Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 127 (2005) 3–16

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

UML Automatic Verification Tool with Formal Methods

Mª Encarnación Beato[1,2]

Escuela Universitaria de Informática
Universidad Pontificia de Salamanca
Salamanca, Spain

Manuel Barrio-Solórzano [3] Carlos E. Cuesta [4]
Pablo de la Fuente [5]

Facultad de Informática
Universidad de Valladolid
Valladolid, Spain

Abstract

The use of the UML specification language is very widespread due to some of its features. However, the ever more complex systems of today require modeling methods that allow errors to be detected in the initial phases of development. The use of formal methods make such error detection possible but the learning cost is high.
This paper presents a tool which avoids this learning cost, enabling the active behavior of a system expressed in UML to be verified in a completely automatic way by means of formal method techniques. It incorporates an assistant for the verification that acts as a user guide for writing properties so that she/he needs no knowledge of either temporal logic or the form of the specification obtained.

Keywords: Formal methods, automatic verification, UML active behaviour, formal UML verification.

[1] Supported by Junta de Castilla y León (Spain) in the Research Project VA117/03

[2] Email: ebeato@upsa.es

[3] Email: mbarrio@infor.uva.es

[4] Email: cecuesta@infor.uva.es

[5] Email: pfuente@infor.uva.es

doi:10.1016/j.entcs.2004.10.024

1 Introduction

The Unified Modeling Language (UML) [1,12] has unquestionable advantages as a visual modeling technique, and this has meant that its applications have multiplied rapidly since its inception. To the characteristics of UML itself must be added numerous tools that exist in the market to help in its use (Rational Rose, Argo UML, Rhapsody ...). However, unfortunately, none of them guarantee specification correctness.

However, it is widely accepted that error detection in the early phases of development substantially reduces cost and development time, as the errors detected are not transmitted to or amplified in later phases. It would thus be very useful to have a tool that would allow the integration of this semi-formal development method with a formal method to enable system verification. This paper presents a tool —TABU (*Tool for the Active Behaviour of UML*)— to carry out this integration by providing a formal framework in which to verify the UML active behaviour.

The tool uses SMV [10] (*Symbolic Model Verifier*) like formal specification, as it has the adequate characteristics for representing the active behaviour of a specification in UML. The main reason for this is that it is based on labeled transition systems and because it allows the user's own defined data types to be used, thus facilitating the definition of variables. It also uses symbolic model checking for the verification, which means that the test is automatic, always obtains an answer and more importantly, should the property not be satisfied generates a means of identifying the originating error.

The tool carries out, with no intervention on the user's part, a complete, automatic transformation of the active behaviour specified in UML into an SMV specification, focusing mainly on reactive systems in which the active behaviour of the classes is represented through state diagrams, while activity diagrams are used to reflect the behaviour of class operations. XMI [11] (*XML Metadata Interchange*) is used as the input format, thus making it independent of the tool used for the system specification.

On the other hand, the tool has a versatile assistant that guides the user in writing properties to be verified using temporal logic. The verification is carried out in such a way that the user needs no knowledge of either formal languages or temporal logic to be able to take advantage of its potential; something which has traditionally been difficult to overcome when deciding on the use of formal methods. In addition, notions of the form of the specification obtained are unnecessary: that is, knowledge of the internal structure of variables or modules obtained is not required for verification. Figure 1 is a graphical representation of the tool's architecture, the engineer only needs

