Ministry of Science and Education (grant 3T11C01128)

a translation form a high-level UML specification to a language accepted by a verification tool seems to be a practical solution.

Our idea is to exploit the tool Verics to verification of UML specifications. Verics is the model checker for real-time and multi-agent systems developed at ICS PAS. It accepts systems defined in terms of a network of timed automata, but is extended with some translators that allow to describe verified systems also in higher level languages, e.g. as Estelle specifications. The interested reader is referred to [2, 3] for more details. Currently, a development of Verics proceeds in several directions. One of them is to build a module to verify systems described in commonly used high level languages such as UML or Java.

In this paper we describe the main concepts of our translation of UML specifications and show how Verics can be used to perform the verification process. The method is illustrated by the example of Alternating Bit Protocol.

The results reported in the paper are only preliminary. In particular, we impose presently several important restrictions on the input language. However, most of the restrictions can be relaxed and hence we plan to extend and improve our results in the near future.

The rest of the paper is structured as follows: in the two following sections we discuss some related work and present briefly the main concepts of UML. Then, the internal language of Verics (Intermediate Language, IL in short) is introduced to give a basis for a description of our UML to IL translation. Finally, a case study is presented and the methodology of verification is exemplified.

2 Related work

The formal verification of systems specified in UML is not a completely new research topic. Since a few years it has been a field of an intensive research. There are a lot of papers and tools dealing with verification of UML, but due to the limited space in this paper it is impossible even to mention all of them. Therefore, we refer to a few related papers and tools only.

All of the approaches below, similarly to ours, make use of the existing model checking environments and perform a translation of UML specification into their input languages.

IF [4] is one of the existing environments that allows for an UML verification. UML diagrams are translated into the internal language of IF and then model checking, simulation, and static analysis tools can be applied.

HUGO/RT [5] is the translator of UML models with time annotations into the internal language of the model checker UPPAAL. A system is defined by a Class Diagram and a set of State Machine Diagrams and the properties tested are described by Collaboration Diagrams.

The authors of [6] report as a case study an attempt to model check the control subsystem of an operational NASA robotics system. It is specified in xUML

[7] – the executable subset of UML. The environment ObjectCheck and the model checker COSPAN is exploited here.

These three approaches, similarly to ours, exploit UML statemachines for specifying a behaviour of a system. Otherwise than *TURTLE* approach [8] which is a special extension of UML (*a profile*) aimed at modelling and formal validation of real-time systems. Here the Activity Diagrams are used to specify the dynamics of a system. There is also a special tool *Ttool* that allows for an edition of UML diagrams using TURTLE profile. Verification and simulation can be performed using *RT-LOTOS* environment.

All of the mentioned approaches, except xUML, make use of the synchronous communication. In our work we use the asynchronous communication via FIFO buffers only, similar to xUML.

3 Introduction to UML

The Unified Modeling Language is a specification and object modelling language widely used in software engineering. It inroduces a standardised graphical notation that allows to specify, visualise, construct and document software systems. Today, when applications are too complex for one person to encircle, whole groups of specialists work on software design processes. Unified Modeling Language has become essential for all the participants of IT projects, including analytics, system designers and programmers. Thanks to UML diagrams that are their common language, the whole team can communicate simply and easily and thus cooperate effectively.

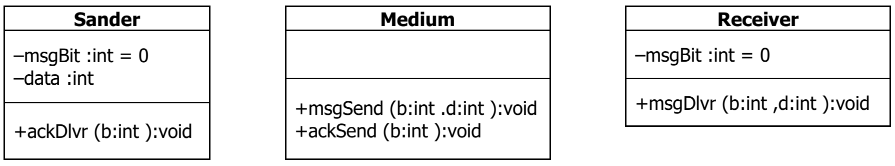

Figure 1. Altermatomg Bit Protocol Class Diagram

UML can be used not only for modelling software systems, bul also for modelling hardware, business processes, organisational structure, and many other fields. However, an imprecise semantics of the language is a serious problem, especially when one wants to get on a „firmal ground”. In this paper we restrict UML to a subset, whose semantics is mostly fixed.

UML in the current version (2.0) consistes of 13 types of diagrams. All of them can be divided into two groups:

- Diagrams modelling the static structure, and
- Diagrams modelling the dynamis behaviour of the system.

In our approach three kinds of UML diagrams are under consideration: Class Diagrams, Object Diagrams, and State Machine Diagrams (Statechart Diagrams).

