8-2013

USMMC: A self-contained model checker for UML state machines

Shuang LIU

Yang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Manchun ZHENG

Bimlesh WADHWA

See next page for additional authors

Citation

LIU, Shuang; LIU, Yang; SUN, Jun; ZHENG, Manchun; WADHWA, Bimlesh; and DONG, Jin Song. USMMC: A self-contained model checker for UML state machines. (2013). *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, August 18-26*. 623-626.
Available at: https://ink.library.smu.edu.sg/sis_research/5010

USMMC: A Self-Contained Model Checker for UML State Machines[*]

Shuang Liu
School of Computing, National University of Singapore
liushuang@comp.nus.edu.sg

Yang Liu
Nanyang Technological University
yangliu@ntu.edu.sg

Jun Sun
Singapore University of Technology and Design
sunjun@sutd.edu.sg

Manchun Zheng
Singapore University of Technology and Design
manchun_zheng@sutd.edu.sg

Bimlesh Wadhwa
School of Computing, National University of Singapore
dcsbw@nus.edu.sg

Jin Song Dong
School of Computing, National University of Singapore
dongjs@comp.nus.edu.sg

ABSTRACT

UML diagrams are gaining increasing usage in Object-Oriented system designs. UML state machines are specifically used in modeling dynamic behaviors of classes. It has been widely agreed that verification of system designs at an early stage will dramatically reduce the development cost. Tool support for verification UML designs can also encourage consistent usage of UML diagrams throughout the software development procedure. In this work, we present a tool, named USMMC, which turns model checking of UML state machines into practice. USMMC is a self-contained toolkit, which provides editing, interactive simulation as well as powerful model checking support for UML state machines. The evaluation results show the effectiveness and scalability of our tool.

Categories and Subject Descriptors:.
D.2.4 [Software/Program Verification]: Model checking
General Terms:.
Verification
Keywords:.
UML state machines, model checking, semantics

1. INTRODUCTION

UML diagrams are gaining increasing usage in object oriented system designs and UML state machines are specially used to model the dynamic behaviors of classes. These diagrams serve as the basis for code development. However, UML specification is documented in natural language, which easily introduces inconsistencies and ambiguities [3]. Another default reported about UML diagrams is the lacking of consistency usage [11] throughout the software development process. As is reported in [11], Software engineers tend to use UML "as long as it is considered useful, after which it is set aside or even discarded". Lacking of tool support is one of the reasons which prevents the usage of UML throughout the software development process (Zeichick [15] found in their survey in 2002 that the reason for one-fourth of the investigate people that do not use UML is because that "their tools do not support UML"). We believe that, in order to make UML helpful throughout the software development process, to enable stimulation the dynamic behavior of UML designs and to turn the idea of automatic finding system design flaws in early stages into practice, rigorous analysis and verification supports on the UML diagrams are necessary. There exists some tools, such as vUML [8], HUGO [6] and TABU [2], which support model checking UML state machines. These tools conduct translations from a UML diagrams to the input language of some model checkers, such as SPIN, UPPAAL, etc. The verification can be accomplished by relying on verification tools for the target languages. But the translation-based approaches suffer from the following defects: (1) Due to the semantic gaps between UML state machines and the target languages, it may be hard to translate some syntactic features of UML state machines, introducing additional but undesired behaviors. For example in [16], extra events have to be added to each process in order to model exit behaviors of orthogonal composite states, which introduces redundant behaviors. (2) Translation-based approaches heavily depend on the target formal languages. Furthermore, the additional behaviors introduced during the translation may significantly slow down the verification; and optimization and reduction techniques (like partial order reduction) may not apply in order to preserve the semantics of the original model. (3) Lastly, for verification, translation-based tools are very sensitive to the updating of the underlying model checkers they depend on. For example the counterexample re-translation functionality of HUGO is disabled because of the updating of Spin.

All the discussed deficiencies make it difficult to apply those tools in real world problems. In order to address these issues, we are motivated to develop a self-contained model checker which is customized for UML state machines.

