A Generalized Stepping Semantics for Model Debugging

Simon Van Mierlo
University of Antwerp
Flanders Make vzw
simon.vanmierlo@uantwerpen.be

Yentl Van Tendeloo
University of Antwerp
yentl.vantendeloo@uantwerpen.be

Hans Vangheluwe
University of Antwerp
McGill University
Flanders Make vzw
hans.vangheluwe@uantwerpen.be

ABSTRACT

Stepping is arguably one of the most important operations for model execution, and model debugging specifically. Each formalism, however, has a different set of supported stepping operations (e.g., big step, combo step, and small step). Furthermore, many tools provide a different terminology for these different steps (e.g., small step, micro step, and epsilon step). As such, the exact semantics of stepping is unknown to the modellers, and might even differ between different tools. Additionally, tool developers have no framework to check whether they have implemented all useful stepping operations for their formalism. In this paper, we define a hierarchical terminology of stepping operations to provide a generalized vocabulary for both users and developers of debuggers. We distinguish four "levels" of stepping operations, related to the available knowledge in the execution trace. From a high to low level of abstraction, we term them simulation stepping, black-box stepping, white-box stepping, and implementation stepping. After introducing the framework and terminology, we apply it to a number of already existing debuggers for a large variety of formalisms.

KEYWORDS

Debugging, Stepping, Simulation

ACM Reference Format:
Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. 2018. A Generalized Stepping Semantics for Model Debugging. In *Proceedings of Second International Workshop on Debugging in Model-Driven Engineering (MDEbug'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Stepping through the execution of a program is arguably one of the most essential debugging operations a developer has available. Instead of observing the effects caused by the full execution of the program, stepping operations provide a way of controlling the execution manually. This serves different purposes: developers can get a view into the evolution of the execution state step-by-step, or even into how the evolution of the state is implemented in the simulator/executor. Stepping is thus a way of *understanding* the execution of a program, by exposing certain details of the execution algorithm and state information to the user.

Given its importance in the programming domain, stepping has frequently been ported to the (executable) model debugging domain as well. In this case, modellers can freely step through the execution of an executable model, as was the case with programs.

MDEbug'18, October 2018,
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Many such executable modelling languages, or formalisms, exist, each with their own set of tools to design, simulate, execute, and debug. When it comes to stepping, formalisms (and their related tools) may offer a set of stepping operations at different level of detail. For example, in program code debugging, a *step into* offers a view into the function called at the current line, while a *step over* jumps over the line and shows the result of calling the function at the current line. In other formalisms, the distinction is often made between a *big step* and a *small step*. For example, in the DEVS debugger presented in [14], a big step jumps from the current state to the next (stable) state. A *small step*, on the other hand, shows the internal simulator steps needed to arrive at the next stable state, thereby exposing implementation details. For Model Transformations (MTs), Tichy et al. [11] transpose *step into, step over, step return, run until* from traditional code debuggers, while Sun and Grey [10] define a step operation that steps through all the operations executed during pattern matching.

In contrast to most operational programming languages, which are line-based, models have a widely varying structure. Some models are similarly line-based (e.g., an explicitly modelled action language), whereas other are graph based (e.g., Petrinets) or a combination of multiple (e.g., Statecharts). The vocabularies used therefore differ from formalism to formalism, and even from debugger to debugger for the same formalism. We consider three potential issues due to this disparity in terminology. First, it makes it difficult for users of a tool to intuitively grasp the semantics of a given stepping operation in a tool for a given formalism. Second, tool developers cannot easily assess the completeness of the provided stepping operations, with respect to the level of detail that is being offered by these stepping operations. Third, developers of a new language would benefit from a unified terminology to use as a prescription for the definition of the language's semantics specification. This would further benefit the area of language design, and, more specifically, the modular design of language fragments.

