Annales UMCS Informatica AI 4 (2006) 244-253

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

The use of model checking and the COSMA environment in the design of reactive systems

Jerzy Mieścicki*

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00 665 Warszawa, Poland

Abstract

The paper discusses how a bridge between the design practice and the formal methods could be maintained. The use of model checking seems to be the most promising approach. Then, the software environment COSMA is presented, implementated in the Institute of Computer Science,WUT. The conceptual framework of COSMA is based upon Concurrent State Machines (CSM) and Extended CSM, which are also briefly summarized and illustrated with a simple example.

1. Introduction

1.1. Formal methods and the design process

One of the most interesting aspects in the up-to-date computer science is a gap between the research on formal methods of system design and rather poor usage of its results in practice. Indeed, the research in formal methods has gained a growing interest in the last decade, both in academia and in the leading research centers. International organizations exist (e.g. FME (*Formal Methods Europe* [1]) and about twenty international conferences devoted to these issues are organized every year (e.g. FME (*Formal Methods Europe*, FORTE (*IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, or ETAPS (*European Joint Conferences on Theory and Practice of Software*). In addition to specialized journals (as, for instance, *Formal Methods in System Design* or *Formal Aspects of Computing Science*), the use of formal methods is frequently discussed in professional international journals. The original formal methodologies are proposed, supported with the appropriate software tools. The reader can find more information in [2].

On the other hand, most of these results, methodologies and tools are not broadly used in practice beyond the research community, despite the fact that

* *E-mail address*: J.Miescicki@ii.pw.edu.pl

their authors work hard to make them be *industrial strength formal methods*. Hardware and IC designers seem to be more aware of advantages of formal methods, but as for software products – the testing procedures (at the consecutive stages of design and implementation) and beta-versions are the only (practically used) means for the system verification.Testing, however, is basically an experimental (rather than formal) method. In the case of reactive systems, involving a communication among concurrent subsystems (modules, threads, ... etc.) as well as between subsystems and the environment – one can hardly expect that all possible sequences of events would be actually tested. Moreover, the testing can improve our confidence that the system *does what it should do*, but can not assure us that it *does not do anything it shouldn't* (get deadlocked, for instance). On the other hand, formal methods are aimed at proving (rather than check experimentally) the required properties of system behavior.

Also, the commercially available CASE tools hardly support the formal verification of the systems under design. The emphasis is put mainly on the consistency with a particular design methodology, structural or object-oriented programming paradigm, on provisions for team work, version control, documentation, design of tests etc. Offered specification formalisms (e.g. UML [3]) are aimed at supporting the project readability and its convenience for users rather than to guarantee the formal correctness of the project. Only a few CASE environments (as, for instance, EDT based on the Estelle language [4,5]) offer a rigorous, formally defined semantics of used programming constructs.

Probably, it is so partially due to just the imperfect nature of software tools (supporting the formal methods) implemented in the research institutions. But in addition to this, the formal verification methods require from the designer new knowledge. He/she has to acquire a bulk of new, quite allien, concepts and ideas as well as to get familiar with new 'technology' of building provably correct systems. One may guess that this also explains why the methods built upon Z notation, VDM or B-method, or theorem proving techniques and tools (e.g. PVS, HOL or Larch) enjoy only a limited (if any) interest in practical designers' community.

Out of a vast collection of formal methods, *model checking* [6-9] seems to be the most promising one just from the viewpoint of potential proliferation among system designers. Indeed, model checking presently gains a growing interest in today's computer science and practice. During the last two decades it has become a known technique for the verification of industrial hardware projects [10], protocols and software [6,7]. A range of software tools (or model checkers) have been implemented aimed at supporting the verification. Among the most frequently referenced ones are SPIN [11,12], SMV [9,13], FormalCheck [14]. A number of other tools of this type have been implemented for research purposes [2].

1.2. Characteristics of model checking

The main idea of the approach is that the system to be verified is modeled as some formal *finite-state* structure M (e.g. a transition system, a graph of reachable system states etc.), representing the system behavior. The desired property (p, say) is also formally expressed (e.g. in a form of a formula of some temporal logic, or a Büchi automaton). Then, the model checker checks if $M \models p$, that is, if p holds for M. The evaluation of $M \models p$ involves the exhaustive inspection of M[1].

