Verification of UML State Diagrams Using Concurrent State Machines

Jerzy Mieścicki

Institute of Computer Science, Warsaw University of Technology
00 665 Warszawa, ul. Nowowiejska 15/19
email: J.Miescicki@ii.pw.edu.pl

Abstract. Numerous research projects are done in academia as well as in industry aimed to support the design process based on UML and Model Driven Architecture with new methods and tools that would help to verify both static and dynamic aspects of UML model, to generate the code from it etc. Much attention is paid to the verification of system's behavior by model checking. In a research project done in the Institute of Computer Science, Warsaw University of Technology, an own model checking environment COSMA is used for these purposes. The approach is based on Concurrent State Machines (CSM), a finite state model well-suited to the representation of systems of concurrent, communicating components. In the paper, the representation of UML state diagrams in terms of CSM is explained and illustrated with an example.

1 Introduction

The progress in the area of new ideas and standards related to Unified Modeling Language (UML, e.g. [1], [2]) is accompanied with an extensive research aimed to support the designer with methods and tools for the verification of static as well as dynamic aspects of a designed system, for generation of code immediately from the UML specification etc. Among other topics, much attention is paid to the behavioral verification of UML models using *model checking* techniques.

The general idea of model checking ([3], [4]) is to construct a *finite-state* formal structure S, representing the behavior of a system to be verified (e.g. a Labeled Transition System, a reachability graph etc.). Then, the property we want to verify (π, say) has to be formally specified: e.g. as a formula of some temporal logic, or a B*ü*chi automaton [4]. Then, we have to check if $S \models \pi$, that is, if π holds for S. The evaluation of $S \models \pi$ involves the exhaustive inspection of S.

Notice that as S is finite, the evaluation of any (properly specified) property is decidable and can be algorithmized, at least if we postpone problems related to the size of S and to the complexity of algorithm. This way, the system designer can be equipped with a set of ready-to-use algorithms and techniques for the analysis of system's properties. Moreover, if the checked property does not hold, he/she can obtain a *counterexample*, i.e. the path of events 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. Unfortunately, finite state methods suffer also some drawbacks. Their very nature prohibits the use of infinite buffers,

Please use the following format when citing this chapter:

Mieścicki, J., 2006, in IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 261–271.

dynamic creation/destruction of processes makes a problem, etc., and the main challenge the model checking is confronted with is the exponential explosion of the model.

Practical implementation of the above general idea of model checking involves multiple decisions. Usually, systems consist of multiple components which share the common resources and communicate among themselves. How their behavior is to be specified and how these individual behaviors are to be composed into one, finite state behavioral model S ? How this model has to be stored, remembering that its size may be of order of $10^{20} - 10^{50}$ states or even more? What should be a form of specification of properties? How to perform effectively an exhaustive inspection of such an large model S? All these questions can be solved in many ways, so that there is a range of different software tools (or model checkers) designed for these purposes. Among the most frequently referenced ones are SPIN [5], SMV [6], FormalCheck [7] and - for checking systems with real-time constraints - Uppaal [8] and Kronos [9]. A few dozen of other tools of this type have been implemented for academic and research purposes.

In the context of the verification of UML models[1] the primary form of behavioral specification of objects are - quite naturally - UML state, collaboration and sequence diagrams, supported by practically all CASE tools. Since the first attempt by Lilius and Paltor (vUML tool, [11]), a most typical approach is the conversion of UML state diagrams (serialized into XMI format) into the input language of some renowned model checker (usually SPIN's Promela language). Later on, the verification itself is entrusted to the model checker. A good example can be project Hugo [14] [12] [13], where UML diagrams are converted into inputs of two separate model checkers (SPIN and Uppaal, the latter one for the verifiation of timed models) and - aditionally - to the third module which has to generate the Java code.