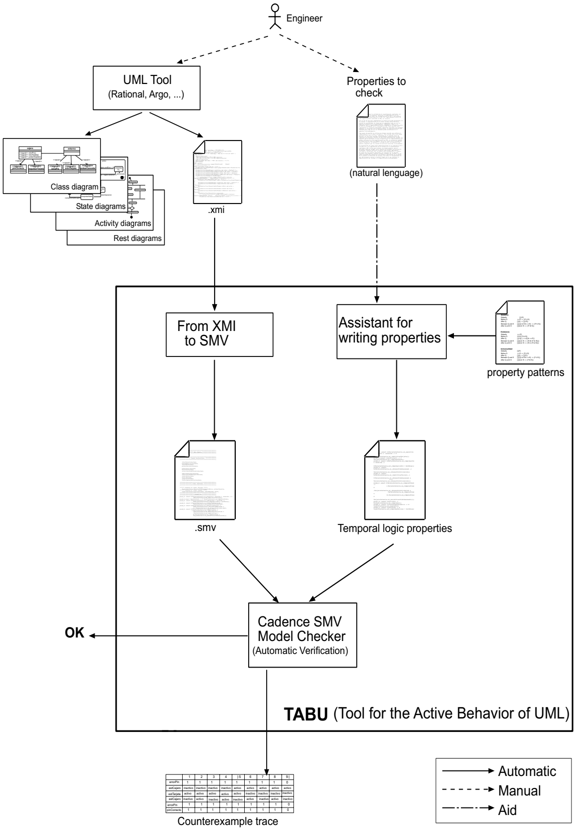


Fig. 1. Tool architecture

knowledge of UML and the system studied, the tool automatically obtains the formal representation in SMV from textual representation in XMI. Parallel, a wizard helps to write properties to be verified using LTL (*Linear Temporal Logic*), moreover if a property is not satisfied, the tool shows a counterexample trace.

The rest of the paper shows the functionalities of the tool illustrated through a case study. It is analysed in terms of two main aspects of the tool: how to obtain a formal specification from the UML diagrams, and how the assistant helps and guides in verifying properties. This is followed by a review of the work in the same field from the literature and, finally, the conclusions are presented along with possible future work.

2 From UML to SMV

The tool input is a UML specification which has been formatted using the XMI exchange syntax. From this input, a SMV specification is automatically generated. Three kinds of diagram are taken into account when transforming the active behaviour from UML into SMV: class, state and activity diagrams. The first provides information concerning the elements that make up the system and their relationships, while the second and third provide information about the behaviour, through time, of each of those elements.

In order to show how the tool works we use the example of an automatic teller machine (figures 2, 3, 4, 5 and 6), both because it is a very well known example, and because it incorporates in its specification most of the existing building blocks of statemachine and activity diagrams.

The following is the basic description of the system. First of all, the user introduces the credit card followed by a pin number. The system checks whether it is correct and, if not it allows the user to try again. If the user introduces three consecutive wrong pin numbers, the card will not be returned to the user. Once the right pin is introduced, the user will be allowed to push the operation button. This operation updates the card information including the available left-over. At any time, the user can push the cancel button that will make the card to be returned and an error signal to be generated.

3 Class diagram

The fundamental concept taken as our starting point is that of the active class. The system is specified in terms of active classes which are associated to the reception of signals. The behaviour of each active class is reflected in a different SMV module, which in turn is instantiated in the main module by each of the class objects.

Each SMV module, representing a class, needs the signals the class receives as its input parameters, and those the class emits as output parameters. Thus, the said signals are reflected in the class diagram using the stereotypes <<send>> and <<signal>> as shown in figure 2. Here, the signals `okPin`, `errorPin` and `updateBalanceCard` correspond to the signals emitted by the `Card` class, while `introPin`, `introCard` and `returnCard` are the ones it receives.

An additional class called `environment` also has to be included. It has no associated behaviour and contains details of the signals produced outside the system and which are input signals.

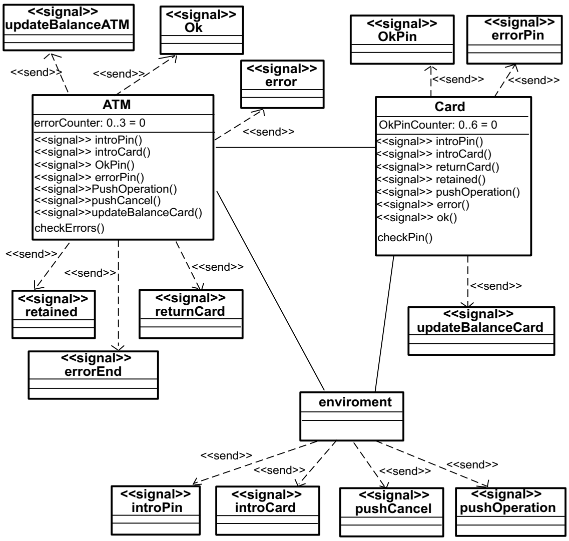


Fig. 2. Class diagram of the ATM system

4 State machines

Behaviour of each of the active objects is reflected through state machine and activity diagrams. To correctly control the evolution of a state machine, the state it is in at any given moment must be known. This is achieved by using a separate variable to store this information for each machine.

