Extending Explicitly Modelled Simulation Debugging Environments with Dynamic Structure

SIMON VAN MIERLO* and HANS VANGHELUWE, University of Antwerp - Flanders Make, Belgium
SIMON BRESLAV, RHYS GOLDSTEIN, and AZAM KHAN, Autodesk Research, Canada

The widespread adoption of Modelling and Simulation (M&S) techniques hinges on the availability of tools supporting each phase in the M&S-based workflow. This includes tasks such as specifying, implementing, experimenting with, as well as debugging simulation models. We have previously developed a technique where advanced debugging environments are generated from an explicit behavioural model of the user interface and the simulator. These models are extracted from the code of existing modelling environments and simulators, and instrumented with debugging operations. This technique can be reused for a large family of modelling formalisms, but was not yet considered for dynamic-structure formalisms; debugging models in these formalisms is challenging, as entities can appear and disappear during simulation. In this paper, we adapt and apply our approach to accommodate dynamic-structure formalisms. To this end, we present a modular, reusable approach, which includes an architecture and a workflow. We observe that to effectively debug dynamic-structure models, domain-specific visualizations developed by the modeller should be (re)used for debugging tasks. To demonstrate our technique, we use Dynamic-Structure DEVS (DSDEVS) (a formalism that includes the characteristics of discrete-event and agent-based modelling paradigms) and an implementation of its simulation semantics in the PythonPDEVS tool as a running example. We apply our technique on NetLogo, a popular multi-agent simulation tool, to demonstrate the generality of our approach.

Additional Key Words and Phrases: Debugging, Dynamic-Structure Formalisms, Visual Modelling and Experimentation Interfaces

ACM Reference Format:
Simon Van Mierlo, Hans Vangheluwe, Simon Breslav, Rhys Goldstein, and Azam Khan. 2019. Extending Explicitly Modelled Simulation Debugging Environments with Dynamic Structure. *ACM Trans. Model. Comput. Simul.* 1, 1 (May 2019), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A plethora of Modelling and Simulation (M&S) tools are in use today. Almost all, however, fail to provide debugging support using the same abstractions as those provided by the modelling formalism(s) used. Instead, they often rely on traditional code debugging techniques with which domain experts may not be familiar, hindering the widespread adoption of M&S. In recent years, research efforts have been directed towards implementing formalism-specific debuggers. Implementing

*While at Autodesk Research.

Authors' addresses: Simon Van Mierlo, simon.vanmierlo@uantwerpen.be; Hans Vangheluwe, hans.vangheluwe@uantwerpen.be, University of Antwerp - Flanders Make, Middelheimlaan 1, Antwerp, 2020, Belgium; Simon Breslav, simon.breslav@autodesk.com; Rhys Goldstein, rhys.goldstein@autodesk.com; Azam Khan, azam.khan@autodesk.com, Autodesk Research, 661 University Ave, Ontario, Toronto, ON M5G 1M1, Canada.

these debuggers is challenging, however, because of the interplay between (1) the semantics of the modelling formalism, as implemented by a simulator or executor; (2) the behaviour of debugging operations (specific to the formalism); and (3) the interaction between the user and the debugging system, potentially through a (or multiple) visual user interface(s). Moreover, the semantics of the modelling formalisms are, in most cases, more complex than the sequential execution of statements in program code. Non-programming notions of time, non-determinism, concurrency, events, and continuous behaviour may be at the heart of the formalism's semantics. To develop debugging environments for these formalisms, traditional software development techniques are not well-equipped, since they cannot express such inherently complex behaviour natively. In earlier work, we have developed a technique that manages this complexity by representing the control flow of simulation algorithms explicitly using Statecharts [15], whose higher level of abstraction allows debugging support to be added by instrumentation [33]. This technique proved to be applicable to a number of formalisms [35, 37].

In this paper, we study the debugging of dynamic-structure models, expressed using dynamic-structure formalisms [3], which allow one to natively describe structure-changing behaviour: during simulation, entities can appear and disappear and their interconnections might change. This makes the construction of a debugging environment particularly challenging: the (visual) modelling environment cannot be repurposed as easily as in our previous contributions, since not all model instances are individually represented in the model diagram. Dynamic-structure simulations are, however, often accompanied by a model-specific visualisation environment which displays the simulation entities during their lifetime. We present a reusable and general workflow and software architecture for constructing debugging environments for dynamic-structure formalisms and their simulation tools, which also instruments these domain-specific visualisations with debugging support. Specifically, the contribution of this paper is fourfold:

(1) A reusable architecture and an explicitly modelled workflow for constructing debuggers (including a visual interface) and instrumenting domain-specific visualizations.
(2) A specification of a debugger for discrete-event, real-time, dynamic-structure, visual, interactive simulation systems. This includes a set of useful debugging operations, as well as an interface for interacting with the debugger.
(3) A prototype that implements the specification and architecture for the DSDEVS [2] formalism as implemented by PythonPDEVS [36], to demonstrate feasibility.
(4) A debugger prototype for NetLogo, a popular multi-agent simulation tool, to evaluate our approach by demonstrating that it can be used to develop debuggers for any dynamic-structure formalism.

Structure. Section 2 provides background for the paper. Section 3 introduces the DSDEVS formalism (and PythonPDEVS) as a running example, which we use to demonstrate our techniques. Section 4 describes our approach to developing (visual) model debugging interfaces for dynamic-structure formalisms. Section 5 discusses two prototypes we built using our technique: the PythonPDEVS debugger, as well as a debugger for NetLogo. Section 6 discusses our contribution, and Section 7 concludes the paper.

2 BACKGROUND

This section provides background information for the remainder of the paper. We start by explaining the Statecharts language, which is used in our approach to describe the behaviour of the debugger and other artefacts. The method for turning modelling environments into debugging environments by explicitly modelling their behaviour is explained next. Last, we present a general modelling and simulation workflow, where particular attention is given to the artefacts created during this process. These artefacts will be reused in the rest of the paper to create debugging environments for dynamic-structure formalisms.

2.1 Statecharts

Statecharts (SC) is a formalism used to model timed, reactive, autonomous systems and was introduced by [15]. A SC model 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).

Events raised in one orthogonal component can be "sensed" by other orthogonal components (broadcasting). SC is a deterministic formalism, and different language variants exist, corresponding to different deterministic orderings and interleavings of events and transitions. We assume the STATEMATE [17] semantics in the remainder of this paper, but other reasonable choices are possible [12, 16]. We assume that a SC model is accompanied by an interface definition to interact with its environment in the form of a set of input and output *ports*. We describe a structure $S =< X, SC_S, Y >$ where:

$X = \{(p, v) | p \in \mathit{IPorts}, v \in X_p\}$ describes the input of the model. *IPorts* is a set of input port names. Each port p has an associated set of possible input events X_p;

SC_S is the SC model describing the behaviour of the system using the above concepts. A formal description of SC can be found in [18];

$Y = \{(p, v) | p \in \mathit{OPorts}, v \in Y_p\}$ describes the output of the model. *OPorts* is a set of output port names. Each port p has an associated set of possible output events Y_p.

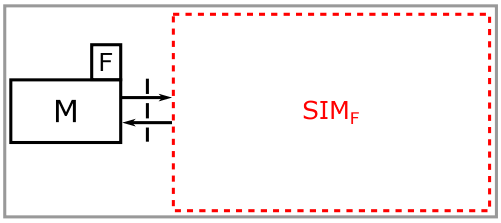
We will use SC extensively throughout the rest of the paper to model the behaviour of each component of our solution. In the models shown throughout this paper, events received on input ports or sent to output ports by the SC model are prepended by *<portname>::* where *<portname>* is the name of the port.

2.2 Explicitly Modelling Model Debuggers

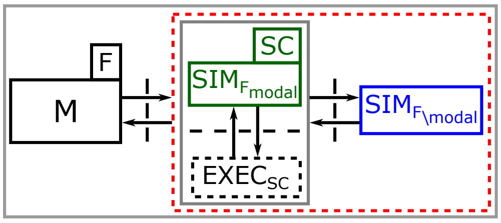
In previous work [35], we have explored the construction of a debugger for the PDEVS formalism. To describe the timed, reactive, autonomous behaviour of the debugger, we use a method called the "de- and reconstruction" of the simulation algorithm [33]. In general, the simulation algorithm for any formalism (see Figure 1a) with behaviour (resulting in a state trace) can be written down in a generic form, as all of them go through three phases:

(1) *Initialization* of simulation time and the simulation state;
(2) *Execution* of simulation 'steps' until an end condition is satisfied (each step computes the new state based on the previous one, and advances the simulated time);
(3) *Finalization* where, for example, the final state of the simulation and the time at which it ended is communicated to the user.

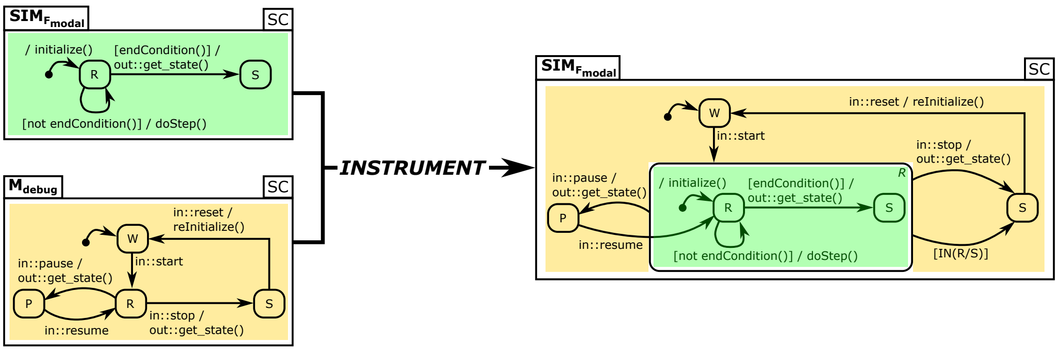
These simulation algorithms can be defined in an appropriate programming language, which results in an executable simulator for that formalism. However, adding debugging support at this abstraction level is difficult, since a debugger is cross-cutting, adding notions of *time* (to run the simulation in real-time, for example), *interruptibility* (at different levels, to implement stepping for example), and *orthogonal behaviour* (to manage breakpoints while the simulation is running, for example). We argue that the simulation algorithm can be deconstructed in its *modal* part (the flow of control, mainly consisting of the "main simulation loop") and its *non-modal* part (consisting of the computation functions that compute the new state during simulation). The modal part of



(a) A model is simulated by an appropriate simulator for its formalism.



(b) The simulator is deconstructed by extracting its modal part.