In contrast to this, in the research project COSMA (Institute of Computer Science, WUT, [15]) an original model checking software environment is used, implemented within the project. The conceptual framework of COSMA are Concurrent State Machines (CSM, [16]). CSM support the communication among system components as well as two aspects of concurrency: possible simultaneous occurrence of communication events (formally - symbols instantaneously broadcasted to all system components) and simultaneous execution of actions of components . No special mechanism for interleaving actions or sequencing the input is assumed. However, single symbols, communication delays, nondeterministic loss of symbols, (finite) buffers as well as specific sender - receiver pairs (instead of broadcast-mode communication) can be also modeled, but as a deliberate decision rather than as an implicit general assumption.

Below, in Section 3 we introduce the idea of a system of CSM and the way the behavior of individual machines is composed into one graph of all-system behavior. To support reader's intuition, the presentation of the CSM model is preceeded by a known example of ATM-Bank system (Section 2). Subsection 3.3 will be devoted to the process of verification, specifically - to the technique of stepwise model reduction [17] which at least helps to overcome the exponential explosion of the model. In Section 4 the main problems with the conversion of UML state diagrams into CSM are summarized.

[1] see e.g. [10] for concise identification of problems and the basic literature.

2 The ATM-Bank example

As an illustration, let us consider a system (frequently used also elsewhere in the literature), consisting of single automatic teller machine (ATM) and a bank computer. Simple UML state diagrams of system components are shown in Fig. 1 and 2. The ATM provides the interface to the User (not shown). ATM communicates to the User by displaying the following texts:

InsertCard, *EnterPIN*, *EnterAmount*, *TakeCard*, *TakeMoney*, *CardInvalid*

while the User is expected to respond appropriately with events like *Card* (inserting the card into ATM slot), *PIN* (entering PIN), *Amount* (entering the amount), *CardRemoved* (removing the card) or - finally - *Money* (signifying that the User gets the money from the machine).

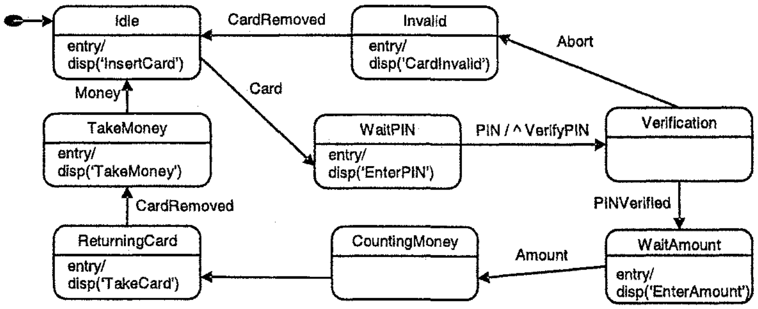


Fig. 1. State machine diagram for ATM

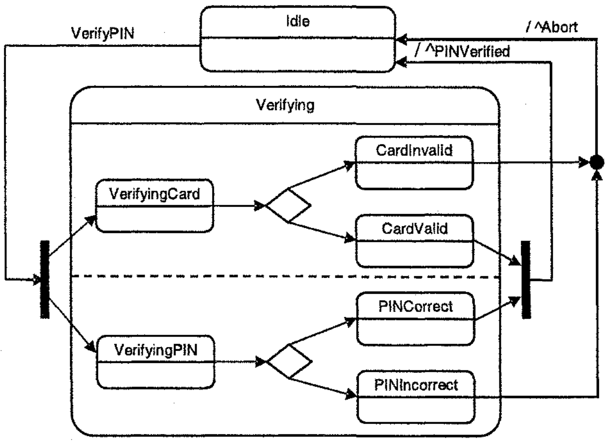


Fig. 2. State machine diagram for Bank

Just for an illustration, assume that we want to model-check the following properties:

- π_1: It is always so that whenever card is inserted then eventually that card is removed before the card can be inserted anew,
- π_2: It is always so that whenever card is inserted then eventually money is paid.

We expect, of course, that for the correct system the first requirement should be evaluated to *True*, while the latter one should be *False*.