In addition, the fact that combined states, both sequential and concurrent, may appear within a machine means that additional variables are needed in order to deal with the submachines. These will be dealt with following the same reasoning as for the main machine, with the exception of the peculiarities they possess with respect to activation and deactivation.

As for the evolution machines, the SMV operator `next` is used. This represents the value taken by the variable in the following step. The state machine is initiated using the `init` operator. As far as the machine for the ATM class is concerned (see Figure 3), the SMV representation of the outermost machine behaviour is as shown below:[6]

```
/***** Statemachine for state: ATM *****/
   st_ATM :{ checkBalance,waitOperation,active,
             inactive,returningCard};
/***** Evolution of statemachine for class: ATM*****/
   init(st_ATM) := inactive;
   next(st_ATM) := case {
```

[6] Each transition is assigned a unique identifier which corresponds to the XMI identifier assigned automatically.

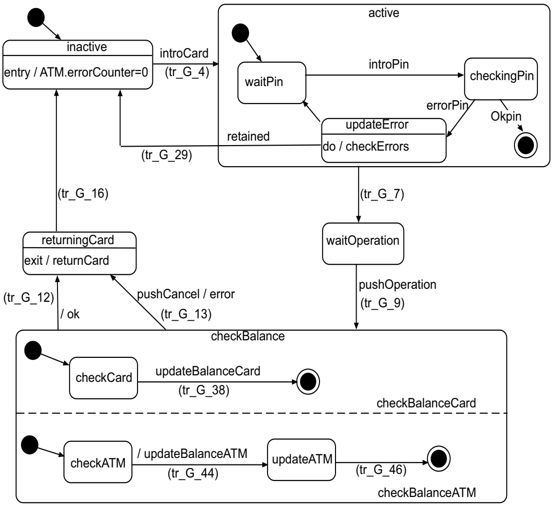


Fig. 3. The state diagram of the ATM class

```
      tr_G_9 : checkBalance;
      tr_G_7 : waitOperation;
      tr_G_4 : active;
      tr_G_29 : inactive;
      tr_G_16 : inactive;
      tr_G_12 : returningCard;
      tr_G_13 : returningCard;
      default : st_ATM;
   };
```

Where `tr_G_9`, `tr_G_7`, `tr_G_4`... represent the firing of transitions `tr_G_9`, `tr_G_7`, `tr_G_4`... The block `default` represents the behaviour where there is no change of state, that is, when no transition present in the machine is fired and it remains in the same state during the following step.

A similar reasoning has been used for the behaviour of the submachines, based on having a different machine for each sequential composite state and for each region of a concurrent composite state. By doing so, the behaviour associated to the concurrent composite state `checkBalance` of Figure 3 is represented in terms of the following machines:

```
/***** Statemachine for state: checkBalanceATM *****/
 st_checkBalanceATM :{updateATM,checkATM,FINAL,DontKnow};

/***** Statemachine for state: checkBalanceCard *****/
 st_checkBalanceCard :{checkCard,FINAL,DontKnow};


/** Evolution statemachine state: checkBalanceATM ***/
   init(st_checkBalanceATM) := DontKnow;
   next(st_checkBalanceATM) := case {
      tr_G_12 : DontKnow;
      tr_G_13 : DontKnow;
      tr_G_9 : checkATM;
      tr_G_44 : updateATM;
      tr_G_46 : FINAL;
      default : st_checkBalanceATM;
   };
/** Evolution statemachine state: checkBalanceCard ***/
   init(st_checkBalanceCard) := DontKnow;
   next(st_checkBalanceCard) := case {
      tr_G_12 : DontKnow;
      tr_G_13 : DontKnow;
      tr_G_9 : checkCard;
      tr_G_38 : FINAL;
      default : st_checkBalanceCard;
   };
```

