An Automatic Approach to Model Checking UML State Machines

Shao Jie Zhang
NUS Graduate School for Integrative Sciences and Engineering
National University of Singapore
Singapore
shaojiezhang@nus.edu.sg

Yang Liu
School of Computing
National University of Singapore
Singapore
liuyang@comp.nus.edu.sg

***Abstract*—UML has become the dominant modeling language in software engineering arena. In order to reduce cost induced by design issues, it is crucial to detect model-level errors in the initial phase of software development. In this paper, we focus on the formal verification of dynamic behavior of UML diagrams. We present an approach to automatically verifying models composed of UML state machines. Our approach is to translate UML models to the input language of our home grown model checker PAT in such a way as to be transparent for users. Compared to previous efforts, our approach supports a more complete subset of state machine including fork, join, history and submachine features. It alleviates the state explosion problem by limiting the use of auxiliary variables. Additionally, this approach allows to check safety/liveness properties (with various fairness assumptions), trace refinement relationships and so on with the help of PAT.**

***Keywords*-UML State Machines; Model Checking; PAT**

I. Introduction

The Unified Modeling Language (UML) [16] has been known as the de facto standard of software modeling language. However, the deficiency of precise and complete semantics, especially for dynamic behavior, impedes the ability to guarantee the correctness of UML models, which has raised many concerns on integrating formal methods with the verification.

Model checking [3] is an automatic technique to verify whether a finite state model satisfies desirable properties by exhaustively exploring the state space of the model. It has been proved as a promising and effective formal approach to the verification of hardware and software systems during the last two decades.

UML state machines are used to describe the behavior and interaction of system components. It depicts the various states that an object may be in and the transitions between those states. This paper presents a translation approach to verifying UML state machines. In particular, this approach utilizes a home grown model checker PAT[1] [18] as the back end to take advantage of its powerful verification capabilities [19], [20]. PAT also supports a wide range of modeling languages including CSP# (short for communicating sequential programs), which shares similar design principle with integrated specification languages like TCOZ [12], [11]. Furthermore, our solution is fully automatic with no intervention on the user's part. Our engineering efforts realize this approach into a prototype tool under the PAT framework. This tool takes XML Metadata Interchange (XMI), the Object Management Goup standard of exchanging UML diagrams, as the input format of UML specification, which makes our tool independent of any modeling tools. Figure 1 shows the work flow of our approach.

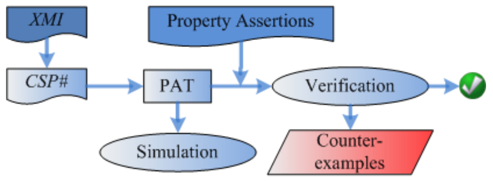

Figure 1. Work Flow of Our Approach

Our contributions are three-folds. Firstly, our approach supports a larger subset of UML state machines than most other works [1], including join, fork, history pseudo states, entry and exit points, which are commonly excluded or not well handled by previous works.

Secondly, shared variables are extensively used to represent every state and event occurrence in state-based intermediate model, which arises the infamous state space explosion problem and hence degrades the performance of model checking. This approach alleviates this problem in the way that system behavior is directly specified in terms of processes and events, yet states are not explicitly represented.

Lastly, the approach enjoys considerable benefits from the model checker PAT. Its simulator allows users to perform various simulation mechanisms on the input model: complete states generation, automatically random simulation, user interactive simulation, trace replay and so on. On the other hand, its verifier enables users to check deadlock, reachability, trace refinement relationship [18], linear temporal logic (LTL) properties with various fairness assumptions [19] and etc.

The rest of the paper is organized as follows: related work is reviewed and discussed in Section II. In Section III, we

[1]available at http://www.patroot.com

present an approach to translating UML state machines to CSP#. Then we demonstrate how our approach works on a case study in Section IV. Section V summarizes this work along with possible future work.

II. Related Work

Many approaches of applying the model checking technique to UML state machines have been proposed in the literature, for example, a comprehensive survey is given in [1]. Generally, these approaches translate UML specification into an intermediate model of some well-known model checker, such as SMV [13], SPIN [7] and FDR [17]. In the following we categorize them based on the model checkers the use, and compare our work to the most relevant works in this area.

