Integration of Runtime Verification into Metamodeling for Simulation and Code Generation (Position Paper)

F. Macias[1], T. Scheffel[2], M. Schmitz[2], and R. Wang[1]

[1] Bergen University College, Norway
{fernando.macias,rui.wang}@hib.no
[2] Institute for Software Engineering and Programming Languages, University of Lübeck, Germany
{scheffel,schmitz}@isp.uni-luebeck.de

Abstract. Runtime verification is an approach growing in popularity to verify the correctness of complex and distributed systems by monitoring their executions. Domain Specific Modeling Languages are a technique used for specifying such systems in an abstract way, but still close to the solution domain. This paper aims at integrating runtime verification and domain specific modeling into the development process of complex systems. Such integration is achieved by linking the elements of the system model with the atomic propositions of the temporal correctness properties used to specify monitors. We provide a unified approach used for both the code generation and the simulation of the system through instance model transformations. This unification allows to check correctness properties on different abstraction levels of the modeled system.

1 Introduction

Modeling is a well-established practice in the development of big and complex software systems. Some of the more widespread approaches (e.g. Unified Modeling Language, UML) comprise the use of several general-purpose modeling languages. The models created with each of these modeling languages are then interconnected or related to one another. In recent years, general-purpose modeling languages are being replaced by Domain Specific Modeling Languages in many cases [6]. These languages define the structure, semantics and constraints for models related to the same application domain [12]. Among the reasons for the adoption of DSMLs one can mention their understandability by domain experts, capacity for high-level abstraction, user friendliness and tailoring to the problem space [6]. Besides, DSMLs inherit some of the advantages of general-purpose modeling, such as an improvement of efficiency for development and simulation.

However, the use of DSMLs does not completely shield the produced software from bugs or man-made mistakes. Software failures may still occur on complex systems due to a variety of reasons such as design errors, hardware breakdown or network problems. These failures require that verification methods are integrated into the development process. The use of such methods during the specification

This work is supported in part by the European Cooperation in Science and Technology (COST Action ARVI) and the BMBF project CONIRAS under number 01IS13029.

of a system can greatly improve their reliability. Unfortunately, testing is seldom exhaustive and cannot always guarantee correctness. An exhaustive option to check every execution path is model checking. But this alternative may suffer the state space explosion problem [10], especially relevant in distributed systems due to their inherent non-determinism. Yet another possibility in the system verification domain is to use runtime verification (RV). RV can cope with the inadequacies of testing by reacting to systems' failures as soon as they occur [9]. Also, it is a much more lightweight technique when compared to model checking, since only one execution path is checked. RV can be used to check whether an execution of a system violates a given correctness property. Such checking can be typically performed by using a monitor [10]. In its simplest form, a monitor decides whether the execution of a system satisfies a given correctness property by outputting either true or false. With runtime verification, the actual execution of the complex system may then be easily checked to ensure that the program does not violate given correctness properties.

This paper aims to effectively integrate runtime verification and domain specific modeling into the development of complex systems. This integration is achieved by linking the elements of domain specific models with the temporal correctness properties.

Related work. Using models and runtime verification during the development of complex systems is not new. For example, in [8], common concepts of runtime models and the provision of a basis for the metamodeling are described, and a metamodeling process for runtime models is presented, which guides the creation of metamodels combining design time and runtime concepts. In [3], a system modeling approach is developed to allow design-time system models to be reused by an autonomous system, and a runtime verification framework is also proposed. A combination of runtime verification and a specific DSML has been used in [5]. The authors used their own modeling framework for component-based systems and extended it by an RV framework. Their approach has a similar direction, but our approach aims at modeling more abstractly while keeping the possibility of verifying low-level properties, with any DSML.

The rest of this paper is organized as follows: Section 2 recaps some basic notion of RV and DSMLs which are used throughout the paper. Section 3 presents the main contribution of this paper: the integration of runtime verification and DSMLs. Finally, a conclusion and an outlook are given in the last section.

2 Background

We view a system as having a state consisting of a set of atomic propositions. Thereby each atomic proposition can either be true or false, and the state of the system at a certain point in time is given by the current value of each atomic proposition. Thus, a run of the system can be seen as an infinite sequence of those states and an execution is a finite prefix of such a run. In RV, we specify correctness properties based on the atomic propositions and generate monitors from them. With this, monitor statements about the correctness of the current execution of the system can then be made.