Notice that as the model M is finite, the checking against any property expressed in terms of system states, transitions, input or output sequences etc. is decidable, and can be algorithmized, at least if we postpone for a while the problems related to the size of M and to the complexity of algorithm. This way, the system designer is offered a set of ready-to-use algorithms and techniques for the analysis of system properties. Moreover, if the checked property does not hold, he/she can obtain a counterexample, i.e. the path leading to the just-identified failure. This provides the feedback information enabling the designer to identify and correct the component which is responsible for a negative outcome of the checking.

Finite state structure M representing the global behavior of a system is usually obtained from the specification of a collection of system components. Each component's behavior is represented either explicitly (in a form similar to statecharts, UML's state diagrams [15,16] etc.) or derived from some primary specification (frequently, having the form of program-like notation, e.g. as SPIN's Promela [12]). For each model checking platform a rule is determined how these local behaviors are composed into a one, large (however still finite) graph or transition system M which is the subject to exhaustive inspection while $M \models p$ is evaluated. Different approach was undertaken in the Bandera project [17]. Here, the Java code (which is actually the final product of the design process) is a primary specification for model checking. From this code, using the techniques of abstraction and slicing [18], some intermediate specification (BIR) is derived, which can be converted into an input language of model checkers like SPIN.

The main challenge the model checking is confronted with is the exponential explosion of the model size, which has been always considered a serious limitation of all finite-state methods. Therefore, an extensive research has been (and is being) done on various techniques that can help to manage the problem. First, a very effective representation for very large graphs has been developed, based on Reduced Ordered Binary Decision Diagrams (ROBDD). It allows the representation of graphs of 10^{20}-10^{50} states or even more. Secondly, the multiple

[1]Of course, the correctness of system's behavior is usually checked against a set of such properties, $\{p_i\}$.

forms of *reduction* of state space are proposed, aimed at removal of the states and transitions which are irrelevant w.r.t. the evaluation of a given formula. The other approach is to calculate the model stepwise, just during the evaluation, as one can expect that in order to obtain the outcome of the evaluation only the *limited model* will do. Still another technique is the *compositional model checking*, where the whole model (too large to be analyzed at once) is decomposed into sub-models of more acceptable size. Accordingly, also the process of verification of model properties is performed in a step-by-step manner rather than in one run.

The present paper is devoted to model checking techniques developed for the COSMA software environment [19], now under implementation in the Institute of Computer Science, (Warsaw University of Technology). COSMA is based on the idea of Concurrent State Machines (CSM) [20], the finite-state model particularly suitable for the modeling of the cooperation and communication among the components of concurrent reactive systems, as well as between the system and its environment.

2. Concurrent State Machines (CSM)

In the CSM framework, the system is a finite set of Concurrent State Machines (CSM), representing the behavior of individual system components. Components can receive (as their input) signals or messages from the environment and from other components. They also can produce signals as their output. Formally, these signals are input or output symbols of an automaton. To any atomic symbol we attribute the atomic proposition which is *True* if (in a particular state) this symbol occurs at the input (or is produced as the output) and *False* otherwise. Let AP be the the finite set of all such atomic propositions. Let BF stand for a universal set of Boolean formulas. Formulas $form \in BF$ are sequences of symbols that obey the well-known syntax:

$$form ::= \mathbf{0} \mid \mathbf{1} \mid prop \mid !form \mid (form + form) \mid (form \star form)$$

where *prop* stands for any atomic proposition $p \in AP$ and !,+, $\star$ stand for Boolean negation, sum and product (respectively). The semantics of Boolean formulas is equally conventional.

Formally, each CSM is a tuple

$$m = < N, edges, form, out, n_0 >$$

where:

- N – finite set of nodes (states of behavior),
- $edges \subseteq N \times N$ – set of directed arcs,
- $form : edges \to \mathcal{BF}$ – labeling function, attributing Boolean formulas to edges,

- $out : N \to \mathcal{P}(AP)$ – output function, attributing each node with a set of propositions that are *True* for this node,
- $n_0 \in N$ is the initial node.

To give the reader just a flavor of CSM modeling, let us consider a simple system of two cooperating processes: Sender and Receiver using a common Buffer. The Sender puts data frames into the buffer, one by one, and stops when N_1 frames have been put. The Receiver observes the buffer and once N_2 data frames are collected, the Receiver turns active, processes the content of the Buffer, clears (resets) it and signals to the Sender that the operation of transmitting the data can be resumed. Of course the system works properly only if $N_1 = N_2$, i.e. the number of frames sent by the Sender matches the number of frames waited for by the Receiver. However, for the sake of illustration, assume that (due to a design error) $N_1 > N_2$.

