Domain Specific Analysis of Statemachine Models of Reactive Systems

Karolina Zurowska

Queen's University
School of Computing
Kingston, ON, Canada
zurowska@cs.queensu.ca

Abstract. Analysis of models is an important aspect of the Model Driven Development (MDD) paradigm. Even though many analysis methods exist (e.g., model checking), they are not easily applicable in the context of MDD tools such as IBM Rational Software Architect Real Time Edition (IBM RSA RTE) and MDD languages such as UML-RT. The major reason for this inapplicability is that they typically require MDD models to be translated to a formal notation, which does not directly support key model features. The research direction proposed in this paper deviates from the standard approaches – it brings analysis "close" to MDD models and introduces domain-specific analysis of UML-RT models. To this end we use a formal representation that preserves the important aspects of the models. This removes the translational effort and, in addition, enables the use of MDD-specific abstractions aiming to support better understanding of models and to improve the scalability of verification. We will define abstractions for data (using symbolic execution), for structure and for behavior. The approach will be implemented as a set of plugins to IBM RSA RTE and evaluated on a variety of UML-RT models.

1 Introduction

The analysis of models early in the development process is one of the promises of the Model Driven Development paradigm. Such analysis should facilitate better understanding of developed systems and should enable formal verification. However this poses many challenges for MDD models, because they have complex dynamic structures and complex behaviors [17]. Additionally, industrial MDD models are large and scalability is essential.

The majority of the analysis techniques proposed in the literature reuse formal verification tools such as SPIN or NuSMV [16, 8, 15]. This can be advantageous, because the tools are mature and implement optimizations, however a translation to a formal language of a model checker is required. This step requires simplifications in models such as flattening of hierarchies, omitting communication details or encoding object-orientation. The goal of our work is to provide more dedicated, domain-specific analysis [19], similar to SLAM or JPF projects

(early efforts by the JPF team to use Promela/Spin were eventually abandoned). This approach reduces the "semantic gap" between the language of a checker and the language of a model. The direct benefit is minimal translational effort, more indirect ones include support for verification methods that are tailored to MDD models. The dedicated approaches to analysis are less often researched, and ours is the first one designed specifically for UML-RT models.

In our approach we introduce a formal representation, which shares important characteristics with MDD models such as hierarchical components, strong encapsulation, message-based communication and state machines. Preserving these features enables definition of "domain specific" abstractions. We propose three types of abstractions: symbolic execution to deal with data, structural abstraction to deal with complex hierarchical structures and state aggregation for state machines. Abstractions simplify the state space for better understanding, but we also use them to improve model checking algorithms. The improvements are thanks to lazy composition, i.e. exploring only those parts of a model that influence the satisfaction of a property.

2 Background and motivation

The language we use in this work as an example of a MDD language is the UML-RT from IBM RSA RTE [1]. A model in this language consists of *capsules*. A capsule may contain *parts*, which are instances of other capsules. Capsules communicate using typed *ports* and the type of a port (called a *protocol*) gathers *messages* sent or received through the port. The behavior of a UML-RT capsule is given with a UML-RT State Machine. UML-RT State Machines contain action code, written in, e.g., Java that updates attributes or sends messages.

Analysis of a UML-RT model (or a model that is similar to it) should increase understanding and should allow for verification of properties of the model. In order to achieve this, the model can be executed in IBM RSA RTE, but the execution is limited to one path, i.e., to one set of input values. This gives some insight, but is insufficient to check properties concerning all possible executions, which is necessary to fully understand and to verify models. The first possibility is to reuse existing model checkers, and to translate the model to e.g. Promela, the input language of the SPIN model checker [15]. Although this enables exhaustive checking, a translation dealing with a sufficiently large subset of UML-RT is complex and difficult to test, and the analysis results are not directly available back in the original model.

The alternative and less followed approach, which we take, is to build a dedicated analysis tool. The disadvantages of such an approach are the effort to develop tools and more limited use of optimizations techniques present in models model checkers (e.g. BDDs). However, the advantages are straightforward translation and directly usable results. Additionally, the domain specific analysis can directly exploit structure and semantics of models. Our hypothesis is that this allows for more modular and, in turn, more scalable analysis and verification.

