JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002
Vol. 1, No. 2, July-August 2002

Debugging UML Designs with Model Checking

María del Mar Gallardo, Pedro Merino, Ernesto Pimentel
Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga 29071 Málaga, Spain

Model Checking is currently one of the most exciting techniques to improve the quality of complex software systems. It is a computer aided verification method that, in many cases, has discovered design bugs in early development steps, thus saving time and costs to produce the final code. Although this technique is successfully applied to many formal description techniques, it is not commonly used by the object oriented programming community, in general, nor by UML developers, in particular. In this paper, we provide a comprehensive overview and rules to integrate model checking into UML-based designs, showing its usefulness from a practical point of view, and giving some guidelines to exploit the benefits of the integration.

1 INTRODUCTION

The aim of this paper is to discuss the integration of UML[BRJ98], which is a "de facto" standard method for modeling complex systems, and Model Checking [CES86], [Holz91], [CW96], which is becoming the standard technique for automatic software verification.[1]

Model Checking represents one of the most useful results of almost twenty years of research in formal methods to increase the quality of software and other related systems. A model checker is an automatic tool that is able to compare two descriptions of the behavior of a given system. Usually, one description is considered as the requirements and the other one as the actual design of a system to meet these requirements. The main constraint is that the description of the design, usually called the model, must be executable. This executability makes it possible to perform an analysis of all the execution paths (exhaustive analysis).

The main usefulness of model checking is its capability to produce a counterexample, or sequence of steps in the model, leading to the violation of a particular property. This feature makes model checkers a very valuable tool for debugging,

[1]This work has been partially supported by projects TIC99-1083-C02-01, 1FD97-1269-C02-02 (TAP) and TIC2001-2705-C03-02

Cite this article as follows: María del Mar Gallardo, Pedro Merino, Ernesto Pimentel: ***Debugging UML Designs with Model Checking***, in Journal of Object Technology, vol. 1, no. 2, July–August 2002, pages 101–117, http://www.jot.fm/issues/issue_2002_07/article1

even when only a partial analysis of a system is performed.

Unified Modeling Language (UML) provides a wide range of notations to model a software system from different perspectives, organizing them in five interrelated views: design view, process view, implementation view, deployment view, and use case view. The role of each of these views is complementary with the others, and the last one must guide a software process. Each view presents both static and dynamic aspects. In this paper, we are focusing on dynamic issues, which are commonly described by means of interaction, state and activity diagrams. Thus, interaction diagrams present the temporal ordering of the messages sent and received by a collection of objects (sequence diagrams), and the structural organization of these objects (collaboration diagrams). State diagrams represent state machines composed of states, transitions, events and activities, and they are used to describe the dynamic view of a system, particularly the behavior of an interface, a class or collaboration. They are very useful to model reactive systems. Finally, activity diagrams show the activity flow of a system, and the objects involved. Their usefulness is oriented to defining the function of a system, and stressing the control flow among objects.

A major deficiency in most current CASE UML tools is that the analysis focuses mainly on structural aspects. Some simple consistency analysis to avoid circular inheritance, or to detect access violation to hidden features (methods in a class, classes in a package, etc.), can be easily carried out by inspecting class, object, component or deployment diagrams. However, it is more complex to detect when a dynamic behavior described by several state diagrams corresponds to an acceptable scenario. Our aim in this work is to show how a very well known technique (i.e. model checking) can be applied to overcome the lack of tools to support exhaustive analysis of diagrams describing dynamic behavior of UML models, specially when modeling concurrent systems with UML.

In fact, the question of how to integrate both major techniques (model checking and UML) is still open. Some authors have recently proposed algorithms and tools to perform a limited kind of model checking of statecharts against temporal logic ([CW96],[LP99],[MLSH98].) However, from our point of view, this way of introducing model checking into daily programming work contains the same traditional difficulties: you cannot convince the user to learn new (logic based) languages apart from his/her usual notations if the benefits are not clear enough. The verification technique has to be introduced gradually, by using (and comparing) existing specifications (UML views), and the use of new specification languages must be considered as a last step for more expert users. The same idea of verifying only UML notations is also in [SKM01] and [Barn01]

The remainder of the paper is basically devoted to answering the following questions: Which are the main technical aspects of a model checker? Where and how is it practical to include model checking into UML based developments? Which are the closest verification tools to inspire the construction of UML-oriented ones? And, finally, can we expect to use the same optimization methods to deal with the well-known state space explosion problem?

2 MODEL CHECKING

For many years, verification of software systems followed the traditional deductive approach. With this approach the software engineer has to write both the specification of the system and the properties in some logic and, in the best of cases, a theorem-proving tool assists in the debugging task. But due to the need of ingenuity to obtain good results, only a small number of teams has been able to apply this approach to industrial-scale software.