The first work based on SPIN model checker is presented in [9]. It presents a translation scheme for UML state diagrams into a Promela model and then invokes SPIN for verification. This work is constrained within the basic elements in state diagrams. Advanced modeling technique such as fork, join, history states, entry and exit behavior of states, variables and multiple state machines are not considered. [10] presents a tool called vUML which transforms a UML state machine to a Promela model. The technical details are not explained in [10]. Clarke and Heinle propose an approach to translating statecharts to the input language of symbolic model checker SMV in [4]. They map every state or event to a single variable. Inter-level transitions are excluded in their works. [5] defines a semantics and a symbolic encoding of UML state machines, and performs verification on NuSMV [2] model checker. They support deferring of messages, concurrent composite states and choice pseudo states.

Closely related to our work is that of [15], [14], [21], which all utilizes CSP as the input language of the intermediate model for model checking. Due to the event driven feature of CSP, the fundamental principle that [15], [14], [21] and our work conform to is encoding an individual state as a CSP process and an event occurrence as a CSP event. Ng and Butler in [15] present a tool to translate UML state diagrams into a classic CSP model. At the same time, they adopt class diagrams to explicitly specify the refinement relation of different CSP processes. Then the refinement relation of CSP models is verified in FDR model checker. In [14], they give a formalization of their approach and extend it to addressing inter-level transitions, implicit and explicit triggered transitions, interruption on on-going activities as well as entry and exit actions.

Yeung et al improve the above-mentioned formalization in [21]. They enhance the definitions and mappings so as to generalize inter-level transition targets to nested targets and meanwhile support prioritizing transitions. Our approach improves their method by formalizing multiple nested sources and targets, i.e. join and fork pseudo states. Entry and exit points are supported for the first time to handle the encapsulation and composition of state machines. History states and guard conditions are included in our works as well. Besides refinement checking, the model checker PAT enables us to check more kinds of properties.

[8] proposes a probabilistic extension of UML state machines. Desired properties are specified in probabilistic logic PCTL, and verified using the model checker Prism [6].

III. Our Approach

In this section, we first give a brief introduction to CSP# modeling language and elements of UML state machine. Then we illustrate the translation rules from state machines to CSP#.

A. CSP#

Hoare's CSP is a formal language for describing patterns of interaction in concurrent systems [17]. CSP# inherits most of its high-level concurrency operators and extends with programmer-favored low-level constructs including shared variables and conditional choice.

In the following, We list out the subset of CSP# that is considered in our work and discuss briefly the semantics for each one of them.

Definition A process P is defined using the grammar:

$$\begin{aligned} P ::= {} & Stop \mid Skip \mid e\{prog\} \rightarrow P \mid P_1;\ P_2 \mid P_1 \Box P_2 \\ & \mid P_1 \mid\mid\mid P_2 \mid P_1 \parallel P_2 \mid [b]P \mid atomic\{P\} \\ & \mid P_1 \bigtriangleup P_2 \mid ch!exp \rightarrow P \mid ch?x \rightarrow P \\ & \mid case\{b1 : P_1;\ b2 : P_2;\ \cdots;\ default : P\} \\ & \mid e ::= name(.exp)* \end{aligned}$$

where P, P_1, P_2 are processes, e is a name representing an event with an optional sequential program *prog*, b, b_1, b_2 are boolean expressions, *ch* is a channel, *exp* is an expression, and x is a variable.

Stop is the deadlock process, while *Skip* represents the process that terminates successfully. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a program such as updating shared variables, the program is executed atomically together with the occurrence of the event. Sequential composition, $P_1;\ P_2$, behaves as P_1 until its termination and then behaves as P_2. The choice $P_1 \Box P_2$ is nondeterministically solved by the occurrence of an event. Parallel composition of processes $P_1 \parallel P_2$ synchronizes common events in the alphabets of both processes. Interleaving process $P_1 \mid\mid\mid P_2$ runs processes independently except for communication through shared variables. Guarded process $[b]P$ executes only when its guard condition b is satisfied. Process $P_1 \bigtriangleup P_2$ behaves as P_1 until the occurrence of the first event from P_2. Regarding case process, the condition is evaluated one by one until a true one is found and then the corresponding process executes. In case no condition is true, the default process will execute. A process may have parameters, and an event may be in a compound form composed of variables.