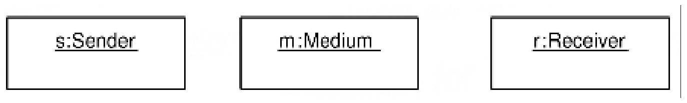

Figure 2. Alternating Bit Protocol Object Diagram

Class Diagrams (see Fig. 1) describe static properties of the systems modeled. We restrict them to the following three elements: Classes, Attributes, and Operations (Methods). A class is a description of a set of objects that consist of the same attributes, operations, associations, and the meaning. An attribute is a named property of a class. We restrict types of attributes to integer, boolean, and defined classes only. An operation is a definition of an activity that can be executed by some instance of the class (object). A list of arguments (with its types) can be associated with an operation.

Object Diagrams (see Fig. 2) are used to define the types and the number of the objects the system consists of and also model the static aspects of the system. Objects are instances of classes that are defined in a class diagram.

State Machine Diagrams (see Fig. 3) are used to specify behaviour of objects, so to model the dynamics of the system. UML State Machine Diagram depicts the various states that an object may be in and the transitions between these states. The transitions are fired in answer to external and internal events such as time events or method calls. In our work to model the time flow we use the timed event *after(t)* which occurs after *t* time units has elapsed since the state with outgoing timed transition is reached.

In this paper we consider the following elements of State Diagrams:

- *States*. We distinguish simple states, complex states, and pseudo-states. A pseudo-state can be initial-, final-, or choice-state. A choice-state is a kind of an intermediate state which must be left immediately after it is entered.
- *Activities*. Entry activities are executed immediately after the state has been entered, whereas exit activities are executed just before the state is left.
- *Transitions*. Transitions describe possible changes of states. Each transition connects a source state with a target state. Transitions are equipped with three arguments, where each of them can be empty:
  - triggered event - the name of an event that enables the transition,
  - guard - a boolean expression. If it is true, then the transition can be fired,
  - list of actions - a sequence of actions that are executed while the transition is fired.

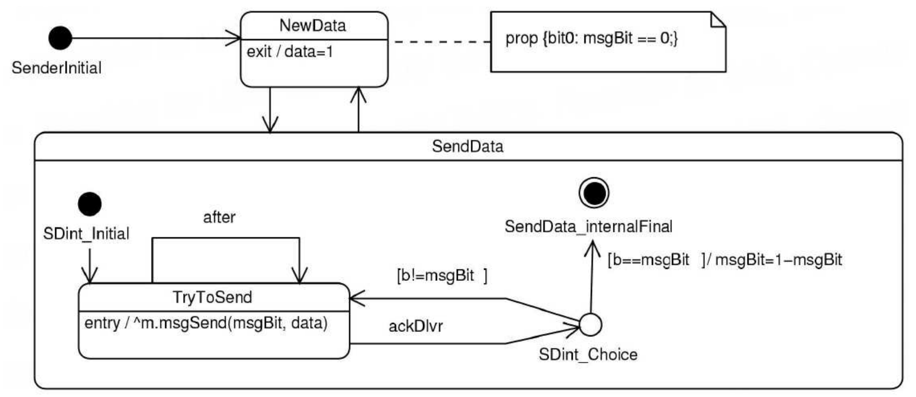

Figure 3. Alternating Bit Protocol Sender State Machine Diagram

4 The verification system Verics and Intermediate Language

Verics is a model checker for untimed, timed, and multi-agent systems. It offers three complementary methods of model checking: SAT-based Bounded Model Checking (BMC), SAT-based Unbounded Model Checking (UMC), and an on-the-fly verification while constructing abstract models of systems. The theoretical background of its implementation has been presented in several papers [9, 10, 11]. The systems to be verified can be specified in a subset of Estelle, in the internal Verics language - Intermediate Language (IL for short), or directly as networks of timed automata.

The properties tested are given as formulas of a branching-time temporal logic, based on CTL. The formulas are built over *propositional variables*, defined directly in the specification of a verified system.

In the sequel we describe a translation from UML into IL. An Intermediate Language program is a parallel composition of sequential communicating processes, where each process is specified in the terms of states and transitions. Communication can be realised trough an asynchronous interchange of messages - via bounded or unbounded buffers - or via shared global variables. An IL program is structured as follows: a program name, a set of declarations, definitions of processes, and a main part. In the declaration section all constants, global variables, and buffers must be defined. A variable definition is a pair consisting of a name and of a type, optionally followed by an assignment of an initial value. Only boolean and integer types of variables are allowed.