Our tool, named USMMC (**U**ML **S**tate **M**achine **M**odel **C**hecker), is implemented based on a formal operational semantics [9] defined for the complete set of UML state machines diagrams. We currently support model checking of safety and liveness properties with different fairness assumptions [13]. We have implemented our tool in a way that all the model checking details are hidden from the users so that users without any model checking background are also able to use our tool. Our tool provides user-friendly graphical interfaces. It takes as input a UML state machine diagram in xmi format, which is compatible with existing UML case tools.

[*]This work is supported by project 9.10.11 "Software Verification from Design to Implementation" of Programme Merlion (official collaborative grant co-funded by France and Singapore).

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2494595

Our tool distinguishes itself from existing UML tools with the following features:

- It is a fully automatic tool for model checking UML state machine diagrams directly and provides user friendly graphical interface.
- It supports model checking of safety and liveness properties with fairness assumptions.
- It supports simulation and verification of multiple UML state machines interactions.
- It reports violations directly in terms of UML state machine execution traces, which are intuitive to follow.

The rest of the paper is organized as follows. Section 2 briefly introduces the operational semantics that our implementation is based on. Section 3 presents the architecture and implementation details of USMMC. Section 4 reports evaluation results of our tool. We discuss related work in Section 5. Section 6 provides further discussions and concludes the paper.

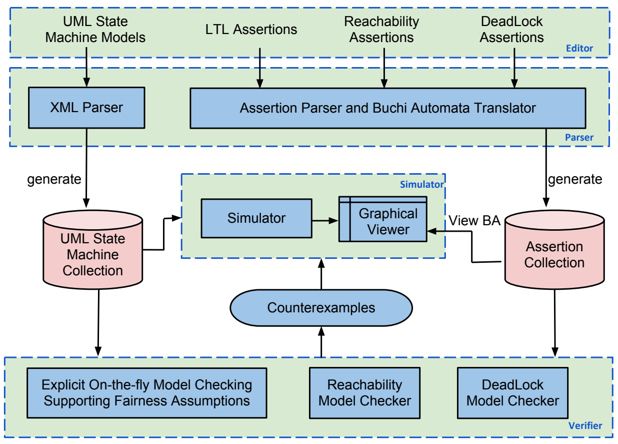

Figure 1: The Architecture of USMMC

2. FORMAL SEMANTICS OF UML STATE MACHINES

To enable direct model checking of UML state machines, we provide an operational semantics for UML state machines [9], which serves as the theoretical foundation for the implementation of USMMC. The operational semantics covers all features of the latest version ($v2.4.1$) of UML state machines [1], including the complex features such as choice, join/fork pseudostates, submachine state which are often left out by existing approaches. We also consider communications between state machines as well as event pool mechanisms, which makes verification of the entire system behavior (instead of a single UML state machine) possible. The syntax structure of our formalization obeys the structure of OMG UML state machines specifications, which adapts well to future changes, such as refinements.

The semantic model of our operational semantics is Labeled Transition System (LTS), which is subject to model checking. In our semantic definition, each LTS step corresponds to an run-to-completion (RTC) step in UML state machines, which may be composed of multiple UML state machine transitions. Thus implementation based on our formal semantics will dramatically reduce the state space generated compared with translation-based approaches. This is also confirmed by the evaluation results in Section 4.

3. ARCHITECTURE DESIGN AND IMPLEMENTATION

In this section, we introduce the architecture design, the functionality and the implementation choices of USMMC.

3.1 Architecture of USMMC

USMMC is a self-contained model checker for UML state machines, which consists of four components, i.e., Editor, Parser, Simulator and Verifier. Fig. 1 shows the architecture design of USMMC. The UML state machines and various assertions can be edited in the text editor component. On clicking the simulation/verification button, the parser is first called to parse the UML state machines and assertions into internal representations; then simulation/model checking components are invoked to perform the simulation/model checking respectively. In the case of an invalid assertion, a counterexample in terms of UML state machines execution trace will be returned. In the following, we are going to introduce each component of USMMC in details.

Editor The editor component provides a text editor which enables editing of both models and properties. Since our tool focuses on providing model checking support for UML state machines instead of graphical modeling, so it takes as input UML state machines in xmi format, which is exported from existing UML case tools. The editor also supports syntax highlighting and multiple document editing.