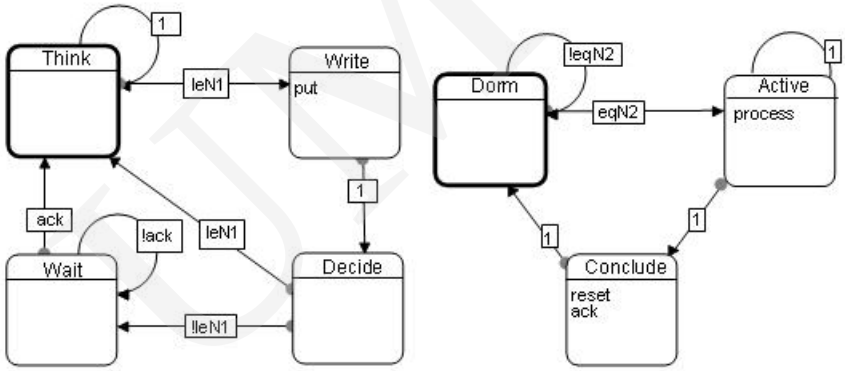

Fig. 1. CSM models of Sender (left) and Receiver (right)

The CSM models of Sender and Receiver are shown in Fig. 1 and Buffer (for the case $N_1 > N_2$) in Fig. 2. Initial states are highlighted with a thicker borderline. Remember that graph edges are labeled with the Boolean formulas rather than with symbols of some input alphabet. If the formula is *True*, the transition is enabled. If more than one transition is enabled then *one of them* is selected as active and executed. The choice is nondeterministic and fair. Formula 1 is always *True* and the edges labeled with it are unconditionally enabled. Thus, if the Sender is in its initial state and the number of data in Buffer is less than N_1 (i.e. leN_1 is *True*), the Sender can (nondeterministically) either remain in *Think* or to pass to *Write* etc. Notice that CSM produce the *sets* of output symbols. For instance, Receiver in *Dorm* produces an empty set of symbols , one-element set {*process*} in *Active* and two-element {*reset,ack*} in *Conclude*.