A buffer is a FIFO queue and has to be given a name and a type. Optionally a capacity can be defined if we want to get a bounded buffer. Buffers are accessible for all processes, similarly to global variables.

A section of a process definition consists of its name, a set of states, local variables definitions (if needed), an initialisation section, and definitions of transitions. Optionally it can be followed by local properties definitions that are later used to construct temporal formulas describing the tested properties of a system.

```
uml_model {
buffer int Medium_msgSend[1] size = 3;
buffer int Sender_ackDlvr[1] size = 2;
buffer int Receiver_msgDlvr[1] size = 3;
buffer int Medium_ackSend[1] size = 2;

process Sender_0 {
 stateset State_5, State_1, State_2, State_4, State_7;
 var int __il_objNum, __il_Par, data, msgBit, b;
 to State_1: { __il_objNum=0; msgBit=0; }
...
  from State_5 to State_7:
    when (b!=msgBit)
    delay [0,0]:urgent /* leave choice point immediately */
      { put(Medium_msgSend[0],0);put(Medium_msgSend[0],msgBit);
        put(Medium_msgSend[0],data); }
...
 prop { bit0: msgBit == 0; in_State_2: in State_2; }
}
...
process Receiver_0 {
 stateset State_15, State_12, State_14, State_13;
 var int __il_objNum, __il_Par, d, msgBit, b;
 to State_12: { __il_objNum=0; msgBit=0; }
...
  from State_15 to State_14:
      { put(Medium_ackSend[0],0); put(Medium_ackSend[0],b); }
...
 prop { bit0: msgBit == 0; in_State_14: in State_14; }
}
...
main { Receiver_0 || Sender_0 || Medium_0 }
prop { Sender_0.bit0; Sender_0.in_State_2;
       Receiver_0.bit0; Receiver_0.in_State_14; }
}
```

Figure 4. Sample of a specification of Alternating Bit Protocol in Intermediate Language

In the main part the instances of processes are created. Actual parameters are assigned to formal parameters in processes definitions, if they have any. Then, the properties definitions are given. The program properties are either boolean expressions over global variables and buffers or local properties of processes. A property is visible only if it is declared in the main section. A sample of a specification of Alternating Bit Protocol in IL is shown in Fig. 4.

5 Translation from UML to Intermediate Language

In this section we give the main ideas behind our translation from UML to IL.

Objects and classes. An Object Diagram specifies a list of objects that a system consits of. Each of objects must be an instance of some class defined in a Class Diagram. Objects are mapped onto processes of Intermediate Language and the number of UML objects corresponds to the number of IL processes.

Attributes. The attributes of objects are translated into process variables. The types allowed are boolean, integer, and object types. The object types are mapped onto integers - each object is equipped with its own unique number, so the communication between objects is possible.

Methods. The methods are translated into arrays of buffers according to the following rules:

- each method of each class is mapped onto one array of buffers,
- the size of an array corresponds to the number of objects of a given class,
- an array index (a single buffer) represents the method offered by a concrete object.

A method call is realized by placing a special element - *call marker* - in the corresponding buffer. If the method called requires some parameters, then they are allocated after the call marker.

States. Each of UML simple- and pseudo-states is mapped onto a state of an IL process. Entry and exit activities are merged with actions of incoming and outgoing transitions in such a way that the exit action (of the source state) is followed by the proper transition action and the latter is then followed by the entry action (of the target state).

Transitions. The transitions in State Diagrams are translated directly into transitions of Intermediate Language processes. A triggered event, a guard, and a sequence of actions can be associated with the transition.

Events. The time events in UML are translated into time constraints of Intermediate Language transitions, using *delay* construction. The latter allows to specify the amount of time that may elapse before certain actions take place. A method call event is mapped onto a guard of an IL transition which is equipped with a set of *get* statements. The role of these statements is to get a method call marker (and optionally its parameters) from the buffer that corresponds to the called method and the referred object. If suitable values exist in the buffer, then they are taken away and the transition is executed. The parameters of the method called are added to the set of local variables of the process.

Guards. The guards in UML are formed using attributes of objects and parameters of the actions called. These expressions are directly transformed into IL guards, using the variables that correspond to UML attributes and parameters.

Actions. In our approach we restrict the actions related to UML transitions to method calls and assignment statements only. The assignments are translated straightforwardly, while the method calls are realised via inserting a call marker (and optionally parameters) into the adequate buffer.