Model Checking has appeared as a clear opponent to the traditional verification method, particularly in developing reliable software for concurrent systems. Although you often have to pay close attention when writing the desired properties, the program can be encoded with formalisms very similar to programming languages, and some current tools can even work with the final code. The verification task consists in ensuring that all, some or no execution paths, in the current design for the software, satisfy a particular (desirable/undesirable) property. This task is carried out automatically by generating and analyzing all the potential (finite) states of the program (exhaustive analysis). When required, the tool produces the counter-examples to know what paths satisfy/violate the property.

The first model checkers worked by constructing the whole structure of states as a prior step to checking the properties. The structure is obtained by executing all potential interleavings of the concurrent program. The properties were mainly liveness properties expressed with Temporal Logic, using formulae such as $\Box$ p ("p is always true"), $\Diamond$ p ("eventually p will be true"), p U q ("q will be true, and p will be true in all previous states") or $\bigcirc$ p ("p will be true in the next state"), p being any kind of proposition or even another temporal formula. Of course, nesting temporal operators increase the complexity of the verification, and decrease the confidence in the meaning of the formula. Nevertheless, the use of logic-based formalisms for debugging UML systems should be studied in the context of OCL. In any case, this is beyond the scope of this work.

Modern model checking tools work by translating the formula into an automaton to recognize the correct/incorrect execution paths. As the verification is performed as soon as the states are produced (and stored), the method does not require the previous construction of the whole state graph. This is why it is called on-the-fly model checking. Maybe SPIN [Holz91] is the most representative tool of this kind. One clear advantage of on-the-fly vs. structure-based analysis is the fact that the on-the-fly method can produce a partial analysis of very large systems, thus giving some information about the analyzed part. In opposition, in these cases, the structure-based method cannot start the verification.

The main problem of model checking is how to deal with very large state space systems. Although there are many real examples where the verification can be done with standard exhaustive verification, fortunately, current tools also implement several optimization techniques to analyze complex systems with more than 100.000 states.

The success of model checking to debug models of concurrent software (writing the models in academic formal description techniques and properties with temporal logic) has made this term relatively popular to describe other verification methods such as: 1) the automatic analysis of general correctness properties of concurrent systems that do not require a particular representation with a property language (absence of deadlock, non-reachable code, arithmetic errors, etc), 2) the automatic verification of more commercial formal description techniques (for example SDL [ITU100] confronted with Message Sequence Chart [ITU120] (MSC)), and 3) the automatic analysis of standard programming languages (C, C++, Java), usually by automatic model extraction from the source code to the input language of existing efficient model checking tools like SPIN.

Nevertheless, the use of model checking for debugging object-oriented software is still a novel topic, which is now being addressed in many international conferences. In particular, the integration of model checking into UML based tools is currently slow. Major commercial tools do not support the automatic analysis of model behavior, and in case they do, they only consider a limited analysis of general properties in statecharts. In the next section, we discuss one approach to introduce model checking in these tools.

3 MODEL CHECKING UML

Identifying the input for the Model Checkers

UML provides a wide range of notations to model a software system, giving details concerning both structure and behavior of the system. In addition, different mechanisms are also used to describe static and dynamic aspects of the same view. To cover all these possibilities, UML includes structural (class, object, component, deployment) and behavioral (use case, sequence, collaboration, state, activity) diagrams. All these diagrams are distributed in the so-called views, giving five different perspectives of the same model, all of them organized around use cases. Dynamic behavior is captured by four different kinds of diagrams: sequence diagrams, interaction diagrams, state diagrams, and activity diagrams. The two first kinds of diagrams are semantically equivalent, and both give the same information, but focusing on different aspects. A similar situation is given with the last two diagrams: a state diagram shows the answer of an entity to external events, and an activity diagram focuses on the flow control of the actions.

Figures 1 to 3 represent some of the views of a compact disk (`CD`) system. The class diagrams in Figure 1 show the invocation relation among the control part (`CD_PLAYER`), the physical drive manager (`CD_DRIVE`) and the user (represented in UML as an actor plus an observer). Figure 2 contains the statecharts for the dynamic behavior of the system, and Figure 3 contains a sequence diagram representing one of the expected scenarios.