SIM_F_modal
SC
/ initialize()
[endCondition()] / out::get_state()
R
S
[not endCondition()] / doStep()
M_debug
SC
W
in::reset / reInitialize()
in::pause / out::get_state()
in::start
P
R
S
in::resume
in::stop / out::get_state()
INSTRUMENT
SIM_F_modal
SC
W
in::reset / reInitialize()
in::start
in::stop / out::get_state()
in::pause / out::get_state()
/ initialize()
[endCondition()] / out::get_state()
R
P
R
S
S
in::resume
[not endCondition()] / doStep()
[IN(R/S)]

(c) The modal part of the simulator is instrumented with debugging operations

Fig. 1. De- and reconstruction of simulators.

the simulator is modeled as a SC model (see Figure 1b) and combined with its non-modal part to obtain a behaviourally equivalent version of the simulator; from this definition, an appropriate code generator for SC can be used to obtain an executable simulator. Next, this modal part is augmented with debugging support (see Figure 1c), and the augmented modal part is now used to generate a debugging-enhanced version of the simulator. This debugging-enhanced simulator can then be coupled to a (visual) modelling and simulation interface to allow a user to debug models.

2.3 Modelling and Simulation Workflow

When modelling and simulating complex systems, a number of artefacts are created. While this paper does not discuss different M&S workflows (this can be found in other works, such as [9]), we do focus on a particular set of simulation systems and assume that a number of artefacts are created by the domain expert.

The domain expert is expected to produce three artefacts: a model of the system, a set of experiments (which could be modelled in an experiment language such as SESSL [13]), and an (optional) model-specific visualization. This last artefact is typical in systems that exhibit agent-like behaviour, where each agent is assigned a visual representation, and the domain expert can follow and potentially influence the simulation by interacting with the visualization. We assume that the domain expert models the behaviour of the visualization in SC, as it is an appropriate formalism to specify the “modes” of the visualization in an explicit control flow model [19]. From this model, code is generated and combined with a visualization library to build an executable model-specific visualization interface. The visualization and simulation model are strictly separate;

the visualization can only interact with the model through its interface (*i.e.*, its input and output ports). If no interactivity is required, the simulation can be run without visualization.

Once these artefacts are created, the domain expert runs the simulation experiments and observes the results (typically, a trace generated by the simulation model and/or the visualization) and check whether these results correspond to expectations. This workflow emphasizes modularity, as it splits the simulation system into separate subsystems. It allows the domain expert to replace components as necessary, and increases scalability, as components can be deployed on separate machines. In many cases, a process such as this can be followed by domain experts to design and evaluate their systems (for example, to simulate occupant behaviour in buildings [7]). The remainder of the paper assumes the domain expert employs this workflow (or a comparable one) for modelling and simulating systems.

3 RUNNING EXAMPLE

As a representative example, we choose the Dynamic-Structure DEVS [2] (DSDEVS) formalism to demonstrate our approach in this paper. DEVS [39] has a long history as a popular discrete-event formalism. DSDEVS is one of its extensions, which defines a *network executive* that can evolve the structure of the model during simulation. Because of its well-defined syntax and semantics, it is an ideal formalism to demonstrate our techniques.

3.1 Dynamic Structure DEVS

DSDEVS [2] is used to model the behaviour of discrete event systems whose structure can change over time. In his paper, Barros extends the DEVS formalism and provides a theoretical basis for modelling dynamic structure by introducing a *network executive*, which is also modelled as a DEVS model, and whose state reflects the current configuration of the DEVS network. The basic building blocks of a DSDEVS model are *atomic DEVS* models, which are structures

$$< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta >$$

where the *input set* X denotes the set of admissible input events of the model. The *output set* Y denotes the set of admissible output events of the model. The *state set* S is the set of sequential states of the model. The *internal transition function* $\delta_{int} : S \longrightarrow S$ defines the next sequential state, depending on the current state. The *output function* $\lambda : S \longrightarrow Y$ defines the output to be raised for a given sequential state, upon triggering the internal transition function. The *external transition function* $\delta_{ext} : Q \times X \longrightarrow S$ with $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ gets called whenever an *external input* ($\in X$) is received. The *time advance function* $ta : S \longrightarrow \mathbb{R}^+_{0,+\infty}$ defines the duration the system remains in the current state before triggering its *internal transition function*.

A network of atomic DSDEVS models is defined by the structure

$$< \chi, M_\chi >$$

where χ is the network executive, and M_χ its model, an atomic DSDEVS model

$$< X_\chi, Y_\chi, S_\chi, \delta_{int,\chi}, \delta_{ext,\chi}, \lambda_\chi, ta_\chi >$$

where X_χ and Y_χ are the input and output sets of the executive. S_χ is the set of states, encoding the network structure

$$< X_\Delta, Y_\Delta, D, M_i, I_i, Z_{i,j}, select >$$

which corresponds to the definition of a coupled model in "classic" DEVS. $\delta_{int,\chi}, \delta_{ext,\chi}$ are the executive's internal and external transition function, which can change the state of the executive, and thus change the network structure. λ_χ and ta_χ are the executive's output and time advance functions. DSDEVS is closed under coupling; coupled models can be nested to arbitrary depth.

An abstract simulator for the parallel version of DSDEVS is described in [4]. Simulator implementations often take a more pragmatic approach to implement the structure-varying functions than relying on the network executive. In tools such as PythonPDEVS [36] and adevs [29], a *model transition function* encodes the structural changes of the model. An atomic DEVS model can decide (after it has executed one of its transition functions) to change, for example, its X or Y sets. It can also signal to its parent (a coupled model) that a change is required at that level, for example to add or remove an atomic model. A simulation step in the algorithm for the classic version of DSDEVS, as implemented by these tools, can be summarized as follows:

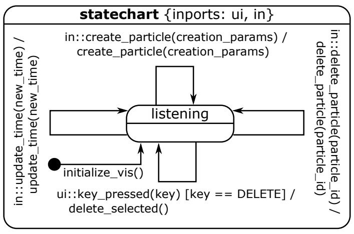
(1) Compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components).
(2) Select one imminent component with a tie-breaking function.
(3) Execute the imminent component's output function, generating an output event.
(4) Route events from sending components to receiving components.
(5) Determine the type of transition to execute for each atomic DEVS model, depending on it being imminent or receiving input.
(6) Execute, in parallel, all enabled internal and external transition functions.
(7) Compute the new structure of the model(s) by executing, for each transitioning model, their model transition function (and their parent's model transition functions as well, if the child requests it).
(8) Compute, for each atomic DEVS model, the time of its next internal transition.

3.2 Example Simulation System

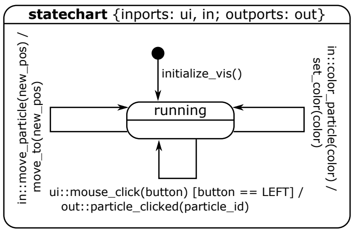
As an example dynamic-structure system, we present a particle interaction simulation, in which a number of particles move in a constrained space, bouncing off walls and each other when they collide. The requirements of the simulation system are listed below:

- A field holds a number of moving particles.
- A new particle is created every second.
- When a particle is created, it chooses a random radius, a random (2D) position within the boundaries of the field, and a random velocity.
- At each frame, a particle updates its current position according to its velocity.
- Each particle deletes itself at a random point in time.
- When two particles collide, they bounce off of each other (by swapping their velocity vectors).
- When a particle bounces against one of the sides of the field, it negates the normal component of its velocity vector to move in the other direction.
- The user can select a particle; this changes its colour to orange and its velocity to 0.
- A selected particle can be deleted by the user.

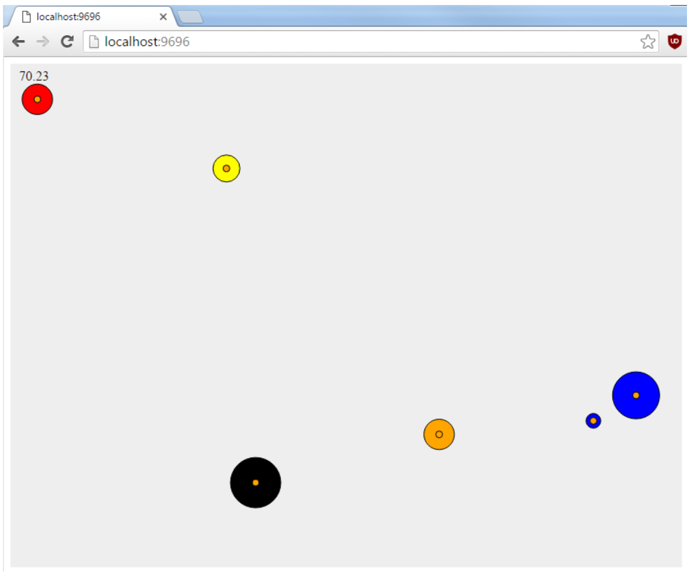
This system clearly exhibits dynamic structure behaviour, and we can use the PythonPDEVS tool to represent its dynamic-structure behaviour. Our model consists of one top-level coupled model called *Field*, which accepts user events (such as *select* or *delete*) and communicates state updates (such as position updates) to the user during simulation. At the start of simulation, a Field is instantiated, which creates two more atomic models are instantiated: a *PositionManager* (responsible for keeping track of the positions of each particle and detecting collisions) and a *ParticleSpawner* (responsible for creating a new particle each second). Each particle computes the new value of its colour and position each simulation step, and then communicates it to the PositionManager and the user. It can receive a message from the PositionManager that notifies the particle a collision has occurred; it can also send a message to the ParticleSpawner, requesting a new particle to be created.


statechart {inports: ui, in}
in::create_particle(creation_params) / create_particle(creation_params)
listening
in::update_time(new_time) / update_time(new_time)
in::delete_particle(particle_id) / delete_particle(particle_id)
initialize_vis()
ui::key_pressed(key) [key == DELETE] / delete_selected()

(a) The SC model of the window.


statechart {inports: ui, in; outports: out}
initialize_vis()
running
in::move_particle(new_pos) / move_to(new_pos)
in::color_particle(color) / set_color(color)
ui::mouse_click(button) [button == LEFT] / out::particle_clicked(particle_id)

(b) The SC model of a particle's visualization.



(c) A screenshot of the model-specific visualization interface.

Fig. 2. The model-specific visualization UI—models and generated application.

We define a suitable model-specific visualization in Fig. 2. Fig. 2a shows the SC model of the top-level window, which receives events generated by the simulation model (*i.e.*, position and colour updates, as well as creation and deletion of particles) and user events (*i.e.*, mouse clicks and key presses) that are translated to output events. Each particle's visualization behaviour is controlled by a separate SC model (shown in Fig. 2b), created by the window when the particle is created.

4 APPROACH

This section explains our approach to construct debugging environments for dynamic-structure formalisms. We start by analyzing a useful set of debugging operations, focusing on their adaptation when taking dynamic structure into account. We then model the debugger's reactive behaviour (an instrumented version of the formalism's simulator) using SC. The debugging-enhanced simulator is then connected to the debugging user interface and the (instrumented) model-specific visualization, resulting in a debugging environment. Throughout this section, we use PythonPDEVS' implementation of the DSDEVS formalism as the running example.

4.1 Debugging Operations

The first step in creating a debugger for a new formalism consists of constructing a set of useful debugging operations for that formalism. In previous work [35], we proposed a set of debugging operations for the PDEVS formalism. Building a debugger for dynamic-structure formalisms is more challenging, since simulation entities can appear and disappear at runtime; we explore the intricacies of implementing operations when faced with dynamic structure. The operations can be divided into three categories: those related to (simulated) time, those related to state (manipulation), and breakpoints. Many more debugging operations can be imagined and addressed in future work;

we do not aim to implement an exhaustive list of all possible debugging operations, but rather choose a number of representative examples in each category and study their behaviour when implemented for dynamic-structure systems.