Where `DontKnow` is the state of a machine which is deactivated. Deactivation can take place either because of the firing of transition `tr_G_13`, the cancel button is pushed, or because both submachines reach the final state and `tr_G_12` is fired by termination. Its syntax is the following:

```
  tr_G_12:=in_checkBalance & in_FINALcheckBalanceATM & in_FINALcheckBalanceCard;
```

5 Actions

The evolution of an active object can lead to different actions, including sending signals and modifying the value of class attributes.

With regard to sending signals, it can happen in any of the following situations: (1) the firing of a transition, if the signal is among the transition effects; (2) the activation of a state, if the signal is among its entry actions; and (3) the deactivation of a state, if the signal is among its exit actions. Taking into account that both state activation and deactivation are due to the firing of some transition, signal evolution can be represented in a similar way to state machine evolution.

As for modifying the value of an attribute that it will be specified through the use of the SMV operators `init` and `next`. Attributes will be initialised with `init` if they have an initial value in the class diagram, whereas their evolution (`next`) will depend on the firing of transitions. For instance, the SMV behaviour for the attribute `errorCounter` in class `ATM`, which keeps track of how many wrong consecutive pin numbers have been introduced, is the following (see Figures 2, 3 and 4).

```
/***** Attribute: errorCounter *****/
   ATM_errorCounter: 0..3;

   init(ATM_errorCounter):=0;
   next(ATM_errorCounter) := case {
      tr_G_55: ATM_errorCounter +1;
      tr_G_29: 0;
      tr_G_16: 0;
```

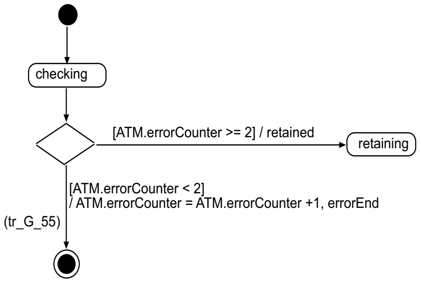

checking
[ATM.errorCounter >= 2] / retained
retaining
[ATM.errorCounter < 2]
/ ATM.errorCounter = ATM.errorCounter +1, errorEnd
(tr_G_55)

Fig. 4. Activity diagram for the activity checkErrors

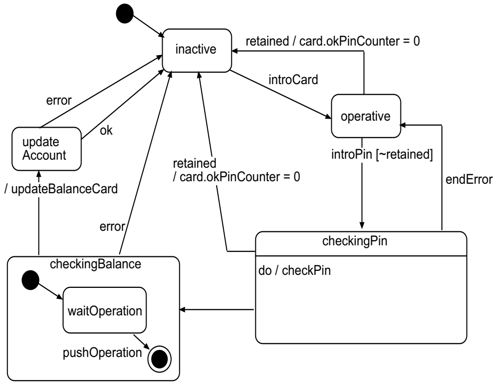


Fig. 5. Statechart of Card class.

```
    default : ATM_errorCounter;
};
```

6 Activity diagrams

A class operation control flow can be modeled using activity diagrams which, fundamentally, show the control flow between activities. Its SMV specification can be found in the module that reflects class behaviour. These activity diagrams are activated whenever a call to an activity is produced within a state using the notation `do` <*activity*>.

Activity diagrams can be considered as a special case of state diagrams where the majority of states are activity states and most transitions are fired by termination. So the mechanism used to represent them is similar to that used for state machines. The only difference is that, for concurrent evolution, the special states of division and union (`fork` and `join`) are used. They are activated whenever any state which has a call to this activity inside it is activated. Likewise, they are deactivated whenever a transition is produced