Entering complex states. If a transition executed leads to a complex state, then the enter actions of this state are added to the transition. If a transition does not lead to any substate of a complex state, then it is understood that it leads to the initial state of the complex state.

Exiting complex states. When a transition leaves some complex state, then the exit actions of this state are added to the transition. In the case when some complex state is a source state for a transition, then this is translated into a set of transitions outgoing all the sub-states of this complex state. Exceptions are the special *terminate transitions*.

Terminate transitions. A terminate transition is a special kind of transitions, outgoing a complex-state and carrying no label, guard, nor event condition. The terminate transition becomes enabled when its source state has finished its task. The terminate transitions are translated into IL transitions outgoing the states that emerged from UML final states.

Final states. A final state is a kind of a pseudostate that has no outgoing transitions. They are translated into special simple states which have to be left immediately. In case that the parent state of the final state is not the top state[1] , then the terminate transitions are enabled. If the parent of the final state is the top state, the entire statechart terminates[2] .

Transitions priorities. According to the semantics of UML state-machines, the transition outgoing a substate has a higher priority than a transition outgoing a parent state. Our translation respects this rule by a special preparation of the guards of the transitions outgoing complex states: this guard is a conjunction of a guard specified in the UML state-machine and the negation of all the guards of all transitions outgoing the substates of this composite state.

Run-To-Completion (RTC). The RTC step is the period of time in which events are accepted and acted upon. Processing an event always completes within a single model step, including exiting the source state, executing any associated actions, and entering the target state. RTC rule is preserved by our translation because communication between processes is asynchronous, each UML transition is translated into one IL transition, and IL transitions are atomic.

Propositional variables. The set of propositional variables can be specified directly in UML state-machine diagrams. To this aim we use a special *note* element that contains a specification of a propositional variable (see Fig. 3 and 5).

[1] The top state represents entire statemachine and all the states in the statemachine are the children of the top state. The top state can not be a source or a target state of any transition.
[2] In this case the corresponding IL process stays forever in its final state.

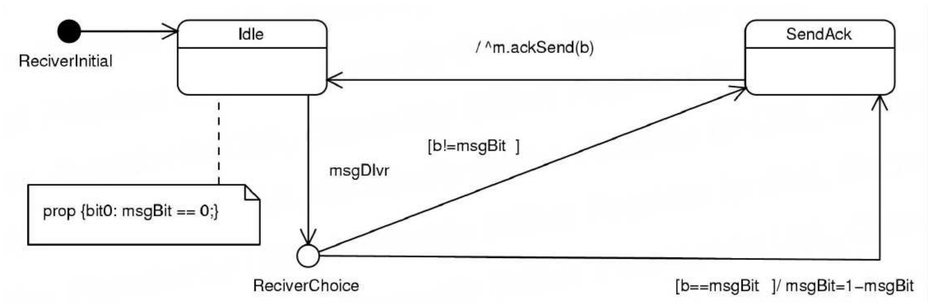

Figure 5. Receiver State Machine Diagram

The specification begins with the keyword *prop*, and then (in the braces) we provide the name of a propositional variable and (after a colon) the boolean expression. In this case, after the translation to IL, we obtain one IL propositional variable. Its value in a state is the same as the value of the boolean expression.

If we need a propositional variable that is true only if the system is in some indicated state, we have to create an UML note element (containing *prop* keyword and the name of the variable, but without the boolean expression) and connect it with appropriate simple or complex state. Then, after the translation to IL, we obtain one IL propositional variable that is true if and only if the system is in one of the IL states that have emerged from the indicated UML state.

Moreover, we can connect an UML note element containing a boolean expression to some UML simple or complex state if we need a propositional variable that is true when the system is in the indicated state and the boolean expression is true. Then, after the translation to IL, we obtain a pair of IL propositional variables and we use the conjunction of them.

Assumptions and Restrictions. This paper reports on our preliminary attempt to verify UML models with Verics. So, it is clear that we have restricted the input language to only a few of the most relevant types of diagrams. For simplicity, we have not allowed for certain features of the diagrams considered in the translation. Below we list the features that are presently ignored:
The elements of Class Diagrams:

- the connections between classes, including inheritance,
- the static attributes - the attributes common for all the objects of the same class,
- the values returned by the methods,
- the visibility of attributes - we treat all attributes as private.

The elements of State Machine Diagrams:

- the concurrent states - in most cases they can be replaced by a set of objects running concurrently,
- the history states,
- the actions different than method calls and assignments.