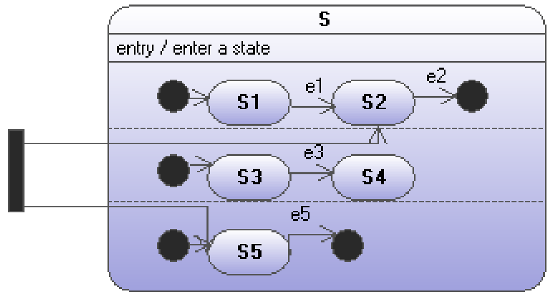

Figure 2. An Example with Fork Pseudo State

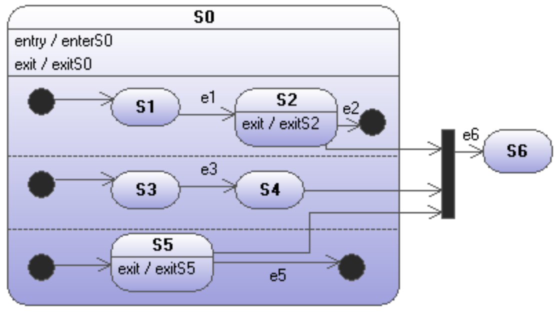

Figure 3. An Example with Join Pseudo State

B. UML State Machine

A state machine describes the lifetime of a single object. It contains states and transitions between them.The complete syntax and graphical notations are presented in [16].

A state models a situation during which some invariant condition holds. It may have three kinds of optional behavior: *entry*/*exit* behavior is a sequence of actions always executing to completion whenever this state is entered /exited; *DoActivity* behavior is executed while being in the state. It starts after *entry* and before *exit*, and can be interrupted by a triggered outgoing transition before completing its own actions.

A state is distinguished as *simple*, *composite* or *submachine*. A *simple* state does not have nested states. A *composite* one has one or more orthogonal regions, each of which contains substates. A *submachine* state specifies the insertion of a state machine. Transitions in the containing state machine use *entry*/*exit points* of the contained state machine as targets/sources.

Besides the source and target states, a transition features an assortment of *trigger*, *guard* and *effect*. Event is used as a *trigger* to activate the transition and can be parameterized to exchange data. A transition can be fired only if the event is dispatched and *guard*, a side effect free boolean expression, is satisfied. Then *effect* behavior is performed during the transition.

In addition, UML state machine offers various pseudo states to facilitate modeling complex behavior. For instance, *join*, *fork* and *choice* pseudo states specify the transitions originating from multiple sources or targeting multiple targets; *initial*, *final* and *history* ones record special moments during the lifetime of an object.

C. Translation Rules

To generate a CSP# specification from UML state machines, we define a series of translation rules assuming the state machines are well-formed. Basically, a state maintains a one-to-one mapping with a process, both of which stand for the behavior pattern of an entity when it interacts with the environment. An event or action naturally corresponds to a CSP event. For convenience, an informal function $f : UML \rightarrow CSP\#$ is defined to illustrate the fundamental mapping rules as Table I shows. In the following, these rules are used to translate advanced modeling behavior involving composite and submachine states.

Fork state deals with the transition from a single source state to several substates in different regions of a composite state. When a transition from a fork state is fired, control passes to all the target states. If one or more regions have no target, then the initial states of all the other regions are implicitly chosen as the targets. The translation rule for fork is more involved than the previous ones. Generally, it reuses the rule for transitions between states at the same level by lifting the fork transition to the composite state, with annotations to describe the actual target, so that the target states are always under the influence of the composite state's own entry behavior and outgoing transitions. In detail, an *n*-parameterized process is used to represent the composite state with fork transitions, in which *n* is the number of regions and every parameter denotes which state in a region is about to be activated. Therefore, a fork state specifies the targets by evaluating parameters, and every parameter is set to zero by default to represent initial state for implicit entry. For example, the fork transition in Figure 2 is translated as follows:

$$\begin{array}{ll} P_S(i,j,k)^2 & = enter\_a\_state \rightarrow \\ & \qquad (P_{r1}(i) \;|||\; P_{r2}(j)) \;|||\; P_{r3}(k)); \\ P_{Fork} & = P_S(2,0,1); \\ \ldots & \\ P_{r2}(i) & = case\{(i == 1) : P_{s3};\ (i == 2) : P_{s4}; \\ & \qquad default : P_{initial2}\}; \end{array}$$

Join state conversely specifies the transition from substates in different regions of a composite state to a target state outside the composite state. A join transition is effective only if all the source states are active. If triggered, it results in all the active substates of the composite state executing their exit behavior starting with the innermost states. Inter-level transition is a special case of a fork/join transition with

[2] P_S denotes the CSP# process representing the state, region or state machine S.

