EXPLICIT MODELLING AND SYNTHESIS OF DEBUGGERS FOR HYBRID SIMULATION LANGUAGES

Simon Van Mierlo
Cláudio Gomes
University of Antwerp
{firstname.lastname}@uantwerp.be

Hans Vangheluwe
University of Antwerp
Flanders Make
McGill University
Hans.Vangheluwe@uantwerp.be

ABSTRACT

Any sufficiently complex system is best described (or specified) with a combination of models in multiple formalisms. To support creating such "hybrid models", recent research focuses on the (syntactic and semantic) combination of formalism fragments. To implement the hybrid language's operational semantics, the simulators of each of the formalisms are combined. Inspired by this principle, we study how hybrid simulators can be instrumented with debugging capabilities. Previous work has shown that an explicit model of any simulator's behaviour can be instrumented with common code debugging operations (e.g., stepwise execution, breakpoints, pause/play) and simulation-specific operations (e.g., (scaled) real-time simulation, event injection). We extend this work by combining debugging-enhanced simulators to create a hybrid simulator, and instrument it with debugging support at the hybrid level. To demonstrate feasibility, we create a debugging-enhanced simulator of the Hybrid Automata formalism, by embedding a Causal Block Diagrams simulator in a Timed Finite State Automata simulator.

Keywords: Debugging, Hybrid Simulation, Modelling.

1 INTRODUCTION

To deal with the increasing complexity of today's (engineered) systems, a system is engineered/studied by: (1) decomposing it into coupled components, and/or (2) identifying different aspects and treating them separately (*e.g.*, the control algorithm of an embedded system versus its power consumption) (Vangheluwe 2008). Ideally, each of these components is modelled at the most appropriate level of abstraction, using the most appropriate formalism(s) (Vangheluwe et al. 2002). For example, an automatic power window system (Denil 2013) is comprised of at least two sub-systems: the software controller and the window dynamics. A model of the control sub-system represents decisions regarding (safe) interactions with the power window (*e.g.*, what happens when the user presses a button, or when an obstacle is detected in the window), while an electro-mechanical model determines the movements of the window.

In order to observe the global behaviour of the system, the behavioural models of each sub-system are coupled. Since they conform to different formalisms, however, their interaction needs to be determined. It can consist of input/output variables that can be accessed by both models during their coupled simulation—as in co-simulation (Gomes 2016), or they can be more complex, realized through *boundary concepts* that do not belong to any of the coupled formalisms—as in co-modelling (Boulanger et al. 2011, Jantsch and Sander 2005). We focus on the latter approach.

SpringSim-TMS/DEVS 2017, April 23-26, Virginia Beach, VA, USA
©2017 Society for Modeling & Simulation International (SCS)

A hybrid formalism needs sound operational semantics. In order to keep mistakes to a minimum and reduce development efforts, existing simulators can be reused. Previous work concluded that a hybrid formalism simulator should act as a coordinator between the simulators of the composed formalisms. In this way, development effort is focused on the definition of the boundary concepts and interaction between the simulators, both non-trivial but essential tasks (Mustafiz et al. 2016). In addition, if the existing simulators provide debugging support, the hybrid formalism simulator should also allow for debugging. Debugging is a well known, well practised, and important activity in the development of behavioural models (*e.g.*, code (Zeller 2005), Statecharts (Mustafiz and Vangheluwe 2013), and DEVS (Van Mierlo et al. 2016)).

In this paper, we propose a novel way to construct hybrid simulators by modelling existing simulators explicitly in a generic form that can be used for modular language composition and can be easily enhanced with debugging support. This form, which we denote as the (hierarchical) canonical form, is based on the commonalities between behavioural formalism simulators. Simulators in the canonical form are easy to understand and can be (1) quickly enhanced with debugging capabilities (Contribution #1), and (2) systematically combined in a way that describes their non-trivial interactions and thus form a hybrid simulator with debugging capabilities (Contribution #2). To demonstrate feasibility, we explore the composition of Timed Finite State Automata (T-FSA) and Causal Block Diagrams (CBD) (Cellier 1991).

Structure Section 2 introduces the T-FSA and CBD formalisms and their simulators, as well as the SCCD formalism, which is used to model the simulators' behaviour. Section 3 explains the canonical form of simulators, as well as how to enhance it with debugging. Section 4 describes the composition of the T-FSA and CBD languages to form the (deterministic) hybrid automata formalism, as well as its debugging-enhanced simulator. Section 5 discusses the validity of our approach. Section 6 presents related work, and Section 7 concludes the paper.

