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 used in the industry such as IBM Rational Software Architect Real Time Edition (IBM RSA RTE) and industrial MDD languages such as UML-RT. The major reason for this inapplicability is that the existing tools typically require MDD models to be translated to a formal notation, which does not directly support key model features. The research direction we follow deviates from the standard approaches – it brings analysis "closer" to MDD models and introduces domain-specific analysis of UML-RT models. To this end we use a formal internal representation that preserves the important features of the models. This minimizes the translational effort and, in addition, enables the use of MDD-specific abstractions aiming to support better understanding of models and improving the scalability of verification. Moreover, we will define abstractions for data (using symbolic execution), structure and behavior. The approach is implemented and evaluated on some experimental UML-RT models.

1 Introduction

Model Driven Development (MDD) is a software development approach based on models, which are continuously refined until code can be automatically generated from them [17]. MDD has been applied in various development areas, but has been the most successful in the development of embedded, reactive systems. The development of such systems is supported with industrial-strength MDD tools such as IBM Rational Software Architect RTE (IBM RSA RTE), IBM Rational Rhapsody [1], Mousetrap from Motorola [6] and Scade Suite from Esterel Technologies [3]. These tools represent a wide spectrum of modeling paradigms,but in order to support the necessary executability of models [17], they all must specify the structure and the behavior of systems. In this research we focus on those modeling approaches, in which structure is provided with hierarchically organized modules and the behavior of these modules is given as UML-like state machines [4] with message-based communication. Such characteristics are incorporated in the modeling languages of IBM RSA RTE, IBM Rational Rhapsody and Mousetrap.

The detection of problems in designs through 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 [23]. 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 [22, 13, 21]. 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 [25], similar to SLAM [8] or JPF [26] 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 [2]. 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 [21]. 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. Our hypothesis is that this allows for more modular and, in turn, more scalable analysis and verification.

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 (CFFSMs). CFFSMs 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 [27] for details). The semantics of CFFSMs is given with a labeled transition system (LTS), called an execution LTS. A state in such an execution LTS contains current execution

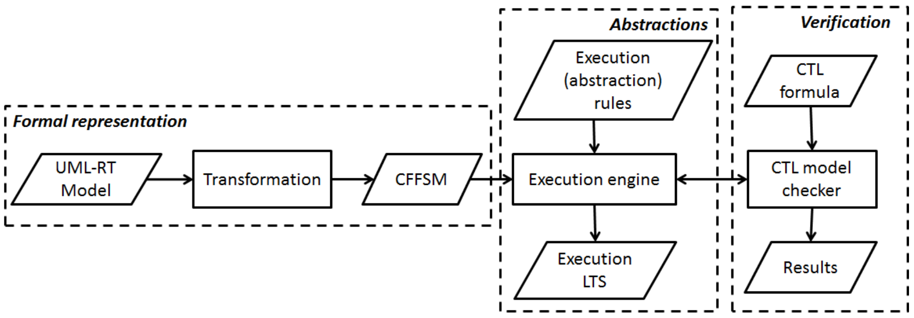

Figure 1: The overview of the analysis process.

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 [16]). 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 LTS for structural abstraction is an *overaproximation*. In this type abstraction the user selects the parts to be abstracted away.

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 types of abstractions mentioned above can be combined, that is, we may use more than one type of abstraction at the same time. For instance, we can abstract away some parts of the model and we may perform symbolic execution using the remaining parts. We can also aggregate states in the parts that are not abstracted away and symbolically execute what is left.

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) [11]. 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 stepwise, the checking of CTL formulas can be performed on-the-fly. To this end, the standard labeling algorithm for CTL properties [11] 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 (for details see [29]).

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 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). We use a family of models of traffic lights with increasing number of intersections.
- a model of Private Branch Exchange (PBX) 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.

We performed several abstractions on the above sets of models. As expected symbolic execution has been very powerful for models that are open, that is, that can receive many signals with data variables from their environment. Enumerating all possible values of such variables leads to state space explosion and explorations cannot be handled. Using symbolic executions makes explorations manageable, which is illustrated in Figure 2. In the figure the upper lines represent concrete and symbolic executions, where the concrete execution uses domains limited to values between -4 and 4. We show the number of explored states, with the dashed lines representing the state space over 100K states, which was too large to be further explored. Note that using symbolic execution reduces the state space even against to very limited domains. However, this technique is the most efficient for models with only several parts, if we can visualize their execution and improve their understanding. For models with more parts we need to reduce such models by the means of structural abstraction or state aggregation. In our experiments structural abstraction was able to reduce the size of the state space from more than 100K to several hundred states.

We also performed also experiments in verifying properties using the experimental UML-RT models. In Figure 2 we present the number of states explored to check properties. We used two properties of the form `A F p` and `E F p`, with p being an internal property of the model. The first property is satisfied if all paths of execution reach a state in which `p` is satisfied. The second property is satisfied if one path reaches such a state. The checking is performed for 2 algorithms we developed. The first algorithm is on-the-fly, that is, we check satisfaction during the exploration of a model. The second algorithm adds laziness to that and we explore only the parts of the model that can influence the property being checked. Note that lazy algorithm performs substantially better in case of models with 2 and 3 intersections. This is because the properties we used refer only to small subset of the all parts used in the models. These results show the substantial reduction of the state space we have to explore, even for model that cannot be fully explored. The achieved reduction is not always possible, as shown for instance for the model with 1 intersection, and it very much depends on the property p and the model.

The above results are only very small part of evaluation we performed. Due to the space restrictions we cannot present more results of experiments.