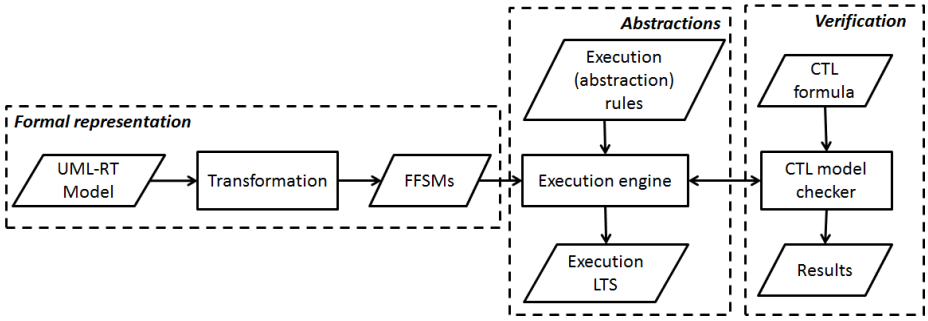

Fig. 1. The overview of the analysis process.

3 Proposed approach

The analysis approach that we propose for UML-RT models is summarized in Figure 1. There are three major parts: representation, abstraction and verification; we describe them below.

The *formal representation* part of the process is responsible for translating UML-RT models into the formal representation: communicating Functional Finite State Machines (FFSMs). FFSMs have similar features as UML-RT. A model consists of a set of *modules*. A module may contain *parts* with types of other modules. *Actions* are used to communicate between parts. We support asynchronous communication, hence modules use *queues*. The behavior of modules is given with state machines, in which transitions have *guards* and *effects* assigned to them. Effects include updates of attributes, sending actions and possibly changes to the structure of the model. Effects are obtained by symbolic execution of action code and by using the results of this execution to represent the code. (see [21] for details). The semantics of FFSMs is given with a labeled transition system (LTS), called an execution LTS. A state in such an execution LTS contains current execution information, i.e., values of the attributes, current states, as well as contents of all queues for all parts in the model.

The goal of using *abstractions* is to reduce the size of the state space. Each type of abstraction is identified with a set of execution rules and these rules are used to generate the execution LTS. We support the following types of abstractions:

1. Symbolic execution. In this type of abstraction concrete values of variables are replaced with symbolic ones (as in [11]). In order to distinguish between branches of execution, path constraints are included in an execution LTS. This type of abstraction is very useful to deal with data and to combine execution states that are different only due to the values of input variables.
2. Structural abstraction. This type of abstraction ignores parts of a model that are irrelevant with respect to the performed analysis. The abstracted parts are treated as if they were removed from a model, but actions that are

delivered by the abstracted parts are assumed to be available at all times. Therefore an execution LTL for structural abstraction is an *overaproximation*. In this type abstraction the user selects the parts to be abstracted away or it is done automatically based on a verified property (see below).
3. State aggregation. This type of abstraction aggregates states of a state machine by combining them into one state, for instance to deal with hierarchical states in state machines. Therefore, as in case of a previous abstraction type, this abstraction is also an overaproximation. The decision which states to aggregate is left to the user, who may use existing hierarchies of states as guidance.

The *verification* part of our approach includes the specification of properties of models and model checking algorithms. The properties of models are expressed with an extension of Computation Tree Logic (CTL) [6]. In the proposed extension atomic propositions include models characteristics such as being in a particular state, a certain queue containing specific actions and certain attributes having certain values. Because the application of execution rules is done step-wise, the checking of CTL formulas can be performed on-the-fly. To this end, the standard labeling algorithm for CTL properties [6] is extended to use the labels from the previous execution steps. Finally, we extend the model checking to use lazy composition. Initially, parts not mentioned in the checked formula are abstracted. All other parts are executed, for instance, symbolically. If one of executed parts requires an action, which can be generated by some abstracted part then this part is explored. This means that applied execution rules are updated and in the next step the execution of the extra part is included.

4 Implementation and evaluation

The process presented in Figure 1 is implemented as a set of Eclipse plugins. The main objectives of the evaluation of our approach are to assess its scalability and applicability. In order to achieve that we will use:

- custom UML-RT models to check the approach in the presence of increasing complexity (e.g., with increasing number of internal parts in the model). In this way we can observe how the verification methods scale.
- PBX model (adapted from a model obtained from our industrial partner) to check how abstractions can support better understanding of a complex model and how verification methods can be used in such a model. The PBX model consists of three subsystems and each subsystem has up to 6 different parts. Code generated from those subsystems has between 3500 and 6000 lines per subsystem.

5 Current status and results

Up until now we have explored symbolic execution in the context of UML-RT models [21, 22]. We also defined a model checking algorithm that uses lazy

composition [20]. In the latter work we showed how exploiting the structure of models can improve the scalability of the analysis; in a few cases we were able to reduce the state space to 10% of the original state space.

Currently we are working on the implementation of the process shown in Figure 1. We implemented the transformation from UML-RT to FFSMs, the execution engine and the model checker. We are now finishing the implementation of rules and are moving towards more thorough evaluation.

