Explicitly Modelling Model Debugging Environments

Simon Van Mierlo
University of Antwerp

Simon.VanMierlo@uantwerpen.be

***Abstract*—Programmers spend a large portion of their time debugging the code they write. This is supported by a variety of debugging techniques such as pause/resume, the setting of breakpoints, stepping over functions, etc. Today, modelling and simulation become increasingly important enablers in the development of complex, reactive, often real-time, software-intensive systems, as they allow rapid prototyping and early validation of designs. Simulation models, though at a higher level of abstraction than code, can however still contain bugs. There is hence a need for model-level debuggers, that are adapted to the semantics of the modelling formalism(s) used, and can properly deal with the timed nature of many of these models. This paper presents a method for constructing model debugging environments for deterministic, operational formalisms. In order to manage the inherent complexity, the timed, reactive behaviour of the debugger is modelled explicitly. The feasibility of the approach is demonstrated by constructing a visual debugging environment for Causal-Block Diagrams.**

I. Introduction

Programmers spend a large portion of their time debugging the code they write [1]. This is supported by a variety of debugging techniques such as pause/resume, the setting of breakpoints, stepping over functions, tracing runtime variables, etc. More recently, advanced techniques such as program slicing, algorithmic debugging, and omniscient debugging attempt to speed up the debugging process. This has lead to what Zeller calls "scientific debugging", where programmers try to find the source of an observable error by systematically formulating and refuting hypotheses [2].

The systems we analyse, design, and develop today are characterized by an ever growing complexity. Modelling and Simulation (*M&S*) [3] 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. Ideally, every aspect of the system is modelled at the most appropriate level(s) of abstraction, using the most appropriate formalism(s) [4]. The M&S approach can only be successful if there is sufficient tool support, including debuggers that enable locating modelling errors. Often, code is generated from models, which can be debugged using traditional methods. In that case, however, the modeller has to make a context switch, and errors in the generated code might be difficult to link back to model elements. There is a need for specialized model debugging environments at the correct level of abstraction using the right notations and operations.

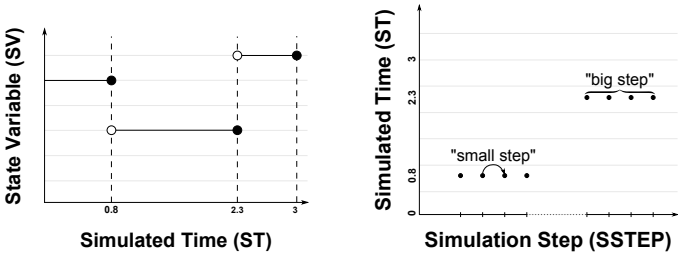
Constructing such environments is an inherently complex task. The interplay of formalism execution semantics, different notions of simulated time such as (scaled) real-time and as-fast-as-possible execution semantics, as well as user interaction through an interface are all challenging to capture and implement correctly using traditional code-centric software development techniques. In this paper, we present a generic method and architecture for constructing model debugging environments for deterministic, operational formalisms. In order to manage the inherent complexity, the timed, reactive behaviour of the debugger is modelled explicitly as a Statechart [5] model. The model is derived from the behaviour of a simulator or executor without debugging support, which is instrumented with debugging operations using a process we call the "de- and reconstruction" of the simulator. This simulator with debugging support is combined with an appropriate visual modelling environment that allows to visualize the state of the system as it evolves over time, and allows the user to interact with the simulator through an appropriate interface.

Section II explains how models can be debugged using a variety of operations. Section III presents our generic method for constructing a debugging environment for a formalism for which only a simulator and modelling environment exist. Section IV applies this method to construct a debugging environment for the Causal-Block Diagrams [6] formalism. Section V discusses related work, and Section VI concludes the paper.

II. Debugging Models

This section explores how models can be debugged. We take an operational view and assume the model is simulated by an appropriate simulator. After simulation, the modeller observes an erroneous result and—similar to what a programmer would do—wants to inspect the dynamics of the simulation. An appropriate debugger should offer functionality similar to code debuggers: stepping, pausing, setting breakpoints, etc. Furthermore, formalism-specific debugging operations should be provided, especially if the model exhibits real-time behaviour.

First, a generic simulation algorithm is presented assuming a formalism in which the state of the system evolves over (simulated) time in steps. Second, the notion of simulated time and its relation to the wall-clock time is explained. Then, different operations on the state of the system are presented. Last, the concept of breakpoint is investigated in the context of model debugging.