In our approach a method call is interpreted as an asynchronous event, and connections between objects in Object Diagrams are ignored.

6 Alternating Bit Protocol

Alternating Bit Protocol (ABP) is a simple network protocol that provides a reliable communication via an unreliable medium due to a retransmission of messages. Below we describe our variant of ABP.

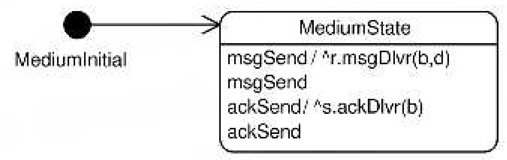

Figure 6. Lossy Medium State Machine Diagram

In our model the three objects are used: Sender, Receiver, and Lossy Medium, as depicted in Fig. 1 and 2. The messages are sent from Sender *s* to Receiver *r* via Lossy Medium *m*. Each message contains a data part and a one-bit sequence number, i.e., a value that is 0 or 1. Sender *s* sends a message continuously with the same sequence number, until he receives an acknowledgement (ACK) from *r* that contains the same sequence number. When this happens, *s* flips the sequence number and starts transmitting the next message. Otherwise, the message is retransmitted by *s* after some period of time or after receiving from *r* the acknowledgement with the opposite value of the bit. When *r* receives a message from *s*, it sends back an ACK with the received sequence number[3]. The Lossy Medium (see Fig. 6) sends or drops messages and ACKs in a non-deterministic way.

7 Verification of the Alternating Bit Protocol

This section illustrates how Verics can be used to verify a system specified in UML. Because of the space limit, we show the verification against one property only so we focus on demonstrating the methodology rather than on the complete verification of the Alternating Bit Protocol.

The first stage of UML verification with Verics consists in the translation of an UML specification into Intermediate Language. Next, the IL specification is

[3] In our example we omit a consumer of data and simplify the producer – data is an integer number with the fixed value 1.

translated into a network of timed automata[4]. The detailed information on this translation can be found in [9].

Before starting the verification, we have to first construct a temporal formula describing the tested property of a system using propositional variables defined in the system specification. In this case, we want to verify the following property: *Always when the Receiver sends ACK and its internal bit is set to 1,and the Sender receives the ACK, then the Sender's internal bit is set to 0*. This property is obviously not true, because following the protocol after the message and an acknowledgement have been received, both (Sender's and Receiver's) internal bits are equal.

To capture this property as a temporal formula we need to define some propositional variables first. In UML state-machine diagrams (see Fig. 3 and Fig. 5) two special note elements are placed. They define two variables named *bit0*, one for the *Sender* and the other for *Receiver* object. Both the notes are connected with the states, in which these variables have the value *true*. After the translation of the UML specification into IL, we obtain two pairs of global propositional variables: *Sender_0.in0*, *Sender_0.in_State_2*, and *Receiver_0.bit0*, *Receiver_0.in_State_14* (see Fig. 4). The variable *Sender_0.bit0* has the value *true* always when the Sender's internal bit is equal to 0. Moreover, the variable *Sender_0.in_State_2* is true iff the Sender process is in the IL state that corresponds to the UML state connected with the note element.

After the translation of the IL specification into the network of timed automata the propositional variables are relocated to the appropriate automata. Their names are similar to the IL propositional variables. While constructing the property tested we have to use the propositional variables defined on the timed automata level. The resulting formula is given as Formula 1.

$$AG\big((\mathrm{Re}ceiver\_0\_in\_State\_14 \wedge \neg \mathrm{Re}ceiver\_0\_bit\_0 \wedge Sender\_0\_in\_State\_2) \Rightarrow (Sender\_0\_bit\_0)\big)$$

Formula 1. Always when the Receiver sends ACK and its internal bit is set to 1,and the Sender receives the ACK, then the Sender's internal bit is set to 0.

Now, we should choose the verification method. At the moment the most effective is BMC that encodes a tested formula and the model of the system (unfolded to the given depth) as a boolean formula. This formula is satisfiable iff the property is true in the model. We check the satisfiability of this formula using *zChaff* [12] SAT-solver. If it is not satisfiable, then we increase the depth of the model and run the BMC and zChaff again. The BMC method requires a formula in the existential form, so applying the negation to Formula 1 we obtain Formula 2.

[4] It is possible to obtain also a global (product) automaton.

$$EF(\mathrm{Re}ceiver\_0\_in\_State\_14 \wedge \neg \mathrm{Re}ceiver\_0\_bit\_0 \wedge Sender\_0\_in\_State\_2 \wedge \neg Sender\_0\_bit\_0)$$

Formula 2. The negation of Formula 1.

