A Statecharts Interpreter and Compiler With Semantic Variability

Joeri Exelmans
Joeri.Exelmans@uantwerpen.be
Universiteit Antwerpen
Flanders Make
Antwerpen, Belgium

Simon Van Mierlo
Simon.VanMierlo@epc.be
EP&C
Turnhout, Belgium

Hans Vangheluwe
Hans.Vangheluwe@uantwerpen.be
Universiteit Antwerpen
Flanders Make
Antwerpen, Belgium

ABSTRACT

The Statecharts formalism has been proven useful for the modeling of complex reactive systems. However, the formalism has no single, precise semantics. Different Statechart tools and standards have made their own (incompatible) semantic decisions. Rather than proposing a new semantics for Statecharts, we present an implementation of a Statecharts interpreter and compiler with configurable semantic variability. Its semantic feature model is based on the existing framework of Big-Step Modeling Languages (BSMLs), and consists of six (mostly) orthogonal dimensions of semantic options, captured in a feature model. We provide an in-depth description of our implementation of the "Priority" semantic dimension, which differs slightly from what BSML prescribes. The feature model also consists of a small set of constraints that prune non-uniquely behaving semantic variants. These constraints were discovered with a novel technique, that we also explain.

CCS CONCEPTS

• **Software and its engineering → Visual languages**; *Compilers.*

KEYWORDS

semantics, variability, statecharts

ACM Reference Format:
Joeri Exelmans, Simon Van Mierlo, and Hans Vangheluwe. 2022. A Statecharts Interpreter and Compiler With Semantic Variability. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3550356.3561569

1 INTRODUCTION

Statecharts are a language for modeling complex, reactive, autonomous/timed and concurrent systems, at a discrete-event level of abstraction, using an intuitive visual notation. When originally introduced by Harel in 1987 [8], no complete or formal semantics were given. In literature, many proposals were made for a formal semantics, each with their strengths and weaknesses. As a result, no single semantics has "won". Meanwhile, implementations (e.g., Statemate, Rhapsody, ROOM, YAKINDU) have made their own, often incompatible, semantic choices. A number of standards have also emerged (e.g., UML State Diagrams, SCXML).

An overview of Statecharts variants was first made by von der Beeck in [13], listing 16 semantic and 10 syntactic variation points. The author also suggested a new variant (without implementation) with the most "desirable" choice for each of the variation points. The variation points were not orthogonal: for instance, "determinism" and "priorities" are variation points, while the former is a language property, influenced by the latter, which is a semantic feature.

More recently, Esmaeilsabzali et al. presented 8 dimensions of semantic features of Statechart-like languages called "Big-Step Modeling Languages" (BSMLs) in [5]. These dimensions are (mostly) orthogonal, meaning that the choices made for each of them minimally impact each other. In this work, no new "optimal" variant is introduced — the authors instead reason that language features should be chosen that are "most fit" for the problem at hand, on a per-model basis. The work did not include an implementation, and the semantic options were explained in natural language, complemented with examples.

1.1 Contribution

In this paper, we present an implementation of a subset of the BSML framework in the form of a Statechart interpreter + compiler with semantic variability, as part of the SCCD project (StateCharts + Class Diagrams, a hybrid language of dynamically created and destroyed Statechart instances, communicating asynchronously [12]). Motivations are:

- Allowing the modeler to select semantic features on a per-model basis.
- Claiming compatibility with existing tools (by mimicking their semantics).
- Seeing whether the BSML framework is indeed implementable.

While the chosen subset of BSML features has been mostly implemented as-is, we motivate a slight deviation in the Priority dimension, calculating priorities and detecting non-determinism statically.

Finally, we present a novel approach for discovering additional constraints on our semantic feature model to prune "duplicated" variants, i.e., variants that behave exactly the same as other variants.

The remainder of this paper is structured as follows: Section 2 presents the semantic dimensions and their options, as supported in our implementation. It also motivates some deviations from the BSML framework, and explains which BSML features were not implemented. Section 3 gives a brief overview of our implementation. Section 4 explains how we identified the subset of semantically unique variants among those executable by our implementation. Section 5 discusses related work, and Section 6 concludes this paper.