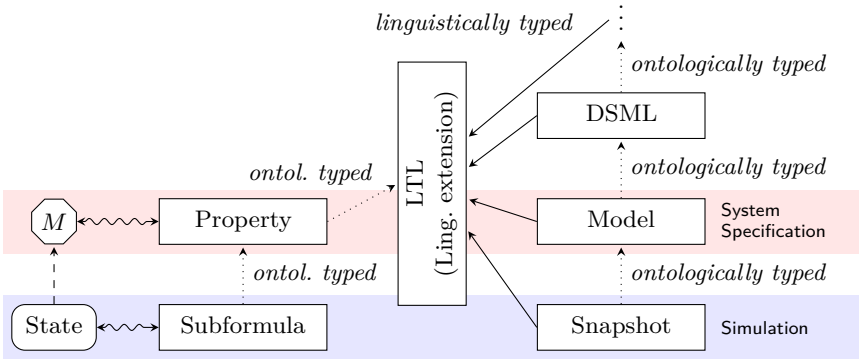

Fig. 1. Underlying multilevel model hierarchy with linguistic extension. Correctness properties of the system can be formulated in LTL, which is connected with the model as a linguistic extension. The property can be translated into a monitor, which accepts prefixes of the words in the language of the property. The System Specification is the abstraction layer of the most specific models which is used for the code generation. A Snapshot of the Model together with the not yet fulfilled Subformula or a State of the Monitor forms one state of the Simulation.

To be compatible with this view of a system, we define a multilevel modeling hierarchy [1] where the DSML and the actual model of the system are included. Moreover, the hierarchy includes instances of the system model that represent the particular state of the system at a given point in time (see Fig. 1). As presented in [11], this hierarchy borrows the concepts of multilevel modelling (with *ontological* typing relations), deep metamodelling [1] and linguistic extension [15] (hence the *linguistic* typing relations). To avoid ambiguities, we call the instances of the system model *snapshots*. A snapshot is also a model, and contains the set of active elements of the system. The way in which the system evolves during the simulation is described using model transformations (MT) that generate a new snapshot from the previous one. See [20] and [7] for similar approaches. In a simplistic way, this MTs remove elements which are not active anymore and create new active ones (see Fig. 2). To link both RV and DSMLs, we associate the atomic propositions, used in RV, with the current state of the system, represented as a snapshot of a domain specific model.

This is done by *matching*: this concept is defined as finding a particular set of elements in the current snapshot (match) or not (no match) in [11].

3 Combining RV and DSMLs

DSMLs used for behavior generally have concepts along the lines of actions being executed and connections among them that define the flow sequence in which the actions are executed. In this section, we introduce our approach using an example DSML which is a simple realization of both kinds of concepts. Together with this DSML, we define the integration of its behavioral semantics with the evaluation of temporal properties. We achieve such integration by linking the elements of the DSML with the atomic propositions used in Linear Temporal

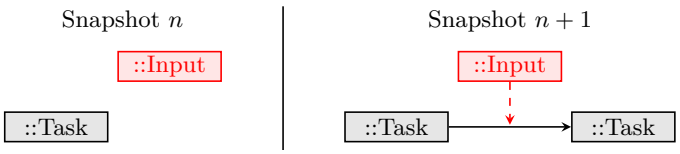

Fig. 2. Sketch of the model transformation for transition triggering. A transition is triggered (its instance appears in a new snapshot) if it is connected to a task and an input in the model, which have active instances in the current snapshot.

Logic (LTL) formulas. All these parts are included in the modeling hierarchy depicted in Fig. 1.

3.1 Example of behavioural DSML

The DSML used in the example defines three types of elements:

Input Used to incorporate information from the environment into the model. Inputs allow the system to react to external stimuli, such as sensor information in a robot. Hence, inputs appear in a snapshot when any of the aforementioned happens, and disappear afterwards. Their appearance cause generally a change of state in the system (new snapshots).

Task A specific action or set of actions executed by the system. A task is running if an instance of it appears in the current snapshot. Multiple tasks can run at the same time.

Transition Represents the order in which tasks are executed. Every transition is connected to a source and a target task, and associated to one input. When the source task is running (i.e. appears in the current snapshot) and the associated input appears, the transition is triggered. After a transition is triggered, a new snapshot is generated in which the transition and the target task start running. Notice that a task may have more than one incoming transition, as well as multiple outgoing transitions. In the first case, as soon as any of the incoming transitions is fired, the target of that transition is activated. In the second case, all of the transitions with the same input are fired at the same time. That is, the target tasks of all fired transitions start running in parallel.