Parser In USMMC, we have implemented a parser for UML state machines and a parser for assertions. The xml parser parses UML state machines (in xmi format) into internal representations, i.e., the syntax structure for UML state machines defined in [9]. The formal semantics of UML state machines is implemented to obtain the LTS models from the internal representations. The LTL models will be consumed by simulation and model checking components in later stages. The assertion parser parses LTL, liveness and deadlock free assertions into internal representations. It supports an assertion language which allows LTL formulae constituted with propositions and events, which compliments existing model checkers. Buchi Automata can be generated from negations of LTL assertions.

Simulator USMMC provides a user friendly simulator which is capable of performing automatic random walking or user-guided step-by-step simulations on UML state machines executions. Since our implementation is based on the formal semantics and the LTS steps are consistent with the UML state machine RTC steps, the LTS model generated by our simulator actually directly reflects the execution of UML state machines. Thus the simulator of USMMC provides good tractability to the original UML state machines model. The simulator also provides other functionalities, such as complete state graph generation, trace playback, counterexample visualization.

Verifier The verifier is capable of conducting on-the-fly explicit state model checking on a variety of properties, including deadlock-free checking, reachability checking and refinement checking [14]. Furthermore, it implements dedicated algorithms for model checking LTL properties under a variety of fairness constraints, including global fairness, event/process level weak fairness and event/process level strong fairness [13], which are often required for verification of liveness properties. Counterexamples are returned as UML state machine execution traces (instead of any intermediate formats), which are intuitive to follow.

3.2 Implementation Choices for USMMC

There are many "semantic variation points" [3] in UML state machines specifications, which introduce difficulties to the semantic formalization as well as tool implementation. We are going to discuss our choices for implementing some of the important semantic variation points in USMMC in this section.

Event Dispatching Mechanism The concept of event pool is introduced to control the event consumption within a UML state machine and to enable communications between different state machines. It guarantees the stabilization of the state machines within a finite number of steps. But the concrete data structure to implement the event pool and event dispatching order are not specified. Our implementation of event pool follows the operational semantics of UML state machines defined in [9], where sets are used to represent event pools. Completion and deferred events are properly considered according to the specification [1].

Conflicting Transitions In UML state machines, multiple transitions can be enabled by a single dispatched event. A maximal non-conflicting set of transitions should be decided to execute and the OMG UML state machines documentation [1] has provided two priority rules to deal with conflicts. However not all conflicts are solvable by the provided priority rules. In our implementation, all the enabled maximal non-conflicting transition sets will be enumerated if more than one maximal non-conflicting transition sets are triggered by the same dispatched event and the priority of them cannot be solved.

Communications between State Machines USMMC supports verification and simulation of multiple state machines communication through signal and call events. In our implementation, signal events are processed asynchronously and call events are processed synchronously, i.e., the caller state machine will be blocked until the callee state machine finishes its execution and returns control. Recursive calls are supported in our implementation.

4. VERIFICATION RESULTS

In this section, we conduct verifications on some commonly used examples with USMMC and HUGO [6]. HUGO is a tool which translates UML state machines into Promela models and utilizes Spin to perform model checking. The latest version of HUGO is based on Spin4.3.0, which is out-of-date. HUGO has compatability problems with Spin5.x and Spin6.x, as a result, the conterexample returned by Spin cannot be translated back to UML execution traces and is hard for human to interpret. The examples we use are *RailCarO* [5], *RailCar* [9] (modifies *RailCarO* to manually introduce bugs. Both examples contain transitions emanating/entering orthogonal composite states, which are not supported by HUGO.), *BankATM* [6], dining philosopher[1] and *TollGate* [7]. The experiments are conducted with an Intel Core $i7-2600$, $3.4GHZ$ CPU and $8GB$ memory machine. Our tool works on Windows7, 64-bit operating system and HUGO works with Spin6.2.3 on Ubuntu10.04 LTS operating system.

The verification results are shown in Table 1. TTime represents the time used to translate UML state machines models into Promela. ETime represents the time used by Spin to do model checking. Our tool finds the manually injected bugs in *RailCar* system, which is out of the capability of HUGO since HUGO does not support transitions emanate/enter orthogonal composite states, which are presented in the *RailCar* model. The results also show that our tool out performs HUGO both in execution time and memory consumption on all the examples.

[1]We model forks and philosophers as separate UML state machines, which execute in parallel.