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9467-3/22/10...$15.00
https://doi.org/10.1145/3550356.3561569

2 SEMANTIC FEATURES

In this section, we'll first explain the semantic variability options supported by our implementation. Then, we'll explain some small deviations that we made from BSML. Finally, we'll mention (and partially motivate) the features from BSML that have not been implemented.

2.1 BSML Basics

Our feature model is largely a subset of the BSML framework [4, 5]. The execution of a BSML happens as a sequence of *big-steps*. A big-step is a reaction to a set of (simultaneous) input events. Big-steps have a run-to-completion (RTC) characteristic: another input cannot be received until the current reaction has completed. This reaction consists of (1) possibly, a change of the internal configuration (by executing transitions), and (2) a (possibly empty) set of output events (generated by the executed transitions). A big-step consists of a sequence of *combo-steps*, which in turn consists of a set of *small-steps*. Combo-steps have no meaning on their own — they are given meaning by certain semantic options. For now, one may assume a small-step consists of the execution of one transition.

The semantic options described by the BSML framework only impact the execution of a *big-step*. BSML does not describe Statechart features such as queuing of input events, history or timed transitions.

2.2 Semantic Dimensions

We'll now explain the semantic dimensions and their respective options, supported by our implementation. The full feature model is shown in Fig. 1. We follow the convention from [5], using Small Caps for semantic options, and Sans-serif for semantic dimensions. The dimensions are:

2.2.1 *Big-/Combo-Step Maximality.*

constrains the transitions that can be taken in a big-step, or combo-step, respectively. Options are Take Many (no constraints — a big-step ends when no more transitions are enabled), Take One (only allow transitions that are orthogonal to each other to be taken within the same big step), and Syntactic (same as Take One, but transitions to states syntactically marked as *unstable* do not "count", and allow for subsequent transitions within the same step).

Note that for any step in any Statechart model, we can order the maximality-options by the number of transitions included in that step: Take Many $\geq$ Syntactic $\geq$ Take One.

Note that the maximality of a combo-step by itself has no meaning — instead, combo-steps are given meaning when other options from other semantic dimensions refer to them, as we will now see.

2.2.2 *Input/Internal Event Lifeline.*

specifies at what point, and for how long, during a big-step, input and internal events become and remain active (such that they can enable transitions). An input event can either be Present in First Small-Step, Present in First Combo-Step or Present in Whole (present in the entire big-step). Similarly, an internal event can be Present in Next Small-Step (to trigger one immediately subsequent transition), Present in Next Combo-Step or Present in Remainder (the event remains active throughout the remainder of the big-step).

By using Present in First Combo-Step and Present in Next Combo-Step, reactions to input events can be separated from reactions to internal events. This is probably the most common usage of combo-steps.

2.2.3 *Memory Protocol.*

controls the "versions" of the values of variables that are read, for evaluating guard conditions and variable assignments. Options are Big-Step, Combo-Step and Small-Step, meaning that values are read as they were at the beginning of the current big-step, combo-step or small step, respectively.

The Big-Step and Combo-Step options can be used to hide the effects of transitions in orthogonal regions from each other. Nevertheless, transitions can still overwrite each other's assignments, so true composability cannot be achieved.

2.2.4 *Priority.*

allows to define (partial) priority orderings on transitions. At run-time, when more than one enabled candidate transition can be executed next, the candidate with the highest priority will be selected. If there is no single highest-priority candidate, non-determinism occurs, and in principle, a candidate is chosen randomly.

In Statecharts, two transitions can only be simultaneously enabled, if they either (a) have the same source state, (b) have orthogonal source states, or (c) have one source state that is an ancestor of the other's source state. This translates itself to complementary priority options for same-state, orthogonal and hierarchical. For the first two, we can either have priority None (no priority) or Explicit priority (i.e., priority is syntactically given). For "hierarchical", we can give higher priority to transitions whose source state is higher up in the hierarchy with Source-Parent, or lower down with Source-Child.

The options for the Priority dimension in our implementation differ somewhat from the options in [5]. We will now motivate this.

2.3 Deviations from BSML

2.3.1 *Priority and Order of Small-Steps.*

Besides Priority, BSML also has a dimension Order of Small-Steps, that determines the order in which transitions that do not disable each other are executed. In our opinion, this is just a matter of selecting the next candidate transition, which is the territory of Priority. Therefore, in our implementation, Order of small-steps became the orthogonal sub-dimension of Priority.