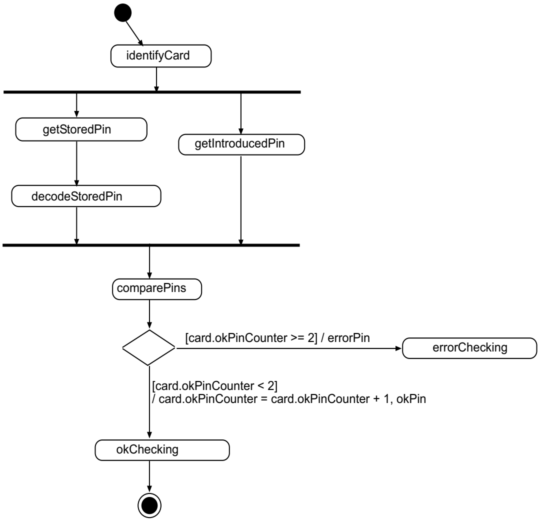


Fig. 6. Activity Diagram of checkPin activity.

that deactivates the state in which it is contained.

7 Verification

Having obtained a system specification in a formal language with a solid mathematical basis means that it is possible to check whether the system complies with certain desirable properties. As with the formal specification methods, the increasing complexity of software systems requires the development of new verification methods and tools to carry it out either automatically or semi-automatically.

In our tool, verification is carried out using the SMV tool model checker. With this, it is possible to make the verification process completely automatic. That is, given a property, a positive or negative reply is always obtained.

The property must be expressed in a temporal logic present in SMV [7] , CTL (*Computation Tree Logic*) or LTL (*Linear Temporal Logic*). This property writing is not a trivial problem. To write them correctly, advanced knowledge of logics and the type of specification obtained from the system is necessary. Our tool overcomes this problem as it has an assistant that guides the user through the writing of properties until the property to be verified is finally obtained following the appropriate syntax.

Our starting point was the pattern classification proposed by Dwyer et al. [4] to which our own cataloguing of the different properties to be automat-

[7] Cadence SMV. http://www-cad.eecs.berkeley.edu/˜kenmcmil/smv

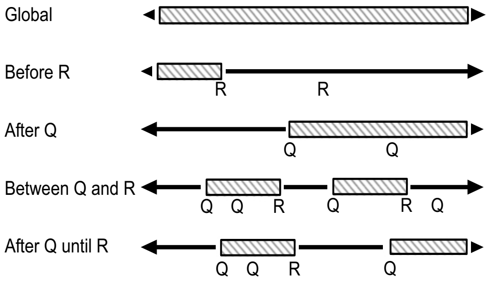


Fig. 7. Specification Scopes.

ically verified has been added.

7.1 Property patterns

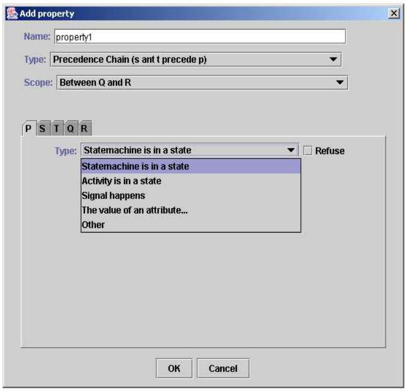
The property writing assistant is based on the pattern scheme proposed by Dwyer et al [4] where it is established a first classification between patterns of occurrence and order. Most of the properties of a system to be verified in practice, fit in with one of these two categories.

Occurrence patterns describe properties with respect to the occurrence of a state or signal during the evolution of a system. These include absence (never), universality (always), existence (sometimes) and bounded existence (appearing a certain number of times). Order patterns establish properties with respect to the order in which they occur. They include: precedence (`s` precedes `p`), response (`s` responds to `p`), and combinations of both: chain precedence (`s` and `t` precede `p` or `p` precedes `s` and `t`), chain response (`s` and `t` respond to `p` or `p` responds to `s` and `t`), and constrain chain (`s` and `t` without `z` respond to `p`).

On the other hand, each kind of pattern has a scope of application which indicates the system execution on which it must be verified. There are five basic scopes: Global (the entire program execution), Before R (the execution up to a given property), After Q (the execution after a given property), Between Q and R (any part of the execution from a given property to another given property) and after Q until R (like between but the designated part of the execution continues even if the second property does not occur). Figure 7 illustrates the portion of an execution that are designated by the different kinds of scopes.