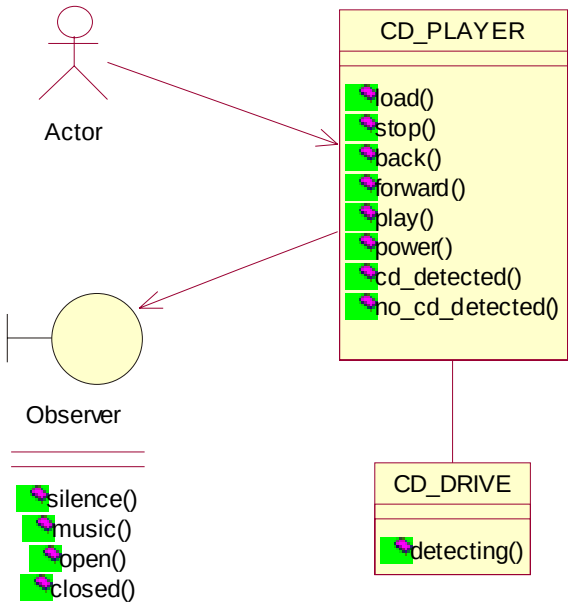


Figure 1: Class diagrams for CD system

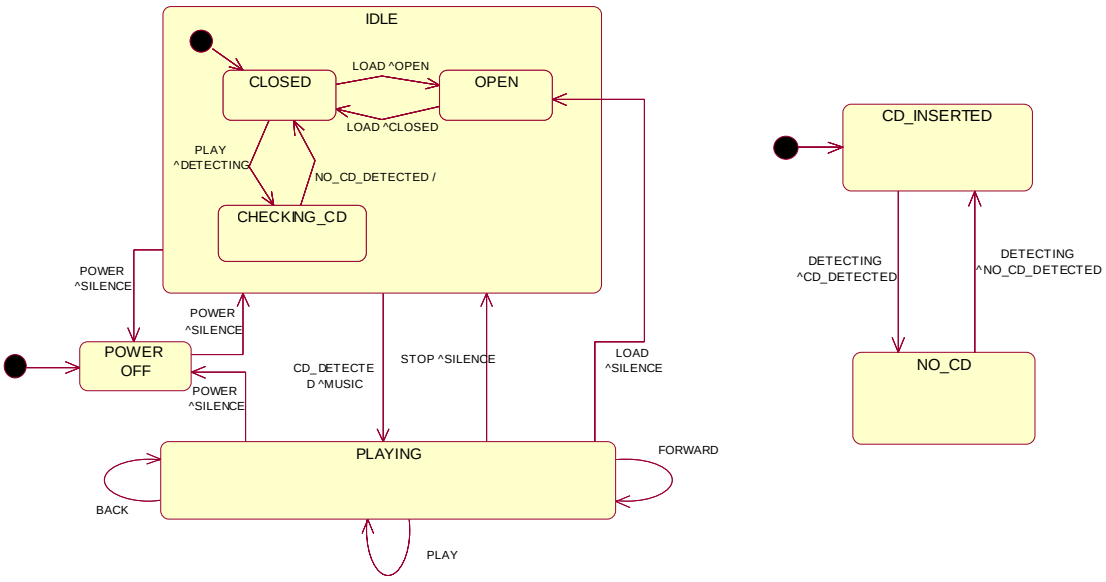


Figure 2: Statecharts for CD_PLAYER (left) and CD_DRIVE (right)

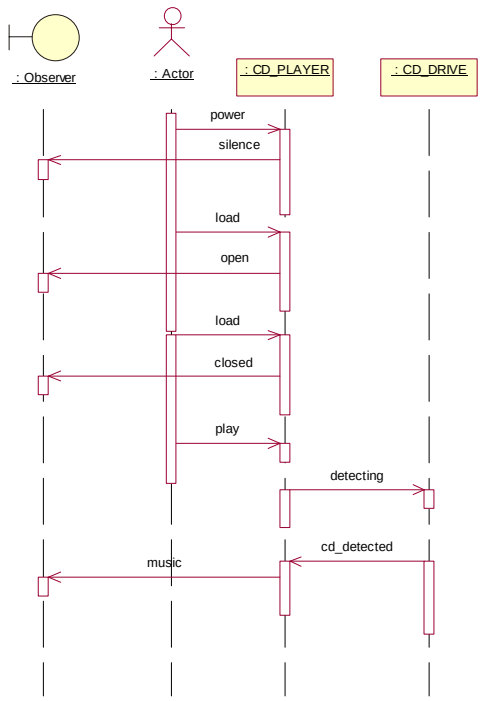


Figure 3: A sequence chart for a desirable behavior

Most of UML-based tools include some possibilities of consistency analysis and automatic code generation. However, only structural information is used to do this. Our idea is to apply model checking to compare the behavior described by a sequence diagram (or a collaboration diagram) with the behavior expressed by a set of state diagrams. This analysis will ensure the right correspondence between a sequence diagram giving the interaction among a group of objects, and the statecharts exhibiting their individual behavior. Obviously, model checking could also be applied to analyze the correspondence between one or more activity diagrams and a collaboration diagram.

Rules for practical Model Checking

If we consider that state and sequence diagrams are the two most representative notations to describe the dynamic behavior of a system, then three main phases have to be covered to apply model checking. In every phase, a set of rules is given to obtain satisfactory results.

