Explicit Modelling of Model Debugging and Experimentation

Simon Van Mierlo

University of Antwerp, Belgium

Keywords: Model-Driven Engineering, Debugging, Simulation, Statecharts

Abstract. In this paper, I present the topic of my PhD: the explicit modelling of model debugging and experimentation. Semantics of modelling formalisms include non-determinism, concurrency, and hierarchy, amongst others. Moreover, simulated time can have different relations to the wall-clock time and supports certain operations such as pausing. Providing debugging support for model simulations is non-trivial using traditional software development techniques. We therefore propose to model simulators, their debuggers, and environments explicitly.

1 Problem Definition

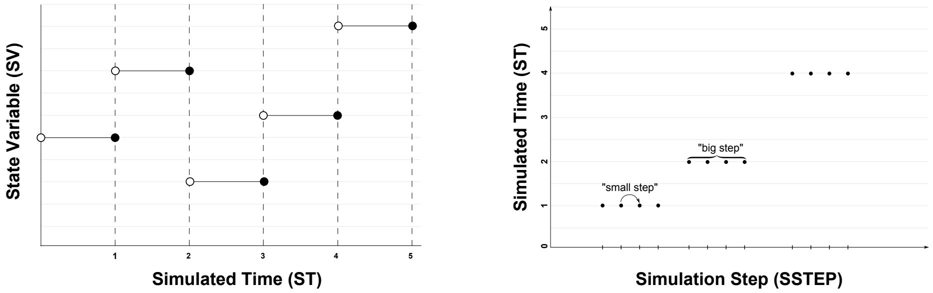
The systems we analyse, design, and develop today are characterized by an ever growing complexity. Modelling and Simulation (*M&S*) become increasingly important enablers in the development of such systems, as they allow rapid prototyping and early validation of designs. Domain experts, such as automotive or aerospace engineers, build models of the (software-intensive) system being developed and subsequently simulate them having a set of "goals" or desired properties in mind. Every aspect of the system is modelled, at the most appropriate level of abstraction, using the most appropriate formalism [1]. The M&S approach can only be successful if there is sufficient tool support, *i.e.*, if the modeller has access to tools which sufficiently support each phase in the M&S approach. This is no different from traditional, code-based software development methods: programmers have access to various helpful tools such as version control software, testing tools, and debuggers. Debuggers allow to locate the source of a defect (which was detected by a failing test, meaning that one of its properties was not satisfied) using breakpoints, stepping, and tracing of runtime variables [2]. Support for simulation debugging is currently limited, and a challenging issue, as the next subsections explain.

1.1 Formalisms

As was mentioned above, different aspects of a system are modelled in a number of different modelling formalisms. This means that, unlike a programming project, where usually one programming language is used, a debugger has to be aware of the different semantic aspects of these formalisms. A few examples of such languages include Petrinets [3] and rule-based model transformations, which allow for *non-determinism*, Statecharts [4], which allow to model *concurrency* and *timed, reactive* behaviour, and Modelica [5], which allows to model sets of mathematical equations. These formalisms can be combined, either using “embedding” (for example, action language in constraints or actions of patterns in the rules of a model transformation), or, for hybrid systems, by composing them heterogeneously, for which semantic adaptation methods were developed [6].

1.2 Time

The notion of *time* plays a prominent role in model simulation. Simulated time differs from the wall-clock time: it is the internal clock of the simulator. In general, a simulator updates some state variable vector, which keeps track of the current simulation state, each time increment. This is shown visually in Figure 1a. The state is updated by some computations, or “steps”: a big step corresponds to the computation of the next value of the state variable, and consists of a number of small steps.



(a) Change of the state variable over time.

(b) Multiple steps executed on a single simulation time instance.

Fig. 1: Simulation time and steps.

Executing program code is always done as fast as possible, *i.e.*, the speed of the program is limited by the machine executing it. Simulations, however, have an additional notion of time: the *simulated time*. A simulation can be run as-fast-as-possible, or in (scaled) real-time, which is useful for simulating models of real-time systems which might be deployed as such on a real-time device. In this case, there is a linear relation between the wall-clock time and the simulated time. The relation of the different notions of simulated time and the wall-clock time is shown visually in Figure 2. Note that there is no linear relation in as-fast-as-possible simulation, meaning that the “current simulation time” is simply a variable in the simulator. Moreover, operations can be performed on simulated time, such as pausing, or stepping back, which are not allowed on wall-clock time.