Table I
BASIC TRANSLATION RULES FROM UML TO CSP#

UML	CSP#	Comment
System *s*	$f(s) = f(sm_1) \mid\mid\mid f(sm_2) \mid\mid\mid \cdots \mid\mid\mid f(sm_n)$ where sm_1, $sm_2, \cdots$, sm_n are state machines.	Replace interleave operator with parallel if broadcast events are defined.
State Machine *sm*	$f(sm) = f(i)$ where i is the topmost initial state of sm.	Same for a region in composite state.
Initial State *i*	$f(i) = f(t_1) \Box f(t_2) \Box \cdots \Box f(t_n)$ where t_1, $t_2, \cdots$, t_n are outgoing transitions from i.	If there is only one transition without guard and effect, skip to the target.
Final State *i*	$f(i) = Skip;$	
Transition *trans*	$f(trans) = [guard](event \rightarrow (f(exit);\ f(effect);\ f(t)))$ where $exit$ is exit behavior of source state and t is target state.	Use compound event if *event* is parameterized
Effect *effect*	$f(effect) = atomic\{e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n\}$ where e_1, $e_2, \cdots, e_n$ is a sequence of actions.	Same for entry and exit behavior
Simple state *s*	$f(s) = f(entry);\ f(doActivity) \bigtriangleup (f(t_1) \Box f(t_2) \Box \cdots \Box f(t_n)))$ where t_1, $t_2, \cdots$, t_n are outgoing transitions from s.	
DoActivity *do*	$f(do) = e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$ where e_1, e_2, $\cdots$, e_n is a sequence of actions.	
Composite state *cs*	$f(cs) = f(entry);\ (f(r_1) \mid\mid\mid f(r_2) \cdots \mid\mid\mid f(r_n) \mid\mid\mid f(doActivity)) \bigtriangleup (f(t_1) \Box f(t_2) \Box \cdots \Box f(t_n)))$ where t_1, $t_2, \cdots$, t_n are outgoing transitions and r_1, $r_2, \cdots$, r_n are regions in cs.	Replace interleave operator with parallel if synchronization events are defined.
Submachine state *ss*	$f(ss) = f(sm)$ where sm is the state machine that ss refers to.	

single target/source. Regarding the translation, a common event is added to force the exit behavior execution of all source states in synchronization, such as the event *join* for the example in Figure 3.

$$\begin{aligned}
P_S(i,j,k) &= entryS0 \rightarrow (P_{r1}(i) \parallel P_{r2}(j)) \parallel P_{r3}(k));\\
P_{s2} &= (e2 \rightarrow exitS2 \rightarrow Skip) \Box\\
&\quad (join \rightarrow exitS2 \rightarrow exitS0 \rightarrow P_{join});\\
P_{s4} &= join \rightarrow exitS0 \rightarrow P_{join};\\
P_{s5} &= (e5 \rightarrow exitS5 \rightarrow Skip) \Box\\
&\quad (join \rightarrow exitS5 \rightarrow exitS0 \rightarrow P_{join});\\
P_{join} &= e6 \rightarrow P_{S6};
\end{aligned}$$

Entry/Exit point is the entry/exit point of a state machine referred by a submachine state. A referred state machine is behaviorally analogous to a subroutine: a transition to an entry point corresponds to a call to a subroutine; transitions in the referred state machine correspond to changes of the subroutine's internal states; a transition to an exit point corresponds to transferring control to the containing state machine. This behavior of subroutine can be described as synchronous channel communication that specifies the transfer of control between state machines. As for the translation of the example in Figure 4, a channel *ch* is defined to carry two constants representing synchronous events: 0 is for the transition to the entry point *active* and 1 is for the one from the exit point *aborted*.

$$\begin{aligned}
P_{S1} &= e1 \rightarrow ch!0 \rightarrow Skip;\\
P_{S2} &= ch?1 \rightarrow P_{S3};\\
P_{SM2} &= ch?0 \rightarrow starting \rightarrow P_{S4};\\
P_{S4} &= abort \rightarrow ch!1 \rightarrow Skip;
\end{aligned}$$

History state adds "memory"to composite state by recording the last substate that was active prior to a transition from the composite state. As for its translation, an integer shared variable is used to record which substate is currently active. So when a transition jumps to the history state, it could find the last active substate according to the value of this variable. More detail is shown in the following case study.