2 BACKGROUND

This section describes the background for the remainder of the paper. We start by explaining SCCD, a hybrid formalism that combines Statecharts and Class Diagrams, with which we model the (debugging-enhanced) simulators' behaviour throughout the paper. Then, we introduce two formalisms: Timed Finite-State Automata and Causal-Block Diagrams. A debugging-enhanced simulator for their combination (a hybrid formalism) is constructed in Section 4.

2.1 SCCD

Statecharts (SC) is a well-known formalism for describing timed, reactive, autonomous system behaviour (Harel 1987). It consists of states and transitions between those states that are triggered by an event (local to the SC model or coming from the environment) or a timeout and optionally have a guard. States can be composed hierarchically in composite states (which have exactly one active child state), as well as orthogonally in parallel regions (where each region has an active state).

SCCD (Van Mierlo et al. 2016) is a hybrid formalism that combines SC and Class Diagrams (CD). An SCCD model is a class diagram where each class has attributes, methods, and an SC model, which describes its behaviour. Its transitions can execute methods and change the attributes of the class instances. Relationships between classes form communication channels (*i.e.*, ports) over which each SC model can communicate with the SC model of other instances by sending events.

Algorithm 1 T-FSA Operational Semantics.

```
 1: function SIMTFSA(M, s_0 evs, Δt)
 2:     clock ← 0
 3:     state ← s_0
 4:     ε ← 0
 5:     while state ∉ FINALSTATES(M) do
 6:         continue ← true
 7:         while continue do
 8:             (evs, e_i) ← POPEV(evs, clock)
 9:             if e_i = ∅ then
10:                 tr ← TRELAP(M, state, ε)
11:             else
12:                 tr ← TREV(M, state, e_i)
13:             end if
14:             if tr ≠ ∅ then
15:                 ε ← 0
16:                 state ← TARGET(M, tr)
17:             else
18:                 continue ← false
19:             end if
20:         end while
21:         clock ← clock + Δt
22:         ε ← ε + Δt
23:     end while
24:     return clock, state
25: end function
```

Algorithm 2 CBD Operational Semantics.

```
 1: function SIMULATECBD(M, maxIters, Δt)
 2:     clock ← 0
 3:     state ← INITSIGNALS(M)
 4:     numIters ← 0
 5:     while numIters < maxIters do
 6:         g ← DEPGRAPH(M, numIters)
 7:         s ← LOOPDETECT(g)
 8:         for c in s do
 9:             if c = {gblock} then
10:                 state ← COMPB(c, state)
11:             else
12:                 state ← COMPL(c, state)
13:             end if
14:         end for
15:         clock ← clock + Δt
16:         numIters ← numIters + 1
17:     end while
18:     return clock, state
19: end function
```

2.2 Timed Finite State Automata

Timed Finite State Automata (T-FSA) is a timed variant of Finite State Automata (Hopcroft et al. 2006), with a single clock (Alur and Dill 1994), and is a simplified version of SC. A T-FSA model consists of a set of states and transitions having optional triggers (an event from the environment or a condition on the amount of time passed in the source state). There is one initial state and a subset of the states are final states.

Algorithm 1 condenses the main tasks of a T-FSA simulator. The main simulation loop processes zero or more triggers, advances time, and keeps track of the current state and elapsed time.

The parameter M is the T-FSA model, s_0 is the initial state of M, $evs = \{(t_0, e_0), (t_1, e_1), \ldots\}$ is a sorted sequence of time-stamped events from the environment, and $\Delta t > 0$ denotes how much to advance the simulated time at the end of each simulation step.

The $state$ variable stores the current state of the T-FSA.

POPEV(evs, t) returns an event whose timestamp matches the parameter t, and the new sequence of events minus the event returned. The null event $\emptyset$ is returned if there is no event at time t.

TRELAP$(M, state, \varepsilon)$ returns a transition that is enabled by the elapsed time.

TREV$(M, state, e_i)$ returns a transition that is enabled by event e_i.

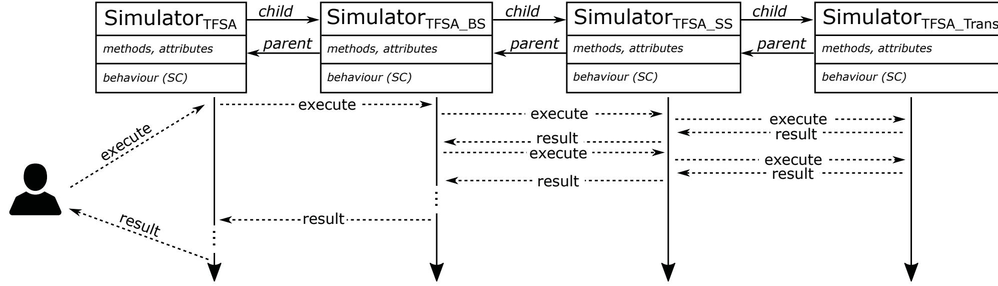


Figure 1: The hierarchical structure of the T-FSA simulator.