4.1.1 Time. Simulated time is the internal clock of the simulator. It is updated as the simulation progresses, and can have different relations to the wall-clock time, as discussed in [33]. We distinguish two "modes" of execution that any debugging tool for formalisms with a notion of time should support:

- **[As-Fast-as-Possible]** In this mode, the simulation runs as fast as the underlying hardware can manage and the operating system allows, until the end condition is satisfied. It allows the user to run the simulation without seeing intermediate results, in case they are only interested in the resulting trace or collected metrics.
- **[Real-Time]** In this mode, simulated time is synchronized with the wall-clock time. A scale factor can be applied to speed up or slow down simulation, while retaining the *linear* relation between simulated time and wall-clock time. A scale factor of 1 corresponds to real-time, while a scale factor smaller or greater than 1 slows down or speeds up simulation proportionally.

Finer-grained control over time is possible and useful when debugging a simulation:

- **[Pause]** Pausing a simulation allows the user to inspect the current state of the system and enables other debugging operations, such as stepping and state modifications.
- **[Big Step and Small Step]** Stepping through a simulation is useful to get insight into how the state of the model evolves. This is usually hidden by the simulator's implementation, but a debugger can offer different levels of stepping, which reveal (parts of) the computations performed by the simulator. For example, in PythonPDEVS, we can discern two levels (see Section 3.1): one iteration of the simulation algorithm can be seen as a "big step", while each of the eight phases in an iteration can be seen as "small steps".

Simulation time does not advance when the simulation is paused. It can only advance during an iteration of the simulation algorithm.

4.1.2 State. Code debuggers allow a user to inspect the state of the running program (consisting of the runtime variables in memory, and the program counter) and to modify that state when the program is paused. This allows for quick verification of a hypothesis without the user having to modify the code and starting a new debugging session. Simulation systems also have state; in case of DSDEVS, the state is an aggregation of the states of its (coupled or atomic) models. We distinguish a direct and an indirect method of changing the state of the system when the simulation is paused:

- **[God Event]** A "god event" allows a user to change the value of a state variable. In case of DSDEVS, this operation changes the current state $(s_{i,curr}, e_{i,curr}) \in Q_i$ to a new state $(s_{i,new}, 0) \in Q_i$ where Q_i is the total state set of the atomic model i.
- **[Event Injection]** For formalisms that have a notion of events, this operation allows the user to manually inject an event to quickly verify how the model reacts. In case of DSDEVS, a user can schedule to inject an event $x \in X_i$ at a specified (future) simulated time instant t. Once time t is reached, the atomic model i receives the input event x, triggering its external transition function.

4.1.3 Breakpoints. A breakpoint is used in code debugging to pause the execution of the program when a specific line of code is reached, and when its associated (optional) condition is satisfied. We transpose breakpoints by allowing the user to specify conditions on the execution state of the

Structural Change / Debugging Operation	(1) Pause	(2) Big Step	(3) Small Step	(4) Reset	(5) God Event	(6) Inject Event	(7) Breakpoint	(8) Visualization
Add or Remove State Variable	Y	Y	Y	Y	Y	Y	N	N
Change Input or Output Set	Y	Y	Y	Y	Y	N	N	N
Add or Remove Connection	Y	Y	Y	Y	Y	Y	N	N
Add or Remove Component	Y	Y	Y	Y	Y	N	N	N
Change Transition Function	Y	Y	Y	Y	Y	Y	Y	N
Change Output Function	Y	Y	Y	Y	Y	Y	Y	N
Change Time Advance Function	Y	Y	Y	Y	Y	Y	Y	N
Change Transfer Function	Y	Y	Y	Y	Y	Y	Y	N
Change Select Function	Y	Y	Y	Y	Y	Y	Y	N

Table 1. Structural changes and their effect on debugging.

simulation (including access to time and current (total) state information). In case of DSDEVS, we choose to model a breakpoint as a function that returns *true* when the simulation should pause, *false* in all other cases, and it receives five parameters: (1) the current simulation time, (2) the current state of the simulation system, which is an aggregation of the states of its atomic DSDEVS components, as well as the current structure of the system, (3) the names of atomic DSDEVS models that executed a transition function in the iteration preceding the triggering of the breakpoint, (4) the output generated by the atomic DSDEVS models that executed their internal transition function in the iteration preceding the triggering of the breakpoint, and (5) the input received by the atomic DSDEVS models that executed their external transition function in the iteration preceding the triggering of the breakpoint. When a breakpoint is triggered, these values are passed to the user in order to help the user understand why the breakpoint was triggered. This allows the user to model breakpoints that trigger at a specific point in time, on a state change, on a structure change (entities that were created/deleted), or one of the components executing a transition function. Breakpoints, similar to a manual pause, can only break *after* a big step has completed (and the system is in a consistent state). More advanced breakpoint conditions, such as conditions on the simulation trace (specified in a domain-specific property language [26]) are left as future work.

4.1.4 Changes to Operations. Our set of debugging operations is based on one which is useful for static-structure formalisms. However, this means that debugging operations which previously relied on the static structure of the model might become meaningless. In Table 1, we identify the debugging operation that must be changed to accommodate dynamic-structure formalisms, and in particular DSDEVS. The first column lists all possible changes to the structure that are allowed by PythonPDEVS in its structure varying function. Each column corresponds to a debugging operation. Each cell is either green (**Y**), meaning the operation is unaffected by the structure change, or red (**N**), meaning it can be affected by the structure change. A number of debugging operations are unaffected:

(1) **[Pause]** Pausing a simulation is independent of structural changes. It pauses the simulation as soon as possible, either after the currently executing big step, or, in the case of real-time simulation, potentially during a waiting period. This ensures the system is in a stable (or quiescent) state that can be inspected by the user.

(2)–(3) **[Big Step and Small Step]** Stepping through a simulation is similarly unaffected. It can be compared to manually pausing after each iteration, or after the execution of a simulation

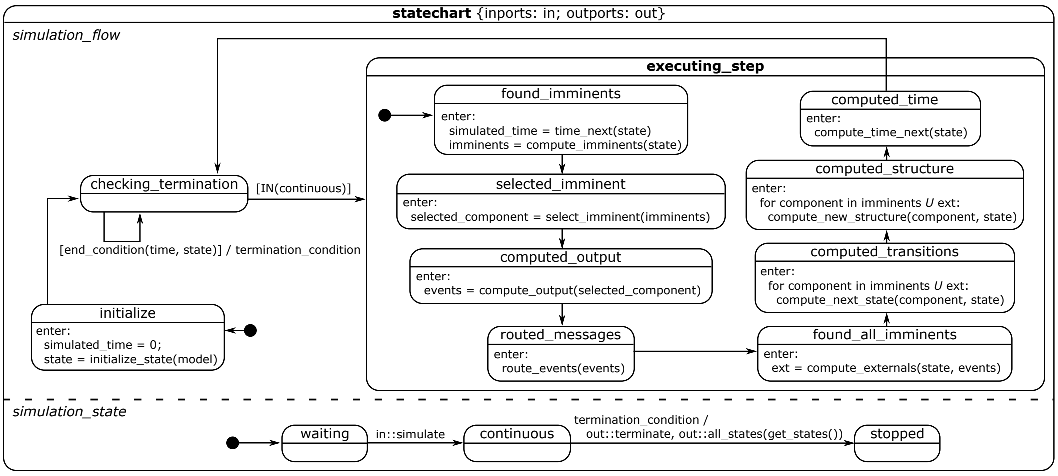

statechart {inports: in; outports: out}
simulation_flow
executing_step
found_imminents
enter:
simulated_time = time_next(state)
imminents = compute_imminents(state)
computed_time
enter:
compute_time_next(state)
checking_termination
[IN(continuous)]
selected_imminent
enter:
selected_component = select_imminent(imminents)
computed_structure
enter:
for component in imminents U ext:
compute_new_structure(component, state)
[end_condition(time, state)] / termination_condition
computed_output
enter:
events = compute_output(selected_component)
computed_transitions
enter:
for component in imminents U ext:
compute_next_state(component, state)
initialize
enter:
simulated_time = 0;
state = initialize_state(model)
routed_messages
enter:
route_events(events)
found_all_imminents
enter:
ext = compute_externals(state, events)
simulation_state
waiting
in::simulate
continuous
termination_condition /
out::terminate, out::all_states(get_states())
stopped

Fig. 3. The SC model of PythonPDEVS's DSDEVS simulation algorithm without debugging support.

phase. For PythonPDEVS, an additional small step (corresponding to the phase which executes the model transition function) was added for communicating the structural changes to the user, but the functioning of the small step operation is not affected by changes in the system structure during simulation.

(4) **[Reset]** Resetting the simulation restores the initial state of the system.
(5) **[God Event]** A god event changes the value of a state variable. It can only change variables that exist when the simulation is paused, and is therefore unaffected.

In contrast, three debugging operations are affected:

(6) **[Event Injection]** A user can inject an event at a specified point in (future) simulation time. Two changes have an effect on this operation: if a component that is the recipient of an injected event is removed, or its input set changed, that injection might become invalid. We solve this by ignoring the injected event in those cases.
(7) **[Breakpoints]** When modelling a breakpoint, the user is not aware of future structural changes. The breakpoint might attempt to access a component, its input/output sets, or its state variables that no longer exist. We disable the breakpoint automatically when it tries to access a non-existent component and warn the user.
(8) **[Visualization]** As the visualization is responsible for displaying the state of the system, and that state now contains the structure of the system, it is affected by all operations that change the structure of the system. The visualization of the state has to be appropriately updated when components, variables, components (and their functions) are added, removed, or changed. The point in time when the visualization is updated depends on the mode of simulation. In as-fast-as-possible simulation, the visualization is updated at the end of simulation. In real-time simulation and while the user is big stepping, the visualization is updated after each big step. When the user is small stepping, the visualization is updated after each small step.

4.2 Modelling the Simulation Kernel

Once a set of debugging operations is known, we need to add these operations to the formalism's simulator. The first step towards enhancing a simulator with debugging support is to split it up into its modal and non-modal parts, as explained in Section 2.2. The non-modal part of the simulation kernel consists of its computation functions. Usually these functions are hidden away in the kernel's implementation, but they can be exposed through an API provided by the developer of the simulator. Some simulation kernels will offer a limited amount of control (for example, only "big stepping" and querying the state of simulation), which limits the amount of debugging operations that can be implemented. For the PythonPDEVS debugger, we have access to the kernel and can adapt the computation functions to choose the level of granularity—we will see an example of a black-box simulator in Section 5. Once the non-modal part is fixed and isolated, we extract the simulator's control flow (its algorithm) and model it in the SC language. The result of this deconstruction step for the PythonPDEVS debugger is shown in Fig. 3. The SC model accepts (user) events on an input port *in* and communicates the simulation results on an output port *out*. It consists of two orthogonal components:

- *simulation_state* keeps track of the current state of the simulation kernel. It waits for an input event from the user that starts the simulation algorithm. When the simulation ends, it communicates the final state of the simulation to the user.
- *simulation_flow* models the flow of the simulation algorithm: it initializes the simulation, and then continuously updates the state in the **do_simulation** state (going through the eight phases) until the end condition is satisfied.