The main reason is that the Promela code generated by HUGO has many local transitions, (thus many local variables are introduced to record the status of the intermediate states, which are memory consuming) which introduce overheads. For example, in the generated *TollGate* promela code, 7 steps are conducted to move from an initial pseudostate to its target state, but it is actually within one RTC step in the UML state machines semantics. Our tool strictly obeys the RTC semantics of UML state machines, where only one step is taken for the above case. The costs introduced by local transitions are exponential in case of non-determinism, such as the dining philosopher example.

The experiment also shows the scalability of USMMC. We check the deadlockfree property (with breadth first search) on dinning philosopher models from $n = 2$ up to $n = 7$[2], and our tool is capable of finding the deadlock within acceptable amount of time. Spin reports out-of-memory error on the models generate by HUGO when $n \geq 4$. The data in Table1 for $n = 4$ is obtained when we set the search depth as the default value, i.e. $1,000,000$. We can see from the result that USMMC can handle large state spaces caused by non-determinism. Reducing further the state space through techniques such as partial-order reduction is the subject of our future work.

5. RELATED WORK

There were some tools developed for model checking UML diagrams. vUML [8] is one of the early tools which translates a UML state machines into Promela models. It supports checking of deadlock, livelock, and reachability properties. But these properties require explicitly annotating states with stereotypes and constraints. LTL properties cannot be checked by vUML without knowing the translated Promela model. HUGO [6] is a tool which aims at verifying the consistencies of UML state machines with properties specified by collaboration diagrams or sequence diagrams. It translates UML state machines into Promela, the input language of the Spin model checker. It supports model checking of deadlock properties and LTL properties. In [16], UML state machines are translated to CSP# and verified by PAT model checker. TABU [2] translates a UML state machines into the input language of SMV model checker. Different from vUML and HUGO, it can support verification the of LTL properties by providing property patterns, which guides the specification of properties. Another tool [12] is based on SMV model checker. It is capable of checking both the static, i.e., well-formed rules, and dynamic properties of a UML state machines. JACK [4] is an integrated environment based on the usage of process algebras, automata and temporal logic. It supports many phases of system development process by integrating different editing and verification tools. The AMC component inside JACK is able to conduct model checking against ACTL properties. But the components of JACK use FC2 as exchange format, which is not widely supported by the state-of-practice tools. Among all the tools discussed here, only HUGO is currently available. All of them except JACK conduct a translation-based approach, which suffers from efficiency and tractability problems. JACK, though directly implements the semantics, is not fully automatic and is unavailable now. Our tool, USMMC, is developed for the purpose of providing fully automatic and direct model checking functionalities, which is more efficient, user friendly and achieves a good coverage of UML state machine features. Since the counterexamples are presented in terms of event execution tracess in UML state machines instead of any intermediate formats, USMMC provides good tractability of design flaws.

[2]n is the number of philosophers.

Table 1: Evaluation results

Model	Property	Result	USMMC				HUGO				
			Time(s)	State	Transition	Mem (KiB)	TTime(s)	ETime(s)	State	Transition	Mem (KiB)
RailCar	Prop1	not valid	0.013	30	34	43, 342	-	-	-	-	-
RailCarO	Prop1	valid	0.011	44	54	43, 058	-	-	-	-	-
BankATM	Prop2	valid	0.009	25	28	917.5	0.231	0.050	578	1, 133	98, 528
TollGate	Prop3	valid	0.110	36	50	43, 345	0.197	0.505	61, 451	256, 807	100, 578
DP2	deadlock	not valid	0.005	39	65	2, 318	0.196	0.111	12, 766	42, 081	98, 918
DP3	deadlock	not valid	0.039	237	589	10, 145	0.242	379.009	4, 626, 838	23, 897, 077	276, 067
DP4	deadlock	not valid	0.34	1, 519	5, 079	21, 059	1.117	8944.754	57, 213, 708	339, 761, 530	3, 059, 807
DP5	deadlock	not valid	3.11	9, 634	40, 366	92, 369	-	-	-	-	-
DP6	deadlock	not valid	27.87	63, 069	324, 275	226, 271	-	-	-	-	-
DP7	deadlock	not valid	232.64	398, 101	2, 385, 361	2, 852, 672	-	-	-	-	-

Prop1=$\Box(alert100 \rightarrow \Diamond arriveAck)$, Prop2=$\Box(retain \rightarrow ((!cardValid \wedge numIncorrect \geq maxNumIncorrect))$, Prop3=$\Box(TurnGreen \rightarrow \Diamond carExit)$.