In this paper, we define a hierarchical terminology of stepping semantics to provide a generalized vocabulary for both users and developers of debuggers. We distinguish four "levels" of stepping operations, related to the available knowledge in the execution trace. From a high to low level of abstraction, we term them simulation stepping, black-box stepping, white-box stepping, and implementation stepping. This leads to a framework, allowing to reason about debuggers for any type of formalism.

Structure. Section 2 introduces the language DEVS and its debugger, to serve as a running example. Section 3 explains our approach, which considers stepping at different levels based on the information exposed to the user. Section 4 applies our approach on a number of representative formalisms by categorizing their stepping

operations using our vocabulary. Section 5 discusses the relevance of our framework to several areas of research. Section 6 discusses related work, and Section 7 concludes.

2 RUNNING EXAMPLE

As a running example for this paper, we present the Discrete Event System Specification (DEVS) formalism [20]. DEVS is a discrete-event formalism with well-defined syntax and semantics. A DEVS model accepts a trace of input events, and produces a trace of output events as the result of simulation. During simulation, the internal state of the DEVS components evolves over (simulated) time. Due to the precisely defined semantics, as well as the clear separation between input event trace, output event trace, and state trace, DEVS is an ideal candidate to demonstrate our approach.

2.1 The DEVS Formalism

DEVS, and in particular Parallel DEVS [2], is used to model the behaviour of discrete event systems. Its basic building blocks are *atomic DEVS* models, which are structures

$$M =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta, q_{init} >$$

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* $\delta_{int} : S \to S$ defines the next sequential state, depending on the current state. The *output function* $\lambda : S \to Y^b$ defines the bag of output events to be raised for a given sequential state, upon triggering the internal transition function. The *external transition function* $\delta_{ext} : Q \times X^b \to S$ 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* $ta : S \to \mathbb{R}^+_{0,+\infty}$ defines the simulation time the system remains in the current state before triggering its *internal transition function*. The *confluent transition function* $\delta_{conf} : S \times X^b \to S$ is called if both an internal and external transition collide at the same simulation time, replacing both functions. The *initial total state* $q_{init} = (s_0, e_0) \in Q$ defines the initial state of the model (s_0), and how long the model has already been in this state (e_0) [17].

A network of atomic DEVS models is called a *coupled DEVS* model. The output of one atomic DEVS model can be connected to the input of other atomic DEVS models using "channels". For each channel, a *transfer function* can be defined to translate output to input events. Parallel DEVS is closed under coupling, which means that coupled models can be nested to arbitrary depth.

An abstract simulator for Parallel DEVS, which computes the next state of the system (a "step") until its end condition is satisfied, is described in [3]. The algorithm can be summarized as follows.

(1) Compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components).
(2) Execute the output function for each imminent component, causing events to be generated on the output ports.
(3) Route events from output ports to input ports, translating them in the process by executing the transfer functions.
(4) Determine the type of transition to execute for the atomic DEVS model, depending on it being imminent and/or receiving input.
(5) Execute, in parallel, all enabled internal, external, and confluent transition functions.
(6) Compute, for each atomic DEVS model, the time of its next internal transition (specified by its time advance function).

We previously presented a debugger for Parallel DEVS [14], which supports three types of steps: (1) a *big step* executes one iteration of the simulation algorithm, (2) a *small step* executes one phase of an iteration, (3) and a *step back* steps back to the previous state in the trace [16]. We will start from these definitions of "steps" to generalize to the vocabulary and framework presented in the remainder of the paper.

2.2 An Example Model with a Failure

To demonstrate the different levels of granularity at which bugs can be found, we consider an example system: a traffic light. To keep it simple, we consider only two requirements: (1) the traffic light should continuously cycle between the three lights in this order: green, yellow, red, green, yellow, red, ..., (2) the traffic light should stay green for 57 seconds, stay yellow for 3 seconds, and stay red for 60 seconds.