Analyzing general properties of Statecharts (Phase 1)

The exhaustive analysis of isolated statecharts or systems composed by several ones is the first step to discard errors. This analysis consists in producing the configu-

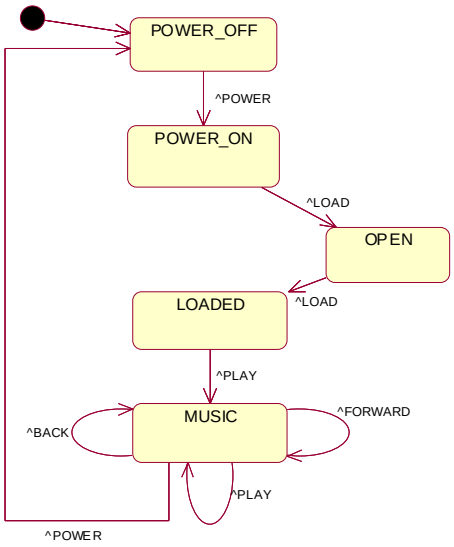


Figure 4: The statechart representing an actor's behavior

rations (global states) of the system. As model checking works automatically, the system to be analyzed must be closed with respect to the environment. This means that the user of this part of the software has to be completely modeled. For example, the statechart of Figure 4 represents a real user (an actor) of the CD system. Verification must start by identifying an initial global state. The initial global state of the system composed by the actor, the CD_PLAYER, and the CD_DRIVE is

```
[POWER_OFF, IDLE, CD_INSERTED]
```

And the state corresponding to the system while the CD system is working could be

```
[MUSIC, PLAYING, CD_INSERTED]
```

If queues are considered to store events, then they are also appended to the global state. With this representation, the model checking tool can interleave the user steps with the CD system in order to inspect general properties such as end-states, conflicting transitions, non-consumed events, etc. The use of interleaving allows us to deal with many semantic variants of statechart, that depend on the application domain. The most efficient way to carry out this search is a depth-first search with a memory to store the visited states. When an error is found, the sequence of states stored in the stack used for searching is employed to construct a path towards the error.

For example, in Figure 5, the sequence diagram represents a discovered potential error consisting of sending a BACK event that is not consumed by the CD_PLAYER statechart. The reason is clear: in the design of the CD_PLAYER, we did not take

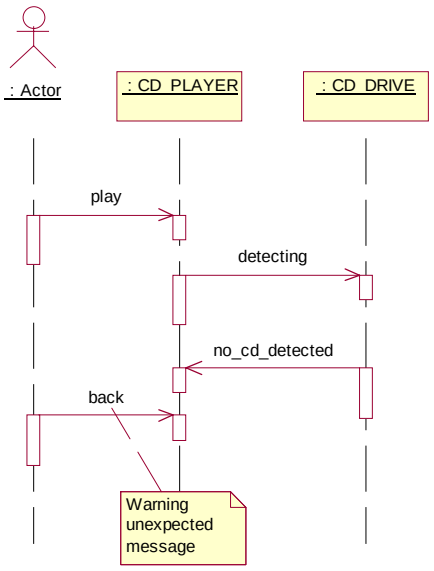


Figure 5: Scenario with unexpected events

into account that the user can press BACK without a disk in the system. There are other two similar errors in the system (for PLAY and FORWARD events). All these errors are found inspecting 29 unique system states, but generating 38 states (note that some states are visited several times). Each state requires between 76 and 180 bytes, whereas the verification time is negligible.

One critical decision at this first verification step is how many details will be considered in the statecharts (including the actors). The complexity increases the number of states (and decreases the quality of the analysis). So, if we need to improve the analysis, accessory details, which are not part of the behavior, must be removed. However a limit must always be observed: reducing the complexity cannot change the basic behavior of the system. For example, removing regions to limit concurrency is not a good idea.

In summary, the rules for successful verification of general properties are the following:

1.1 Construct the smallest environment to check the system

1.2 Check for a few sets of general errors, and remove these errors before analyzing the cause of the others

1.3 Do not move to the following phase without a precise knowledge about how the current state diagrams work. It is necessary to know the reasons for errors and warnings.

Verifying Statecharts against desirable Sequence diagrams (Phase 2)

The second phase in verification usually consists in checking for very particular properties of the designed system. The most practical way is to employ a sequence diagram to describe a potential desirable behavior in the system. If we want to check that this behavior is possible in the statecharts, then the model checking tool can translate this sequence diagram into an automaton to inspect the evolution of the global states previously inspected in Phase 1 (again with the environment included). For this purpose, states are associated to the sequence diagram at the points where some *interesting* event occurs in some entity, and its state is included in the global state. For example, in the extended global state

```
[LOADED,CHECKING_CD,CD_INSERTED,sent(CD_DRIVE,cd_detected)]
```

the state of the automaton is represented by sent `(CD_DRIVE, cd_detected)`, and it captures that event `cd_detected` was sent by `CD_DRIVE`.

The automaton evolves synchronized with the system being debugged, observing events in the statecharts, and when it reaches a final state (the last event in the sequence diagram), then the property is *verified*. Of course, the automaton is never accessible to the user, and the verification tool must keep it hidden, but it allows extending the basic verification method in Phase 1 to deal with the sequence diagram verification. For example, by producing and inspecting 10 states, it is possible to verify the property in Figure 3 (and partially in Figure 4). Thus, we can ensure that this scenario can be reached by the interaction of the statecharts in the current design.