3 Concurrent State Machines (CSM)

3.1 Definition of CSM

Let *AP* stand for an universal set of atomic propositions. From these atomic propositions, constants $\mathbf{0}, \mathbf{1}$, operators $!, +, *$ (Boolean negation, sum, product, respectively), and parentheses, we build Boolean formulas, obeying the well-known, conventional syntax and semantics. Let $\mathcal{BF}$ be an universal set of all Boolean formulas. The alphabet of formula f (denoted $\alpha(f)$) is the set of atomic propositions referred to in it[2]. Notice that $\alpha(\mathbf{1}) = \alpha(\mathbf{0}) = \emptyset$, as actually neither $\mathbf{0}$ nor $\mathbf{1}$ refer to any atomic proposition.

Formally, a Concurrent State Machine m is a tuple

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

where:

- N - finite set of nodes (states of behavior), $n_0 \in N$ is the initial node,
- $edges \subseteq N \times N$- set of directed arcs,
- $form : edges \rightarrow \mathcal{BF}$ - labeling function, attributing Boolean formulas to edges,
- $out : N \rightarrow 2^{AP}$ - output function, attributing to each node a set of atomic propositions $p \in AP$ that are *True* for this node,

It is convenient to think of CSM models as of labeled graphs (Fig. 3). Rounded boxes represent states, initial state is highlighted with a thicker borderline. In upper part of the box the state name is identified (e.g. *Idle*, *CardOK*, *Verifying*, *InvCard*) and below a set of propositions that are *True* for this state is enumerated (so-called *output set* of a given state). Directed edges of the CSM graph define the next-state relation. Edges are labeled with Boolean formulas rather than with individual symbols from some input alphabet. We require that a machine has to be *complete*, i.e. for any state, the Bolean sum of formulas at outgoing edges equals $\mathbf{1}$.

In the context of behavior modeling we usually understand the atomic propositions as the communication symbols (signals, messages etc.) produced by the machine in a given state as its output and received (or "watched for") as its input. Machine's *output alphabet* (denoted $Out(m)$) is the union of output sets of states. For instance, for machine from Fig. 3, the output alphabet is:

$$Out(BankMain) = \{PINVerif, verCompl, doVerif, Abort\} \tag{1}$$

[2] We require the formulas $f \in \mathcal{BF}$ be 'minimal', in a sense that their alphabets are minimal. So, for instance, $\mathbf{1}$ is used instead of $(a+!a)$, a instead of $(a * b + a*!b)$ etc.

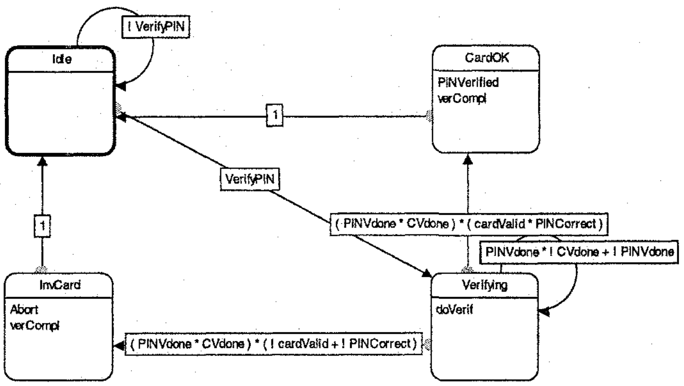

! VerifyPIN
Idle
CardOK
PINVerified
verCompl
1
VerifyPIN
1
(PINVdone * CVdone) * (cardValid * PINCorrect)
PINVdone * ! CVdone + ! PINVdone
InvCard
Abort
verCompl
Verifying
doVerif
(PINVdone * CVdone) * (! cardValid + ! PINCorrect)

Fig. 3. Example Concurrent State Machine (BankMain)

In the CSM framework it is assumed that the truth value of *all* propositions from the output alphabet of the machine are fully determined by the output function of the (present) state. In other words, we assume that as long as the machine is in state $n \in N$, all propositions $p \in out(n)$ are *True* while all the other ones from machine's output alphabet (i.e. $q \in Out(m) - out(n)$) are *False*. This can be represented by the *state output formula* (denoted $\varphi(n)$) which is *True* for state n. For instance, for state *InvCard* (Fig. 3) - the state output formula is:

$$\varphi(InvCard) = !PINVerif * verCompl * !doVerif * Abort \qquad (2)$$

as $out(InvCard) = \{Abort, verCompl\}$, etc. We say that in state *InvCard* the machine 'produces' two output symbols: *Abort* and *verCompl*, in *Verifying* - one symbol (*doVerif*), while in *Idle* no output symbol is produced.