Our system consists of one atomic model, defined in Equation 1.

$$Light = \langle X_l, Y_l, S_l, \delta_{int,l}, \delta_{ext,l}, \delta_{conf,l}, \lambda_l, ta_l, q_{init,l} \rangle \quad (1)$$

$$\begin{aligned}
X_l &= \{\} \\
Y_l &= \{display_red, display_green, display_yellow\} \\
S_l &= \{red, green, yellow\} \\
\delta_{int,\,l} &= \{green \to yellow, yellow \to red, red \to green\} \\
\delta_{ext,\,l} &= \{\} \\
\delta_{conf,\,l} &= \{\} \\
\lambda_l &= \{green \to [display_green], \\
&\quad\ yellow \to [display_yellow], \\
&\quad\ red \to [display_red]\} \\
ta_l &= \{green \to 57, red \to 60, yellow \to 3\} \\
q_{init,\,l} &= \{red, 60\}
\end{aligned}$$

We assume that a test suite is also constructed that checks for the satisfaction of both requirements. When executing the test suite, a failure is noticed, thereby meaning that at least one of the requirements is not satisfied. This will form the starting point of our debugging efforts.

3 APPROACH

We now present our hierarchical stepping operations and the proposed unified vocabulary, applied to the running example. It is based on an analysis of the different levels of traces in a simulation or execution run, where each level increases the amount of detail offered to the user of the debugger. As presented in [13], a failure occurs when a requirement of the system is not satisfied by the design of the system (which, we assume, is described in a model). Once that failure is observed, the source of the failure (a defect) can be found by, amongst others, observing the behaviour of the system in the form of *traces*.

Depending on the debugger, different levels of detail can be offered. We argue that there are four such levels, and we claim that they are hierarchical, as shown in Figure 1. Intuitively, when switching to a lower level, more information is exposed to the user. The

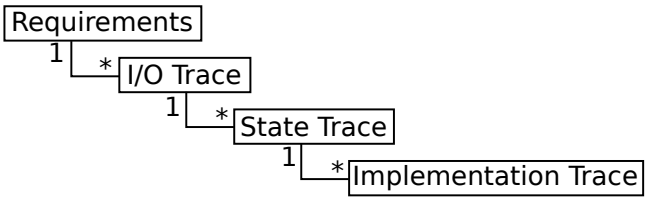

Figure 1: The different levels at which a defect can be observed by the user.

first level is the requirements level, which determines which requirement is not satisfied. An arbitrary number of input/output traces exist that can lead to the requirement failing; at the input/output (second) level, the externally observable behaviour of the model that lead to the requirement not being satisfied is exposed. Similar observations hold for lower levels of abstraction. At the state trace (third) level, which shows the trace of internal (globally consistent) states that lead to the faulty input/output trace. At the implementation (fourth) level, the outcome of the intermediate (i.e., non-stable) simulator states that lead to the faulty state trace is exposed. All four levels are related to the execution trace, presenting to the user different levels of detail.

3.1 Simulation Step

At the top-most level (the requirement level), the corresponding step is the *simulation step*. This step simulates the whole system and reports back the result: either the requirement is satisfied or it is not. As a debugging operation, it merely signals which requirement was not satisfied. While it is not a traditional debugging operation, as it is exactly the same as ordinary simulation, it is important to include it here at the root of our hierarchy. A single simulation step takes the model to its final state, defined using a termination condition (e.g., after 1000 seconds of simulated time).

Running Example. In the context of our running example, simulation stepping for the two test cases made for validating the requirements signals that requirement 2 is violated: the traffic light does not display the correct colour for a long enough period of time. At this level, we have no further insights in the model; we will have to go to a lower level of abstraction.

3.2 Black-Box Step

One level down from the simulation step is the *black-box step*. Since generally the validation of the requirements is based on some type of query over the input/output trace, this step makes use of the input/output trace generated during the simulation in which the requirement was not satisfied. It provides a view on the externally observable behaviour: a trace of the input and output of the model under study. These steps are "black-box", in the sense that they consider the model as a black box that accepts input and produces output. Our framework is agnostic to the type of trace: it could be discrete (a finite number of events in any finite time interval) or continuous (an infinite number of events in any finite time interval). In our example, we focus on discrete traces.