2.3 Causal Block Diagrams

Causal Block Diagrams (CBD) provide a way to represent a set of differential equations by means of blocks and connections. Each block has (optional) inputs and one output, and it can either represent an algebraic mathematical operation (*e.g.*, summation or multiplication) or a time-sensitive operation (integral or derivative). A CBD model can additionally have input/output ports, representing signals (functions over time) coming from/going to the environment.

The main simulation loop steps through simulated time and, at each time point, computes the output of every block, based on its inputs. This is captured in Algorithm 2, adapted from (Vangheluwe et al. 2014).

The parameter M is the CBD model, $maxIters$ controls the number of simulation steps to be performed, and $\Delta t > 0$ denotes how much to advance the simulated time at the end of each simulation step.
The $state$ variable is a vector with an entry per signal.
$\textsc{DepGraph}(M, numIters)$ returns the dependencies between the blocks.
$\textsc{LoopDetect}(depGraph)$ returns the schedule of blocks and strongly connected components that represent an algebraic loop.
$\textsc{CompB}(block)\ and\ \textsc{CompL}(loop)$ compute the new value of the output signals of a block and an algebraic loop, respectively.

More details about CBD simulation can be found in (Gomes et al. 2016).

3 MODELLING SIMULATION ALGORITHMS

Most often, simulators are implemented in program code. In previous work, we argue that program code is not the most appropriate formalism to model the simulator's behaviour, in particular when the simulator needs to be augmented with debugging operations (Van Mierlo 2014) or combined with other simulators to form a hybrid simulator (Mustafiz et al. 2016). Instead, a formalism capable of modelling the simulator's timed, reactive, autonomous behaviour natively is more appropriate. This section reviews the canonical form of simulation algorithms and proposes an improved version based on hierarchical nature of many algorithms. To properly model that hierarchy, we argue SCCD (Van Mierlo et al. 2016) is an appropriate formalism.

3.1 A Generic Simulator Template

We introduced the simulation algorithms for the T-FSA and CBD formalisms in Section 2 (Algorithm 1 and Algorithm 2). At a high level of abstraction, these algorithms are very similar, as they go through a set of phases: (i) *Initialization* of simulation time and the simulation state; (ii) *Execution* of simulation 'steps'

until an end condition is satisfied (the core of the algorithm, where a new state is computed based on the previous one, and the simulation time is advanced); (iii) *Finalization* where, for example, the final state of the simulation and the time at which it ended is communicated to the user.

Algorithm 3 Generic simulation algorithm.

```
1: function SIMULATE(M, params)
2:     initialize(params)
3:     while not endCondition() do
4:         executeStep()
5:     end while
6:     finalize()
7:     return getState(), getTime()
8: end function
```

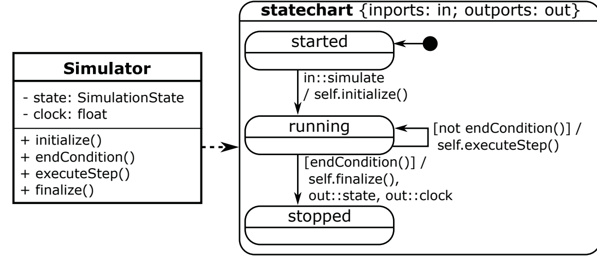


Figure 2: The canonical form of the generic simulation algorithm.

These phases yield a generic template, shown in Algorithm 3. Converting this high-level control flow to a SCCD model is possible using the 'de- and reconstruction' technique presented in previous work (Van Mierlo 2014). The result is shown in Figure 2: we model a class *Simulator* which has the necessary attributes and methods (whose implementation depends on the formalism being simulated). The behaviour of this class is modelled in a SC model which executes the phases of simulation outlined above.

3.2 Hierarchical Canonical Representation

For the purposes of debugging and simulator composition, the instruction $executeStep()$ in Algorithm 3 needs to be further refined. There is a single notion of 'step': it is a computation which brings the simulation from a state to the next and increases the simulated time. Upon closer inspection of Algorithm 1 and Algorithm 2, an additional level of 'steps' can be discerned. Each 'step' (from now on: 'big step') consists of a series of 'small steps'. In the case of T-FSA, a 'big step' executes as many transitions as possible, while a 'small step' executes one such transition. In the case of CBD, a 'big step' computes the new value of all signals in the model, while a 'small step' computes the value of a single signal. This means the executeStep() function can be expanded as a *while*-loop, which is preceded by an initialization phase and succeeded by a finalization phase. In (Mustafiz et al. 2016), the authors differentiate analogously between a 'macro' and a 'micro' step, and propose a flattened canonical form of the simulator. Their model does not mimic the hierarchical nature of the simulation algorithm, however, neither does it exploit the similarity between the control flow of the two levels. This makes it challenging to add debugging support in a modular, reusable way. We propose an improved canonical form, where each level is modelled in a separate SCCD class, following the template of Figure 2.