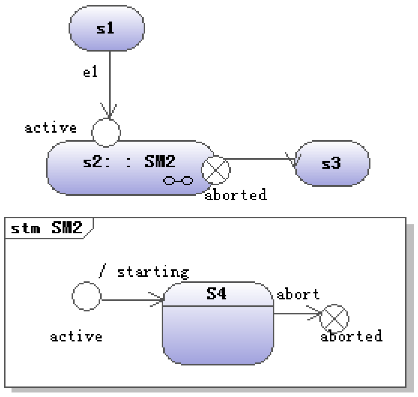

Figure 4. An Example with a Submachine State

IV. CASE STUDY

In this section, we demonstrate how to apply the translation approach developed in Section III using a CD player case study.

Figure 5 shows a UML state machine diagram modeling a CD player. Initially, the player stays at composite state *NONPLAYING*. Thus, the whole process starts at *NONPLAYING*.

$$CDPLAYER() = NONPLAYING(0);$$

The composite state *NONPLAYING* starts at *CLOSED* state, which denotes the CD drawer is closed. When a user

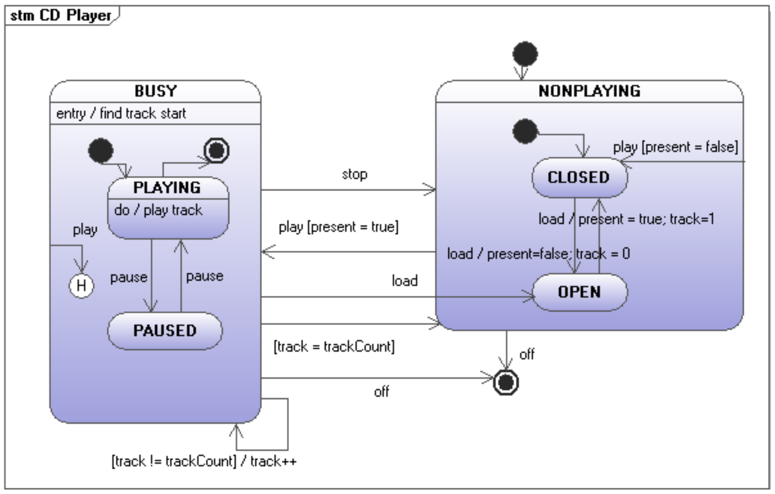

Figure 5. State Machine Specification for a CD Player

presses “load”button, the drawer opens. If he/she presses it again, the drawer closes and the track ready to be played and *track* is set to the first one in CD. When the user presses “play”button, if no CD is in the drawer, then the drawer keeps closed; otherwise, the player goes to *BUSY* state. Using the rules in Section III to map *NONPLAYING* state to a CSP# process we obtain the following CSP# program.

```
//Variable declaration
$var\ \ present = false,\ \ track = 0;$
//Process definition
$NONPLAYING(i) =$
                $case\{$
                $(i == 0) : CLOSED()$
                $(i == 1) : OPEN()$
                $\} \triangle ($
                $([!present](play \rightarrow NONPLAYING(0)))$
                $\Box\ ([present](play \rightarrow BUSY(0)))$
                $\Box\ (off \rightarrow Skip));$
```

```
$CLOSED() = load$
            $\rightarrow open\{track = 0;\ present = false\}$
            $\rightarrow OPEN();$
$OPEN()\quad = load$
            $\rightarrow close\{track = 1;\ present = true\}$
            $\rightarrow CLOSED();$
```

When it enters *BUSY* state, the player firstly locates the track to play and starts to play one by one until the last track is finished and after that the player goes back to *CLOSED* state. If “play”button is pressed with the player in *BUSY* state, the player enters the history state. In other words, it plays the current track if *PLAYING* is the last active substate; it restarts the current track but remains paused if *PAUSED* is the last one. Since history state is used, a shared variable *j* is defined to retain the last visited substate. Its valuation behaves as the entry behavior of every substate. Thus, *BUSY* state is described as the following.

```
//N denotes the total number of tracks.
$var\ j = 0;$

$BUSY(i) =$
    $find\_track\_start \rightarrow$
    $case\{(i == 0) : PLAYING()$
        $(i == 1) : PAUSED()\}$
    $\triangle\ ((load \rightarrow NONPLAYING(1))$
    $\Box\ ([track! = N]$
        $(\{track = track + 1\} \rightarrow BUSY(0)))$
    $\Box\ ([track == N]NONPLAYING(0))$
    $\Box\ (stop \rightarrow NONPLAYING(0))$
    $\Box\ (off \rightarrow Skip)$
    $\Box\ (play \rightarrow BUSY(j));$

$PLAYING() = \{j = 0;\ \} \rightarrow (play\_track \rightarrow Skip)\ \triangle$
             $((pause \rightarrow PAUSED())\ \Box\ Skip);$
$PAUSED()\ \ = \{j = 1;\ \} \rightarrow pause \rightarrow PLAYING();$
```