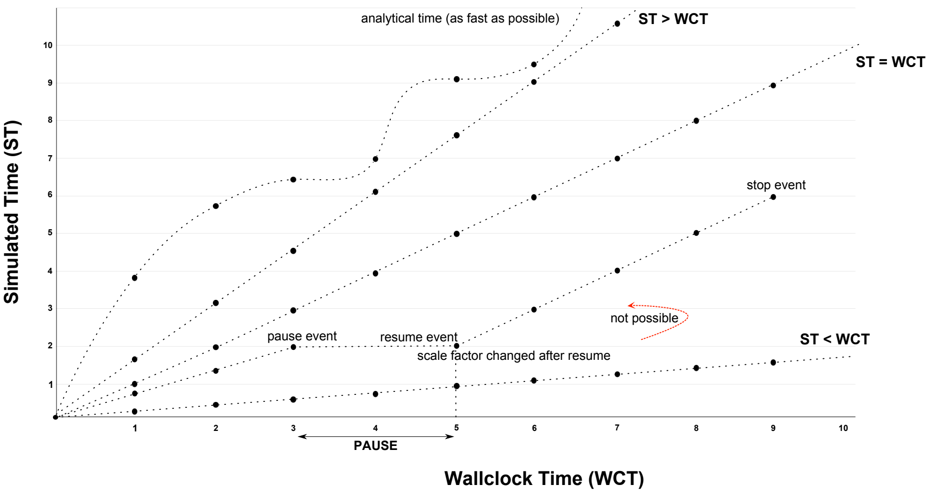


Fig. 2: Different notions of simulated time.

1.3 User Interaction

Users interact with a simulation through the simulation environment. The interleaving of user events coming from the environment with the real-time, interruptible behaviour of the simulator (or interacting simulators, in the case of hybrid system simulation), is non-trivial.

The challenge is to manage the inherent complexity of constructing model debugging and experimentation environments. The interplay of formalism execution semantics, different notions of simulated time, and user interaction makes this a challenging task if traditional software development methods are used. While examples exist of model simulation debuggers implemented in code, it is clear that, especially for multi-formalism environments, techniques are needed to overcome this complexity.

2 Related Work

One research question of the PhD is how concepts of program debugging are mapped onto model simulation. For example, what does it mean to step over or into a state in a Statecharts model, or a Place in a Petrinets model? In some cases, this mapping is straightforward: stepping into a composite Statechart state will change the scope of the simulation to its contained states. Sometimes, such a mapping cannot be constructed. Mannadiar and Vangheluwe [7] survey the state-of-the-art in debugging and explore how these concepts can be translated to the realm of Domain-Specific Modelling.

Model debugging has received some attention in the literature on the Modelica language. In [8–11], the authors develop techniques for debugging equation-based models, which differ greatly from sequential programs, where each statement is executed one after the other. They look at static and dynamic debugging, as well as how to make the debugging techniques scalable for large models.

In [12], Mustafiz and Vangheluwe tackle the problem of constructing a debugging environment for debugging Statecharts by explicitly modelling it as a Statechart. In essence, the debugged Statechart is embedded into the Statechart describing the behaviour of the environment.

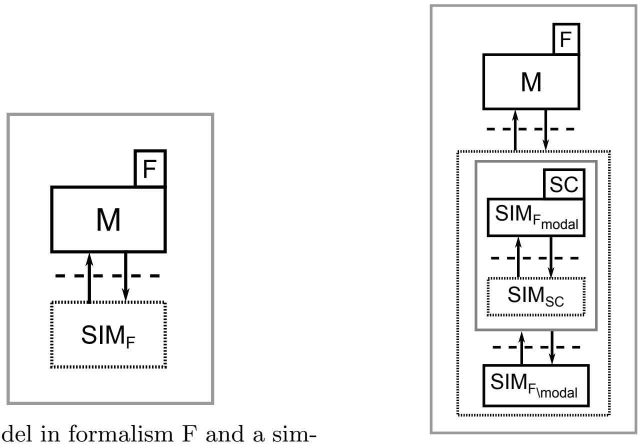
3 Proposed Solution and Expected Contributions

The goal of this thesis is to construct a set of techniques for developing useful debuggers for model simulations, taking inspiration from the code debugging world and the simulation-specific concepts discussed in Section 1. In this section, we further split this up into a number of sub-goals.

A first goal is to construct a set of debugging environments for a number of well-known general-purpose modelling formalisms: Petrinets, Statecharts, Causal Block Diagrams (CBDs) [13], DEVS [14], Modelica, and rule-based model transformations (MoTif [15] in particular). Most simulators and simulation environments are implemented in code. On top of the inherent complexity, this software development method brings with it accidental complexity, as it is not the most appropriate language to develop a timed, reactive system with complex interleaving of user and simulator events. We therefore propose to explicitly model the debugging environment using Statecharts. In general, each simulator has a "main simulator loop". This comprises a number of states and transitions between them, performing some action that updates the state of the model. In the most naive case, there is a single state which represents the main loop of the simulator and a transition going from and to that state, each time performing one simulation "step". This is the so-called "modal part" of a simulator, which is intuitively represented as a Statechart.