As an example, Figure 1 demonstrates the hierarchical canonical form for the T-FSA simulator: instead of having one *Simulator* class, we have four: one for each level. The top-level simulator creates a new big step simulator and requests it to compute the next iteration of the simulation until the simulation end condition is satisfied. The big step simulator, in its turn, creates a new small step simulator and requests it to compute the next state until the big step end condition is satisfied (*i.e.*, no more transitions are enabled). The small step simulator creates the lowest level simulator, which will simply be a function call that executes a transition. Each level finalizes and communicates its results to the level above when its end condition is satisfied. The

user can start the simulation (by instantiating the top-level simulator) and wait until it finishes execution, then inspect its results.

Analogously, a hierarchical canonical version for the CBD simulator can be constructed. In the next subsection, we will explain how debugging operations can be added to the simulator's behaviour.

3.3 Debugging

In general, there are three categories of debugging operations, which are either inspired by program debugging operations or are simulation-specific:

[Time] These operations manipulate simulated time, a central concept to most simulations. Simulated time can have different *relations* to the wall-clock time. Simulation can be run *as-fast-as-possible* (*i.e.*, as fast as the hardware of the executing platform and operating system allow), which means that the simulation clock is simply a variable of the simulation. Models of timed systems can also be run in *(scaled) real-time* mode. In that case, the simulated time is synchronized with the wall-clock time. The simulation then proceeds at the same pace as the system it models. An optional scale factor can speed up or slow down simulation, while retaining the linear relation between simulated time and wall-clock time. Simulations can also be *paused*. An important consideration is the level of the simulation at which a pause occurs. We have opted to only pause at the simulation level, immediately after a big step. This avoids ending up in an inconsistent simulation state where some, but not all, small steps have been computed.

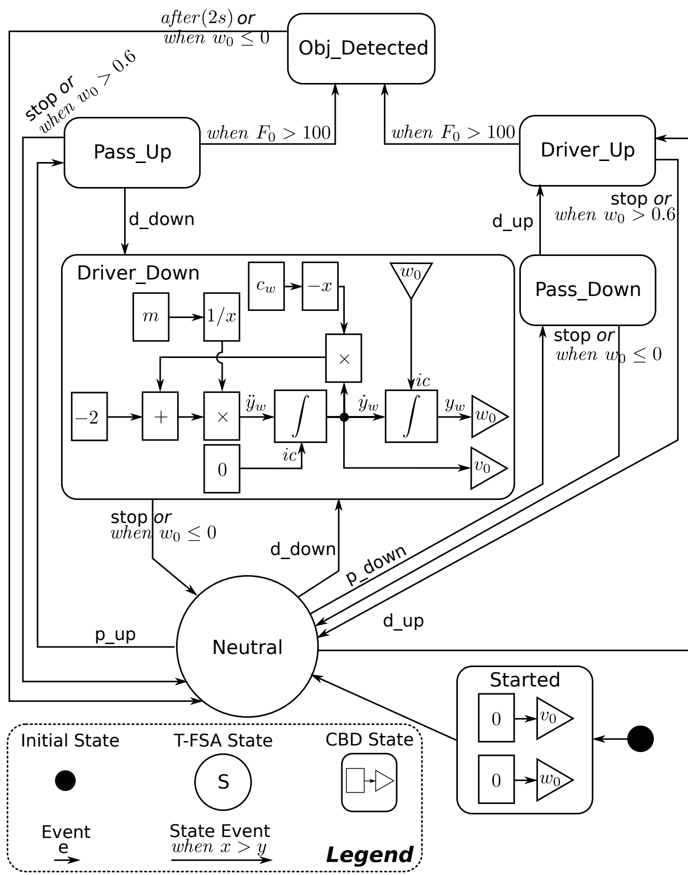
[Control] These operations manipulate the control flow of the simulation. The user can *step through* the simulation. The user can be given control at any of the levels in the canonical form. For example, in a two-level simulation algorithm, the user can be given a *big step* and *small step* operation. A *breakpoint* is an automatic pause, depending on a constraint on the simulation state. This pause, similar to a manual pause, occurs at the simulation level.