Similarly, the *input alphabet* of machine m (denoted $Inp(m)$) is the union of alphabets of all edge formulas. Any proposition $p \in Inp(m)$, when *True*, signifies that the symbol p (signal, message, ...) is present in machine's input. For instance, for machine from Fig. 3:

$$Inp(BankMain) = \{VerifyPIN, PINVdone, CVdone, cardValid, PINCorrect\} \qquad (3)$$

Notice that it is not required that input and output alphabets have to be disjoint.

The next-state semantics of machine's behavior is as follows. At any instant of time the machine is in exactly one of its states; initially - in the initial state. In any (present) state n, machine produces its output symbols (making some atomic propositions *True* and the other ones *False*) and simultaneously evaluates the formulas on the edges outgoing from n. If a formula is *True*, then its edge is *enabled*[3]. If only one edge is enabled (deterministic case) - it becomes *active*. If more than one edge is enabled then one of them is selected as active. The choice is nondeterministic and fair[4]. If the

[3] Due to completeness, there is always at least one enabled edge.

[4] Of course, the next-state semantics refers to a single execution of the machine. However, in the context of model checking, all the edges that are enabled in a given state point out to *reachable* states. Thus, the reachability graph of the machine (as well as of the whole system, se below) contains all the edges which are labeled with non-zero formulas.

selected active edge (n,n') points out to a state $n' \neq n$ (different than the present one) then the machine executes the transition to n'. Transition is instantaneous (zero time). Otherwise, i.e. if $n' = n$ - machine remains in n. Notice that formula $\mathbf{1}$ is always *True*, so the edges (n,n') (where $n' \neq n$) labeled with it represent spontaneous transitions, executable regardless of machine's input. Similarly, formula $\mathbf{0}$ would mean that the edge is never enabled: such an edge can be simply removed from the graph.

3.2 System of CSM and its product

Now, consider a finite (nonempty) set M of Concurrent State Machines. For any two machines $m_i, m_j \in M$, if $Out(m_i) \cap Inp(m_j) \neq \emptyset$ then there is a communication from m_i to m_j (m_i and m_j are 'communication partners'). If $Inp(m_i) \cap Inp(m_j) \neq \emptyset$ then the two machines share the same input. A set M of CSM is a *system* of CSM, iff either $|M| = 1$ (one-component system) or any $m \in M$ has at least one communication partner or shares the input with at least one other machine.

The overall output alphabet of system M (denoted $OUT(M)$) is the union of output alphabets of all $m \in M$. Similarly, the input alphabet of M ($INP(M)$) is the union of input alphabets of all $m \in M$. The set difference $E(M) = INP(M) - OUT(M)$ is the set of atomic propositions which are inputs of machines $m \in M$ but are not produced inside the system. We assume that these symbols $p \in E(M)$ come from an unknown environment of system M and at any instant of time they can be either *True* or *False*[5].

The global behavior of a system of CSM is represented by system's reachability graph RG. The algorithm of obtaining RG has been developed and implemented as one of modules of COSMA environment [16]. Its idea is as follows. The state of the system is a vector of states of system components. Algorithm starts from system initial state which is the vector of initial states of components. In a given system state, system produces the *set union* of outputs of components. As the system output alphabet $OUT(M)$ is known, for any system state $\bar{n}$ the state output formula $\varphi(\bar{n})$ is determined, analogously as in Eq. 2. From state $\bar{n}$, a set of states is *hypothetically* immediately reachable. The hypothetical edge that would lead from $\bar{n}$ to some $\bar{n}'$ should be labeled with the Boolean product of $\varphi(\bar{n})$ and the product of appropriate edge formulas of individual system components. If this product equals $\mathbf{0}$, then the state (although it was *hypothetically* reachable) proves not to be *actually* reachable and is not included into the emerging graph. Otherwise, the state is included and the edge with an appropriate labeling formula is created [6]. The process continues until no new reachable states emerge.