We have implemented Priority as follows: Based on the chosen priority options, a directed graph is statically built, whose nodes are transitions, and whose edges mean "has higher priority than", which is a transitive relation. An example of such a graph is shown in Fig. 2. Next, this graph is checked for cycles (when a cycle is found, the program terminates with an error), and subsequently it is checked if any pair of transitions of equal priority can be simultaneously enabled, based on whether their source states are orthogonal to each other, or ancestors of one another. If this is the case, we have detected a possible case of non-determinism, and present an error message to the user. If this is not the case (i.e., the model is guaranteed to be deterministic), a total ordering is constructed, consistent with the partial priority ordering, and stored in a list. Then, at run-time, this list is used to always quickly find the highest-priority enabled transition.

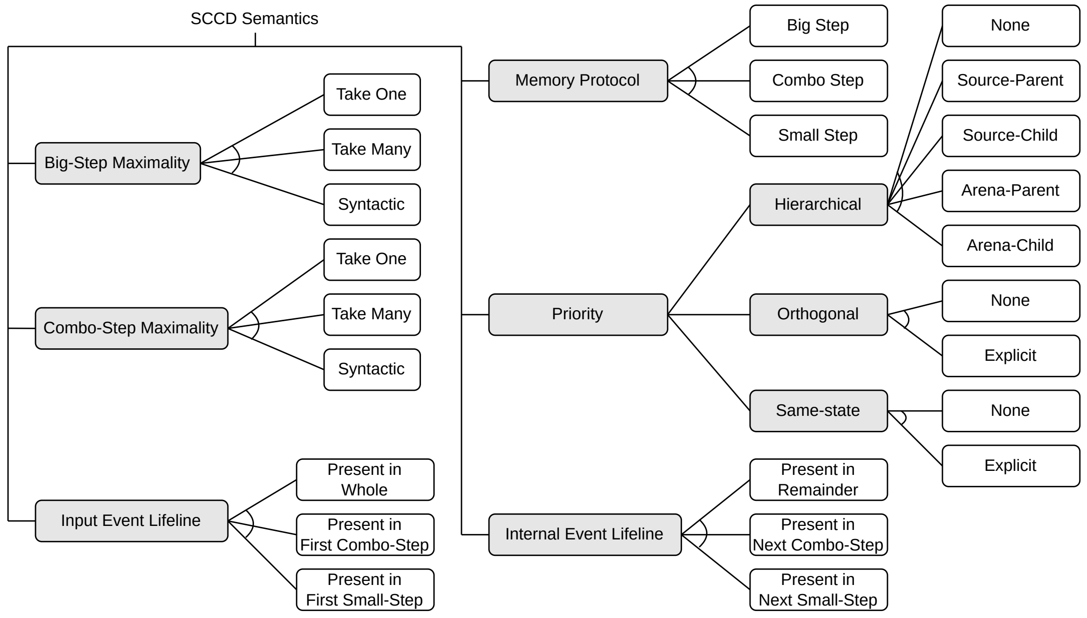

Figure 1: Feature model of implemented semantics

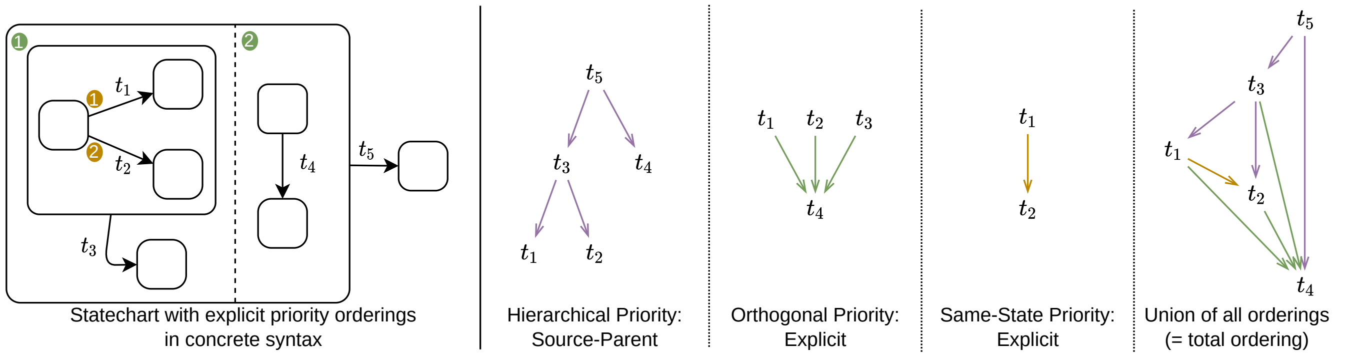
1
2
t_1
t_2
t_3
t_4
t_5
Statechart with explicit priority orderings in concrete syntax
Hierarchical Priority: Source-Parent
Orthogonal Priority: Explicit
Same-State Priority: Explicit
Union of all orderings (= total ordering)