We combine this SC model of the modal behaviour of the simulation kernel with its non-modal part in the reconstruction step. We then regenerate the—behaviourally equivalent—code of the simulation kernel using an appropriate SC compiler.

4.3 Modelling the Debugger

The debugging operations presented in Section 4.1 affect the control flow. Indeed, to be able to step through a simulation instead of executing it continuously, the control flow has to be adapted in order to give appropriate control to the user. We make sure that instrumenting the SC model of the simulation kernel does not alter its original execution semantics. This ensures continuity: if the user does not make use of the debugging operations, the debugging-enhanced simulation kernel will produce the same trace as the original simulation kernel. The result of the instrumentation step is presented in Fig. 4. A number of orthogonal components have been added:

- The *injection_monitor* allows the user to inject an event on an input port of the DSDEVS model at a specified point in simulated time.
- The *breakpoint_manager* allows the user to add, delete, and toggle breakpoints.
- The *reset_monitor* allows a user to reset a running simulation, which re-initializes the simulated time and state.

Two simulation modes have been added in the *simulation_state* component: the simulation can now be stepped through, or run in (scaled) real-time.

Stepping behaviour is implemented in the *simulation_flow* component: the termination check will bring the component to the **do_simulation** state, which will execute one iteration of the simulation algorithm. At the end (when transitioning back to the **check_termination** state), the system raises a *big_step_done* event, which will result in the *simulation_state* component to transition back to the *paused* state. This ensures no second step is executed, as the *simulation_flow* state remains in the **check_termination** state until the *simulation_state* changes. The user can perform small steps when the simulation is paused. There is no state in the *simulation_state* to denote the user is small stepping: it stays paused. Instead, the user can manually advance the *simulation_flow* by sending *small_step* events, which will cycle through the phases of the simulation in the **do_simulation** state. After each phase, relevant information is passed to the user. A special state *checking_small_step* was

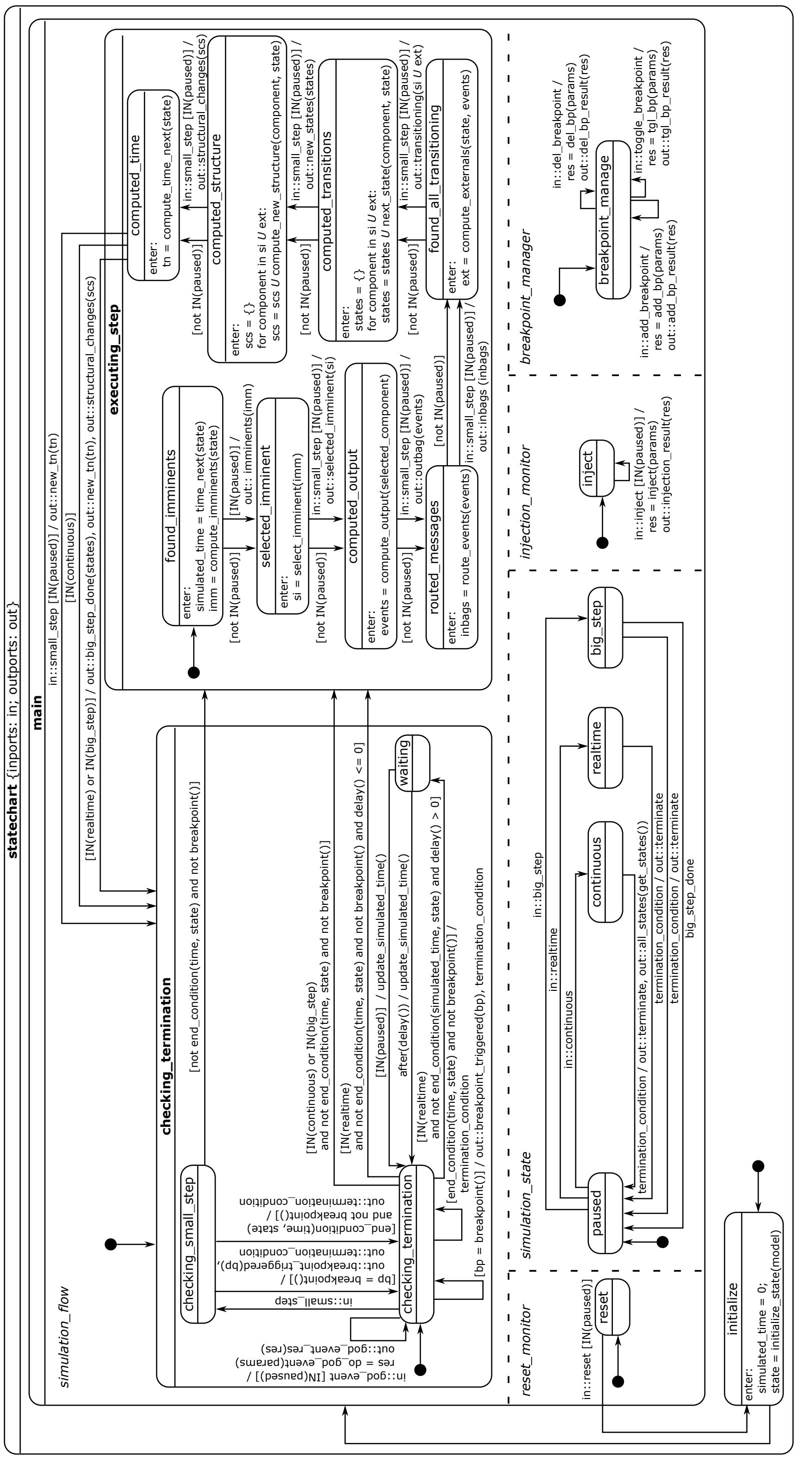

statechart {inports: in; outports: out}
main
simulation_flow
executing_step
computed_time
enter: tn = compute_time_next(state)
computed_structure
enter: scs = {} for component in si U ext: scs = scs U compute_new_structure(component, state)
computed_transitions
enter: states = {} for component in si U ext: states = states U next_state(component, state)
found_all_transitioning
enter: ext = compute_externals(state, events)
found_imminents
enter: simulated_time = time_next(state) imm = compute_imminents(state)
selected_imminent
enter: si = select_imminent(imm)
computed_output
enter: events = compute_output(selected_component)
routed_messages
enter: inbags = route_events(events)
[not IN(paused)]
in::small_step [IN(paused)] / out::new_tn(tn)
[IN(continuous)]
[IN(realtime) or IN(big_step)] / out::big_step_done(states), out::new_tn(tn), out::structural_changes(scs)
in::small_step [IN(paused)] / out::structural_changes(scs)
in::small_step [IN(paused)] / out::new_states(states)
in::small_step [IN(paused)] / out::transitioning(si U ext)
in::small_step [IN(paused)] / out::inbags (inbags)
[IN(paused)] / out:: imminents(imm)
in::small_step [IN(paused)] / out::selected_imminent(si)
in::small_step [IN(paused)] / out::outbag(events)
checking_termination
checking_small_step
checking_termination
waiting
[not end_condition(time, state) and not breakpoint()]
[IN(continuous) or IN(big_step) and not end_condition(time, state) and not breakpoint()]
[IN(realtime) and not end_condition(time, state) and not breakpoint() and delay() <= 0]
[IN(paused)] / update_simulated_time()
after(delay()) / update_simulated_time()
[IN(realtime) and not end_condition(simulated_time, state) and delay() > 0]
[end_condition(time, state) and not breakpoint()] / termination_condition
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
in::small_step
[bp = breakpoint()] / out::breakpoint_triggered(bp), out::termination_condition
[end_condition(time, state) and not breakpoint()] / out::termination_condition
in::god_event [IN(paused)] / res = do_god_event(params) out::god_event_res(res)
breakpoint_manager
breakpoint_manage
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
injection_monitor
inject
in::inject [IN(paused)] / res = inject(params) out::injection_result(res)
simulation_state
paused
continuous
realtime
big_step
in::big_step
in::realtime
in::continuous
termination_condition / out::terminate, out::all_states(get_states())
termination_condition / out::terminate
termination_condition / out::terminate
big_step_done
reset_monitor
reset
in::reset [IN(paused)]
initialize
enter: simulated_time = 0; state = initialize_state(model)

Fig. 4. The SC model of the simulation kernel with debugging support.

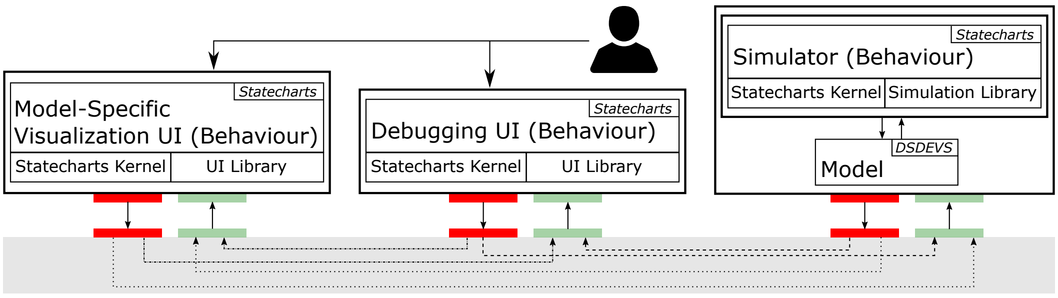


Fig. 5. The architecture of our debugger.

added in **check_termination**: since the user is able to execute small steps when the simulation is paused, this state checks whether no breakpoint was triggered or the simulation has ended. If this extra check were not there, a user could execute a small step past the simulation end. Once this check is done, the *found_imminents* state is entered, and the transition to the *selected_imminent* component is taken unconditionally, since the user has already requested a small step.

Additionally, the **check_termination** state is expanded with real-time behaviour, which makes the simulation wait until the current time specified in the time advance functions of the DSDEVS model has passed in wall-clock time. The waiting behaviour is achieved by cycling between the *check_termination* state and the *wait* state, with a minimal delay. This cycling behaviour allows the user to pause the simulation while the simulation algorithm is waiting until the model executes its next transition (while still updating the simulated time). When in real-time or big step mode, the new state of the system is communicated to the user. This allows the user to track changes to the state during a debugging session. The debugging-enhanced simulation kernel also checks whether any breakpoint was triggered in the *check_termination* state. If so, it raises the *termination_condition* event, which makes the *simulation_state* transition to *paused*. The user is also notified that a breakpoint triggered.

From this instrumented model, we generate the behaviour of the debugging enhanced simulation kernel of PythonPDEVS using an appropriate SC compiler.