However, this verification work is not so easy. What happens when the system produces an unexpected event with respect to the sequence diagram? Must it be ignored or must the tool discard the current path and try another one? The most convenient answer is *"it depends on the event"*. If the unexpected event is taking an active part in the scenario represented by the sequence diagram, then it cannot be ignored. Otherwise, the analysis can go on with the same path, ignoring the last event. The general principle is that the underlying automata representing the sequence diagram only moves or discards an execution path when it finds *observable events*. So the model checking tool must report success in the verification if all observable events in the trace are consistent with the partial ordering of events that is defined by the sequence diagram.

In order to have a practical verification tool, the set of observable events should be defined for every verification, according to the following rules:

2.1 When you are interested in obtaining at least the sequence of events in the sequence diagram, even if other events occur, then specifically define as observable the events which are in the diagram. Others will be ignored (and allowed in the trace produced by the statecharts.)

2.2 When the sequence diagram represents exactly the desired sequence of events,

and no others should occur, then it is necessary to define all events in the system as observable.

If the verification of the statecharts against the sequence diagram is not possible with these rules, then conclude that the desired scenario is not reachable by the system. In case of doubt, proceed by extending or restricting the sequence diagram until the verification, with one of the previous rules, works. For example, a property like the one in Figure 6 is not verified by the system composed by the statecharts in Figures 1 and 3, whichever observability rule you use.

Verifying Statechart against non-desirable Sequence diagrams (Phase 3)

However, the verification of a set of desirable sequence diagrams should not be considered sufficient when debugging the dynamic behavior of UML systems. An additional verification of undesirable scenarios will exclude some critical errors. For this purpose, a set of simple diagrams could be built and checked in the same way, and with the same rules, as before. But now we expect that the model checking tool will never produce success in the verification. Verification means errors. For example, a diagram like the one in Figure 6 represents a situation where the CD system plays music without an inserted disk (see the two last events in the sequence diagram.) The verification tool analyzes 18 states looking for a trace to satisfy the scenario. And finally, it discards the verification of this malicious scenario.

Finally, actions could be considered equivalent to events from the verification point of view. Actions could be present in the statecharts and sequence diagrams, and when defined as observable, the actions in the trace should occur in the same order as expected in the sequence diagram.

4 ABOUT EFFICIENT VERIFICATION TOOLS

Verification with state and sequence diagrams has to be carried out automatically and efficiently. Using the composite global state (with the underlying automata-based representation for the sequence diagram), a model checker can automatically perform the three verification phases. Of course, the exhaustive analysis has to remove duplicate states and to detect loops when performing the depth first searching. Furthermore, when debugging realistic systems, the model checker must also implement strategies to solve the state explosion problem. Some of these strategies are well known [Holz91]. *Partial order reduction* is a method for replacing several interleaved sequences of events (or actions) by only one that represents the whole set. The *State compression* method reduces the use of memory by compressing the representation of the states without losing information, but increasing the verification time. *Bit-state* analysis represents states as bits in a hash table instead of storing the whole global state, so in many cases the analysis is only partial.

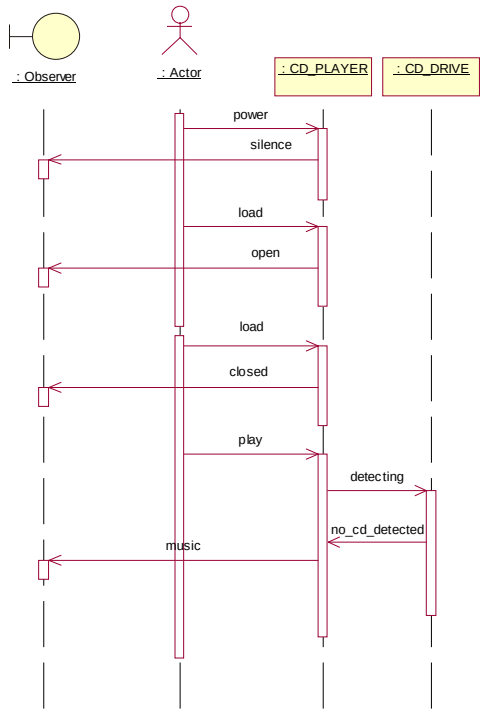


Figure 6: A sequence diagram for an erroneous behavior

The efficiency of these well-known techniques can be further augmented with the more recent *Abstraction method* [CGL94]. In general, abstracting the model of a system means to construct a simpler specification which is equivalent to the initial one with respect to the properties to be verified. The abstract model can be verified using less time and memory. The classical method for constructing the abstract model consists in generating the state graph prior to verification, and developing ad hoc new model checking tools depending on the way of abstracting. An alternative approach is to produce a new textual model by transformation of the initial one, in such a way that it can be analyzed with the same model checker employed with the original one. Furthermore, new transformations can be applied to the already abstracted versions. Based on our successful experience employing this approach with the languages Promela [GM99] and SDL [GM00], we can argue that the same method could be employed to improve the verification of statecharts against sequence diagrams.