(a) Change of state over simulated time.

(b) Multiple steps at a single simulated time.

Fig. 1: Simulation time and steps.

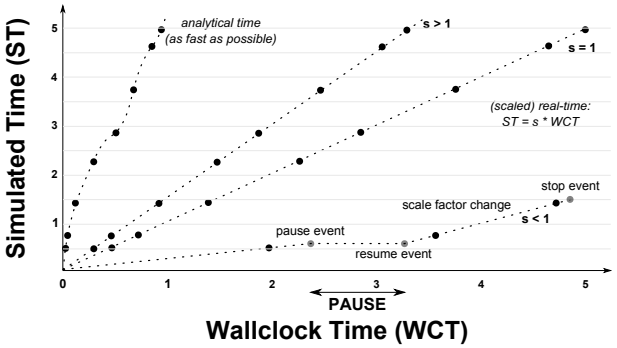


Fig. 2: Different notions of time.

A. A Generic Simulator

Algorithm 1 A generic simulation algorithm.

```
1: time ← 0
2: state ← INITIALIZE_STATE(model)
3: while not END_CONDITION(state, time) do
4:     state ← COMPUTE_NEXT(model, state)
5:     time ← INCREMENT_TIME(model, state, time)
6: end while
```

Algorithm 1 presents the pseudocode of a generic simulation algorithm, capable of simulating models in a particular formalism. First, in lines 1–2, the value of the simulated time is initialized, as well as the model state. Then, lines 3–6 contain the "main loop": the simulation evolves the state of the system until some end condition has been satisfied. A "step" consists of updating the state of the system (as dictated by the model) and incrementing the simulated time. The end condition can depend on the state and the simulated time.

Fig. 1b depicts the evolution of the state (variable) over simulated time. Three time steps (iterations of the *while* loop) are shown. Note that the discontinuities and the fact that state updates are performed in non-equidistant intervals assume a variable step size or discrete-event formalism (such as DEVS [7]). Simulators for discrete-time formalisms update the state in equidistant intervals, while the state evolution of continuous models approaches a continuous function.

The "meat" of the algorithm is the call to the function that computes the next state, on line 4. This call constitutes one "step" of the simulation, and after it has completed successfully, the system is in a valid state. In between, however, a number of small computation steps may be involved. This is visualized in Fig. 1b, where one "big step" is broken up into a number of "small steps". Note that the simulated time stays constant in between small steps, and only increases after a big step has been completed.

B. Time

The notion of time plays a prominent role in model simulation. Simulated time differs from the wall-clock time: it is, as explained above, the internal clock of the simulator. Simulated time can, however, have a well-defined relation to the wall-clock time.

Program code is always executed as fast as possible (*i.e.*, the speed of the program is limited by the resources of the machine executing it). Simulations, however, can either be run as-fast-as-possible, or in (scaled) real-time. The latter 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 different relations of the simulated time to the wall-clock time are depicted in Fig. 2.

In as-fast-as-possible simulation, there is no relation between simulated time and wall-clock time, meaning that simulated time is simply a variable in the simulator. In real-time simulation, simulated time is synchronized with the wall-clock time. This implies that the simulation steps have a hard real-time deadline (*i.e.*, the values of the runtime variables have to be computed before the wall-clock time reaches the simulated time). A scale factor s can be applied to speed up or slow down simulation, while maintaining the linear relation between simulated time and wall-clock time.

Moreover, operations can be performed on simulated time, such as pausing, or stepping back, which are obviously not allowed on wall-clock time.

C. State

As explained, the system state evolves over (simulated) time during simulation. Usually, the modeller knows the initial state (since it is captured in the model), as well as the end state (the result of simulation often is an aggregation of values found herein). Inspecting the state during simulation is an important part of debugging, as it allows to see how the system evolves over (simulated) time. Certain simulators already allow to define tracers which textually or visually output the state of the system during simulation. What constitutes state depends heavily on the formalism—in case of program code, it is the data values found in memory, the program counter, the stack, etc.

A debugging environment can, next to state visualization, allow to manually change the state of the system during simulation. This helps in refuting hypotheses related to the source of an error: changing the value of a suspect variable and observing how the system dynamics change can rule out that particular value as being the cause of the error, for example.