[State] These operations manipulate the simulation state. A *god event* allows the user to manually change the simulation state. Its definition depends on the formalism. The user might be allowed to change the state directly (in T-FSA, for example, changing the current state manually to a different state, or triggering a particular transition that changes the current state) or indirectly (in T-FSA, for example, by injecting an event at the current simulation time, which then triggers a particular transition).

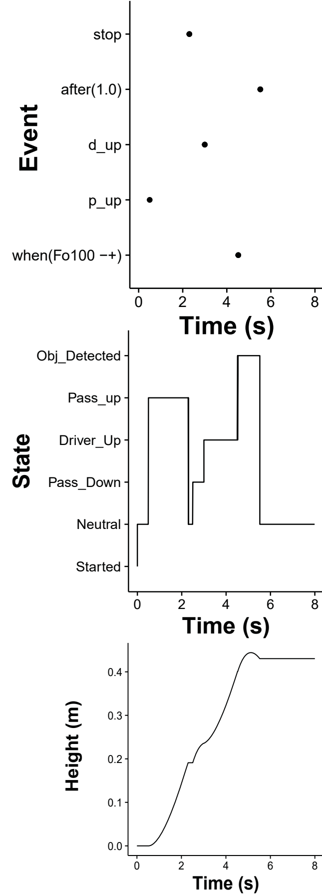
In previous work (Van Mierlo 2014) we explain how the Statechart's representation of a simulation algorithm can be instrumented to enhance it with debugging operations. We apply the same procedure in this paper and highlight the differences with previous work. Each level has to be instrumented, but only with relevant operations. **[Time]** operations and breakpoints are only relevant at the simulation level (*i.e.*, the outer while-loop of Algorithm 1 and Algorithm 2), since that is the only level which has a notion of a clock progressing and can be paused without leaving the simulator in an inconsistent state. The same is true for **[State]** operations, which we assume can only be executed when the simulation is paused and bring the simulation to a new consistent state. **[Control]** operations are relevant for all levels which execute a loop. Indeed, those are the points at which we want to give control to the user to iterate manually through the loop instead of automatically.

By enhancing the simulation algorithm(s) with debugging support, their interfaces change, giving more control to the user to interrupt and control the simulation algorithm using events:

- **continuous** runs the simulation as-fast-as-possible.



(a) Example hybrid model of the power window system.



(b) Example trace.

Figure 3: The example model and its simulation trace.

- **realtime** runs the simulation in real-time. It accepts an optional parameter: the *realtime scale*.
- **pause** pauses a running simulation after the current big step has finished (or, if the simulation is in real-time mode and in a waiting period, it returns immediately).
- **add_breakpoint**, **del_breakpoint**, **toggle_breakpoint** are used for breakpoint management.
- **god_event** changes the current state. In T-FSA, the user can manually enable a transition, which is executed in the next small step. In CBD, the user can manually change the value of a signal.

This debugging interface is used in the next section to create a debugging-enhanced hybrid simulator.

4 HYBRID AUTOMATA

The Hybrid Automata formalism inherits its syntax from both T-FSA and CBD. It has states and transitions, which are triggered by events and optionally have a guard. States, however, can now contain a CBD model, which is simulated when that state is entered. A transition whose source state contains a CBD model can be triggered by a *boundary crossing condition*, which depends on an output value of the CBD model.

Figure 3a shows the (partial) hybrid model for the power window system. The overall modes of the power window are depicted as a T-FSA. The *Driver_Down* state's CBD model is expanded, representing the

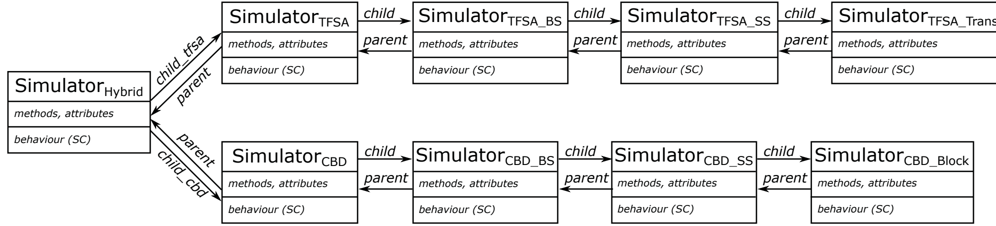


Figure 4: The hierarchical structure of the T-FSA-CBD simulator.

dynamics of the window when going down. Multiple transitions are triggered by a boundary crossing condition: when an object is detected, or when the window reaches the bottom and top of its frame. Figure 3b shows an example simulation where the passenger raises the window until an object is detected (at time 4). The model moves to state *Obj_Detected* where the window is being lowered for 1 second.