A single black-box step takes the model from one externally observable event to another. This means that we extend the set

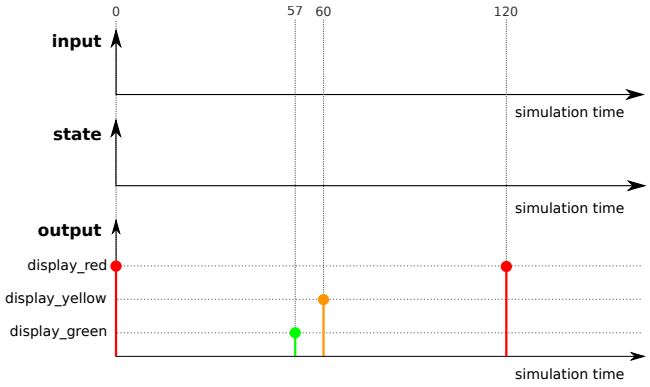

Figure 2: Stepping through an input/output trace.

of reachable configurations to those points in time where an input or output event occurs. All intermediate configurations are unreachable and might not even exist.

Running Example. In the context of our running example, black-box stepping would give us the trace shown in Figure 2. As can be seen, this simple model has no input events, but has several output events. From these outputs, it is clear that the requirements are indeed not satisfied, as, for example, the time between the *display_green* and *display_yellow* is not equal to 57. To attempt to find the cause, we will have to observe the state of the model.

3.3 White-Box Step

One level down from the black-box step is the *white-box step*. This step makes use of all the knowledge from the black-box step, but additionally includes information on the internal state of the model. The internal state can evolve autonomously (without an input event driving the state change) and does not necessarily lead to an output event. But the internal state trace leading up to behaviour visible in the input/output trace is important towards finding the source of a failure. A single white-box step takes the model from one globally consistent state to another. This means that we further extend the set of reachable configuration with the point in time where a state change occurs, even if the state changes to itself. All intermediate configurations are unreachable and might not even exist.

Running Example. In the context of our running example, white-box stepping gives us the trace shown in Figure 3. There are no additional points in simulated time that we can step to, but we gain insight into the state evolution of the model. In this case, we see that the state and events are desynchronized: the event *display_green* is shown right before the state changes to *yellow*. This gives an additional direction for our search, but it does not pinpoint the error in the model yet. For that, we will have to dig deeper.

3.4 Implementation Step

One level down from the white-box step is the *implementation step*. This step makes use of all the knowledge from the white-box step, but additionally includes information about the execution order within the simulator. State changes are always the result of some (complex) processing steps within the simulator. These steps expose implementation details of the simulator used, which are usually

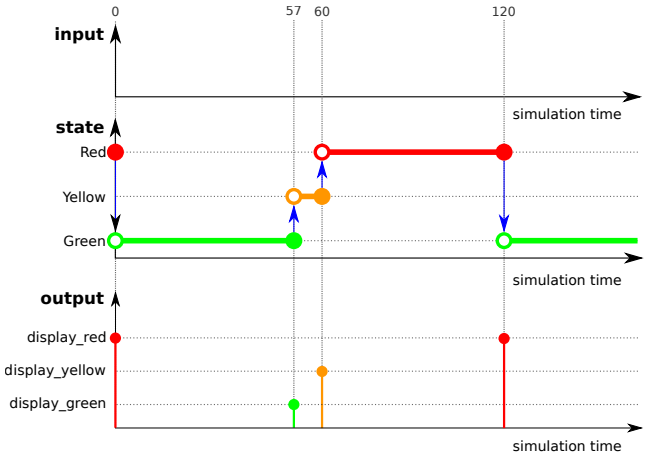

Figure 3: Stepping through a state trace.