When the system is simulated, new snapshots are generated using model transformations. These transformations are not explained in detail here due to space limitations. A richer example similar to ours can be found in Table III of [16]. We adapted the syntax that the authors use in our example. Generally speaking, these model transformations are responsible for the simulation of inputs appearing, transitions being fired and tasks finishing (disappearing). Figure 2 shows an illustrative example of the MT for the triggering of a transition.

Inputs are used in our DSML to model all the possible happenings that may cause a change of state. The monitors used to evaluate temporal properties require that a snapshot is generated only when there is a change of state of the system, i.e. no two consecutive snapshots are the same. Besides, the monitors need to be aware of any registered input, even if it does not trigger any transition.

Hence, inputs are modeled as triggers for transitions, but the appearance of an input in a snapshot is independent from the transitions that it may trigger.

Note that the execution of our system does not include any notion of time. In some cases it is nevertheless useful to model the expiration of a certain amount of time. We do this by introducing timers which raise a timeout after a specified time. A timer may be started on activating a task. In the model we abstract away from the time passing by and represent timeouts as regular inputs. The timeout inputs can be used like any other input in order to trigger a transition. In a nutshell, this allows us to handle time while keeping the discrete LTL semantics defined on a sequence of states.

3.2 Linking a DSML with temporal properties

As presented in [11], we implement the syntax of a temporal logic as a linguistic extension. This extension is orthogonal to the model hierarchy composed by the DSML, the particular system model, and the current snapshot (see Figure 1).

The key concept of a linguistic extension for our work is the possibility to connect *any* type of element or set of elements in the modeling hierarchy to the model representing a temporal property. In this work, we will connect single elements in the snapshots to the temporal properties. As a consequence of this way of modeling LTL properties, an atomic proposition is a fragment of a model instance that may appear in a snapshot. The atomic propositions are evaluated as follows: If at least one match of the fragment appears in the current snapshot of the system, the atomic proposition is evaluated to true otherwise to false.

This connection of elements and atomic propositions allows us to look at the sequence of snapshots of a system as the execution of that system, from both RV and modelling points of view. So we can do runtime verification with temporal logics in a natural way based on those snapshots because they represent the states of the system during the execution. An example for this can be seen in Figure 3. In this example we modeled a robot driving around. The figure shows how inputs are part of the model of the system, and at the same time are linked to the atomic propositions of the LTL formula. The correctness property given in Figure 3 states that obstacles found in front of the robot disappear if the robot moves backwards. Otherwise the robot has found a moving obstacle coming after it. This property describes the environment of the robot and hence is specified using atomic propositions linked to inputs. Atomic propositions can be linked to tasks in the same way if the property expresses the behavior of the robot in a more direct way.

The connection of the atomic propositions to the elements enables us to expand our approach to different LTL semantics like LTL_3 [2] in order to report final fulfillment or violation of the correctness properties as soon as possible when monitoring. Also, we can expand to LTL extensions like timed LTL [14] that allows us to express real-time properties or add other theories for using variables instead of only boolean propositions [4], and quantified propositions [19].

Notice that the possibility of connecting any element (or set of them) to atomic propositions greatly enriches LTL via its atomic propositions.

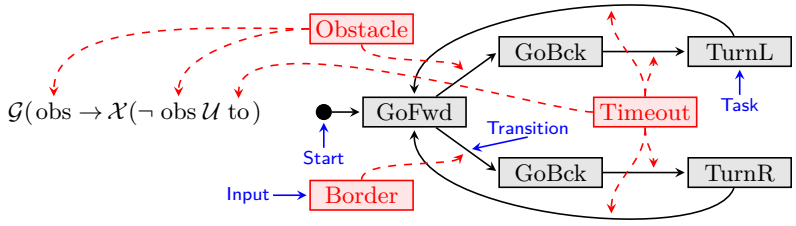

Fig. 3. Modeling of the behavior of a robot with an attached correctness property detecting moving obstacles. The robot can move forwards (GoFwd), move backwards (GoBck) and turn left or right (TurnL, TurnR). The transitions are activated by the inputs and activate the subsequent task. Obstacle and Input represent sensor inputs and Timeout represents the expiration of a timer started by GoBck, TurnL and TurnR. That is, Timeout is used to represent the amount of time that a Task takes to finish.