So far, we have demonstrated how we could use the translation rules defined in this paper to describe the behavior of a CD player modeled using UML state machines. The CSP# model obtained from the formalization is fed into PAT for simulation and model checking. Suppose now we want to check if the design adheres to some basic requirements of the system. For example, a CD player should guarantee two safety properties.

1) The number of any playing track is always within the amount of tracks in the current CD.
2) It never plays when the player does not locate any track.

We represent these two basic requirements in LTL as follows, where $\Box$reads as always.

1) $\Box((track >= 0) \bigwedge (track <= N))$
2) $\Box\neg((track == 0) \bigwedge (play\_track))$

The results from PAT suggest that the design satisfies the safety properties stated above. Also, other kinds of property assertions could be verified against the model with the help of PAT, such as liveness, fairness and trace refinement properties. Take a liveness property as an example: a CD player should ensure that when there is a CD in the drawer, if the user presses "play" button, then the player will eventually play some track. This liveness property is specified in LTL as the following, where $\Diamond$reads as eventually.

$$\Box((present == true) \bigwedge play \rightarrow \Diamond play\_track)$$

When we model-check the property against the model, it turns out that the model satisfies it.

V. Conclusion

In this paper we have defined a translation scheme for a class of UML models composed of asynchronously executing, hierarchical state machines. The main purpose of this approach is two-fold: first, to provide a completely automatic approach for transforming a model of state machines to the input model of PAT model checker; second, to effectively handle advanced modeling techniques in state machines such as fork, join, history and submachine features.

In the near future we plan to support deferred events, time events and conduct more industrial case studies to evaluate our approach. We also wish to extend our approach by looking at UML sequence diagram. In addition, we will investigate various reduction techniques to optimize the translation to make model checking state machines more efficient.

References

[1] P. Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *CoRR*, cs.SE/0407038, 2004.

[2] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[4] E. M. Clarke and W. Heinle. Modular translation of statecharts to smv. Technical report, Carnegie-Mellon University School of Computer Science, 2000.

[5] J. Dubrovin and T. Junttila. Symbolic model checking of hierarchical UML state machines. Technical Report B23, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2007.

[6] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS06), volume 3920 of LNCS*, pages 441–444. Springer, 2006.

[7] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[8] D. Jansen, H. Hermanns, and J. Katoen. A probabilistic extension of uml statecharts: Specification and verification. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 2469, pages 355–374, Oldenburg, Germany, 2002. Springer.

[9] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, V11(6):637–664, December 1999.

[10] J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. *Automated Software Engineering, International Conference on*, 0:255, 1999.

[11] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 95–104, Kyoto, Japan, 1998.

[12] B. P. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[13] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[14] M. Y. Ng and M. Butler. Towards formalizing uml state diagrams in csp. In *In: 1st International Conference on Software Engineering and Formal Methods, IEEE Computer Society*, pages 138–147, 2003.

[15] M. Y. Ng and M. J. Butler. Tool Support for Visualizing CSP in UML. In *ICFEM '02*, pages 287–298, London, UK, 2002. Springer-Verlag.

[16] OMG. Unified Modeling Language Superstructure Version 2.2. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/, Feburary 2009.

[17] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[18] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA 2008*, pages 307–322. Springer, 2008.

[19] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, pages 702–708, Grenoble, France, June 2009.

[20] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *ICFEM 2008*, volume 5256 of *LNCS*, pages 318–337. Springer, 2008.

[21] W. L. Yeung, K. R. P. H. Leung, J. Wang, and W. Dong. Improvements Towards Formalizing UML State Diagrams in CSP. In *APSEC '05*, pages 176–184, Washington, DC, USA, 2005. IEEE Computer Society.