The resulting graph is again a single CSM called a *product* of machines. The product is commutative and associative, which supports the compositionality of the model.

The overall organization of the example system from Section 2 is shown in Fig. 4. It consits of two subsystems (ATM and Bank), where ATM is a single CSM (Fig. 5) and Bank itself is composed from three components: Bank-Main (Fig. 3)

[5] Notice that by the above definition the alphabet of propositions coming from the environment and produced inside M are disjoint.

[6] It should be emphasized that the propositions $p \in OUT(M)$ are eliminated from these formulas. Indeed, for any system state $\bar{n}$ the truth value of all output propositions is known so that we can substitute $\mathbf{0}$ for propositions that are *False* in this particular state and $\mathbf{1}$ otherwise.

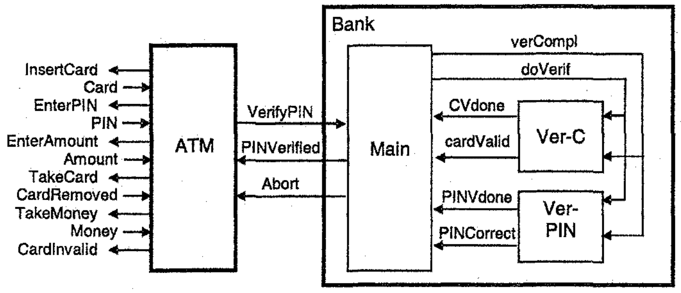


Fig. 4. Structural block diagram of the example system

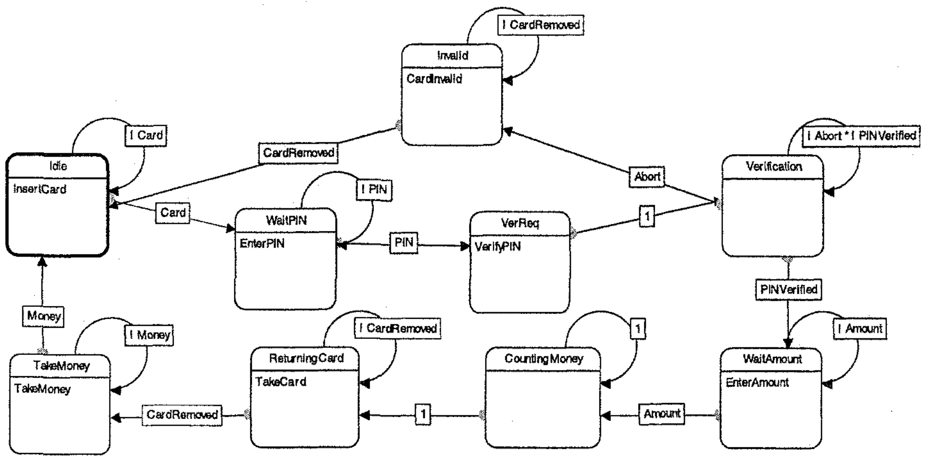


Fig. 5. CSM model of ATM

and two orthogonal machines, one for the verification of the card, the other for the verification of PIN (Fig. 6). Directed arrows in the block diagram from Fig. 4 indicate the communication between machines: for instance, *VerifyPIN* is the output symbol from ATM and input proposition for BankMain, etc. These communication relationships can be easily specified in terms of intersections of input/output alphabets.

Additionally, we prepare the CSM model of expected behavior of the User (not shown for the sake of the economy of space). It has 10 states and 16 edges and generally is analogous to the ATM (Fig. 5) to/from which it communicates. The CSM product of the whole system is a new machine:

$$System = User \otimes ATM \otimes BankMain \otimes VerC \otimes VerPIN \tag{4}$$

It has as few as 28 (reachable) states (out of $10 \times 9 \times 4 \times 4 \times 4 = 5760$ elements of Cartesian product of sets of components' states) and 46 labeled edges.