6 Related work

The current practice of analysis and verification of MDD and UML-based models builds mostly on model checking. The proposed works are primarily concerned with the translation of models to the input languages of existing model checkers. For instance, UML State Machines are analyzed using SPIN [16], UML Activity Diagrams using nuSMV [8] and Statecharts, after translation to Java, using Java Pathfinder [14, 3]. There also exist translations of UML-RT to Promela/SPIN [15] and to the AsmL language used in SpecExlorer [12]. In contrast to those works the research proposed here is based on a more straightforward translation, which reduces the semantic gap between languages of models and model checkers [19].

Beside translations to model checkers, there are approaches built around UML-like state machines. Giese et al. [9] explore compositional aspects of models based on parallel composition and synchronous communication between UML Statecharts. This approach is implemented in the FUJABA tool suite [5] and uses compositional verification inspired by the popular assume-guarantee paradigm [2]. Our approach goes beyond the compositionality and also explores abstraction to improve scalability of the analysis.

Abstractions have been used to improve model checking techniques, but they are usually limited to data abstractions [7, 13, 10]. Another data-driven abstraction is symbolic execution, which, in its original version [11], simply replace concrete values of variables with expressions that represent them. Symbolic execution has been applied also to state based models. For instance, to Statecharts [18] or UML State Machines [4] or, after translation to Java using SPF [3]. The proposed research uses symbolic execution only as one of the abstractions and symbolic execution is defined for modular and hierarchical models as already introduced [21, 22].

Acknowledgments

Author wishes to acknowledge the support of Dr. Juergen Dingel, NSERC, IBM Canada, and Malina Software.

References

1. IBM Rational Software Architect, RealTime Edition, Version 7.5.5., http://publib.boulder.ibm.com/infocenter/rsarthlp/v7r5m1/

2. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: Modularity in model checking. In: Computer Aided Verification. pp. 521–525 (1998)
3. Balasubramanian, D., Pasareanu, C., Whalen, M., Karsai, G., Lowry, M.: Improving symbolic execution for statechart formalisms. In: MoDeVVa'12 (2012)
4. Balser, M., Baumler, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of UML state machines. In: ICFEM 2004 (LNCS Vol.3308). pp. 434 – 48 (2004)
5. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The FUJABA real-time tool suite: model-driven development of safety-critical, real-time systems. In: ICSE '05
6. Clarke, E.M., Grumberg, O.J., Peled, D.A.: Model checking. Cambridge, Mass. : MIT Press (1999)
7. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems 16(5) (1994)
8. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
9. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. In: ESEC/FSE 2003. pp. 38–47 (2003)
10. Ioustinova, N., Sidorova, N.: Abstraction and flow analysis for model checking open asynchronous systems. In: Software Engineering Conference, 2002. (2003)
11. King, J.: Symbolic execution and program testing. Communications of the ACM 19(7), 385 – 394 (1976/07)
12. Leue, S., Stefanescu, A., Wei, W.: An AsmL Semantics for Dynamic Structures and Run Time Schedulability in UML-RT. Tech. rep., University of Konstanz, Germany (2008), http://kops.ub.uni-konstanz.de/volltexte/2008/5781/
13. Manna, Z., Colón, M., Finkbeiner, B., Sipma, H., Uribe, T.: Abstraction and modular verification of infinite-state reactive systems. In: Requirements Targeting Software and Systems Engineering (1998)
14. Mehlitz, P.: Trust your model - verifying aerospace system models with Java pathfinder. In: IEEE Aerospace Conference (2008)
15. Saaltink, M., Meisels, I.: Using SPIN to analyse RoseRT models. Tech. rep., ORA Canada (1999)
16. Schafer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electronic Notes in Theoret. Comp. Science 55(3) (2001)
17. Selic, B., Gullekson, G., Ward, P.T.: Real-time Object Oriented Modeling and Design. J. Wiley & Sons (1994)
18. Thums, A., Schellhorn, G., Ortmeier, F., Reif, W.: Interactive Verification of Statecharts. In: INT 2004 (LNCS Vol.3147) (2004)
19. Visser, W., Dwyer, M., Whalen, M.: The hidden models of model checking. Software and Systems Modeling 11(4), 541–555 (2012)
20. Zurowska, K., Dingel, J.: Model checking of uml-rt models using lazy composition. In: Models'13 - to appear (2013)
21. Zurowska, K., Dingel, J.: Symbolic execution of UML-RT state machines. In: SAC Software Verification Track (2012)
22. Zurowska, K., Dingel, J.: Symbolic Execution of Communicating and Hierarchically Composed UML-RT State Machines. In: NASA Formal Methods 2012