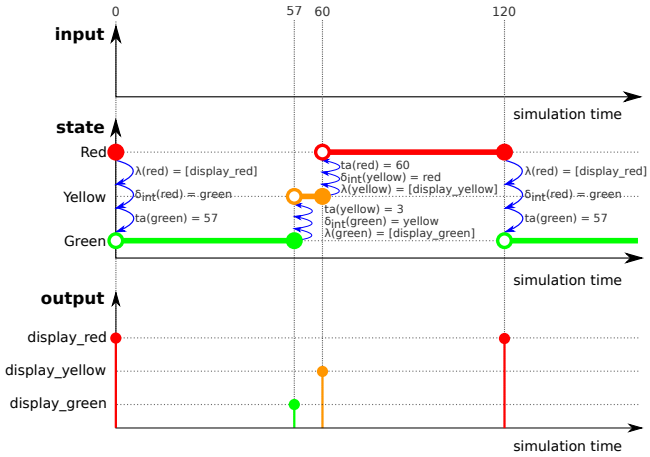

Figure 4: Stepping through an implementation trace.

shielded: models specify *what* a system does, not *how* it does it. As a result, an implementation step can expose non-reachable (globally inconsistent) states in a set of "phases" the simulator goes through to compute the next reachable state. For example, in the context of DEVS, implementation steps are at the level of computing output functions and performing transitions. One implementation step might thus cause the execution of the output functions, and the next implementation step might compute the internal transitions.

Running Example. In the context of our running example, implementation stepping gives us the trace shown in Figure 4. As can be seen, the information is exactly the same as before, but now the "instantaneous" jumps between stable states get broken down in the various implementation steps that were necessary. By looking at the sequence of these calls, as well as the arguments passed, we notice that for each transition, the output function is invoked before the transition itself. Given this knowledge, which is part of the DEVS execution semantics, our defect becomes clear: when transitioning to the green state, the output function is invoked on the red state (with $\lambda(red) = show\_red$). This is an often made mistake when modelling with DEVS; it can easily be fixed by updating the output function to the one shown in Equation 2.

$$\begin{aligned}\lambda = \{&red \rightarrow [display\_green],\\ &green \rightarrow [display\_yellow],\\ &yellow \rightarrow [display\_red]\}\end{aligned} \tag{2}$$

Rerunning the test suite shows that the model satisfies both requirements now.

4 UNIFICATION OF EXISTING FORMALISMS

To demonstrate the usefulness of our vocabulary as a framework for classifying debugging operations, we apply it to a number of representative formalisms. The goal is to show that we can apply the framework to formalisms with widely differing semantics. Each formalism is presented in turn, with some motivation for its inclusion, and then its stepping semantics is discussed. The "simulation step" is the same for all formalisms (i.e., run the simulation from beginning to end) and is therefore not discussed.

Causal Block Diagrams (CBD) [1]. Allows to model mathematical equations using a dataflow notation similar to Synchronous Data Flow. It is implemented in tools such as Simulink by the Mathworks. A model in CBDs consists of blocks representing mathematical operations such as addition, division, etc. These blocks are connected with signals: each block has a number of input and output signals. These operations can be time-dependent: for example, a block can integrate its incoming signal value over time. Algebraic loops can be created if the input signal of a block depends on its own output value; this requires to feed all strongly connected components (representing a (linear) set of equations) to a solver. The simulation algorithm for CBDs updates the output signal values of each block each time step. To do this, each iteration a dependency analysis is made to discover the strongly connected components, and to create a schedule that decides the order in which the signal values will be computed. Then, the simulation algorithm loops over all blocks, computing their new output signal value. This effectively discretizes the behaviour. We map our steps onto CBDs as follows:

- *Black-Box Step*: step to the next sampled input/output signal value.
- *White-Box Step*: step over one iteration of the simulation algorithm, showing the new values of all signals. Due to the discretization and the fact that each block is recomputed in every iteration, this is exactly equal to black-box stepping.
- *Implementation Step*: step through the inner loop that computes the new values of the components in order, or optionally solve an algebraic loop.