Due to space constraints, the simulation algorithm for the hybrid T-FSA-CBD simulator cannot be shown. It is, however, a merge of the algorithm for T-FSA presented in Algorithm 1 and the CBD simulator in Algorithm 2. Intuitively, the simulation algorithm can be broken down as follows:

1. The simulation is initialized as in the T-FSA algorithm, with one difference: the Δt parameter is set to the minimum of the Δt parameters for the two simulators, guaranteeing the same level of accuracy as in the individual simulators.
2. An outer-while loop executes the model until an end state has been reached, or if at any point, the currently executing CBD model has reached its maximum number of iterations.
3. At the start of a big step, the algorithm checks whether the current state contains a CBD model. If this is the first time the state was entered, the model is initialized. Then, its next iteration is computed, by invoking the child CBD model's simulator.
4. After the iteration is computed, the algorithm checks whether any state events occur: these are boundaries that are crossed by continuous variables in the CBD simulation. These boundary crossings are translated to T-FSA events and can trigger a transition.
5. A T-FSA small step is executed as usual: the next event is read from the environment and a transition is executed if any is enabled. Transitions can now also be triggered by state events, but events from the environment get priority.

This can be seen as a *protocol* which meaningfully combines the semantics of both simulators and is implemented by the hybrid simulator, which appropriately calls its child simulators. The implementation of this protocol is made possible only when the child simulators' interface provides adequate control. We observe that in our explanation of the algorithm above we refer to three debugging operations:

- In Item 3, one *big step* of the CBD simulation algorithm is executed.
- In Item 4, a transition is triggered by a state event. It is, for the child T-FSA simulator, an outside force which enables it, corresponding to a *god event*.
- In Item 5, one *small step* of the T-FSA simulation algorithm is executed.

In Figure 4, the hierarchical composition of the hybrid simulator is shown. The protocol of the simulation algorithm, as well as boundary concepts (such as state event detection) is implemented by the hybrid simulator. It has two child simulators, whose behaviour it controls through their exposed debugging interfaces.

The hybrid simulation algorithm can be fit into the generic template of Figure 2. The concept of 'big step', 'small step', state, and time operations remain unchanged from the 'master' T-FSA algorithm. The T-FSA

simulator processes states containing a CBD model as normal T-FSA states, and it controls how simulated time advances. The hybrid simulator is responsible for invoking the CBD simulators for states containing a CBD state. God events can enable a transition at the T-FSA level.

These debugging operations are also valid at the CBD level, however. A user debugging a hybrid simulation might want to step through CBD block computations or change a signal value through a (CBD) god event. Because of the hierarchical nature of the formalism (CBD models are contained in T-FSA states), we define a *step into* debugging operation. This operation is valid when the simulation is paused, and the current T-FSA state contains a CBD model. It switches the execution context to the CBD simulator and allows regular debugging interaction at that level. The user can then execute a big step, a small step, or a god event. When a big step finishes, control is returned to the hybrid simulator. Our implementation can be found online: https://msdl.uantwerpen.be/git/claudio/MLE/src/master/debugging_fsa_cbd_composition.

5 DISCUSSION

We propose a modular, generic, and repeatable method for constructing a debugging-enhanced hybrid simulator which composes two (or more) existing simulators. It is modular, since the child simulators of the hybrid simulator do not have to be modified and can be unaware of their role in the hybrid simulation. It is generic, as we rely on a hierarchical canonical form modelled in SCCD. It is repeatable, since the de- and reconstruction technique has been established before and consists of a fixed number of steps that can be applied to any formalism. We put debugging operations forward as the main enablers of meaningful semantic composition of formalisms.

The resulting hybrid simulator satisfies the properties outlined in (Mustafiz et al. 2016):

- **Language Continuity**: the (hybrid) simulator's behaviour needs to behave as the original simulator if it simulates a plain T-FSA model. If no states contain a CBD model, then the hybrid simulator will never invoke the CBD simulator, and no state events occur. Consequently, only the T-FSA simulator will be invoked throughout the simulation.
- **Step Progression**: for valid models, the simulation always advances and will eventually meet the end condition. A big step of the hybrid simulator consists of executing a big step of the T-FSA simulator, which, for valid models, always terminates and advances time.
- **Step Synchronization**: there is an algebraic relation between the simulated time of the different simulators, or between the rates at witch they progress. In the hybrid simulator, the CBD and T-FSA simulated times are not the same but they advance at the same rate.

Additionally, the hybrid simulator offers debugging operations and satisfies the following properties:

- **Continuity**: if the user does not use any of the debugging features, the simulator's behaviour does not differ from its non-instrumented version.
- **Soundness**: a pause request by the user will pause the simulation in a *consistent* state.
- **Big Step-Small Step Correspondence** - A big step and a small step in the debugger of the T-FSA formalism has the same behaviour as a big step and small step in the hybrid formalism. The same applies to the CBD big step and small step if the user *steps into* a state containing a CBD model.