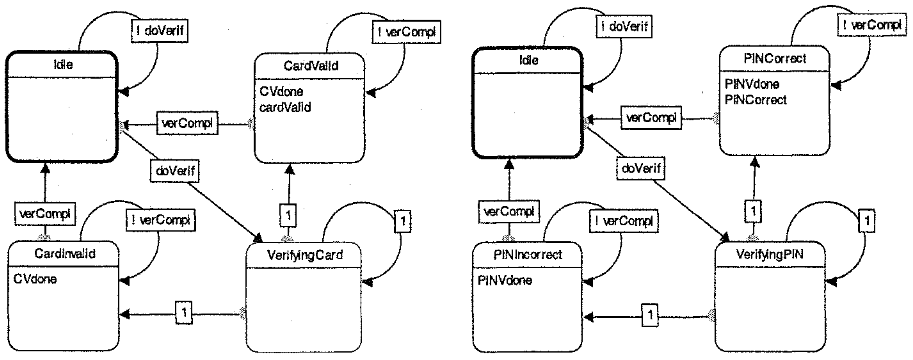


Fig. 6. CSM models of Ver-C (left) and Ver-PIN (right)

3.3 Multi-phase computation of CSM product

Notice that due to the associativity of CSM product, we can obtain the *System* in several steps instead of the one ('flat') operation, as in Eq. 4. For instance, we can compute *System* as a sequence of partial products:

$$System = User \otimes (ATM \otimes (BankMain \otimes VerC \otimes VerPIN)), or \quad (5)$$

$$Bank = BankMain \otimes VerC \otimes VerPIN \quad (6)$$

$$ATMandBank = ATM \otimes Bank \quad (7)$$

$$System = User \otimes ATMandBank \quad (8)$$

However, if we know what properties are to be verified, we can significantly reduce the partial products before they are used in the next step of product computation. In our example we want to verify the properties π_1, π_2, specified at the end of Section 2. They refer only to propositions *Card*, *CardRemoved* and *Money* (in the interface between User and ATM).

Now, suppose that we have just computed the partial product *Bank* (as in Eq. 6. Actually, it has 15 states and 32 edges. However, from the viewpoint of the next step (Eq. 7) the only relevant states are the ones which either produce or receive symbols to/from ATM, i.e. *Verify*, *PINVerified*, *Abort* (easily identifiable in the block diagram from Fig. 4). Remaining (irrelevant) states and edges can be merged in order to obtain compressed, much smaller version of the partial product. The algorithm for partial product compression (given a set of relevant symbols) has been implemented as a part of COSMA environment. The result of its application to *Bank* (or *NewBank*) is shown in Fig. 7[7]. Notice that *NewBank* has only 4 states and 7 edges (compared with 15/32 of the 'original' *Bank*).

The same procedure can be continued with successive subproducts. We substitute *NewBank* instead of *Bank* in Eq. 7, compute *ATMandBank*, compress again the resulting product into *NewATMandBank* (leaving as relevant symbols only these from ATM-User

[7] The algorithm attributes new, technical identifiers to merged states

interface). Finally, we compute and compress (*NewSystem* = *User* $\otimes$ *NewATMandBank*). This time, compression involves hiding all propositions except *Card*, *CardRemoved* and *Money*, (necessary and sufficient) for the evaluation of π_1 and π_2. The result, shown in Fig. 8, is so elementary that one can analyze it just by naked eye. Indeed, the graph shows that *NewSystem*1 $\models \pi_1$ (it is true that whenever *Card* is inserted then eventually *CardRemoved*) while *NewSystem*1 $\not\models \pi_2$ (it is not true that whenever *Card* is inserted then eventually *Money* is paid).