Statecharts (SCs) [6]. Offers abstractions to model the timed, reactive, autonomous behaviour of a system. Its main abstractions are states, which model the configuration the system is in, and transitions between these states, which model the dynamics of the system and can be triggered by (input) events or by a timeout, optionally accompanied by a condition on the state variables of the system. States can be composed orthogonally (where each orthogonal region has an active state) or hierarchically in composite states. Similar to DEVS, it is a discrete-event formalism: state changes occur at discrete points in (simulated) time, and are triggered by

discrete events. Nonetheless, DEVS and Statecharts have different notions of modularity and hierarchy, making them sufficiently different. We map our steps onto SCs as follows:

- *Black-Box Step*: step to the next event in the I/O trace.
- *White-Box Step*: step to the next globally consistent state, which executes applicable transitions, possibly concurrently.
- *Implementation Step*: step through the inner loop that detects enabled transitions, selects the applicable ones, and atomically executes a single one even if multiple are to be executed concurrently.

Action Language (AL) [15]. Is a line-based formalism and is comparable to procedural programming languages. Nonetheless, its syntax and semantics are explicitly modelled using a metamodel and model transformations, respectively. Its syntax consists of the usual imperative programming constructs (variable declaration and assignment, while- and for-loops, function definitions), which are executed sequentially in a line-by-line fashion. It is not timed, although a *sleep* function is provided. AL provides facilities for external input and output, similar to *stdin* and *stdout*, through the use of special *input* and *output* functions. These "events" are timestamped by their wall-clock time, relative to the start time of the program's execution. We map our steps onto AL as follows:

- *Black-Box Step*: step to the next call to the input/output functions in the program.
- *White-Box Step*: step through the execution of the program line-by-line.
- *Implementation Step*: show how statements are executed by the runtime. This reveals implementation details, such as program counter and memory locations, as well as how the instructions of the target platform are executed one-by-one to execute the phases of the (currently executing) statement.

Petri Nets (PN) [8]. Allows to model non-deterministic systems. Its main abstractions are *places* (containing *tokens*) and *transitions*, which have a number of input places and a number of output places. A transition is enabled if there are at least as many tokens in its input places as the weight of the link. An enabled transition can be triggered, thereby removing tokens from its input places and adding tokens to its output places. Petri Nets are mainly used to evaluate safety properties of systems, such as making sure they cannot reach an unsafe/invalid configurations (corresponding to a marking, which lists the number of tokens for each place). The analysis of a PN results in a *reachability graph*, listing all reachable markings, and the transitions that lead from one marking to another. Stepping through an analysis is possible, but here we restrict ourselves to the *simulation* of a *PN* model: instead of analyzing the full model, we simulate one possible trace through possible reachable markings. We map our steps onto PN as follows:

- *Black-Box Step*: there is no input/output, except for whether or not the system has reached a safe state. So the black-box step is equivalent to the simulation step.
- *White-Box Step*: execute one enabled transition, out of potentially many concurrently enabled transitions.
- *Implementation Step*: show how the set of enabled transitions is computed; then, show how one of them is chosen to trigger. Even more detailed, a step could allow manual intervention, by allowing to choose the enabled transition to be triggered.

5 DISCUSSION

This section discusses the relevance of this contribution to several areas where it can be useful: analyzing the completeness of tools, modularly building hybrid languages, and the development of domain-specific languages.

5.1 Tool Completeness

For tool developers, and in particular developers of model debuggers, there currently is no way of analyzing whether the set of operations the tool provides is *complete*, in the sense that all useful operations are implemented. Our vocabulary can act as a framework for these developers, with which they can decide at which level debugging support is offered. While from the user's point of view it is ideal to have debugging support at all the levels we defined, this might not be feasible or even desirable for practical reasons (e.g., too low level of abstraction), as well as for the protection of intellectual property (e.g., do not reveal simulation algorithm). But even then, our vocabulary allows to reason about such concerns and make an informed decision at which level(s) stepping operations are provided.