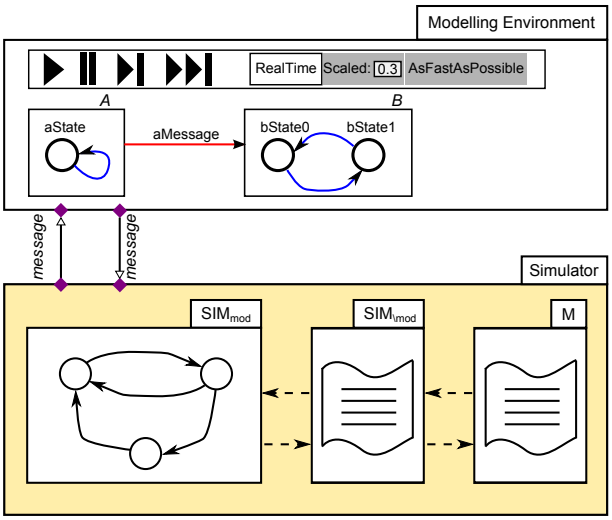


Fig. 3: The generic architecture.

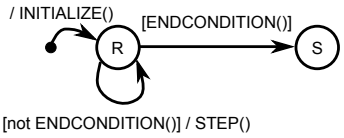


Fig. 4: The modal part of the generic simulation algorithm.

D. Breakpoints

Manually pausing simulated time was discussed above. Breakpoints allow to automatically pause the execution when a condition is satisfied. In program code, breakpoints allow to pause when a specific line of code is reached. Additionally, it can be augmented with a condition on the runtime state. A model debugger might expose, similarly, a way of setting breakpoints that depend on the runtime state of the system.

III. Method

Taking the requirements from the previous section, this section describes a method for constructing a model debugging environment, starting from a simulator without debugging support, and a (visual) modelling environment. First, a high-level view of the architecture is presented. Then, a method is given for modelling the behaviour of the simulator to add debugging support.

A. Architecture

Fig. 3 shows the generic architecture for a debugging environment. At the top, a model with two elements *A* and *B* is shown inside a visual modelling environment. The formalism this model conforms to is not defined, as it is irrelevant for the discussion in this section. It could be a model of two communicating processes which each have internal state and can send and receive messages.

A simulator capable of simulating models in that formalism is shown below. The internals of the simulator will be discussed in the next subsection. The simulator can simulate a model *M*, which is an exported (possibly compiled) version of the model from the modelling environment. It exposes some interface to control the simulation—in the simplest case, allowing the simulation to be started and collecting the results upon termination of the simulation. This is depicted by the purple rectangles. The modelling environment was augmented with a toolbar that allows to control the simulation— pressing a button will send a message to the simulator, which will react appropriately. This means that the model editing environment can also serve as a simulation environment.

We require the modelling environment to have a software interface as well, such that it can be interacted with, not just interactively through a UI, but also programmatically. This interface should at least support Create, Read, Update, and Delete (CRUD) requests to alter the model, as well as functions to highlight elements. This allows the simulator to alter the state of the model during simulation, which is necessary to display the state of the model during simulation, for animation, and ultimately debugging.

B. Adding Debugging Support

Any simulation algorithm can be written in the form presented in Algorithm 1. This form separates the *modal* part of the simulator (the flow of control, mainly consisting of the "main simulation loop") from the *non-modal* part. This modal part can be "lifted out" and modelled in an appropriate formalism, for which we choose Statecharts [5]. Fig. 4 shows the modal part of the generic simulator. When this model is combined with an appropriate executor that implements the semantics of the Statecharts formalism, and combined with the non-modal part of the simulator (implemented in the functions called in the actions of the Statechart transitions), we obtain a version of the simulator that implements identical semantics. It is now simply broken up in two parts. In Fig. 3, its components SIM_{mod} and $SIM_{\setminus mod}$ are shown.

In Fig. 5, the last step in creating a simulator which supports debugging is shown. We *merge* the modal part of the simulator behaviour model with a model capturing the debugging operations we want to add. This results in an instrumented model of the modal behaviour of the simulator. The last step is to replace SIM_{mod} in Fig. 3 with this instrumented model. For continuity reasons, the behaviour of the simulator should be unaltered 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 example shown, the debugger includes the concepts of *start*, *pause*, *resume*, and *stop*. The simulator only has two states: *running*, and *stopped*. This is a trivial (and fictional) example, but it demonstrates the process which we call de- and reconstruction of the simulator.