4.4 Architecture

Once a debugging-enhanced simulation kernel is obtained, it can be connected to other components to create a debugging environment. In previous work, the architecture connected the debugging-enhanced simulation kernel to an instrumented version of the (visual) modelling interface for the formalism. In this paper, we extend the architecture to include the model-specific visualization. This is particularly useful for dynamic-structure formalisms, since simulation entities can appear and disappear during simulation; the model-specific visualization can then help provide an overview.

The architecture of our solution is shown in Fig. 5. It consists of three components, whose behaviour is specified in SC models:

- The simulator, whose behaviour is described by

$$S_{sim} =< X_{sim}, SC_{sim}, Y_{sim} >$$

- The debugging user interface, whose behaviour is described by

$$S_{ui\_debug} =< X_{ui\_debug}, SC_{ui\_debug}, Y_{ui\_debug} >$$

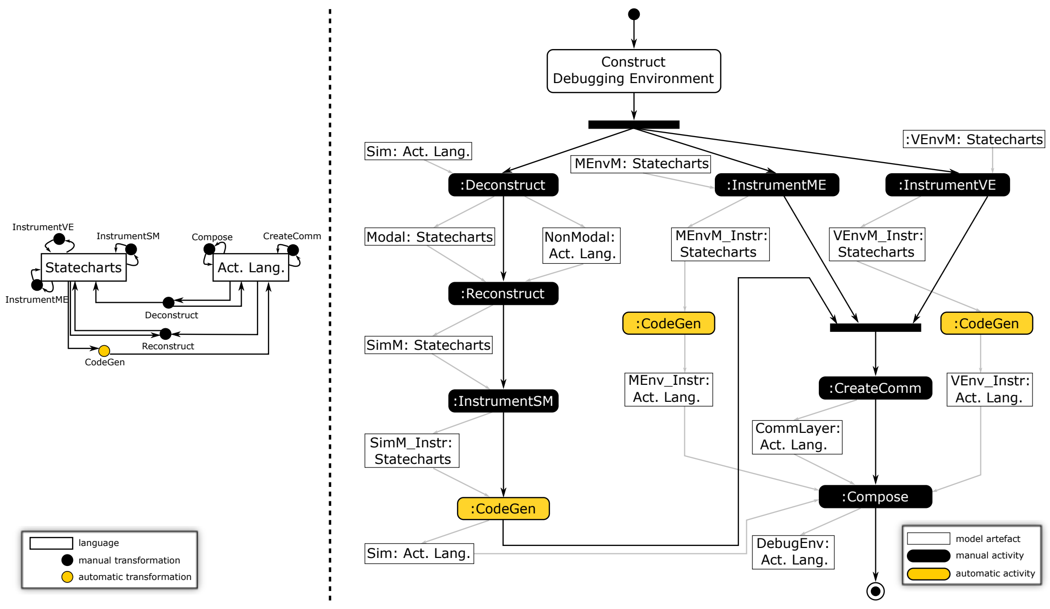


Fig. 6. Workflow (in an FTG+PM language) to construct a debugging environment.

- The model-specific visualization interface (instrumented with debugging operations), whose behaviour is described by

$$S_{ui\_vis} =< X_{ui\_vis}, SC_{ui\_vis}, Y_{ui\_vis} >$$

The first two models, S_{sim} and S_{ui_debug}, have to be defined once and can be reused to debug any model created in the simulator's formalism (in this case, DSDEVS). The third, S_{ui_vis}, is model-specific and has to be adapted to specific simulation models. The states of the components in the architecture are kept synchronized: the bus connecting the components delivers the output events (after an optional translation) of all components to the others. Each component can then react by updating their state, as defined in the reactive behaviour specified in their SC model.

4.5 Workflow

The workflow for constructing a debugger for dynamic-structure formalisms starting from an existing simulation kernel, a modelling and simulation environment without debugging support, and a model-specific visualization UI, is shown in Fig. 6. The workflow is modelled in a Formalism Transformation Graph and Process Model (FTG+PM) language [22, 28]. FTG+PM allows one to model engineering workflows whose deliverable is a set of (software) artefacts; a modeller can relate each artefact to the language it is specified in, as well as specify transformations between the used languages. The right part of an FTG+PM model represents a workflow consisting of activities that require user input (manual activities) and tasks that do not (automatic transformations). The left part describes a formalism transformation graph consisting of a number of formalisms and (automatic/manual) transformations between them. Tasks in the workflow are explicitly typed by transformations in the formalism transformation graph, while the artefacts created are typed by the formalisms in the formalism transformation graph.

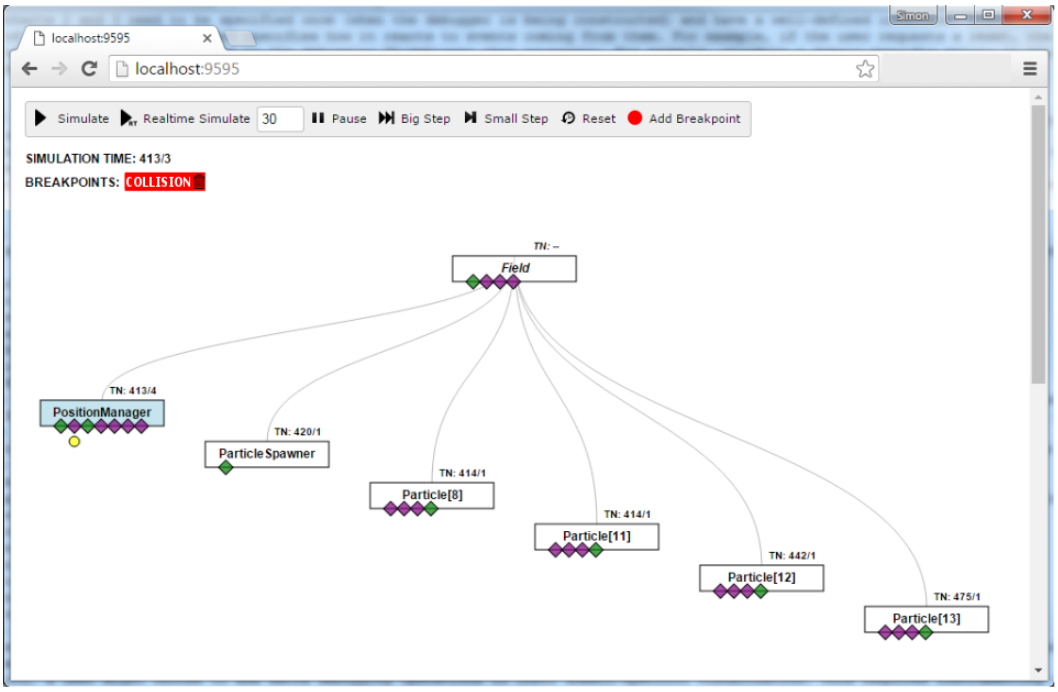


Fig. 7. A screenshot of the DSDEVS debugging interface.

There are four tasks in our workflow: de- and reconstructing the simulation kernel, instrumenting the modelling environment, instrumenting the model-specific visualization UI, and creating a communication layer. These four tasks were explained in the previous subsections.

5 DEBUGGER PROTOTYPES

This section explains how we applied our techniques to create a debugger for PythonPDEVS, a simulation kernel we are familiar with that can simulate DSDEVS models, and Netlogo, a multi-agent modelling environment. By doing so, we demonstrate that out techniques are transferable to tools other than those we know the source code of. The implementation of both prototypes can be found on *https://msdl.uantwerpen.be/git/simon/DSDEVS-Debugger*.

5.1 PythonPDEVS (DSDEVS) Debugger

We discussed the PythonPDEVS debugger's behaviour (and how to model it) in the previous section. This section shows the result of combining the different components (the debugging-enhanced simulation kernel, the debugging user interface, and the model-specific visualization) using our architecture to obtain a debugger for PythonPDEVS.

5.1.1 Debugging User Interface. This section presents a basic visual debugging interface, which allows for full control over the simulation algorithm using the set of debugging operations defined in Section 4.1 and implemented in Section 4.3. The behaviour of the debugging interface is modelled using SC. Each user action corresponds to an input event to this model and is translated to an output event that is sent to bus of the architecture. Conversely, any events arriving on the bus serve as input to the SC model of the debugging interface. A screenshot of the interface is shown in Fig. 7.

At the top, a tool bar allows the user to interact with the running simulation by pressing buttons. Below that, information related to the running simulation is visualized: the current simulated time and the current set of breakpoints are shown at the top, and below that, the current structure of the system. Each DSDEVS model is represented by a rectangle. The name of each component is displayed (coupled models have their names printed in italics), as well as their input (green) and output (purple) sets in the form of ports. Hovering over a port displays the name of the port as well as any incoming or outgoing connections to other ports. Hovering over (atomic) models displays their current state. The next time ("TN") at which an internal transition is scheduled is displayed above each atomic model. This is not the case for coupled models (such as *Field*): their next transition time is the minimum of its constituent components' next transition times. Clicking on an input port allows the user to inject an event. Right-clicking on an atomic model is used for a god event (changing the value of a state variable). The hierarchical structure of the DSDEVS model is displayed as a tree—children of a coupled DSDEVS model are shown below their parent. Models at the same level are staggered, to allow for more elements to be displayed side-by-side. Additionally, this layout helps with the drawing of incoming and outgoing connections between ports. While the example shows a two-level tree, this is not necessarily always so: the tree can be arbitrarily deep (depending on the current model structure).

When the user executes a small step, the debugging interface visualizes useful information regarding the executed phase. This visualization, including an explanation of each small step, is shown in Table 2. Small steps allow the user to debug the system at a more fine-grained level, inspecting closely what happens during a simulation step.

5.1.2 Model-Specific Visualization. For debugging purposes, if some DSDEVS models are visualized by entities in a UI, those entities can be instrumented with extra debugging information that is sent by the simulator. And vice versa, certain interactions with entities in the model-specific visualization UI might highlight the corresponding DSDEVS model in the debugging UI. For this to work, the domain expert has to instrument the SC model of the model-specific visualization UI with appropriate transitions that raise events on its output port. The same tasks have to be performed if the visualization UI has to respond to events from the instrumented simulation kernel. In the case of our example DSDEVS system, we augmented the SC model in Fig. 2b with a transition that listens to the user pressing the right mouse button, generating an event on its output port; this event is translated to an input event of the debugging interface, which reacts by highlighting the corresponding atomic model. The debugging interface, on the other hand, generates an output event if the user presses the key to reset the simulation; this event is translated to an input event of the model-specific visualization interface, which resets the visualization.

5.2 NetLogo Debugger

NetLogo is a tool for modelling multi-agent systems. In this section, we explain how we used our techniques to develop a debugger for NetLogo in a short period of time.