5.2 Hybrid Languages

An interesting application of our vocabulary is the design of hybrid languages. Such languages are often the result of composing both the syntax and semantics of two existing languages. These compositions are often either parallel (i.e., two simulation algorithms are interleaved), or hierarchical (i.e., the simulation algorithm of the "child" formalism is called at certain points in the simulation algorithm of the "parent" formalism). Languages such as BCOoL [19] offer a way of defining such interleavings. Our vocabulary can assist such methods by explicitly defining at which stepping level the formalisms are combined. This possibly leads to an extension of our framework, since there will be two orthogonal dimensions of input/output traces, state traces, and implementation traces.

5.3 Domain-Specific Languages

Defining debuggers for new domain-specific languages is now mostly done ad-hoc, and is based on the knowledge of the language designer. For a user of these languages, it is not necessarily clear from the onset what each stepping operation performs if no common vocabulary is used. Our vocabulary can therefore be used as an "interface" between the language designer and the language user to avoid misunderstandings and improve the usability of the language and its tools. For the language designer, it becomes possible to approach the design of stepping operations from the bottom up, instead of top-down as we presented in this paper. Since the language designer has control over the simulation algorithm, it might be easier to view the implementation trace as the source of all information. Subsequent higher levels of traces (the state trace, input/output trace, and simulation trace) can then be obtained by successively filtering the traces. This can aid the language developer to more intuitively define stepping semantics at each level.

6 RELATED WORK

The step operation has its roots in code debugging [21]. Procedural languages traditionally provide a number of stepping operations:

step over (the currently executing line), *step into* (the function called at the current line), and *step return* (the currently executing function). These functions all assume a view of the code, and as such would be classified as *white-box steps* in our vocabulary. Certain debuggers also offer stepping support at the assembly instruction level, which in our vocabulary would be classified as an *implementation step*.

Most model debuggers offer a stepping operation. For example, Vangheluwe *et al.* [18] define a *big step* and *small step* operation for the *Causal Block Diagrams* formalism, based on the simulation algorithm presented earlier in this paper. They do not consider input/output steps, however. Van Mierlo defines debugging support for a large set of formalisms [12]. For each formalism, he explicitly models the simulation algorithm and subsequently instruments this with debugging support, resulting in stepping operations at different level of detail. However, the vocabulary ranges from *small step*, *combo step*, *big step*, *time step*, and is not consistent between formalisms. In their recent work, Jukss et al. [7] build a debugger for model transformations, and observe the different levels at which a transformation can be debugged (schedule, rule, matching algorithm), and the different levels of steps this leads to. Their debugging operations all reside at the *white-box* level. Similarly, Corley *et al.* [4] look at debugging support for model transformations, thereby defining (white-box) stepping operations that can go forward and backwards (implementing *omniscient* debugging) in the state trace.

In other work, the notion of a simulator "step" is important as well, either to classify language variants based on how they implement a hierarchy of steps, or to analyze simulation algorithms and possibly combine them at different levels of steps. Esmaeilsabzali *et al.* [5] analyze the semantics of big-step modelling languages, such as Statecharts, and discern four levels of steps: *big step* (which processes one *event*), *combo step* (which executes one transition for each *concurrent region*) and *small step* (which executes one transition). They then define a family of possible big-step modelling languages semantics, based on how events are managed, whether transitions are executed in parallel, etc. While they introduced the stepping semantics of big-step modelling languages, it is unclear how their terminology can be intuitively transposed to other languages (e.g., action language). Mustafiz *et al.* [9] define a method for merging languages at the semantic level, by building a *canonical form* of the simulators that distinguishes between *macro-* and *micro-steps*. Since the canonical form is known, they can be *merged* at the level of these steps. In our vocabulary, macro-steps are *white-box* steps, while micro-steps are *implementation* steps.

7 CONCLUSION

This paper defines a vocabulary to classify stepping operations at different levels of granularity. We distinguish between *simulation*, *black-box*, *white-box*, and *implementation* steps. These correspond to the level at which information is presented to the user: respectively at the *requirement*, *input/output trace*, *state trace*, and *implementation trace* level. We apply this vocabulary on a running example in the *DEVS* formalism and then generalize to a set of semantically varying formalisms, demonstrating its wide applicability. In future work, this vocabulary can be refined and used as a basis for building new debugging techniques and tools for (domain-specific) formalisms, and used as a basis for other, related research areas such as modular language design.