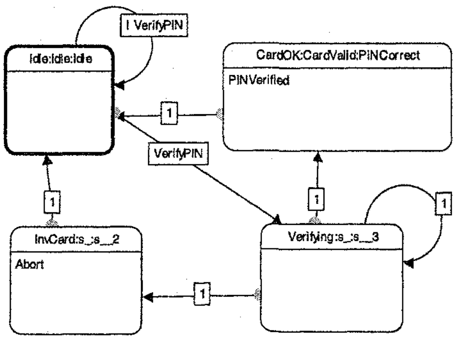
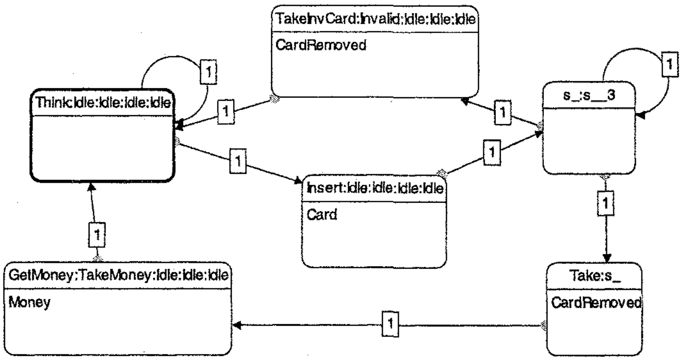


Fig. 7. *NewBank* or compressed product $\otimes\{BankMain, VerC, VerPIN\}$



TakeInvCard:Invalid:Idle:Idle:Idle
CardRemoved
Think:Idle:Idle:Idle:Idle
1
1
1
s_:s__3
1
1
Insert:Idle:Idle:Idle:Idle
Card
1
1
GetMoney:TakeMoney:Idle:Idle:Idle
Money
Take:s_
CardRemoved
1

Fig. 8. *NewSystem* or compressed product for the evaluation of π_1 and π_2

It should be mentioned that in the case of larger (also: uncompressed) graphs the verified properties are expressed as formulas in QsCTL (a version of CTL) and evaluated using one of modules of COSMA environment ([18]), with possible edition of counterexamples etc.

4 Conversion of UML state diagrams into CSM

The example discussed above shows that the CSM model and COSMA tool is a noteworthy conceptual framework for behavioral verification of systems. Multi-phase product computation and compression of partial products seem to be an important advantage, as a powerful technique that can overcome (or relax, at least) the exponential model explosion and provide readable evaluation results. However, if such COSMA-style model checking has to support MDA approach - we should have algorithms and tools for converting UML state diagrams into Concurent State Machines. The software module for converting UML state diagrams (from their XMI specification) into CSM is now under implementation within the COSMA project. Below, we briefly comment on main problems encountered during the implementation. Unfortunately, the results of algorithmic conversion are hardly readable in practice, so that the CSM models of ATM and Bank discused in preceding sections have been prepared manually, just to provide illustrative examples highlighting the nature of CSM model.

First of all, CSM are best tailored to modeling of *control-dominated* systems. Simple types of data (not only boolean, but also short integers, like counters etc.) are acceptable, but may significantly increase the size of product. Also, infinite buffers are excluded and finite ones have to be modeled as separate machines, which may lead to a substantial complication of the model. Dynamic object creation/destruction also contradicts the finite-state nature of the CSM model. On the other hand, the same limitation face practically all finite state methods and model checkers. Moreover, it should be mentioned that the COSMA environment supports also Extended CSM (ECSM, [19]), which allow for the definition of all types of variables and attributing the pieces of C/C++ code to states and transitions of CSM. Of course, systems of ECSM are no longer model-checkable: they can be either simulated or excuted, but we can verify their control- and communication flow 'skeletons' before the code is added.

Conversion of "flat" UML diagrams, like the ATM from Fig. 1, is rather a simple task (compare Fig. 5). However, in CSM the outputs are attributed to states (like in Moore automata) rather than to transitions (like in Mealy automata and state diagrams), therefore in order to produce *Verify* message to Bank the additional CSM state is introduced (*VerReq*). The "self-loops" at CSM states (making the conditions of staying in states explicit) are merely a technical trick.