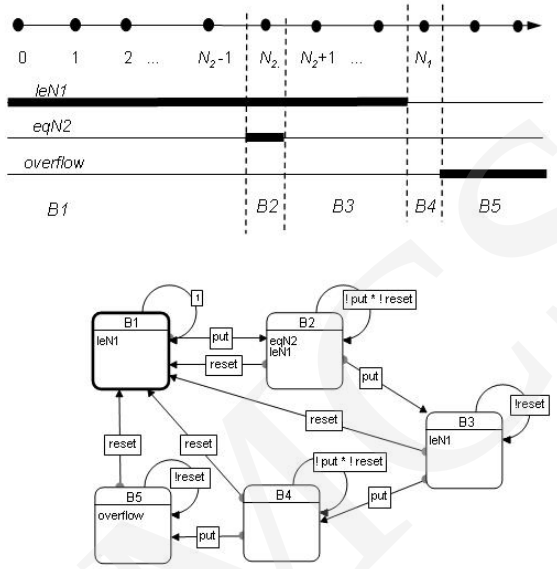

Fig. 2. CSM model of Buffer for $N_1 > N_2$

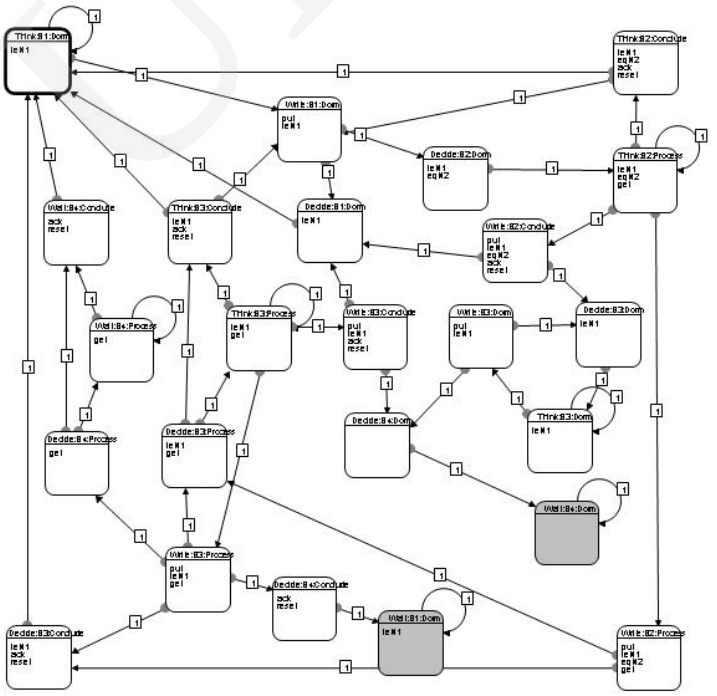

Fig. 3. Graph of reachable system states

For the system of CSM, the algorithm of obtaining the graph of reachable system states (GRSS) has been developed [20] and implemented as a module of COSMA environment. GRSS for the example system is shown in Fig. 3. It has as few as 24 reachable states (out of $4\times3\times5 = 60$ elements of Cartesian product of sets of components' states) so that it can be analyzed just by naked eye. At no surprise, the example system performs incorrectly: there are two reachable deadlock states (shadowed grey) in which the system unconditionally remains for ever. Moreover, at some states both *put* and *get* occur simultaneously, which means that the access to the Buffer is not properly synchronized.

The example above was purposefully simple and small in terms of the size of the reachability graph. The COSMA environment (described in the next section) can be used for the verification of much more practical and challenging systems.

3. The COSMA environment

The overview of COSMA software ([19]) is sketched in Fig. 4. A central role is played by the repository, which stores the system components specified in a form of text files in CXL language (based on XML). The COSMA control module (not shown) supports creation/edition of workspaces and projects as well as communication to/from other modules. The functions of main COSMA modules are the following:

- *Grapher* provides the user interface for graphical specification/edition of the CSM graphs and their conversion to/from CXL text files,
- *Product Engine* performs the conversion of CSM models from CXL to ROBDD data structures, computes the reachability graph of a given project (using a state-of-the-art BDD library) and (if needed) converts the resulting CSM graph back to CXL. Product Engine supports also algorithms for multi-phase reduction of the product [21,22],
- *TempoRG* evaluates the required properties (expressed in a form of formulas in QsCTC, a version of CTL [23,24]) in the Reachability Graph of a system,
- *Cntrexmple Editor* processes the counterexamples provided by *TempoRG* in the case of a negative evaluation,
- *UML2COSMA* (now under implementation) supports import of UML state diagrams into the COSMA CXL format and sequence diagrams into temporal requirements to be evaluated.

The COSMA tool supports also the extended CSM model (ECSM, [25]). The extensions consist in defining conventional variables, attributing actions[2] to CSM states and/or transitions, using expressions over these variables as the additional propositions in Boolean formulas etc. Also, for the performance evaluation purposes, the random or constant times can be attributed to states and

[2]Practically – the sequential programs and functions in C.

for non-deterministic choices the probabilities can be determined. Of course, the Extended CSM are no longer finite-state models and are analyzed by the simulation rather than the model checking. This role plays *ECSM Simulator*.

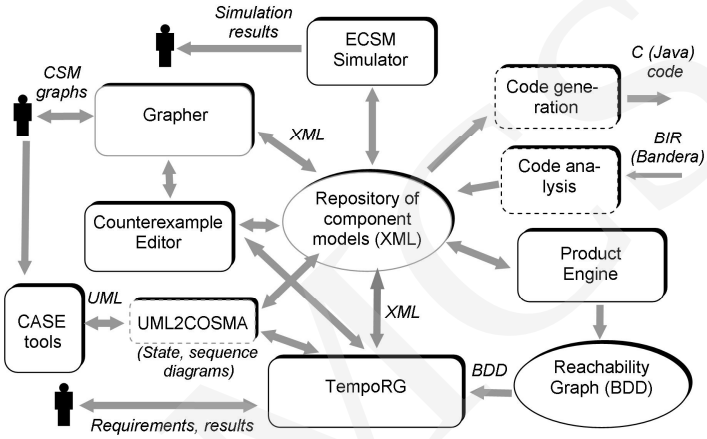

Fig. 4. The COSMA software environment

Two additional modules exist in an experimental form and are not included into the present version of COSMA. The *Code generator* was designed to generate the C code from ESCM specification, while *Code analyzer* was aimed to accept Java programs (in a BIR form, produced by Bandera [17,18] and to convert them into the CSM projects. The *Code analyzer* has been fully implemented [26], however, the results were rather discouraging. It seems that the CSM model scan be effectively obtained and model-checked only for a very limited subset of Java.

4. Conclusions

An attempt to maintain a bridge between the design practice and formal verification must involve the decision as to the place of the verification within the design process. Our experience shows that the most promising approach is the use of model checking at the early phase of a design. The verification of a coordination and communication among main concurrent components may help to identify and correct the coordination errors, which then do not propagate to consecutive design stages. Just verified general component models can serve as the templates for a more detailed implementation in a form of programs or pieces of hardware.