The result of reconstructing the simulator is an instrumented version of the original simulator, enriched with debugging capabilities. From this model, a debugging and experimentation environment for formalism F can be automatically generated using a Statecharts compiler.

IV. Example: CBD Debugging Environment

Causal Block Diagrams (CBDs), also known as Synchronous Data Flow, is a modelling language that consists of

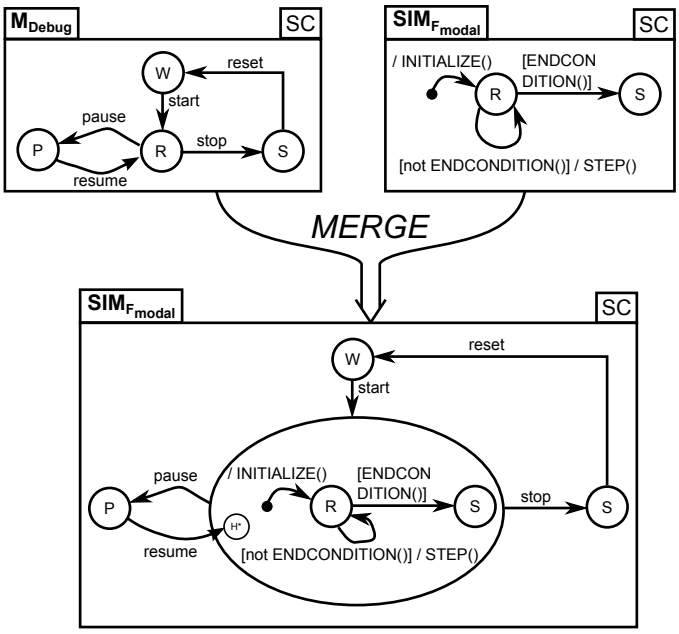


Fig. 5: Merging the modal part of the simulator with debugging concepts.

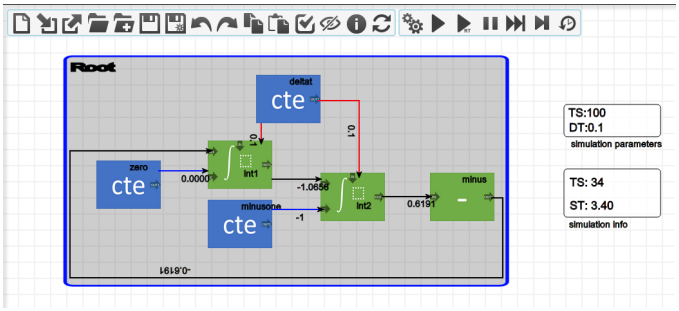


Fig. 6: A generated visual modelling and simulation/debugging environment for Causal Block Diagrams.

a network of *blocks* denoting mathematical operators, expressions that evaluate to Boolean values, memory, and constants. Connections between blocks denote signals (values as function of time).

An example CBD is shown in Figure 6. It models a set of mathematical equations. It is built in AToMPM [8], the visual environment we use for modelling language engineering. AToMPM can synthesize visual modelling environments from a definition of the abstract and concrete syntax of a modelling language. In this case, we have modelled the syntax of CBDs and modelled the example in the generated environment. AToMPM is web-based and exposes a public interface, which can be called using HTTP requests. It can furthermore be extended with plug-ins, that are exposed to the user through toolbars. We will use these two mechanisms to extend the behaviour of AToMPM and implement the architecture shown in Fig. 3.

A CBD is simulated by updating the values of all its blocks in every step. The pseudocode for the algorithm of the CBD simulator is shown in Algorithm 2.

Algorithm 2 The CBD simulator's "main loop".

```
1: time ← 0
2: while not END_CONDITION(time) do
3:     schedule ← LOOPDETECT(DEPGRAPH(cbd))
4:     for gblock in schedule do
5:         COMPUTE(gblock)
6:     end for
7:     time ← time + δ_t
8: end while
```

Three functions are central to the algorithm:

- *LOOPDETECT* computes all loops found in the CBD. A loop exists when the input of a block is computed by another block reachable from the original block. The function returns a collection of all blocks and loops, in the order that they need to be computed.
- *DEPGRAPH* computes dependencies between blocks, which is needed for the loop detection algorithm.
- *COMPUTE* contains the code that performs each block's computation on its input signals, updating the value of its outgoing signals (for example, the current value of the output signal of a sum block equals the sum of the current values of its input signals).