Figure 2: Priorities between transitions: Example Statechart (left) and its priority graphs (right)

2.3.2 Negated Event Triggers. In BSML, a complementary way of expressing priorities between transitions exists, called *negated event triggers*. For instance, in the case where a transition t_1 is enabled by event a, and t_2 is enabled by event b, in the situation where both a and b are active, t_1 can be given a lower priority by altering its trigger to $a \wedge \neg b$, indicating that t_1 is not enabled whenever b is. In the early Statechart days, negated event triggers were considered the main mechanism for defining priorities [11].

Our implementation supports negated event triggers, but we consider it not a priority mechanism. Instead, it is a mechanism for restricting the transitions that can be simultaneously enabled.

When our priority graph is being checked for non-determinism, currently, only the source state of transitions is taken into account to detect whether two transitions can be simultaneously enabled. This could be relaxed (and still only let deterministic models pass) to also take into account negated event triggers: it can be statically guaranteed that two transitions $t_1 : a \wedge \neg b$ and $t_2 : b$ can never be simultaneously enabled.

2.3.3 Enabledness and Assignment Memory protocol. BSML distinguishes between Enabledness Memory Protocol and Assignment Memory Protocol. The former specifies what version of variables to read when evaluating guard conditions, and the latter specifies what version of variables to read when evaluating variable assignments. While our implementation does support these two dimensions, in the context of this paper, we consider them a single Memory Protocol dimension, because we are unaware of any use case for using separate options for each.

2.4 Non-included BSML Features

The following semantic features from BSML were not implemented:

2.4.1 Concurrency. allows multiple transitions to be taken in the same small-step, given that these transitions are commutative (i.e., the order in which they are executed does not matter). A small-step then becomes an unordered set of transitions. No priority needs to be defined between transitions that execute concurrently. The main difficulty is guaranteeing that transitions are indeed commutative. While [5] only takes the source and target states of transitions into account to determine commutativity, we believe that reads and writes on internal variables must also be considered. This could be done with concurrency analysis [3], a static analysis technique for finding possible race conditions. We consider this feature future work.

2.4.2 Dataflow. is a semantic option of the Priority-orthogonal dimension, that executes *writes before reads*, i.e., transitions that write to a variable are given a higher priority than transitions that read the same variable. Similar to Concurrency, static analysis on action language fragments would be necessary to support this feature. We consider this future work.

2.4.3 Non-causal Event Lifeline Options. For Internal Event Lifeline, BSML defines two additional options: Present in Whole makes an internal event active since the beginning of a big-step, which means the event can be active even before it was raised. This can yield non-causal behavior: A pair of transitions $t_1 : a/^\wedge b$ (is triggered by event a and raises event b) and $t_2 : b/^\wedge a$ can trigger *each other*. Some Statecharts variants allow this [10]. Because the execution of such transitions is not traceable to an input event, it is called *non-causal*. Due to its non-intuitiveness, this option was not implemented. Present in Same makes an internal event active in the same small-step, and therefore only makes sense in combination with Concurrency enabled. This option can also cause non-causal behavior, and was therefore not implemented.

2.4.4 Interface Variables / Events. as defined in BSML, are an additional way to compose different models together. We consider these not a Statecharts feature, and consider input- and output-events the only way of interacting with an environment, or possibly other Statechart models.

2.4.5 Non-syntactical Input/Output Events. BSML has several options for distinguishing between input, internal and output events. We always require explicit and syntactical distinction. We think that such distinction does not impact the essence of Statechart semantics.