A NetLogo model defines *agents* (called *turtles*), which each have their own state and a set of *operations*, which act upon that state. Typically, there is a *main function* that encodes the main simulation loop by coordinating the set of turtles. Turtles can be created and deleted (making it a dynamic-structure modelling language); they can communicate by calling each other's operations. NetLogo comes with an interface to interact with the running simulation: *widgets* (such as buttons and sliders) allow a user to send commands (that call operations) to the simulation and to view the current/aggregated state of the model (by viewing a visualization of the current state of the "world", or viewing statistics in a plot). Time advances in *ticks*: after one iteration of the simulation algorithm, a tick ends and the next iteration is started. The semantics of the NetLogo execution

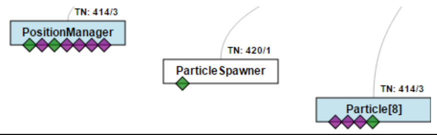
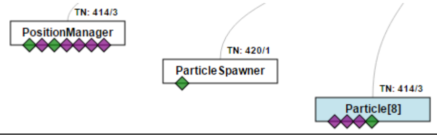
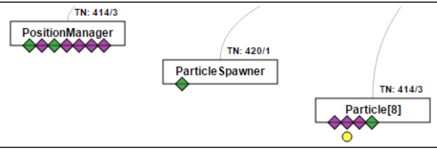
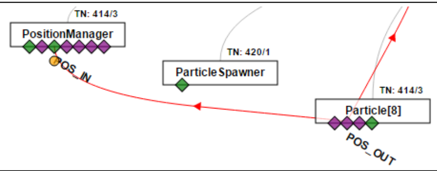
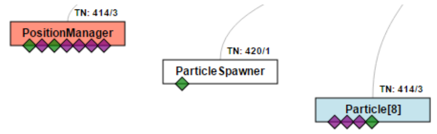
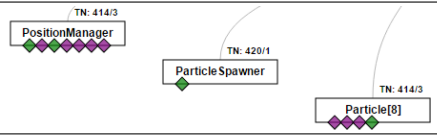
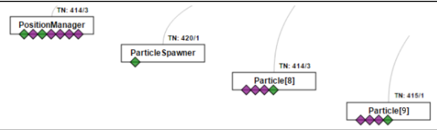
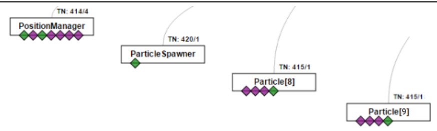
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(1) All imminent components are highlighted in blue.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(2) Only the selected imminent component is highlighted in blue.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(3) The output message(s) generated by the imminent component are shown beneath the corresponding port(s). Clicking the message will log its contents in the console.
 TN: 414/3 PositionManager POS_IN TN: 420/1 ParticleSpawner TN: 414/3 Particle[8] POS_OUT	(4) The message is routed from the output port(s) to the input port(s) along the connections, and translated in the process.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(5) All transitioning components are highlighted: red for components that will execute their external transition function, blue for those that will execute their internal transition function.
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8]	(6) The new state of the components is computed (no visual change, but hovering over the atomic models will show their new state).
 TN: 414/3 PositionManager TN: 420/1 ParticleSpawner TN: 414/3 Particle[8] TN: 415/1 Particle[9]	(7) Structural changes are displayed: an atomic model was created.
 TN: 414/4 PositionManager TN: 420/1 ParticleSpawner TN: 415/1 Particle[8] TN: 415/1 Particle[9]	(8) The time at which the next internal transition function is scheduled is updated for each component.

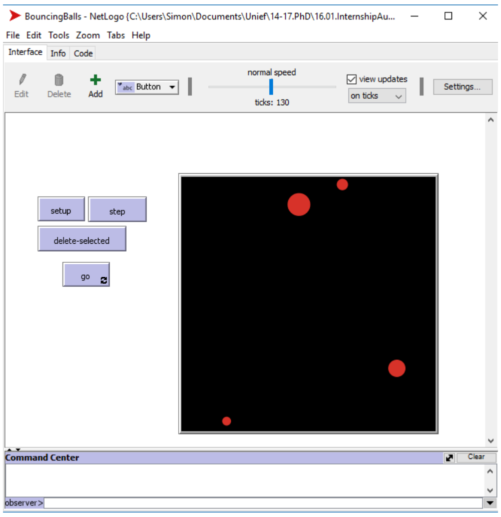
Table 2. Small step visualization: the eight phases of a simulation iteration.

engine are not formalized. Instead, a modeller has to encode the execution semantics of each model in its operations. An extensive set of example models and documentation is available[1].

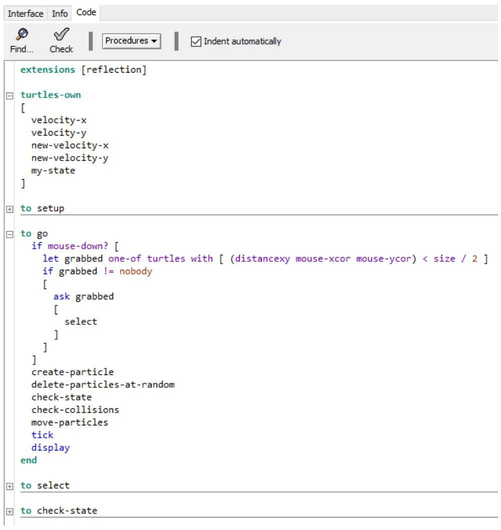
We can model the example particle interaction system (see Section 3.2) in NetLogo. Figure 8a shows the visual user interface of NetLogo. It displays the current state of the "world": a number of particles are moving in the constrained space. A user can click on a particle to select it or press one of the five buttons to interact with the simulation:

- *setup* is used to initialize the simulation;

[1]https://ccl.northwestern.edu/netlogo/docs/



(a) The visual user interface of NetLogo.



```
extensions [reflection]

turtles-own
[
  velocity-x
  velocity-y
  new-velocity-x
  new-velocity-y
  my-state
]

to setup

to go
  if mouse-down? [
    let grabbed one-of turtles with [ (distancexy mouse-xcor mouse-ycor) < size / 2 ]
    if grabbed != nobody
    [
      ask grabbed
      [
        select
      ]
    ]
  ]
  create-particle
  delete-particles-at-random
  check-state
  check-collisions
  move-particles
  tick
  display
end

to select

to check-state
```

(b) Model excerpt in NetLogo.

Fig. 8. Modelling the example particle interaction system in NetLogo.

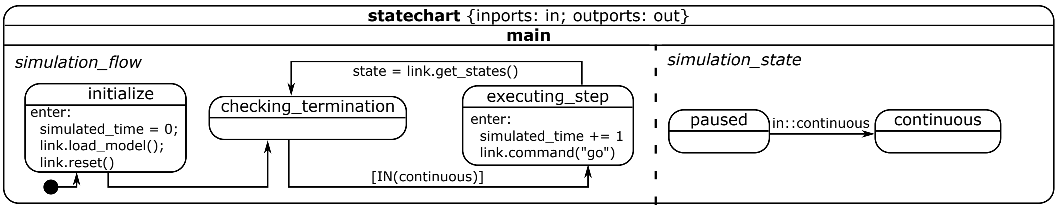


Fig. 9. Modelling the NetLogo executor's modal behaviour using SC.

- *step* advances the simulation one iteration by calling the *go* operation (see below);
- *delete-selected* deletes all selected particles;
- *go* continuously executes the *go* operation, effectively simulating the model.

An excerpt of the model is shown in Fig. 8b. The *go* operation encodes a simulation iteration:

- It starts by checking whether the user clicked on a particle;
- *create-particle* creates a new particle every 30 ticks;
- *delete-particles-at-random* deletes a random particle with a certain probability;
- *check-state* sets the colour and velocity of selected particles;
- *check-collisions* swaps the velocity vectors of two particles when they collide;
- *move-particles* moves all particles according to their velocity;
- *tick* and *display* let the visual environment know that an iteration has ended.

Modelling this example system in NetLogo is quite different from modelling it in DSDEVS. With DSDEVS, the simulator ensures that the correct transition functions of the DEVS models are called, messages are routed, etc. In NetLogo, on the other hand, the semantics are explicitly encoded in operations, which can query and act upon the state of the system. It is a more direct way of encoding the semantics, and as such it is interesting to look at how we can create a debugger for such a system.

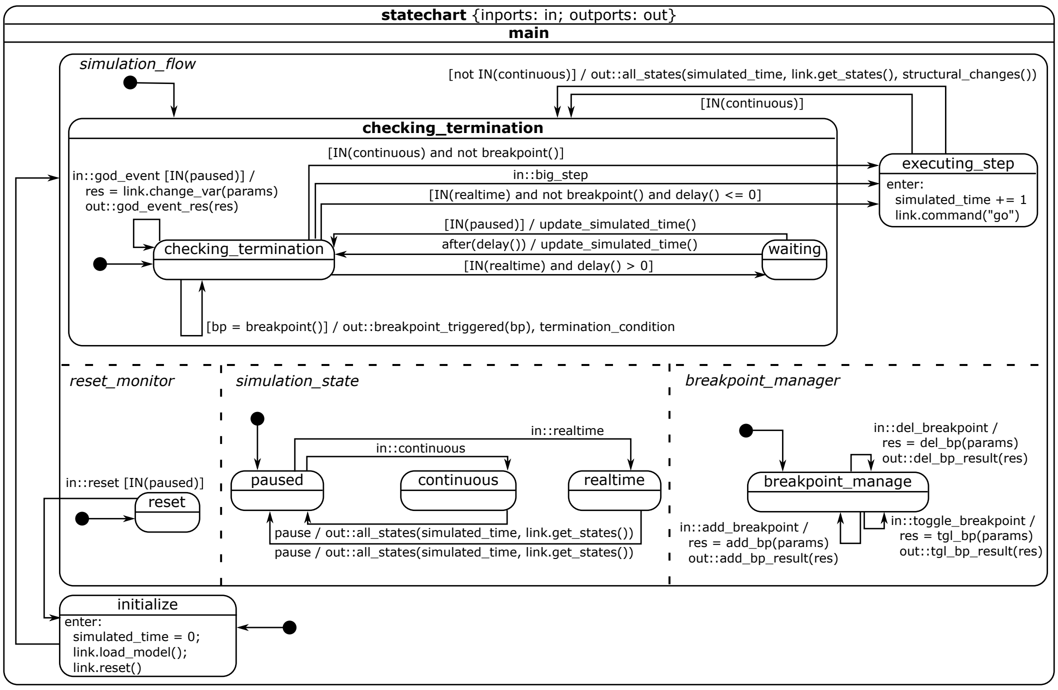

statechart {inports: in; outports: out}
main
simulation_flow
[not IN(continuous)] / out::all_states(simulated_time, link.get_states(), structural_changes())
[IN(continuous)]
checking_termination
[IN(continuous) and not breakpoint()]
executing_step
in::big_step
in::god_event [IN(paused)] / res = link.change_var(params) out::god_event_res(res)
[IN(realtime) and not breakpoint() and delay() <= 0]
enter: simulated_time += 1 link.command("go")
[IN(paused)] / update_simulated_time()
after(delay()) / update_simulated_time()
checking_termination
waiting
[IN(realtime) and delay() > 0]
[bp = breakpoint()] / out::breakpoint_triggered(bp), termination_condition
reset_monitor
simulation_state
breakpoint_manager
in::realtime
in::continuous
in::del_breakpoint / res = del_bp(params) out::del_bp_result(res)
in::reset [IN(paused)]
paused
continuous
realtime
breakpoint_manage
reset
pause / out::all_states(simulated_time, link.get_states())
pause / out::all_states(simulated_time, link.get_states())
in::add_breakpoint / res = add_bp(params) out::add_bp_result(res)
in::toggle_breakpoint / res = tgl_bp(params) out::tgl_bp_result(res)
initialize
enter: simulated_time = 0; link.load_model(); link.reset()