7.2 Property classification

The different properties to be verified have been catalogued to establish limits for the scopes (Q and R) and to specify the order of properties when more



Add property

Name: property1

Type: Precedence Chain (s ant t precede p)

Scope: Between Q and R

Type: Statemachine is in a state □ Refuse

Fig. 8. Property classification.

than one must be determined (`s`, `t` o `z`), so that the user does not need to know or understand the structure of the specification obtained in SMV to carry out verification. The established property types are as follows (see Figure 8).

- A state machine with one active object is in a particular state.
- An object activity is in a particular state.
- A signal or event is produced.
- Value comparison of an attribute.

The tool will automatically generate the property in the adequate format, in accordance with the chosen option and the selected pattern and scopes. Once we have the properties to be verified, it is possible, using the tool itself, to execute the SMV *model checker* to carry out the verification. If the property is not satisfied, it generates a trace showing a case where it is not verified.

For example, for the automatic teller machine, it would be possible to verify that the card is never retained; this means that the signal `retained` never happens. The property description is as follows:

Description	*The card is never retained*
Pattern	*Absence*
Scope	*Global*
Property Classification	*A signal is produced*
Specification automatically obtained	`G(~retained)`

As expected, the result of the checker is false. If the generated counterexample trace is analyzed (see next table), it can be seen that the card is retained when there has been 2 wrong pin numbers and again errorPin is generated.

Step	**62**	**63**	**64**	**|:65**
st_ATM	active	active	active	active
st_active	checkingPin	updateError	updateError	updateError
st_checkErrors	DontKnow	checking	BRANCH	retaining
errorPin	**1**	0	0	0
ATM_errorCounter	2	2	2	2
retained	0	0	0	**1**

8 Related work

Most works dealing with UML verification using formal method techniques focus on state diagram verification. Many such papers are based on previous works that, using formal methods, verified the classic *statechart* Harel [6] on which the UML state diagrams are based.

Yet today, as the use of UML is so widespread, researchers are looking to take a step further in UML verification with formal methods. Research is focusing on obtaining tools that automatically give the system specification in a formal language without the user's intervention and, as far as is possible, to achieve an automatic verification process.

The tools developed for verifying UML system specifications can be classified according to the formal language used, as a language prior to verification, for representing the system.

- Promela Language (tool Spin). Most work done on UML verification has been developed for the *model checker* Spin. The main contributions in this field are:
  - vUML [9,13]. vUML is a tool for the automatic verification of UML models, focusing on state diagrams. It is easy to use, generating automatically a representation of the system in Spin and performing an automatic verification to check that states of error are never reached and that states catalogued as desirable are reached.
  - Latella, Majzik & Massink [8] work with UML state diagrams, encoding them through hierarchical automata (HA) from which they generate the specification for Spin. The transformation is not carried out automatically, although they have achieved this in later works using XMI DarMajBen02.
  - HUGO [14,7]. This project includes a set of tools to apply *model checking* to UML state diagrams and collaborating ones. The latter are used to check whether the interaction represented in the collaboration diagram can be performed using state machines. That is, for the verification. The step from UML to Spin is carried out automatically using XMI.
- SMV language. There are also some works in the literature which try to verify UML by using the SMV model checker.
  - VeriUML [2] are a set of integrated tools developed in the University

of Michigan that allow UML state diagrams to be verified and to check whether the model is syntactically correct.

This first set of tools, developed in the year 2000, was later extended [15,16], allowing verification of the model's static part. The points of entry for this approach are the class diagram, the restrictions imposed on the said diagram written in OCL and an object diagram, all in XMI format. But the verification is not automatic.

- TCM (*Toolkit for Conceptual Modeling*), is a set of tools developed by R. Eshuis [5] which allows only activity diagrams to be verified by converting them to transition systems that can be verified using nuSMV. The transformation is automatic, although it is not based on XMI.