6. CONCLUSION AND DISCUSSIONS

We developed a self-contained model checker and simulator for UML state machines, which is able to check safety and liveness properties and conduct step-wise simulation of UML state machines executions. Our tool is implemented based on a formal operational semantics defined for UML state machines. The experiment results show the effectiveness and efficiency of our tool. USMMC has been implemented as a stand-alone tool in C# with user-friendly graphical interfaces. Starting from 2012, USMMC has come to a stable stage with solid testing and 11 built-in examples. It has been applied to verify many real-time systems ranging from classical concurrent algorithms, such as the dining philosopher problem, to real world problems, such as the railcar system. Our future works include further reducing the state space through techniques such as partial order reduction. Detail information about USMMC (including the video demonstration and the deliverable tool) can be found in our website `www.comp.nus.edu.sg/~lius87`. We list some known issues about our tool and possible solutions as follows.

Compatability problem about XMI format Although OMG had released XML Interchange Format (XMI) as the standard exchange format of UML diagrams, different tools adopt different versions of XMI, which causes compatability problems between the models exported by different tools. The models exported by one case tool cannot be properly displayed by the other tools, as is reported by [10]. This is a known open issue. Since our tool takes the UML state machine models (in xmi format) exported by those tools as input and it is infeasible for us to support all those incompatable formats, we support the output format of Enterprise Architect in the current stage. Providing our own graphical modeling front-end for UML state machines may thoroughly solve the problem and this is subject to our future work.

Structure of the Event Pool Currently, the event pool is implemented as a set in our tool. We are planning to provide more structures, such as queue, bag, and user-defined structures for the event pool implementation in order to meet more modeling requirements.

Action language In the current implementation, we do not provide any specific language for modeling actions and behaviors of UML state machines, just a subset of Object Constraint Language (OCL), arithmetic and boolean calculations are supported. So we are planning to support more complex languages, such as imperative programming languages (C/C#/java), as the description language of events, actions and activities. This will make our tool coincide with existing graphical UML editing tools and is capable of conveying more meticulous system design concerns.

7. REFERENCES

[1] OMG unified language superstructure specification (formal). Version 2.4.1, 2011-08-06. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`.

[2] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente. UML automatic verification tool with formal methods. *Electronic Notes in Theoretical Computer Science*, 127:3–16, 2005.

[3] H. Fecher, J. Schönborn, M. Kyas, and W. de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In *Formal Methods and Software Engineering*. Springer, 2005.

[4] S. Gnesi, D. Latella, and M. Massink. Model checking UML statechart diagrams using JACK. In *HASE*, 1999.

[5] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30:31–42, 1997.

[6] A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In *Proceedings of 5th Workshop on Tools for System Design and Verification, Technical Report*, 2002.

[7] J. Kong, K. Zhang, J. Dong, and D. Xu. Specifying behavioral semantics of UML diagrams through graph transformations. *Journal of Systems and Software*, 82:292–306, 2009.

[8] J. Lilius and I. P. Paltor. vUML: A tool for verifying UML models. In *ASE*, 1999.

[9] S. Liu, Y. Liu, E. André, C. Choppy, J. Sun, B. Wadhwa, and J. S. Dong. A formal semantics for complete uml state machines with communications. In *iFM*, 2013.

[10] B. Lundell, B. Lings, A. Persson, and A. Mattsson. UML model interchange in heterogeneous tool environments: An analysis of adoptions of XMI 2. In *MODELS*. Springer, 2006.

[11] M. Petre. UML in practice. In *ICSE*, 2013.

[12] W. Shen, K. Compton, and J. Huggins. A toolset for supporting UML static and dynamic model checking. In *COMPSAC*, 2002.

[13] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV*, 2009.

[14] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. More anti-chain based refinement checking. In *ICFEM*, 2012.

[15] A. Zeichick. Modeling usage low; developers confused about uml 2.0, mda. Technical report, BZ Research, 2002.

[16] S. Zhang and Y. Liu. An automatic approach to model checking UML state machines. In *SSIRI-C*, 2010.