The most powerful abstraction of statecharts to verify sequence diagrams is the abstraction of events. This kind of abstraction consists in using a single name of event (an abstract event) to represent a set of real events. For example, the abstract event `button` could represent all the events due to the actor requests `(power, load, play, forward and back)`, and the abstract event `effect` could represent the answers from the `CD_PLAYER`. If we make abstraction of events, then we have to transform the statecharts to work with the abstract events. Fortunately, this work

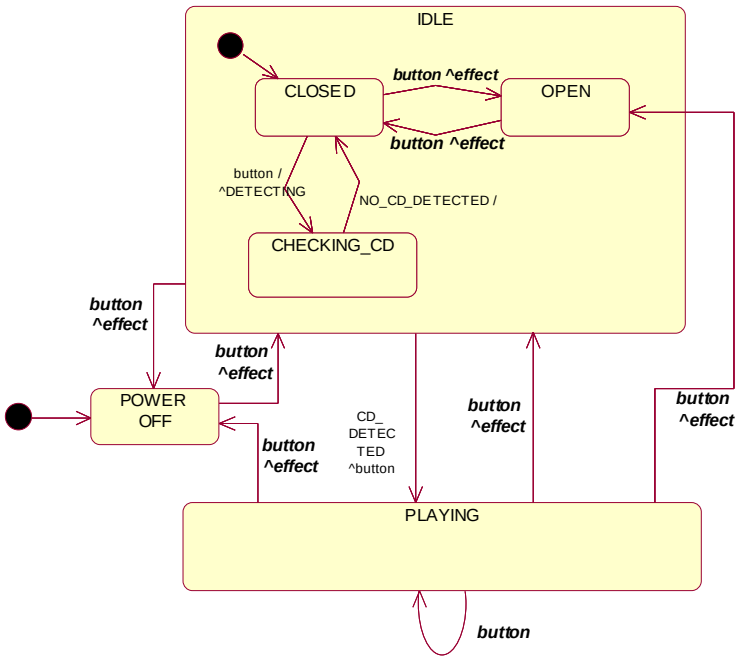


Figure 7: Abstract version of CD_PLAYER

can be done fully automatic. For example, Figure 7 shows an abstract version of CD_PLAYER with the two abstractions of events described above. Note that now the statechart exhibits more non-deterministic behaviors. For example, two transitions starting at state CLOSED can be fired with the same event button. This is due to the loss of precision introduced by the abstract event button: How to know if button is representing play or load to fire the correct transition? We need to introduce non-determinism in order to preserve at least the same potential execution sequences as in the initial statechart. This is a key point to obtain useful verification results.

The verification with abstraction of events works as follows. Given a model M composed by a set of statecharts and a sequence diagram SD, we proceed by constructing and checking a more abstract model M^* against an abstract sequence diagram SD^* until deciding about the satisfaction of SD against M, or until finding specific errors (phases 2 and 3 in our proposed methodology.) The abstraction in both diagrams is done using the same abstract events.

The verification of undesired behaviors (Phase 3) is done directly: If M^* does not verify SD^*, then M does not verify SD. Therefore, M can be employed to continue the development cycle (e.g. code generation). For example, let us assume that the simulation of statecharts in Figure 2 exhibits too many unexpected errors, probably because it is a very early version. If we want to have a minimum confidence about its correctness, we could try to check sequence diagrams considering the number of events that the actor and the system send, without taking care about the events themselves. For instance, the sequence diagram in Figure 8 (left) shows a non-

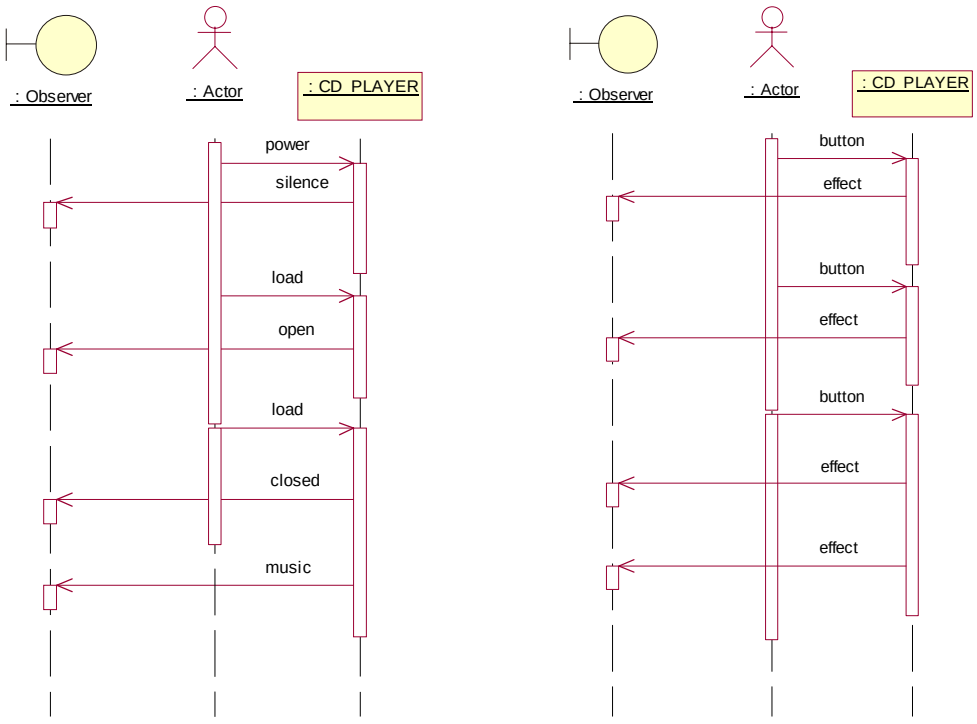


Figure 8: An undesirable behaviour (left), and its abstract version (right)