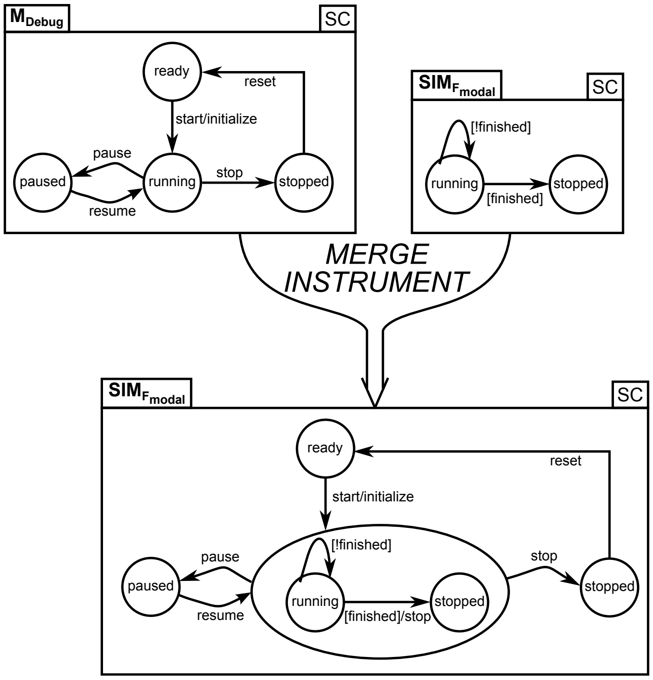
The process of extracting this modal part, which we call de/reconstructing the simulator, is shown in Figure 3. The first step, deconstrucing the simulator, extracts the modal part of the simulator in a Statechart (SC) model called $SIM_{F_{modal}}$. This model is combined with a Statechart simulator, interpreter, or compiler called SIM_{SC} to give it operational semantics. The Statechart together with its executor interface with the non-modal part of the simulator for formalism F ($\mathrm{SIM}_{F_{\setminus modal}}$, which, in this case, consists of the coded functions to run the simulator). The combination of the modal and non-modal part of the simulator results in a behaviourally equivalent simulator to SIM_F. From the user's point of view, the black-box containing the model to be simulated and its simulator is unchanged.

In Figure 3c, the last step in creating an instrumented simulator is shown. We *merge* the modal part of the simulator for F with the behavioural



(a) A model in formalism F and a simulation kernel for F.

(b) De/Reconstructing the simulator.



(c) Merging the debugging concepts with the modal behaviour of the simulator.

Fig. 3: The workflow for explicitly modelling the simulator's behaviour.

model of the debugger. This results in an instrumented model of the modal behaviour of the simulator. The last step is to replace $SIM_{F_{modal}}$ in Figure 3b with this instrumented model. Again, this should not change the behaviour of the simulator in any way if the user does not make use of the debugging functionality. Extra behaviour has been added, but running the simulator as before is still possible. In the (trivial, but representative) example shown, the debugger includes the concepts of *start*, *pause*, *resume*, and *stop*. The simulator only has two states: *running*, and *stopped*. It runs the main loop of the simulator until the *finished* condition is satisfied, signalling that the simulation is done.

In a more advanced stage of the thesis, the idea is to apply the same techniques for environments which allow heterogeneous model composition, for which multiple simulators and semantic adaptation are necessary. It is our intuition that explicitly modelling the debugger and simulator will facilitate this.

4 Preliminary Work

The techniques described in the previous section have been successfully applied to model an experimentation and debugging environment for Causal-Block Diagrams. We took an existing CBD simulator, identified its modal part, and extracted it as a Statechart model. We then instrumented this model with debugging support. A graphical user interface allowed users to simulate (as-fast-as-possible or in (scaled) real-time), pause simulation, and step through the simulation (either "big step", meaning the values of all blocks for the next iteration were calculated at once, or "small step", where only the value of one block was computed). It also allowed to set breakpoints and define the maximum number of simulation iterations.

5 Evaluation and Validation

From the very onset of the project, the goal is adoption of the developed methods, techniques, (prototype) tools and processes in industry (and academics). During each phase of the project, interaction with companies is planned as to align industrial needs and scientific developments. This will ensure the results are relevant and usable in an industrial context. Moreover, the results of my work are to be published in a number of different communities. For the simulation side, conferences such as *SpringSim* and *WinterSim* are ideal venues for early validation of results. In a later stage, results will be published in the *SIMULATION* journal. For the model-driven engineering side, the *MODELS* and ECMFA conferences, and the journals *Software and Systems Modelling* (SoSyM) and *Science of Computer Programming* are ideal. The ICGT and ICMT conferences, as well as the GraBaTs workshop, are ideal venues for the validation of results when using model transformations.