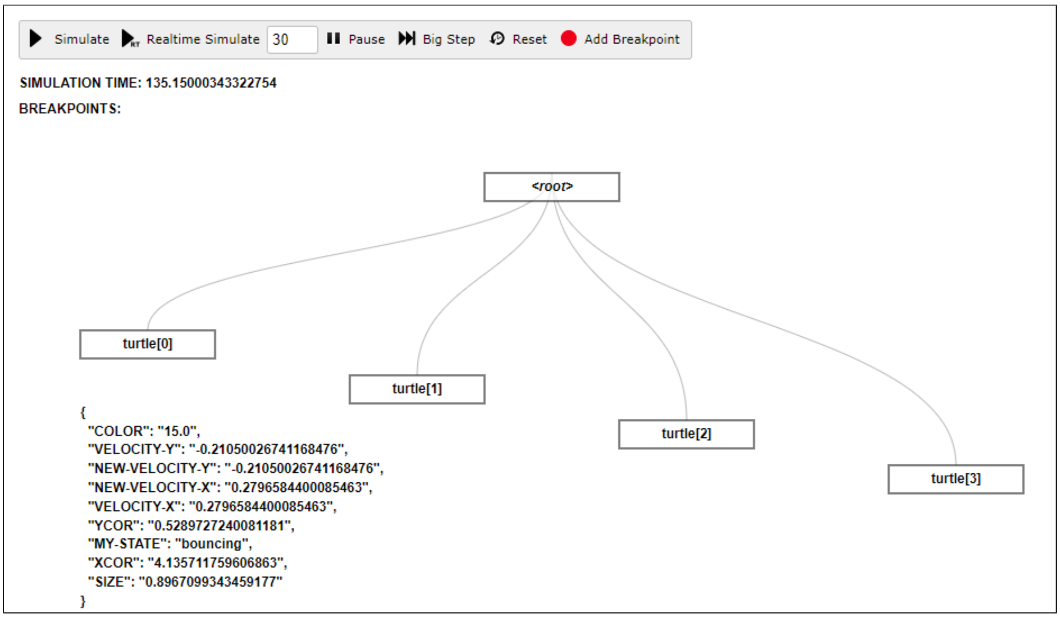
Fig. 10. The SC model describing the behaviour of the NetLogo debugger.

We start by modelling NetLogo's execution semantics using SC, shown in Fig. 9. Since the semantics of the model are mainly encoded in its operations, this model is quite minimal. We rely mainly on NetLogo's controlling API (the *link* variable is a reference to NetLogo's API), which allows us to control a model from outside NetLogo (by calling the model's operations). Moreover, we use Netlogo's extension API to add an operation to the non-modal part of NetLogo that returns the current set of turtles (including the values of their variables). Comparing the model of NetLogo's execution semantics, shown in Fig. 9, to the model of the execution semantics of DSDEVS, shown in Fig. 3, a similar structure can be discerned: the simulator's *main loop* is encoded by the transitions between the *checking_termination* and *executing_step* states, which runs indefinitely while the *simulation_state* orthogonal region is in the *continuous* state.

We then model the debugger's behaviour in a SC model which calls the appropriate operations, and offers the following operations:

- *simulate* the model continuously;
- *realtime simulate* the model (with an optional scale factor);
- *pause* a running simulation;
- *big step* through the simulation (executing one iteration);
- *reset* the simulation;
- *breakpointing* to automatically pause the simulation when a condition is satisfied;
- *god event* to change the value of one of the turtle's variables.

To implement this behaviour, we can reuse parts of the SC model describing the behaviour of the PythonPDEVS debugger. The SC model of the NetLogo debugger is shown in Fig. 10. The similarities with the model for the PythonPDEVS debugger are obvious. The model is slightly smaller, since there is no support for small steps, only big steps (since we cannot alter the non-modal



SIMULATION TIME: 135.15000343322754

BREAKPOINTS:

```
{
 "COLOR": "15.0",
 "VELOCITY-Y": "-0.21050026741168476",
 "NEW-VELOCITY-Y": "-0.21050026741168476",
 "NEW-VELOCITY-X": "0.2796584400085463",
 "VELOCITY-X": "0.2796584400085463",
 "YCOR": "0.5289727240081181",
 "MY-STATE": "bouncing",
 "XCOR": "4.135711759606863",
 "SIZE": "0.8967099343459177"
}
```

Fig. 11. Reusing the debugging UI to debug the NetLogo model.

part of NetLogo, only extend it). This is why the states relating to small steps (the child states of *executing_step*, the *checking_small_step* and the *big_step* states) are not present in this model.

By synchronizing the time and state of the simulation, we can implement realtime simulation, as well as breakpoints. Incidentally, by externalizing this synchronization, our implementation of real-time simulation performs better than the built-in one. NetLogo allows, natively, to set a *framerate* for the simulation, corresponding to the number of ticks per second the model needs to advance. The native NetLogo support for real-time simulation assumes, however, that computations are instantaneous. So, if a framerate of 30 is set, a new tick will be computed every $\frac{1}{30}$ seconds. This means that, if the time to compute a new tick is significant, the simulation will lag behind. Our implementation of realtime simulation takes into account the amount of time it took to compute a tick, resulting in a much more realistic realtime simulation.

From the instrumented SC model, we can generate the code of a debugging-enhanced NetLogo simulator. Once we have the debugging-enhanced simulator, we can couple it to a debugging user interface. We reuse the interface we used for PythonPDEVS to show the current simulation state and provide buttons with which the user can control the simulation. Fig. 11 shows the debugging user interface for NetLogo, which is paused. The four turtles (shown in Fig. 8a) are shown, and the values of *turtle[0]*'s variables is displayed, as the user is currently hovering the mouse pointer over it. The remaining operations are identical to the PythonPDEVS debugger.

6 DISCUSSION

We contribute to the ongoing effort of bringing debugging techniques to simulation systems by presenting a reusable architecture and workflow. With this workflow, it is possible to turn modelling and simulation environments for dynamic-structure formalisms (with a coded simulation kernel) into full-fledged debugging environments. In this section, we first compare our work to the state-of-the-art, and then explore possible extensions in future work.

6.1 Related Work

Techniques and tools for program code debugging have been thoroughly researched. While new contributions are continuously made to this field, they are not the focus of this paper. Zeller presents an overview of the state of the art in his book [40], and introduces what he calls "scientific debugging", which allows programmers to optimally use existing techniques and tools to find, fix, and manage defects in software. The operations that our debugging environment supports are transpositions of well-established operations from code debugging to the realm of modelling and simulation.

Recently, model debugging and simulation debugging have gained increasing interest from the research community. It is an essential part in the ongoing efforts to improve the techniques and tool support for modelling and simulation. Different authors approach the problem in various ways, however, and we give an overview here.

We believe debugging support should be provided at the most appropriate level of abstraction, and take into account formalism-specific semantics. A number of works address this issue. In [8], the authors explain how debugging support was added to the Möbius modelling and simulation framework, which provides a formalism-independent discrete-event simulator. They add support for stepping, model state modification and model state visualization to their existing kernel. The back-end simulation process is separated from the front-end visualization by a communication layer. [30] argue that debugging support has to be provided at the most appropriate level of abstraction, which depends on the particular simulation formalism used to model systems. They adapt the OpenModelica environment to add debugging support, allowing the user to find modelling errors in their object-oriented equation-based models. [14] develop a different approach, extending statistical debugging techniques traditionally used for debugging software systems, which makes them effective to debug simulation systems as well. They assume the models are developed using a programming language, however, instead of a high-level modelling language. [5] acknowledges the need for testing and debugging techniques for dynamic-structure models. He focuses on a hierarchical testing technique to discover an error in the specification, after which debugging techniques need to be employed to find the source of the error.

Closely related are debugging techniques for Model-Driven Development (MDD), and more specifically for the Unified Modelling Language (UML), a family of (executable) languages for specifying the structure and behaviour of software systems. In [20], the authors map code debugging concepts onto the Story Diagrams formalism, and build a visual debugging user interface in the Eclipse open-source development environment. Their architecture consists of a debugging client that communicates with a debugging server, controlling the execution of the interpreter. Both [25] and [21] extend the fUML—a subset of the UML for which precise semantics are defined—with debugging support.

Model Transformations (MTs) are an essential part of the Model-Driven Engineering (MDE) approach to designing and developing complex systems. Two works approach the debugging of MTs from two perspectives. In [31], the authors make the observation that MTs are inherently non-deterministic. To capture that non-determinism, they develop a domain-specific language on top of Petrinets [27] to execute and debug MTs. By representing models and rules as Petrinets models, techniques such as formal verification and step-wise execution of Petrinets can be reused. [11], on the other hand, extend the model transformation debugging capabilities of the AToMPM visual modelling environment [32] with omniscient debugging. To achieve this, they transpose "step forward" and "step back" operations from code debugging to appropriate debugging operations that allow traversal of the MT execution history in both directions.

While general-purpose modelling languages are useful for modelling a wide variety of systems, in some cases a more specialized language, specific to one domain, is more appropriate. These are called Domain-Specific Languages (DSLs). [38] first recognized the need for debugging support at the DSL level. They map DSL debugging operations to code debugging operations, performed on the generated code. [24] then transposed code debugging operations to the DSL level, without relying on generated code. [1] demonstrate how to add visual execution and debugging to a DSL whose syntax is described in a metamodel, and its semantics as a set of graph transformation rules. The Moldable Debugger [10] is a reusable framework for developing debuggers for DSLs. It allows the implementation of a set of debugging operations such as stepping, state querying and visualization at the most appropriate, domain-specific, level of abstraction with minimal effort. [6] describe a partly generic debugger that can be extended with domain-specific trace management functions. They allow the definition of a set of debugging operations that traverse, query, and manage these execution traces.

We take inspiration from these approaches to transpose code debugging operations to a set of useful debugging operations for simulation formalisms. Moreover, many of these approaches support visualization of the system execution in a notation familiar to the modeller—often reusing the language's concrete syntax in the debugging interface, potentially augmented with additional (runtime) information. This paper focuses, however, on the reusable workflow and enabling architecture for constructing debugging environments for any (in particular, dynamic-structure) modelling formalism. We build on on the techniques developed in our previous work [33, 35] to implement debugging techniques for dynamic-structure models as well. Specifically, we distinguish two roles involved in the debugging process: the simulation expert (*i.e.*, the person implementing the tools related to a modelling formalism, such as simulators, modelling and simulation environments, and debuggers) and the domain expert (*i.e.*, the person using these tools to model complex systems in a particular domains). We make the observation that to understand and debug simulation systems (in particular dynamic structure systems), a formalism-specific debugging environment might not suffice, and model-specific visualizations might be required to help the domain expert find defects in the system. Not only does the simulation expert have to be assisted as much as possible with tools, frameworks, and workflows to develop effective formalism-specific debugging interfaces, the work involved for the domain expert to instrument the model-specific visualization should be minimized.