ACKNOWLEDGMENTS

This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO). This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

[1] François E. Cellier. 1991. *Continuous System Modeling* (first ed.). Springer-Verlag.
[2] Alex Chung Hen Chow and Bernard P. Zeigler. 1994. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 1994 Winter Simulation Multiconference*. 716–722.
[3] Alex Chung Hen Chow, Bernard P. Zeigler, and Doo Hwan Kim. 1994. Abstract simulator for the parallel DEVS formalism. In *AI, Simulation, and Planning in High Autonomy Systems*. 157–163.
[4] 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.
[5] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. 2010. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering* 15, 2 (2010), 235–265. https://doi.org/10.1007/s00766-010-0102-z
[6] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274.
[7] Maris Jukšs, Clark Verbrugge, and Hans Vangheluwe. 2017. Transformations Debugging Transformations. In *Proceedings of MODELS 2017 Satellite Events*.
[8] Tadao Murata. 1989. Petri Nets: Properties, analysis and applications. In *Proceedings of the IEEE*. 541 – 580.
[9] Sadaf Mustafiz, Cláudio Gomes, Bruno Barroca, and Hans 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)*. San Diego, CA, USA, 29:1—-29:8.
[10] Yu Sun and Jeff Gray. 2013. End-User Support for Debugging Demonstration-Based Model Transformation Execution. In *Modelling Foundations and Applications*, Pieter Van Gorp, Tom Ritter, and LouisM. Rose (Eds.). Lecture Notes in Computer Science, Vol. 7949. Springer Berlin Heidelberg, 86–100. https://doi.org/10.1007/978-3-642-39013-5_7
[11] Matthias Tichy, Luis Beaucamp, and Stefan Kögel. 2017. Towards Debugging the Matching of Henshin Model Transformations Rules. In *Proceedings of MODELS 2017 Satellite Events*.
[12] Simon Van Mierlo. 2018. *A Multi-Paradigm Modelling Approach for Engineering Model Debugging Environments*. Ph.D. Dissertation. University of Antwerp.
[13] Simon Van Mierlo, Erwan Bousse, Hans Vangheluwe, Manuel Wimmer, Martin Gogolla, Matthias Tichy, and Arnaud Blouin. 2017. Report on the 1st International Workshop on Debugging in Model-Driven Engineering (MDEbug'17). In *Proceedings of MODELS 2017 Satellite Events*, Vol. 2019. CEUR-WS.
[14] Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. 2017. Debugging Parallel DEVS. *SIMULATION* 93, 4 (2017), 285–306.
[15] Yentl Van Tendeloo. 2015. Foundations of a Multi-Paradigm Modelling Tool. In *MoDELS ACM Student Research Competition*. 52–57.
[16] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2017. Time- and Space-Conscious Omniscient Debugging of Parallel DEVS. In *Proceedings of the 2017 Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS '17, part of the Spring Simulation Multi-Conference)*. Society for Computer Simulation International, 1001 – 1012.
[17] Yentl Van Tendeloo and Hans Vangheluwe. 2018. Extending the DEVS Formalism with Initialization Information. *ArXiv e-prints* (2018). arXiv:1802.04527
[18] Hans Vangheluwe, Daniel Riegelhaupt, Sadaf Mustafiz, Joachim Denil, and Simon Van Mierlo. 2014. Explicit Modelling of a CBD Experimentation Environment. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS '14, part of the Spring Simulation Multi-Conference)*. Society for Computer Simulation International, 379–386.
[19] M. Vara Larsen, J. DeAntoni, B. Combemale, and F Mallet. 2015. A Behavioral Coordination Operator Language (BCOoL). In *Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*.
[20] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation* (second ed.). Academic Press.
[21] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.