Table 1. Experimental results for the Formula 2.

Depth	BMC time[s]	zChaff time[s]	Clauses	Literals	Result
1	0.27	0.01	44098	106978	unsat
2	0.54	0.11	86723	210683	unsat
3	0.85	0.20	129348	314388	unsat
4	1.10	0.65	171973	418093	unsat
5	1.44	0.40	214598	521798	unsat
6	1.70	1.46	257223	625503	unsat
7	1.96	1.57	299848	729208	unsat
8	2.26	2.40	342473	832913	unsat
9	2.55	3.17	385098	936618	unsat
10	2.84	7.22	427723	1040323	unsat
11	3.19	6.05	470348	1144028	unsat
12	3.46	6.24	512973	1247733	unsat
13	**3.73**	**5.74**	**555598**	**1351438**	**SAT**

The experiments shows that Formula 2 is satisfiable in the model on depth 13. This means that the property expressed by the Formula 1 is not true in the verified system. In the Table 1 the detailed information on verification of Formula 2 is presented. The tests have run on Pentium M 1.73GHz with 512MB RAM running under Linux 2.6.12-9. The total time of all necessary translations is about two seconds.

8 Conclusions and future work

The paper reports on the preliminary results of our work dealing with automatic verification of UML using Verics. The translator of UML models has been implemented that handles (with some restrictions) Class Diagrams, Object Diagrams, and State Machine Diagrams. Moreover, it is possible to specify the propositional variables directly in UML specifications.

We are planning to continue and extend our work. In the near future we will introduce the synchronous communication methods, extend the translated subset of UML with new diagrams, and relax some of the current restrictions. This will allow to model systems in a more natural way. Moreover, we plan to provide a possibility to specify tested properties in a graphic form. This should enable the verification of UML systems by users having no knowledge about temporal logic, Intermediate Language, and timed automata.

References

1. OMG: *Unified Modeling Language (UML)*, version 2.0. http://www.omg.org/technology/documents/formal/uml.htm (2005).
2. Nabiałek, W., Niewiadomski, A., Penczek, W., Półrola, A., Szreter, M.: *VerICS 2004: A model checker for real time and multi-agent systems*. In: Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'04). Volume 170(1) of Informatik-Berichte., Humboldt University (2004) 88-99.
3. Dembiński, P., Janowska, A., Janowski, P., Penczek, W., Półrola, A., Szreter, M., Woźna, B., Zbrzezny, A.: *VerICS: A tool for verifying timed automata and Estelle specifications.* In: Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03). Volume 2619 of LNCS., Springer-Verlag (2003) 278-283.
4. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: *The IF toolset.* In Bernardo, M., Corradini, F., eds.: SFM. Volume 3185 of Lecture Notes in Computer Science., Springer (2004) 237-267.
5. Knapp, A., Merz, S., Rauh, C.: *Model checking - timed UML state machines and collaborations*. In: FTRTFT. (2002) 395-416.
6. Sharygina, N., Browne, J., Xie, F., Kurshan, R., Levin, V.: *Lessons learned from model checking a NASA robot controller*. Formal Methods in System Design **25** (2004) 241-270.
7. Starr, L.: *Executable UML: The models that are the code*. In: Model Integration, LLC. (2001).
8. Apvrille, L., Courtiat, J.P., Lohr, C., de Saqui-Sannes, P.: *TURTLE: A real-time UML profile supported by a formal validation toolkit.* In: IEEE Trans. Software Eng **30**(7) (2004) 473-487.
9. Doroś, A., Janowska, A., Janowski, P.: *From specification languages to timed automata.* In: Proc. of CS&P the Int. Workshop on Concurrency, Specification and Programming (CS&P'02). Volume 161(1) of Informatik-Berichte., Humboldt University (2002) 117-128.
10. Penczek, W., Woźna, B., Zbrzezny, A.: *Bounded model checking for the universal fragment of CTL.* Fundamenta Informaticae **51(1-2)** (2002) 135-156.
11. Woźna, B., Penczek, W., Zbrzezny, A.: *Reachability for timed systems based on SAT-solvers.* In: Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'02). Volume 161(2) of Informatik-Berichte., Humboldt University (2002) 380-395.
12. Zhang, L.: *zChaff.* http://www.ee.princeton.edu/~chaff/zchaff.php (2005).