Having both industrial and scientific feedback from a strong community will ensure the quality of the work, and the relevance of the results. Currently, I do not plan usability studies of the proposed approaches. The main contribution of my work are the techniques for constructing advanced debugging environments by explicitly modelling them. Validation of the techniques is provided by implementations for specific (combinations of) formalisms.

6 Current Status

Currently, most of my efforts are directed towards developing the required foundations. In particular, I'm working on a new metamodelling framework, which enables modular language design and uniform access to models called the *Modelverse*. This framework, as well as its web-based front-end called AToMPM [16] will be used to implement the techniques described in this paper.

I started in September of 2013. The end of my PhD is planned in December 2017. Below is a course-grained schedule, with expected contributions per year:

- **2014**: In the coming months, I will be working on a debugging environment for Parallel DEVS. I will also redo the work of [12], but extending the set of debugging operations, as well as apply the techniques which I propose here, instead of embedding the model directly.
- **2015**: In this year, I will define a set of debugging environments and model environments explicitly for a number of other formalisms, including rule-based model transformations, Petrinets, and Modelica.
- **2016-2017**: I will validate my techniques by interacting with industry. Furthermore, I will extend the techniques to multi-formalism environments and domain-specific languages.

I will also continue working on the language engineering topics - in particular, the explicit modelling of entire modelling languages using the Modelverse, and the effect on explicit modelling of transformation and the technique of RAMification [17].

Acknowledgements. This work was partly funded with a grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

1. P. J. Mosterman and H. Vangheluwe, "Computer Automated Multi-Paradigm Modeling: An Introduction," *Simulation*, vol. 80, pp. 433–450, Sept. 2004.
2. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.
3. T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
4. D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, June 1987.

5. P. Fritzson and P. Bunus, “Modelica – a general object-oriented language for continuous and discrete-event system modeling,” in *IN PROCEEDINGS OF THE 35TH ANNUAL SIMULATION SYMPOSIUM*, pp. 14–18, 2002.
6. F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet, “Semantic adaptation for models of computation,” in *Application of Concurrency to System Design (ACSD), 2011 11th International Conference on*, pp. 153–162, June 2011.
7. R. Mannadiar and H. Vangheluwe, “Debugging in domain-specific modelling,” in *Software Language Engineering* (B. Malloy, S. Staab, and M. Brand, eds.), vol. 6563 of *Lecture Notes in Computer Science*, pp. 276–285, Springer Berlin Heidelberg, 2011.
8. M. Sjölund and P. Fritzson, “Debugging Symbolic Transformations in Equation Systems,” in *EOOLT, Linköping Electronic Conference Proceedings vol. 56*, pp. 67–74, 2011.
9. M. Sjölund, F. Casella, A. Pop, A. Asghar, P. Fritzson, W. Braun, L. Ochel, B. Bachmann, and P. Milano, “Integrated Debugging of Equation-Based Models,” in *Proceedings of the 10th International ModelicaConference*, pp. 195–204, 2014.
10. A. Pop, M. Sjölund, A. Asghar, P. Fritzson, and C. Francesco, “Static and Dynamic Debugging of Modelica Models,” in *Proceedings of the 9th International Modelica Conference*, pp. 443–454, Nov. 2012.
11. A. Asghar, A. Pop, M. Sjölund, and P. Fritzson, “Efficient Debugging of Large Algorithmic Modelica Applications,” in *Proceedings of MATHMOD 2012 - 7th Vienna International Conference on Mathematical Modelling*, 2012.
12. S. Mustafiz and H. Vangheluwe, “Explicit modelling of statechart simulation environments,” in *Summer Simulation Multiconference*, pp. 445 – 452, Society for Computer Simulation International (SCS), July 2013. Toronto, Canada.
13. H. Vangheluwe, D. Riegelhaupt, S. Mustafiz, J. Denil, and S. Van Mierlo, “Explicit modelling of a CBD experimentation environment,” in *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, TMS/DEVS ’14, part of the Spring Simulation Multi-Conference, pp. 379 – 386, Society for Computer Simulation International, 2014.
14. B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation.* Academic Press, 2 ed., 2000.
15. E. Syriani and H. Vangheluwe, “A modular timed graph transformation language for simulation-based design,” *Software and Systems Modeling (SoSyM)*, vol. 12, no. 2, pp. 387–414, 2013.
16. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, “AToMPM: A web-based modeling environment,” in *MODELS’13 Demonstrations*, 2013.
17. T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, “Explicit Transformation Modeling,” in *MoDELS Workshops, Lecture Notes in Computer Science vol. 6002*, pp. 240–255, Springer, 2009.