If a comparison is made between the work presented in this paper and the above-mentioned work, it can be concluded that the main characteristics of this paper focus on the possibility of performing an automatic verification of the behaviour of a UML specification in which the said behaviour is reflected through state and activity diagrams and is also semi-transparent for the user. Most works do not carry out an automatic verification and vUML [9,13], which does, does not use the potential of temporal logic, implementing a limited verification based on checking that it is impossible to reach error states. These error states are introduced by the user in the diagrams, so the diagrams are more complicated.

It should also be pointed out, though it has not been discussed here through lack of space, that the representative elements of both state and activity diagrams are included in this approach (except for synchronization states, events with parameters, and the dynamic creation and destruction of objects), something that cannot be said of other contributions in this field, in which few of the characteristics provided by UML (history states, deferred events, transitions fired by termination...) are dealt with.

9 Conclusions and future lines of work

This paper presents a tool whose main aim is to integrate formal methods with non-formal ones in such a way as to be transparent for the user. To be precise, it verifies the UML active behaviour using SMV. Although this is not a new idea, as far as we know at the present time, nowhere are activity and state diagrams jointly verified, using the former to represent the behaviour of the class operations.

However, the most innovative characteristic of the tool is that, in spite of using the potential of temporal logic to verify systems, the user need have no knowledge of its working. In addition, the user needs no knowledge of the structure of the specification obtained either, thus eliminating one of the major inconveniences of using formal methods.

As for future lines of work, some kind of treatment of the traces obtained in the verification when the property is not satisfied would seem to be of great interest. More precisely, that the representation of the traces should be visual instead of written, by using either some of the UML diagrams or an animated representation of the state and activity machines which could help the user to locate the error source very quickly.

References

[1] Booch G., Rumbaugh J. and Jacobson I.: The Unified Modeling Language. Addison-Wesley (1999)

[2] Compton K., Gurevich Y., Huggins J. and Shen W.: An Automatic Verification Tool for UML. University of Michigan CSE-TR-423-00 (2000)

[3] Darvas A., Majzik I. and Beny B.: Verification of UML Statechart Models of Embedded Systems. Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (2002) 70–77

[4] Dwyer M. B., Avrunin G. S. and Corbett J. C.: Patterns in Property Specifications for Finite-State Verification. Proceedings of the 21st International Conference on Software Engineering (1999)

[5] Eshuis R.: Semantics and Verification of UML Activity Diagrams for Workflow Modelling. PhD. University of Twente. (2002)

[6] Harel D.: STATECHARTS: A visual Formalism for Complex Systems. Science of Computer Programming, North Holland **8** (1987) 231–274

[7] Knapp A. and Merz S.: Model Checking and Code Generation for UML State Machines and Collaborations. Proc. 5th Wsh. Tools for System Design and Verification (2002) 59–64

[8] Latella D., Majzik I. and Massink M.: Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing. The International Journal of Formal Methods, Springer **6**, n. 11 (1999) 637–664

[9] Lilius J. and Porres I.: vUML: a Tool for Verifying UML Models. Turku Centre for Computer Science, Åbo Akademi University TUCS 272 (1999)

[10] McMillan K. L.: Symbolic Model Checking. An approach to the state explosion problem. Carnegie Mellon University. CMU-CS-92-131. (1992)

[11] OMG: XML Metadata Interchange (XMI) v2.0. OMG Document 2003-05-02. (2003)

[12] OMG: UML 2.0 Diagram Interchange Final Adopted specification OMG Document ptc/03-09-01. (2003)

[13] Porres I.: Modeling and Analyzing Software Behavior in UML. Åbo Akademi University TUCS (2001)

[14] Schäfer T., Knapp A. and Merz S.: Model Checking and Code Generation for UML State Machines and Collaborations. Proc. Wsh. Software Model Checking of Electronic Notes in Theoretical Computer Science **55**, n. 3 (2001) 1–13

[15] Shen W., Compton K. and Huggins J.: A validation Method for UML Model Based on Abstract State Machines. Proceeding of EUROCAST (2001) 220–223

[16] Shen W., Compton K. and Huggins J.: A Toolset for Supporting UML Static and Dynamic Model Checking. COMPSAC IEEE Computer Society (2002) 147–152