Each time step, the simulator iterates over all blocks in the correct order and computes the values of each block's outgoing signals. Debugging a CBD simulation requires "lifting out" the outer *while* loop, allowing us to interactively step through each iteration. On a more fine-grained level, the inner loop can be lifted out too, allowing one to interactively step through the computation of individual blocks.

The end condition depends on the simulated time. It is modelled in the "simulation parameters" element in the graphical user interface (see Fig. 6). Here, the simulation will end after 100 time steps. Furthermore, the time increment (δ_t) is set to 0.1. The model and parameters are compiled to the format the simulator requires (this is not necessarily the same as the format the user interface requires). When the simulator completes a run, it returns the final state, which is displayed in the user interface. A mapping between elements in the user interface model and the compiled model (and back) is necessary in order to realize this visualization.

Fig. 7 shows the Statecharts model resulting from de- and reconstructing the CBD simulation algorithm, as described in the previous section, now with debugging features woven in. It supports the following debugging operations:

- **Continuous simulation** runs the simulation until completion, and then shows the final state of the system.
- **Realtime simulation** runs the simulation until completion, but synchronizes the simulated time with the wall-clock time. The state is shown each time step, such that the modeller can see how the system evolves over time. A scale factor can be applied to slow down or speed up simulation.

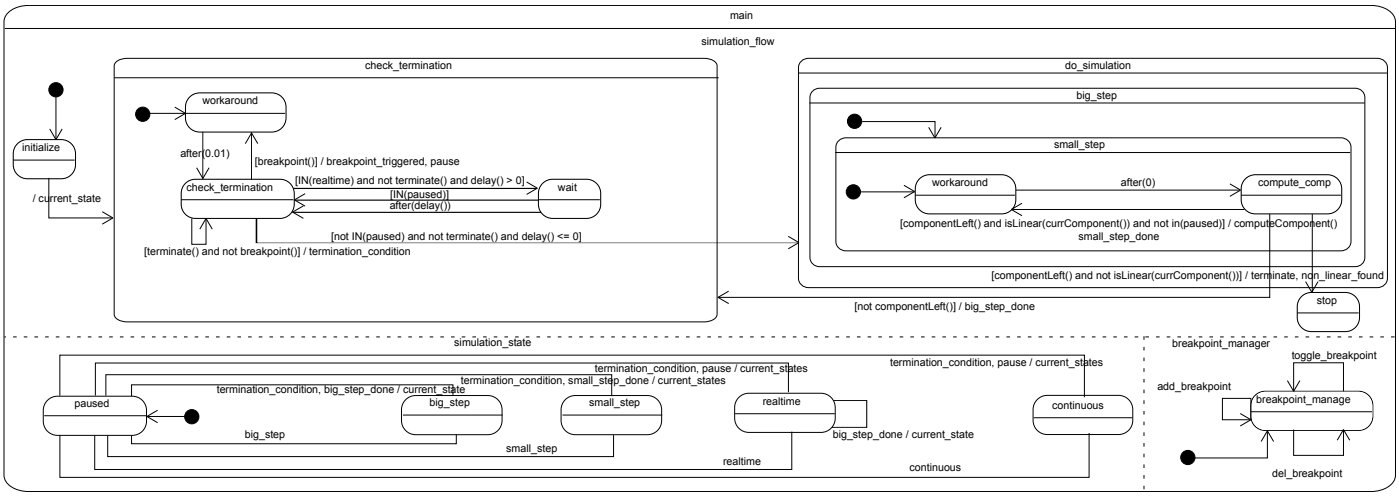


Fig. 7: The modal part of the CBD debugger.

- **Pause** interrupts a running simulation. In continuous simulation, it stops after the currently executing "big step" has finished, ensuring the system is in a stable state. In realtime simulation, however, the simulation is paused as soon as possible. This means that the simulator can be "in between" two big steps, as it might have been waiting for the appropriate wall-clock time to elapse when the pause was requested.
- **Big step** advances the simulation with one "big step" (one iteration of the outer *while* loop) and shows the system state.
- **Small step** advances the simulation with one "small step" (one iteration of the inner *while* loop) and shows the system state. This allows stepping through individual block computations.
- **Reset** allows to reset the simulation to the initial state.
- **Breakpoints** can be modelled using a breakpoint block. This pauses the simulation when its input signal becomes 1. This allows to model arbitrary conditions in the CBD model on which to pause.
- If non-linear (unsolvable) components are found, the debugger indicates which blocks belong to that component, in order for the modeller to be notified so appropriate changes can be made to the model.