CSM model and COSMA offer a promising approach to the research in this field. Specification of components behavior in terms of Concurrent State Machines is reasonably intuitive and understandable to anyone familiar with

such basic notions as a state, a transition, a Boolean formula. Moreover, CSM support two aspects of concurrency: simultaneous occurrence of communication events (formally – symbols of the input alphabet) and simultaneous execution of component actions. No special mechanism for interleaving actions or sequencing the input is assumed. The system of CSM performs as if it was embedded in a communication medium which instantaneously and faultlessly broadcasts to all system components the set union of output symbols produced by the environment and components themselves. However, the delays, nondeterministic loss of symbols, (finite) buffers as well as the specific sender – receiver pairs (instead of broadcast-mode communication) can be also modeled, but as a deliberate designer's decision rather than as an implicit general assumption.

Further research of the CSM methodology and the COSMA tool would involve mainly:

- development of an effective tool for conversion of commonly known UML state, sequence, activity and cooperation diagrams to/from the CSM model,
- development of compositional model checking techniques, especially the multi-phase reduction method that helps relax the exponential explosion problem,
- introduction of time constraints to the CSM model (Timed CSM),
- use of the Extended CSM model as a tool for the refinement of systems of finite state CSM components into concurrent programs.

References

[1] *Formal Methods Europe*: http://www.fmeurope.org/
[2] http://archive.museophile.sbu.ac.uk/formal-methods.html
[3] *Unified Modeling Language*: www.omg.org/technology/documents/formal/uml.htm
[4] Budkowski S. et al., *The Estelle Development Toolset*, Institut National des Télécommunications, Evry, France, 1998, http://www-lor.int-evry.fr/edt.
[5] *Information Processing Systems. Estelle: A Formal Description Techinque Based on an Extended State Transition Model*, ISO/TC97/SC21, 1997.
[6] Peled D.A., *Software Reliability Methods*. Springer Verlag, (2001).
[7] Berard B., (ed.) et al., *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Verlag, (2001).
[8] Clarke E.M., Grumberg O., Peled D.A., *Model Checking*. MIT Press, (2000).
[9] McMillan K.L., *Symbolic Model Checking*. Kluwer Academic Publishers, (1993).
[10] Kropf T., *Introduction to Formal Hardware Verification*. Springer Verlag, (1999).
[11] Holzmann G.J., *The Model Checker SPIN*. IEEE Trans. on SE., 23(5) (1997) 279.
[12] *SPIN*: http://spinroot.com/spin/whatispin.html
[13] *SMV*: http://www-2.cs.cmu.edu/ modelcheck/smv.html
[14] *FormalCheck*: www.cadence.com/datasheets/formalcheck.html
[15] Harel D., *StateCharts: A visual formalism for complex systems*. Science of Computer Programming. (8) (1987) 231.
[16] Harel D. et al., STATEMATE: *A Working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering, 16(4) (1990) 403.
[17] *Bandera* www.bandera.projects.cis.ksu.edu/

[18] Hatcliff J., Dwyer M., *Using the Bandera tool set to model-check properties of concurrent Java software*. Proc. CONCUR 2001, (2001) 39.
[19] *COSMA*: www.ii.pw.edu.pl/cosma/
[20] Mieścicki J., *Concurrent State Machines, the formal framework for model-checkable systems*. ICS Research Report, 5 (2003).
[21] Mieścicki J., *Multi-phase model checking in the COSMA environment*. ICS Research Report, Warszawa, 14 (2003).
[22] Mieścicki J., Czejdo B., Daszczuk W.B., *Multi-phase model checking in the COSMA environment as a support for the design of pipelined processing*. ICS Research Report, Warszawa, 16 (2003).
[23] Daszczuk W.B., *Verification of temporal properties in concurrent systems*, Ph.D. thesis, Warsaw University of Technology, Faculty of Electronics and Information Technology, Warszawa, (2003).
[24] Daszczuk W.B., *Temporal model checking in the COSMA environment (the operation of TempoRG program)*. ICS Research Report, Warszawa, 7 (2003).
[25] Krystosik A.,: *ECSM – Extended Concurrent State Machines*. ICS Research Report, 2 (2003).
[26] Fusik P., *Model checking of concurrent Java programs using Bandera and COSMA environments*. M.Sc. thesis, Institute of Computer Science, WUT, (2004).