Enhancing a simulator with debugging operations is well-suited to (partly) automate once the simulators are in the hierarchical canonical from. We leave this as future work, as well as the automatic generation of the hybrid simulator, which was touched upon in (Mustafiz et al. 2016).

The hybrid simulation algorithm relies on the debugging operations implemented by the individual simulators. Our approach is thus only valid on debugging-enhanced simulators. Currently, our approach is white-box, since we assume the source code is available to de- and reconstruct the simulator. We also assume a set of specific debugging operations are implemented, which is not necessarily the case for arbitrary debugging-enhanced simulators. For example, we need a 'big step' and 'small step' operation, as well as a particular 'god event' operation. If there is a mismatch between useful debugging operations and useful operations for language composition, our approach might not be valid for all simulators, especially those that have a limited interface (or none).

One interesting related area is co-simulation, where different simulators are linked together by an orchestration algorithm. In the FMI standard (Blockwitz et al. 2012), each simulator only exposes a limited interface allowing the orchestrator to advance the algorithm one 'step'. As we have shown, if more flexible language composition is required, each simulator has to expose a finer grained interface (*e.g.*, distinguishing between big and small steps).

6 RELATED WORK

While program debugging is an active research area, it is not the focus of this paper. It does, however, provide a foundation. (Zeller 2005) presents an overview of the state of the art in program debugging. We are inspired by (Mannadiar and Vangheluwe 2011), where the authors describe how program debugging operations can be transposed onto the modelling domain. Many of our operations are transpositions of existing debugging operations, although we add a number that are simulation-specific.

Both (Buchanan and Keefe 2014) and (Pop et al. 2014) describe how a simulator can be fit to provide debugging support. In (Pop et al. 2014), the focus is on the Modelica language, whereas in (Buchanan and Keefe 2014), the authors target a language-independent discrete-event simulator. In contrast, our approach can potentially be used to provide debugging support to any hybrid simulator, resulting from the composition of two simulators. We apply the 'de- and reconstruction' technique, already applied to Parallel DEVS (Van Mierlo et al. 2016) and CBDs (Vangheluwe et al. 2014), to obtain a generic, hierarchical, canonical version of each simulator, needed to create the debugging-enhanced hybrid simulator.

In (Boulanger and Hardebolle 2008), the ModHel'X allows a user to code a semantic adaptation between a main model (*e.g.*, the T-FSA controller) and an embedded one (*e.g.*, the CBD dynamics). This adaptation is done at the model level, and can thus be specialized for different models. In our case, the adaptation is done at the simulator level, independent of the concrete models being simulated.

Co-simulation can be seen as a more restricted approach towards semantic adaptation, where simulators are allowed only to communicate at the end of macro-steps. In (Denil et al. 2015), the authors propose a Domain Specific Language (DSL) for the specification of common semantic adaptations. The work builds on (Boulanger and Hardebolle 2008), so the adaptations are specific to the models being coupled, as the authors point out. Ptolemy (Eker et al. 2003) offers debugging capabilities and allows the modeller to compose models from different formalisms. While a very flexible approach, the adaptations between the formalisms are fixed, in contrast to our work.

7 CONCLUSION

We present a novel method for constructing a debugging-enhanced simulator for hybrid formalisms composed of two or more formalisms. To add meaningful debugging to each simulator, we introduce a new, hierarchical

canonical form that reflects the hierarchical nature of many simulation algorithms. We assume each formalism's simulation algorithm is modelled in its canonical form as a SCCD model. The simulation algorithm (*i.e.*, the protocol between the existing simulators of the child formalisms) is implemented by modelling it explicitly using SCCD as well. It makes use of the debugging interface provided by its child simulators for operations such as 'big step', 'small step', and 'god event'. We show that these operations are necessary to build a meaningful hybrid simulator. Debugging support is added at the hybrid level, and a new 'step into' debugging operation is implemented to allow switching to the simulator of the lower-level formalism. In the future, we envision (partly) automating our approach. We also plan to prove correctness properties on the constructed hybrid simulator. Last, we will apply our techniques to commercial simulators to demonstrate their generality.

ACKNOWLEDGEMENTS

This work was funded by Flanders Make vzw, the strategic research centre for the manufacturing industry, and with PhD fellowships from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

Alur, R., and D. L. Dill. 1994. "A theory of timed automata". *Theoretical Computer Science* vol. 126 (2), pp. 183–235.