4 Conclusion and Outlook

Our approach integrates behavioral models of DSMLs and correctness properties for RV through the whole software engineering process of design, simulation and code generation. For design, we provide a framework which allows to specify correctness properties as a linguistic extension of the DSML modeling hierarchy. In the simulation, a monitor and a model can be executed at once by synchronizing the model simulation and the monitor execution. Thereby the simulation is defined as a sequence of snapshots which are enriched with atomic propositions based on the match of certain elements in the snapshots. Finally, this synchronization between correctness properties and model execution is reused in order to execute the monitors and the designed system in the generated code for the target platform. The connection between the correctness property and the modeled behavior of the system is kept consistent throughout the whole software engineering process.

We implemented this approach in Eclipse EMF[1] and generated Python code for the ev3dev platform[2] in order to control Lego EV3 robots. The Python code for the monitors is generated by using the LTL_3 semantics with the logic and automata library LamaConv[3].

We plan to extend this approach for designing and verifying asynchronous distributed systems. Such systems generally have a huge state space generated by the environmental influence because of the high asynchronous fashion. Hence static verification approaches become difficult to use. With metamodeling, such systems can be designed in a more abstract way and monitors can be connected on multiple abstraction layers to the agents they should observe. To achieve such extension, we need to be able to model single agents of the distributed system and connect their actions with each other such that we can model their communication. Our goal is to increase the applicability of existing approaches of RV for asynchronous distributed systems like the ones in [13], [18] and [17] by combining them with DSMLs as presented in this paper.

[1] `eclipse.org/modeling/emf` [2] `ev3dev.org` [3] `isp.uni-luebeck.de/lamaconv`

References

1. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling 7(3), 345–359 (2008)
2. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14:1–14:64 (2011)
3. Callow, G., Watson, G., Kalawsky, R.: System modelling for run-time verification and validation of autonomous systems. In: System of Systems Engineering (SoSE). pp. 1–7. IEEE (2010)
4. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. Software Tools for Technology Transfer (STTT) 18(2), 205–225 (2016)
5. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software & Systems Modeling 14(1), 173–199 (2015)
6. Fowler, M.: Domain-specific languages. Pearson Education (2010)
7. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Software Tools for Technology Transfer (STTT) 14(1), 15–40 (2012)
8. Lehmann, G., Blumendorf, M., Trollmann, F., Albayrak, S.: Meta-modeling runtime models. In: Models in Software Engineering, LNCS, vol. 6627, pp. 209–223. Springer (2010)
9. Leucker, M.: Teaching runtime verification. In: Runtime Verification (RV). LNCS, vol. 7186, pp. 34–48. Springer (2011)
10. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
11. Macias, F., Rutle, A., Stolz, V.: A Property Specification Language for Runtime Verification of Executable Models. In: Nordic Workshop on Programming Theory (NWPT). pp. 97–99 (2015), Tech. Rep. RUTR-SCS16001, School of Computer Science, Reykjavik University
12. Mellor, S.J.: MDA distilled: principles of model-driven architecture. Addison-Wesley Professional (2004)
13. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: Parallel and Distributed Processing Symposium (IPDPS). pp. 494–503. IEEE (2015)
14. Raskin, J., Schobbens, P.: The Logic of Event Clocks – Decidability, Complexity and Expressiveness. J. Autom. Lang. Comb. 4(3), 247–286 (1999)
15. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. Formal Aspects of Computing 26(6), 1115–1152 (2014)
16. Rutle, A., MacCaull, W., Wang, H., Lamo, Y.: A metamodelling approach to behavioural modelling. In: Behaviour Modelling-Foundations and Applications. pp. 5:1–5:10. ACM (2012)
17. Scheffel, T., Schmitz, M.: Three-valued asynchronous distributed runtime verification. In: Formal Methods and Models for Codesign, MEMOCODE. pp. 52–61. IEEE (2014)
18. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient Decentralized Monitoring of Safety in Distributed Systems. In: Software Engineering (ICSE). pp. 418–427. IEEE (2004)
19. Stolz, V.: Temporal assertions with parametrized propositions. J. Log. Comput. 20(3), 743–757 (2010)
20. Wang, H., Rutle, A., MacCaull, W.: A formal diagrammatic approach to timed workflow modelling. In: Theoretical Aspects of Software Engineering (TASE). pp. 167–174. IEEE (2012)