3 IMPLEMENTATION

Our implementation was written in Python, and consists of a frontend (a textual parser), and two backends: an interpreter and a compiler (which generates Rust code). The only concrete syntax currently supported is a dialect of SCXML.

The action language component was developed as part of this project, but is nevertheless usable on its own: just like the Statechart component, it has its own concrete syntax, its own interpreter and its own (Rust) code generator. At each of these levels, the action language is *embedded* into the Statechart language.

The Statchart language of course consists of an implementation of BSML (i.e., big-step execution and semantic variability), but also of Statechart features like timed transitions, history, (scaled) real-time / as-fast-as-possible simulation, and queuing of input events. Apart from BSML's run-to-completion semantics, we also support the *synchrony hypothesis* [2], meaning that computations happen infinitely fast (in simulated time, that is). This makes it possible to model timeouts that happen precisely on time.

To verify that our implementation of BSML is correct, we have implemented the relevant examples from [5] as test cases. For an in-depth technical reference, we refer to [6]. The implementation can be found on our git server [7].

4 DISCOVERY OF UNIQUE VARIANTS

Our selection of implemented BSML features resulted in the feature model shown in Fig. 1. This model has 1944 variants, and every variant is executable, but not every variant has unique behavior: For instance, if we choose Take Many for combo-steps and Take One for big-steps, the combo-step still has to end when the big-step ends, and therefore the behavior would be identical to choosing Take One for combo-step maximality.

In order to come up with a set of constraints on our feature model that rule out "duplicated" variants, duplicated variants were detected in a "brute-force" manner: We created a Statechart model that was supposed to behave uniquely for each semantic option. A fragment of this model is shown in Fig. 3. Every semantic dimension is represented by one orthogonal region in the model. After executing one big-step with input `input0` from its initial configuration, the model is supposed to tell us the option chosen for each dimension, by ending up in a state with a label corresponding to the right option. Our repository includes the full model and source code for this experiment in the `examples/semantics` directory.

We have only considered those variants where Priority is explicitly defined at the orthogonal and same-state levels (ruling out non-deterministic variants), which reduces the number of variants from 1944 to 648. Then, for each of the 648 variants, we initialized the model and ran one big-step, and detected 312 unique final configurations. Finally, from observing the non-unique variants, we manually constructed a small set of rules (coded in Python, our implementation language) that prune the variants with non-unique behavior. The rules are "explainable", as follows:

- Combo-Step Maximality cannot be bigger than Big-Step Maximality. This was already explained.
- If a non-maximality option refers to combo-steps (i.e., input event lifeline is First Combo-Step, internal event lifeline is Next Combo-Step or memory protocol is Combo-Step), then big-step maximality must not be Take One: when this *is* the case, a combo-step will always end at the same time as a big-step, and hence, a combo-step has no distinctive meaning. Any option referring to combo-steps can then be rewritten to refer to the big-step. For instance, Present in First Combo-Step behaves identically to Present in Whole.
- If no non-maximality option refers to combo-steps, then combo-step maximality must be Take One (i.e., the default option). By only allowing one option for combo-step maximality here, this dimension does not create additional variants.