desirable SD (it is forbidden for this system to start the music automatically after inserting the CD; the user is required to explicitly push the button play.) Figure 8 (right) shows its corresponding abstract version (SD^*). The model checker will now report that the system (using the statechart in Figure 7) will never produce a sequence of abstract events like the one in Figure 8 (right), and we could conclude that the initial system never produces a sequence like the one in Figure 8 (left.) It is worth noting that the computational effort when verifying the abstract version has been reduced with respect to the verification of the initial model. In addition, as an abstract sequence diagram represents a set of standard ones, the abstraction reduces the number of sequence diagrams to be analyzed against the original model to obtain the same result.

However, the loss of precision in the events and the addition of non-determinism in the statecharts make the verification of desirable behaviors (Phase 2) a more demanding task. One abstract event represents a set of real events, so the occurrence of the abstract event is not conclusive to decide about the occurrence of a real one. The same problem appears when the tool reports an error in the verification of undesirable behaviors (Phase 3). In both cases, we have to use the abstract trace in order to check whether a particular sequence of events represented by this one, is possible in the initial model. If the sequence exists in M, then we have information to give a result regarding the behavior of M. Otherwise, we have to revise the abstraction (maybe using a different one), produce a new abstract model, and repeat

the verification.

Apart of abstracting events, it is also possible to abstract other elements in the specification like variables (attributes), or transition guards. But a discussion about this is out of scope of this work.

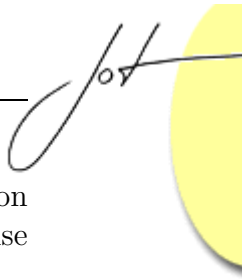
Although our proposal is to construct specific model checking tools for UML, taking into account that the UML1.4 specification [OMG02] considers a number of semantic variation points for statecharts, one practical way to get experience with our verification method is the translation from UML designs into input languages for existing tools. A direct mapping can be made from statecharts to SDL specifications, and from sequence diagrams to MSCs. Then, we may use existing tools to perform the three phases reported above, and also to support the optimization techniques, including the abstraction method implemented for SDL. For example, the verification experiences described in this paper were done with this translation scheme.

5 CONCLUDING REMARKS

Our recommendations to verify statecharts and sequence diagrams provide a well structured methodology to debug complex UML systems prior to implementation. Verification can be fully automatic and no new specifications have to be created. The method is specially useful to find design errors in the first versions of the model.

Our proposal is complementary to other related work in several aspects. The use-case tree [Barn01], proposed by Barnard, represents the whole set of completed scenarios expected in the model. It is assumed that the tree is constructed from the statecharts. Although this representation of the state space is constructed automatically, the analysis of scenarios with use-case trees was proposed to be done manually due to the fact that the scenarios (the use cases) were given informally. But, even if the scenarios were formalized (e. g. as sequence diagrams) the construction of the whole use-case tree (as a graphical view of the system) limits the possibilities for automatic verification for two reasons. The first is that a path in the use-case tree could contain all model events, so the scenarios should also consider every event, including those produced by objects which are not actually participating in the scenario. The second is the impossibility to store the whole tree for realistic systems. It is very frequent to have millions of composite states for real systems, and use-case tree analysis will exhibit the same constraints as old structure-based model checking.

These problems in Barnard's method are overcome in our proposal by considering partial verification as a debugging method. Our observability conditions combined with the objective (desirable/undesirable sequence diagram) for just one property in each verification reduces the size of the scenarios. The underlying automata-based verification for sequence diagrams allows us to explore the state space with an on-demand generation of states (on-the-fly model checking). This strategy, combined



with discarding paths that satisfy/violate the current property under verification and with the abstraction method, allows the analysis of very large systems, because we do not need to store the whole graph prior to verification.

The method for debugging UML by comparing sequence and state diagrams is also complementary and close to other methods based on translating statecharts into Promela, the input language of the model checker SPIN. All these approaches ([LMM99], [LP99], [MLSH98], [SKM01]) consider on-the-fly model checking, and the main differences among them are in the translation scheme to Promela and the way of considering the properties to be verified.

Latella et al.'s [LMM99] proposal for using temporal logic to define desirable (undesirable) scenarios also corresponds to partial verification, because a practical temporal formula only represents very specific fragments of the real executions in the system. The work by Mikk et al.[MLSH98] also considers temporal logic to represent the properties. These two works employ different semantics for statechars. The first one describes a new semantics, while the second one uses the statechart semantics implemented by the commercial tools of I-logix. Being temporal logic outside the UML framework, our approach is more suitable for the way UML software engineers normally work. A related and non-explored alternative closer to UML could be the use of OCL as a property language.