Composite states (like the AND-state *Verifying* in Fig. 2) cause more problems. First, not only the diagram itself (here: BankMain), but also each of nested subdiagrams (Ver-C and Ver-PIN) must be separate CSM. If so, Ver-C and Ver-PIN have to remain in *some* CSM state even though a higher-level diagram (BankMain) had just returned to *Idle*. Generally, if the composite state can be entered through H or H* pseudostates, then upon exit from this (UML) state all the nested machines have to remain "frozen" in their present (CSM) states. If for the subdiagram the default initial state is specified - then the same trigger which pulls off the higher-level diagram from (UML) composite state forces all the nested sub-machines to get back to their initial (CSM) states. This calls for additional (appropriately labeled) edges in CSMs, from each state back to the initial one. Moreover, in order to keep the sub-machines frozen while the higher-level machine is not in "their" composite state, to each composite state a default technical output symbol is attributed (not provided by the designer at UML level) which multiplies (in a sense of Boolean product) all the formulas at the transitions in its sub-machines. This way these transitions are temporarily disabled. It is the above conventions why algorithmically generated CSM models are hardly readable.

Fortunately, the mentioned technical symbols can be easily hidden during compression and do not influence the readability of final evaluation results.

Among other problems is the conversion of other pseudostates, like Fork - Join bars as well as junction and branch pseudostates. They involve a specific exchange of synchronization symbols among sub-machines, but still can be rather naturally modeled in terms of CSM (see Fig. 3 and 6). Notice that for a subsystem of CSM, aimed to represent a nested composite state we can compute a local CSM product, as we did e.g. for Bank (Eq. 6). This operation "flattens" the behavioral specification and helps to understand the details of cooperation among machines.

The most challenging problem for the COSMA project is now the introduction of real-time constraints to CSM. In this paper we have used just a basic version of the CSM model, where the the only representation of the flow of time are states, in which a machine can nondeterministically remain for an unspecified but finite time (e.g. *CountingMoney* in Fig. 5 or *VerifyingCard*, *VerifyingPIN* in Fig. 6). The research on the theory and implementation of Timed CSM is in progress.

References

1. *Unified Modeling Language*: www.omg.org/technology/documents/formal/uml.htm,
2. B. P. Douglass: *Advances in the UML for Real-Time Systems*, The Addison-Wesley object technology series, 2004.
3. B. Berard (ed.) et al.: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer Verlag, 2001,
4. E. M. Clarke, O. Grumberg, D. A. Peled: *Model Checking*, MIT Press, 2000.
5. SPIN: http://spinroot.com/spin/
6. SMV: http://www-2.cs.cmu.edu/ modelcheck/smv.html
7. FormalCheck: www.cadence.com/datasheets/formalcheck.html
8. Uppaal: http://www.uppaal.com/
9. Kronos: http://www-verimag.imag.fr/TEMPORISE/kronos/
10. M. Gallardo, P. Merino, E. Pimentelis: Debugging UML Designs with Model Checking, *Journal of Object Technology*, vol. 1, no. 2, July-August 2002, pp. 101-117.
11. J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering*, IEEE Press, 1999.
12. T. Schafer, A. Knapp, S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001.
13. A. Knapp, S. Merz, Ch. Rauh, Model Checking Timed UML State Machines and Collaborations, W. Damm and E.-R. Olderog (Eds.): *FTRTFT 2002*, LNCS 2469, pp. 395-414, Springer-Verlag, 2002.
14. Project Hugo: http://www.pst.informatik.uni-muenchen.de/projekte/hugo/
15. *COSMA*: www.ii.pw.edu.pl/cosma/
16. J. Mieścicki: Concurrent State Machines, the formal framework for model-checkable systems, *ICS Research Report*, 5/2003,
17. J. Mieścicki, B. Czejdo, W. B. Daszczuk: Multi-phase model checking in the COSMA environment as a support for the design of pipelined processing. *Proc. European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS 2004*, Jyväskylä, Finland, 24-28 July 2004.
18. W. B. Daszczuk: Temporal model checking in the COSMA environment (the operation of TempoRG program). *ICS Research Report*, 7/2003, Warszawa, 2003.
19. A. Krystosik: ECSM - Extended Concurrent State Machines. *ICS Research Report* 2/2003,