5 Contributions

The work we presented resulted in the following contributions:

1. Formal representation of a subset of UML-RT language that includes modules, hierarchical composition, communication, dynamic part creation and state machines.
2. Definition of symbolic execution, structural abstraction and state aggregation.

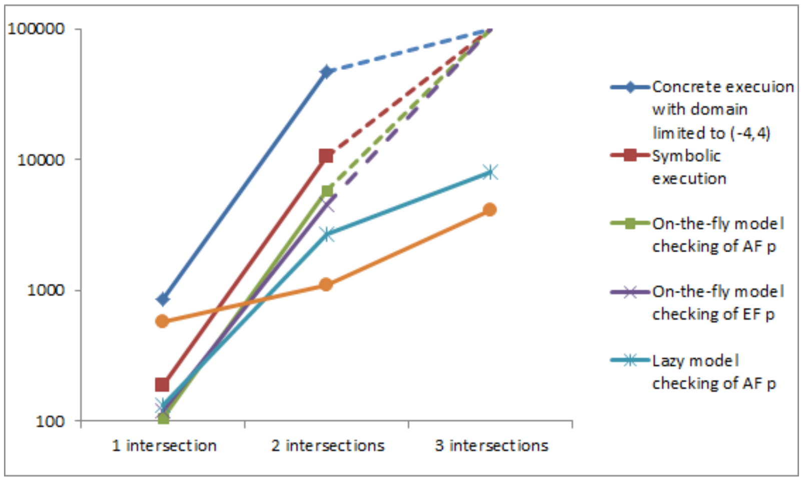

Figure 2: Number of explored states for traffic lights models (Y-axis is logarithmic with base 10).

3. Design of verification algorithms: on-the fly model checking and model checking based on lazy composition.
4. Implementation of engines to executed models with abstractions and implementation of model checking algorithms.

6 Related work

The current practice of analysis and verification of MDD models builds mostly on model checking. The proposed works translate models to the input languages of existing model checkers. For instance, UML State Machines are analyzed using SPIN [22], UML Activity Diagrams using nuSMV [13] and Statecharts, after translation to Java, using Java Pathfinder [20, 7]. In case of UML-RT there exists translations to Promela/SPIN [21] and to the AsmL language [18]. In contrast to those works our research is based on a more straightforward translation, which reduces the semantic gap between languages of models and model checkers [25].

Beside translations to model checkers, there are approaches built around UML-like state machines. Giese et al. [14] explore compositional reasoning based on parallel composition and synchronous communication between UML Statecharts. This approach is implemented in the FUJABA [10] and uses popular assume-guarantee paradigm [5]. 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 [12, 19, 15]. Another data-driven abstraction is symbolic execution, which, in its original version [16], 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 [24] or UML State Machines [9] or, after translation to Java using JPF [7]. The presented work uses symbolic execution only as one of the abstractions and symbolic execution is defined for modular and hierarchical models as already introduced [27, 28]. We also extended the symbolic execution to be applied to collections of state machines, which has not been proposed elsewhere.

Acknowledgments

Author acknowledges the support of Dr. Juergen Dingel, NSERC, IBM Canada, and Malina Software.

References

[1] IBM Rational Rhapsody, http://www.ibm.com/developerworks/ rational/products/rhapsody/

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

[3] Scade Suite (Esterel Technologies), http://www.esterel-technologies.com/products/scade-suite/

[4] Unified Modeling Language (UML 2.0) Superstructure, http://www.uml.org/

[5] 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)

[6] Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. MODELS 2005 (LNCS Vol.3713) (2005)

[7] Balasubramanian, D., Pasareanu, C., Whalen, M., Karsai, G., Lowry, M.: Improving symbolic execution for statechart formalisms. In: MoDeVVa'12 (2012)

[8] Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM 54(7), 68–76 (Jul 2011), http://doi.acm.org/10.1145/1965724.1965743

[9] 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)

[10] 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

[11] Clarke, E.M., Grumberg, O.J., Peled, D.A.: Model checking. Cambridge, Mass. : MIT Press (1999)

[12] Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems 16(5) (1994)

[13] Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)

[14] 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)

[15] Ioustinova, N., Sidorova, N.: Abstraction and flow analysis for model checking open asynchronous systems. In: Software Engineering Conference, 2002. (2003)

[16] King, J.: Symbolic execution and program testing. Communications of the ACM 19(7), 385 – 394 (1976/07)

[17] Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional (2003)

[18] 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/

[19] 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)

[20] Mehlitz, P.: Trust your model - verifying aerospace system models with Java pathfinder. In: IEEE Aerospace Conference (2008)

[21] Saaltink, M., Meisels, I.: Using SPIN to analyse RoseRT models. Tech. rep., ORA Canada (1999)

[22] Schafer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electronic Notes in Theoret. Comp. Science 55(3) (2001)

[23] Selic, B., Gullekson, G., Ward, P.T.: Real-time Object Oriented Modeling and Design. J. Wiley & Sons (1994)

[24] Thums, A., Schellhorn, G., Ortmeier, F., Reif, W.: Interactive Verification of Statecharts. In: INT 2004 (LNCS Vol.3147) (2004)

[25] Visser, W., Dwyer, M., Whalen, M.: The hidden models of model checking. Software and Systems Modeling 11(4), 541–555 (2012)

[26] Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)

[27] Zurowska, K., Dingel, J.: Symbolic execution of UML-RT state machines. In: SAC Software Verification Track (2012)

[28] Zurowska, K., Dingel, J.: Symbolic Execution of Communicating and Hierarchically Composed UML-RT State Machines. In: NASA Formal Methods 2012

[29] Zurowska, K., Dingel, J.: Model checking of uml-rt models using lazy composition. In: MoDELS. pp. 304–319 (2013)