Tool vUML [LP99], although keeps Promela hidden to the user, only supports the verification of general properties of statecharts, corresponding to Phase 1 in our methodology. The authors employ collaboration diagrams to create the Promela configuration to be analyzed, but not as a notation for properties.

HUGO [SKM01] also employs Promela and SPIN as the core verification technology, but its users work with UML descriptions. HUGO verifies whether the desirable behavior described by a collaboration diagram is feasible for a set of UML state machines (equivalent to our Phase 2.)

Our work is mainly focused on the use of existing tools for SDL and MSC, because it is expected that SDL and UML will have a parallel development in the next years. For example, the ITU has approved a recommendation for using both formal methods combined [ITU109], and the OMG has recommended SDL as an action language for UML1.4 [OMG02]. So, SDL-based tools seem to be more suitable to support debugging UML with model checking than Promela-based tools.

Our conclusion is that the proposed partial verification method is clearly supported by current trends in the use of formal methods for automatic analysis of software systems[CW96]. Furthermore, we believe that usual practice in object oriented design is to start by giving desirable use-cases in UML but limiting attention to only one piece of the behavior. All the views are extended with new details at the same time, so the verification of statecharts and sequence diagrams can be done incrementally. This practice satisfies our proposal of partial scenarios.

REFERENCES

[Barn01] Barnard J., Use-Case Tree, Journal of Object Oriented Programming 13(10):19–24, Feb.2001.

[BRJ98] Booch G., Rumbagugh J., Jacobson I., The Unified Modelling Language User Guide, Addison Wesley, 1998

[CES86] Clarke E.M., Emerson, E. A., Sistla A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Trans. on Programming Languages and Systems, 8 (2):244–263, 1986.

[CGL94] Clarke M., Grumberg O., Long D.E., Model Checking and Abstraction, ACM Transaction on Languages and Systems, 16(5): 1512–1245, 1994.

[CW96] Clarke E.M., Wing J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, 28(4):626–643, 1996.

[GM00] Gallardo M.M., Merino P.: A Practical Method to Integrate Abstractions into SDL and MSC based Tools. 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems, GMD Report 91, 2000.

[GM99] Gallardo M.M., Merino P.: A Framework for Automatic Construction of Abstract PROMELA Models. In Theoretical and Practical Aspects of SPIN Model Checking, Lecture Notes in Computer Science 1680:184–199, 1999.

[Holz91] Holzmann G.J.: Design and Validation of Computer Protocols. Prentice-Hall, 1991.

[ITU100] ITU-T, Z.100 — Specification and Description Language (SDL), ITU-T, 2000

[ITU109] ITU-T, Z.109 — SDL combined with UML, ITU-T,1999.

[ITU120] ITU-T, Z.120 — Message sequence chart (MSC), ITU-T,2000.

[LMM99] Latella D., Majzik I., Massink M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Modelchecker. Formal Aspects of Computing 11(6):637–664, 1999.

[LP99] Lilius J., Porres Paltor I., vUML: a Tool for Verifying UML Models, 14th IEEE International Conference on Automated Software Engineering (ASE'99), pp. 255–258 1999.

[MLSH98] Mikk E., Lakhnech Y., Siegel M. and Holzmann G.J, Implementing Statecharts in Promela/SPIN, Proceedings of Workshop on Industrial-strength Formal Specification Techniques (WIFT'98), 1998.

[OMG02] Object Management Group (OMG) , OMG Unified Modelling Language Specification (Action Semantics), UML 1.4 with Action Semantics, http://www.omg.org, January, 2002.

[SKM01] Schäfer T., Knapp A. and Merz, S., Model Checking UML State Machines and Collaborations, Electronic Notes in Theoretical Computer Science 55(3):19–24, 13 pages, 2001.

ABOUT THE AUTHORS

María del Mar Gallardo is Associate Professor at the Department of Computer Science of the University of Málaga (Spain). She received a Ph.D. in Computer Science from the same University in 1997 with a thesis on the use of abstract interpretation to improve the execution of concurrent logic languages. Her work is currently focused on abstract model checking. She can be reached at gallardo@lcc.uma.es and http://www.lcc.uma.es/∼gallardo.

Pedro Merino got his PhD in Computer Science in 1998. He is Associate Professor at the University of Málaga, Spain. He works on model checking for software and for communication protocols. He is also interested in active networks, especially in the integration of performance and safety analysis. He can be reached at pedro@lcc.uma.es and http://www.lcc.uma.es/∼pedro.

Ernesto Pimentel is Associate Professor at the University of Malaga. He got his PhD in Computer Science in 1993, and his research activity is related with the application of formal methods to software engineering, including topics like models for concurrency, component-based software development, and abstract model checking. More information is available at: http://www.lcc.uma.es/∼ernesto.