The toolbar in Fig. 6 allows the modeller to send requests to the debugging-capability augmented simulator, as they generate HTTP requests that are translated to event instances and sent to the input port of the Statechart, which will react appropriately. Visualization of state is performed by translating the state returned by the Statechart to CRUD requests that are sent to the interface for changing the structure of the model. In Fig. 6, the simulation is paused and the current signal values are shown on the links between blocks. The "simulation info" block shows how many time steps have passed and what the current simulated time is. This allows the modeller to see how the model evolves over time and to control the simulation in order to isolate any observed bugs.

V. Related Work

Simulation debugging is a relatively new research area. In [10], Mannadiar and Vangheluwe address the need for debugging models in domain-specific languages and propose a mapping of code debugging concepts to model-based design.

Debuggers for simulation formalisms do exist, though they are typically hand-crafted. A debugger for Modelica for example was developed by hand, not modelled [11].

Modelled debuggers for some formalisms already exist. In [9], Mustafiz and Vangheluwe construct a debugging environment for Statecharts with a technique similar to ours. They embed the model to be debugged directly in the model of the debugger, however. This technique only works because both models are in the same formalism (Statecharts). In constrast, in our approach, we merge the Statechart models of the simulator and the debugger.

We have already used the techniques described in this paper to prototype debugging environments for other formalisms. A previous version of a CBD debugger, which had no visual user interface, and hence did not use the generic architecture described in this paper, was developed in [12]. A debugger for the Parallel DEVS formalism was developed in [13].

VI. Conclusion

This paper describes a method for constructing debugging environments for deterministic, operational formalisms. Starting from an existing simulator (implemented in code), we build a Statecharts model of its modal behaviour. This model is combined with a model of debugging operations such as step, pause, real-time simulate, etc., producing a model of the behaviour of a simulator with debugging, from wich code is synthesized. This simulator is coupled with a visual modelling interface that allows two-way communication and model manipulation. This allows modellers to simulate and debug their models in the modelling tool in which they built the model.

To demonstrate feasibility, we used our technique to construct a visual debugging environment for Causal-Block Diagrams using AToMPM [8].

ACKNOWLEDGMENT

This work was funded by a PhD fellowship from the Agency for Innovation by Science and Technology in Flanders (IWT).

REFERENCES

[1] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[2] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[3] H. Vangheluwe, "Foundations of modelling and simulation of complex systems," *ECEASST*, vol. 10, 2008.

[4] P. J. Mosterman and H. Vangheluwe, "Computer automated multi-paradigm modeling: An introduction," *Simulation*, vol. 80, no. 9, pp. 433–450, Sep. 2004. [Online]. Available: http://sim.sagepub.com/cgi/doi/10.1177/0037549704050532

[5] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(87)90035-9

[6] F. E. Cellier, *Continuous system modeling*. New York: Springer-Verlag, 1991.

[7] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, 2nd ed. Academic Press, 2000.

[8] 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.

[9] S. Mustafiz and H. Vangheluwe, "Explicit modelling of Statechart simulation environments," in *Summer Simulation Multiconference*. Society for Computer Simulation International (SCS), July 2013, pp. 445 – 452, toronto, Canada.

[10] R. Mannadiar and H. Vangheluwe, "Debugging in domain-specific modelling," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, B. Malloy, S. Staab, and M. Brand, Eds. Springer Berlin Heidelberg, 2011, vol. 6563, pp. 276–285.

[11] 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*, Nov. 2012, pp. 443–454. [Online]. Available: http://www.ep.liu.se/ecp_article/index.en.aspx?issue=76;article=46

[12] 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*, ser. TMS/DEVS '14, part of the Spring Simulation Multi-Conference. Society for Computer Simulation International, 2014, pp. 379 – 386.

[13] S. Van Mierlo, Y. Van Tendeloo, S. Mustafiz, B. Barroca, and H. Vangheluwe, "Explicit modelling of a Parallel DEVS experimentation environment," in *Proceedings of the 2015 Symposium on Theory of Modeling and Simulation - DEVS*, ser. TMS/DEVS '15, part of the Spring Simulation Multi-Conference. Society for Computer Simulation International, 2015, pp. 860 – 867.