This leads to the choice of SC for modelling all parts of the simulation system as it is an appropriate formalism to describe timed, reactive, autonomous systems. Simulation experts can focus on the essential complexity of the system: finding a set of useful debugging operations and instrumenting the simulation kernel, as well as building a reusable debugging interface. Domain experts then only need to perform extra tasks if they want additional domain-specific debugging support. The processes and architecture presented in this paper make their respective tasks repeatable and structured.

6.2 Future Work

We intentionally leave the creation of more advanced visual interfaces to visualization experts. We observe, however, that our debugging interface can easily be replaced by a more advanced one, as long as its (SC) interface stays the same. For example, the interface presented by [23] provides a complete view of the state trace, as well as a historical trace of messages sent and received. This might be more appropriate for a debugging tool, coupled with the other components of our architecture. The architecture allows the components involved in the debugging process to be replaced, enhanced, and adapted at will. Ultimately, this architecture enables the construction of a library of reusable components from which simulation and domain experts may pick and choose to build debugging environments.

We cover a set of essential debugging operations in this paper, and show how they can be implemented. More work is necessary to evaluate this set and possibly implement more advanced operations. This can be accompanied by user studies that validate the debugging interface's effectiveness at finding bugs. We focus on the technical aspect instead; our workflow and architecture serve as a basis onto which future debuggers for dynamic-structure systems can build.

The techniques described here can be transposed to other formalisms as well. The set of debugging operations will likely be different, but the presented techniques are applicable to a large family of formalisms whose semantics results in a trace that evolves over (simulated) time. This is, in general, possible, even for continuous-time formalisms [33]. Slight variations are possible, such as for variable step-size solvers: in that case, an extra phase (or "small step") can be inserted for determining the next step size. When the logic becomes too complex and needs to be debugged as well, a completely new "layer" of debugging (*e.g.*, microsteps) can be inserted to give the user finer-grained control. Additionally, formalisms that exhibit fundamentally different semantics, such as non-determinism or a-causal behaviour are interesting targets to see how the architecture, workflow and instrumentation change. Moreover, building debuggers for hybrid formalisms is ongoing work [34].

The SC models describing the behaviour of the PythonPDEVS and NetLogo simulation algorithms are manually constructed and manually augmented with debugging support. Future research can look into the possibility of automating this process. For example, the debugging *aspect* could be modelled generically and *merged* into a model of the formalism's semantics. Possibly, at a certain level of abstraction, the modal part of the simulation behaviour for different formalisms can be modelled generically as well: correctly combining that model with the simulator's non-modal part would allow us to regenerate the simulator's behaviour (and augment it with debugging support) from this generic specification.

7 CONCLUSION

This paper presents a reusable workflow and architecture for constructing debugging environments for dynamic-structure modelling and simulation formalisms. We start from DSDEVS, a formalism used to model dynamic-structure, discrete-event systems, to come up with a set of useful debugging operations. The architecture of the solution considers the model-specific visualization created by the domain expert as an integral part of the simulation system. By instrumenting the visualization with domain-specific debugging interactions, it becomes a useful asset in the debugging process. This is achieved by linking the domain-specific visualization with the generic, visual modelling and simulation interface and simulation kernel. The effort for the domain expert is minimized—new operations are added by defining user interaction in the model-specific visualization and linking it to the generic components. To demonstrate applicability, we apply our approach to PythonPDEVS, a simulator that can simulate DSDEVS models, and NetLogo, a popular multi-agent modelling tool. The NetLogo debugger reuses many aspects of the PythonPDEVS debugger, proving that the approach is reusable across dynamic-structure formalisms. Our approach has many applications (in other domains, and to other formalisms) and can be combined with more advanced state tracing and visualization techniques.

ACKNOWLEDGMENTS

This work was partly funded by the Automotive Partnership Canada in the NECSIS project, by Flanders Make vzw, and with a PhD fellowship from the Agency for Innovation by Science and Technology in Flanders. We thank Yentl Van Tendeloo for his support of and with the PythonPDEVS simulation kernel. We thank the anonymous reviewers for their insightful comments on earlier versions of this manuscript.

REFERENCES

[1] Nils Bandener, Christian Soltenborn, and Gregor Engels. 2010. Extending DMM Behavior Specifications for Visual Execution and Debugging. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers.* 357–376.

[2] F. J. Barros. 1995. Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation. In *Proceedings of the 27th Winter Simulation Conference.* 781–785.

[3] F. J. Barros. 1997. Modeling Formalisms for Dynamic Structure Systems. *ACM Trans. Model. Comput. Simul.* 7, 4 (Oct. 1997), 501–515.

[4] F. J. Barros. 1998. Abstract Simulators for the DSDE Formalism. In *Proceedings of the 30th Winter Simulation Conference (WSC '98).* IEEE Computer Society Press, Los Alamitos, CA, USA, 407–412.

[5] F. J. Barros. 1998. Hierarchical Testing of Dynamic Structure Models: A Practical Approach. *Trans. Soc. Comput. Simul. Int.* 15, 4 (Dec. 1998), 181–189.

[6] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry. 2015. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015).* ACM, New York, NY, USA, 137–148.

[7] S. Breslav, R. Goldstein, A. Tessier, and A. Khan. 2014. Towards Visualization of Simulated Occupants and Their Interactions with Buildings at Multiple Time Scales. In *Proceedings of the Symposium on Simulation for Architecture & Urban Design (SimAUD '14).* Society for Computer Simulation International, San Diego, CA, USA, Article 5, 8 pages.

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

[9] D. Çetinkaya, A. Verbraeck, and M. D. Seck. 2015. Model Continuity in Discrete Event Simulation: A Framework for Model-Driven Development of Simulation Models. *ACM Trans. Model. Comput. Simul.* 25, 3, Article 17 (April 2015), 24 pages.

[10] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz. 2015. Practical Domain-specific Debuggers Using the Moldable Debugger Framework. *Comput. Lang. Syst. Struct.* 44, PA (Dec. 2015), 89–113.

[11] J. Corley, B. P. Eddy, E. Syriani, and J. Gray. 2016. Efficient and scalable omniscient debugging for model transformations. *Software Quality Journal* (2016), 1–42.

[12] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu. 2010. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering* 15, 2 (2010), 235–265.

[13] R. Ewald and A. M. Uhrmacher. 2014. SESSL: A Domain-specific Language for Simulation Experiments. *ACM Trans. Model. Comput. Simul.* 24, 2, Article 11 (Feb. 2014), 25 pages.

[14] R. Gore, P. F. Reynolds Jr., D. Kamensky, S. Diallo, and J. Padilla. 2015. Statistical Debugging for Simulations. *ACM Trans. Model. Comput. Simul.* 25, 3, Article 16 (April 2015), 26 pages.

[15] D. Harel. 1987. Statecharts: a Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274.

[16] D. Harel and H. Kugler. 2004. *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report.* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML), 325–354.

[17] D. Harel and A. Naamad. 1996. The STATEMATE Semantics of Statecharts. *ACM Trans. Softw. Eng. Methodol.* 5, 4 (Oct. 1996), 293–333.

[18] D. Harel, A Pnueli, J. P. Schmidt, and R. Sherman. 1987. On the formal semantics of Statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science.* 54–64.

[19] G. Kistner and C. Nuernberger. 2014. Developing User Interfaces using SCXML Statecharts. In *Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML*, D. Schnelle-Walka, S Radomski, T. Lager, J. Barnett, D. Dahl, and M. Mühlhäuser (Eds.). 5–11.

[20] A. Krasnogolowy, S. Hildebrandt, and S. Wätzoldt. 2012. Flexible debugging of behavior models. In *Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT).* 331–336.

[21] Y. Laurent, R. Bendraou, and M. Gervais. 2013. Executing and Debugging UML Models: An fUML Extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13).* ACM, New York, NY, USA, 1095–1102.

[22] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. 2013. *FTG+PM: An Integrated Framework for Investigating Model Transformation Chains.* Springer Berlin Heidelberg, Berlin, Heidelberg, 182–202.

[23] M. Maleki, R. Woodbury, R. Goldstein, S. Breslav, and A. Khan. 2015. Designing DEVS Visual Interfaces for End-user Programmers. *SIMULATION* 91, 8 (Aug. 2015), 715–734.

[24] R. Mannadiar and H. Vangheluwe. 2011. Debugging in Domain-Specific Modelling. In *Software Language Engineering*, Brian Malloy, Steffen Staab, and Mark Brand (Eds.). Lecture Notes in Computer Science, Vol. 6563. Springer Berlin Heidelberg, 276–285.

[25] T. Mayerhofer. 2012. Testing and Debugging UML Models Based on fUML. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1579–1582.

[26] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer. 2014. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In *Software Language Engineering*. Lecture Notes in Computer Science, Vol. 8706. Springer International Publishing, 1–20.

[27] T. Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.

[28] S. Mustafiz, J. Denil, L. Lúcio, and H. Vangheluwe. 2012. The FTG+PM Framework for Multi-paradigm Modelling: An Automotive Case Study. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling (MPM '12)*. ACM, New York, NY, USA, 13–18.

[29] J. J. Nutaro. 2016. adevs. http://www.ornl.gov/~1qn/adevs/.

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

[31] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer. 2010. *Models in Software Engineering: Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Catch Me If You Can – Debugging Support for Model Transformations, 5–20.

[32] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. AToMPM: A Web-based Modeling Environment. In *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition @ MODELS'13*, Vol. 1115. CEUR, 21–25.

[33] S. Van Mierlo. 2015. Explicitly Modelling Model Debugging Environments. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*. 24–29.

[34] Simon Van Mierlo, Cláudio Gomes, and Hans Vangheluwe. 2017. Explicit Modelling and Synthesis of Debuggers for Hybrid Simulation Languages. In *Proceedings of the Symposium on Theory of Modeling & Simulation (TMS/DEVS '17)*. Society for Computer Simulation International, San Diego, CA, USA, Article 4, 12 pages. http://dl.acm.org/citation.cfm?id=3108905.3108909

[35] S. Van Mierlo, Y. Van Tendeloo, and H. Vangheluwe. 2016. Debugging Parallel DEVS. *SIMULATION* (August 2016). arXiv:http://sim.sagepub.com/content/early/2016/07/29/0037549716658360.full.pdf+html

[36] Y. Van Tendeloo and H. Vangheluwe. 2016. An Overview of PythonPDEVS. In *JDF 2016*. 59–66.

[37] H. Vangheluwe, 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 (DEVS '14)*. Society for Computer Simulation International, 379–386.

[38] H. Wu, J. Gray, and M. Mernik. 2008. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience* 38, 10 (2008), 1073–1103.

[39] B. P. Zeigler. 1984. *Theory of Modelling and Simulation*. Krieger Publishing Co., Inc., Melbourne, FL, USA.

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

Month YearMonth Year

Received Month Year