Blockwitz, T., M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. 2012, nov. "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models". In *9th International MODELICA Conference*, pp. 173–184. Munich, Germany, Linköping University Electronic Press; Linköpings universitet.

Boulanger, F., and C. Hardebolle. 2008. "Simulation of Multi-Formalism Models with ModHel'X". In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pp. 318–327.

Boulanger, F., C. Hardebolle, C. Jacquet, and D. Marcadet. 2011. "Semantic Adaptation for Models of Computation". In *11th International Conference on Application of Concurrency to System Design (ACSD)*, pp. 153–162.

Buchanan, C., and K. Keefe. 2014, September. "Simulation Debugging and Visualization in the Möbius Modeling Framework". In *Proceedings of the 11th International Conference on Quantitative Evaluation of Systems (QEST)*, pp. 226–240.

Cellier, F. E. 1991. *Continuous system modeling*. New York, Springer-Verlag.

Denil, J. 2013. *Design, Verification and Deployment of Software Intensive Systems - A multiparadigm approach*. Ph. D. thesis, University of Antwerp.

Denil, J., B. Meyers, P. D. Meulenaere, and H. Vangheluwe. 2015. "Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-Simulation". In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, edited by SCS, pp. 99–106. Alexandria, Virginia.

Eker, J., J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. 2003, jan. "Taming heterogeneity - the Ptolemy approach". *Proceedings of the IEEE* vol. 91 (1), pp. 127–144.

Gomes, C. 2016. "Foundations for Continuous Time Hierarchical Co-simulation". In *ACM Student Research Competition (ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems)*, pp. to appear. Saint Malo, Brittany, France.

Gomes, C., J. Denil, and H. Vangheluwe. 2016. "Causal-Block Diagrams". Technical report.

Harel, D. 1987, June. "Statecharts: a Visual Formalism for Complex Systems". *Science of Computer Programming* vol. 8 (3), pp. 231–274.

Hopcroft, J. E., R. Motwani, and J. D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Jantsch, A., and I. Sander. 2005. "Models of computation and languages for embedded system design". *IEE Proceedings - Computers and Digital Techniques* vol. 152 (2), pp. 114–129(15).

Mannadiar, R., and H. Vangheluwe. 2011. "Debugging in Domain-Specific Modelling". In *Software Language Engineering*, edited by B. Malloy, S. Staab, and M. Brand, Volume 6563 of *Lecture Notes in Computer Science*, pp. 276–285. Springer Berlin Heidelberg.

Mustafiz, S., C. Gomes, B. Barroca, and H. Vangheluwe. 2016. "Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation". In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS '16, pp. 29:1—-29:8. San Diego, CA, USA.

Mustafiz, S., and H. Vangheluwe. 2013. "Explicit Modelling of Statechart Simulation Environments". In *Proceedings of the 2013 Summer Computer Simulation Conference*, pp. 21:1—-21:8.

Pop, A., M. Sjölund, A. Ashgar, P. Fritzson, and F. Casella. 2014. "Integrated Debugging of Modelica Models". *Modeling, Identification and Control* vol. 35 (2), pp. 93–107.

Van Mierlo, S. 2014. "Explicit Modelling of Model Debugging and Experimentation". In *Proceedings of the Doctoral Symposium at MODELS'14*.

Van Mierlo, S., Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe. 2016. "SCCD: SCXML Extended with Class Diagrams". In *3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*.

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2016. "Debugging Parallel DEVS". *SIMULATION*.

Vangheluwe, H. 2008. "Foundations of Modelling and Simulation of Complex Systems". *EASST* vol. 10.

Vangheluwe, H., J. De Lara, and P. J. Mosterman. 2002. "An introduction to multi-paradigm modelling and simulation". In *Proceedings of AIS2002 (AI, Simulation & Planning)*, pp. 9–20, SCS.

Vangheluwe, H., D. Riegelhaupt, S. Mustafiz, J. Denil, and S. Van Mierlo. 2014. "Explicit Modelling of a CBD Experimentation Environment". In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, TMS/DEVS '14, pp. 379–386, SCS.

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

AUTHOR BIOGRAPHIES

SIMON VAN MIERLO is a PhD student at the University of Antwerp (Belgium). For his PhD, he is studying how modelling formalisms, environments, and simulators can be enhanced with debugging support.

CLÁUDIO GOMES is a PhD student at the University of Antwerp (Belgium). The topic of his PhD is the foundations of co-simulation.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium), an Adjunct Professor at McGill University (Canada) and an Adjunct Professor at the National University of Defense Technology (NUDT) in Changsha, China. He heads the Modelling, Simulation and Design (MSDL) research lab. His research interest is the multi-paradigm modelling of complex, software-intensive, cyber-physical systems.