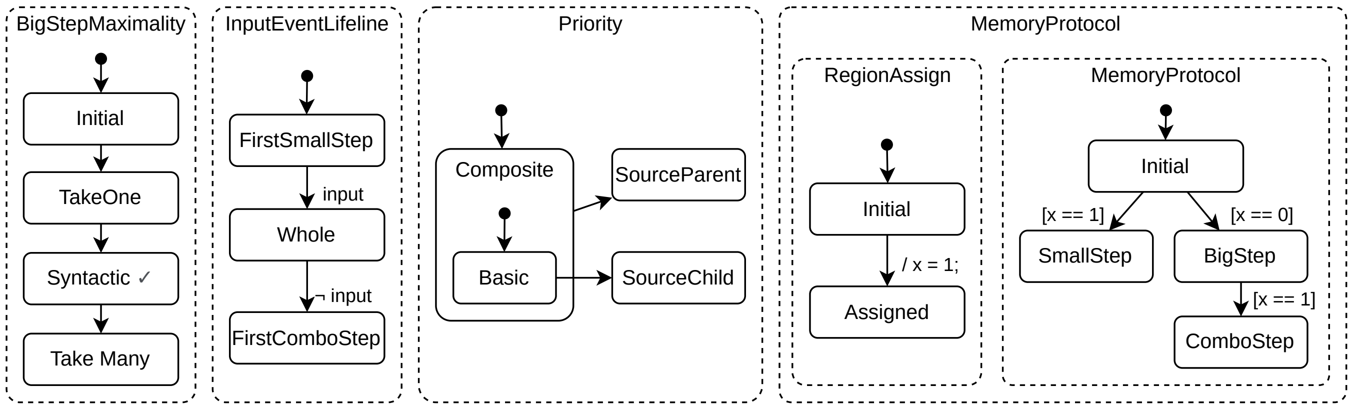

Figure 3: Fragment of Statechart designed to behave uniquely for each unique semantic variant

4.1 Discussion

It seems that, apart from interactions between combo-steps and other options referring to combo-steps, the implemented semantic dimensions truly are orthogonal. This still does not mean that each of the 312 unique variants is a good choice. Some thoughts:

- Probably some mechanism to limit the number of transitions in a big-step is desirable: Using Take Many in combination with Present in Whole or Present in Remainder could result in ever-enabled transitions, and hence, never-ending big-steps.
- For event lifeline, there is a strong intuitive coupling between options for input and internal events. We can probably distinguish between two classes of semantics: one the one hand, we have asynchronous, or "one event at a time" configurations, such as:
  - Present in First Small-Step + Present in Next Small-Step
    + (big) Take One
  - Present in First Combo-Step + Present in Next Combo-Step
    + (combo) Take One + (big) Take Many

  On the other hand, we have synchronous, or "all events simultaneously" configurations, such as:
  - Present in Whole + Present in Remainder + (big) Take One

  which may fit nicely with an "all transitions concurrently" approach (i.e., concurrency semantics, which hasn't yet been implemented). These two classes should not be mixed up. Restricting Input Event Lifeline and Internal Event Lifeline to these three configurations, reduces the number of non-duplicated variants further down to 74.
- For hierarchical priority, there is also the argument that a higher priority should always be given to higher-level transitions, because *refining* a state by adding sub-states (and sub-transitions) to it should not alter the existing behavior of that state.

5 RELATED WORK

An implementation of a subset of BSML was also made by Luo et al. [9], providing semantically configurable Statecharts within the mbeddr [1] modeling environment. Compared to our implementation, it has additional support for Concurrency semantics, but lacks support for combo-steps. Its implementation of Priority is more restrictive than ours: While explicit priorities between same-state transitions are supported, the compiler always falls back to *document-order* (of transition declarations, in textual concrete syntax) when no priority is given. While this guarantees determinism, it is a redundant priority mechanism, when also supporting explicit priority. Further, transitions are ordered by their priority at run-time, as opposed to our implementation, which resolves priorities statically, while detecting and rejecting non-deterministic models.

6 CONCLUSION

We have presented an implementation of a Statechart interpreter + compiler with semantic variability. We have implemented a subset of the BSML framework, mostly as-is, with only a few (motivated) deviations in the Priority dimension. Once we had an implementation, we could detect unique semantic variants, which led to the creation of simple rules to prune "duplicated" variants.

6.1 Future Work

For future work, we could further extend our implementation with Concurrency and Dataflow semantics. We could also enhance the static analyzer to detect if transitions can be concurrently enabled with respect to (negated) event triggers, in order to become less strict about its requirement for explicit priorities, while still guaranteeing determinism.

ACKNOWLEDGMENTS

Author J. Exelmans is an SB PhD fellow at FWO (1S70622N). This work was also partially funded by Flanders Make.

REFERENCES

[1] 2016. mbeddr. http://mbeddr.com/ Accessed: 2022-07-29.

[2] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

[3] Evelyn Duesterwald and Mary Lou Soffa. 1991. Concurrency Analysis in the Presence of Procedures Using a Data-Flow Framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991*, William E. Howden (Ed.). ACM, 36–48.

[4] Shahram Esmaeilsabzali, Nancy A Day, Joanne M Atlee, and Jianwei Niu. 2009. *Big-step semantics.* Technical Report. University of Waterloo, Cheriton School of Computer Science.

[5] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. 2010. Deconstructing the semantics of big-step modelling languages. *Requir. Eng.* 15, 2 (2010), 235–265. https://doi.org/10.1007/s00766-010-0102-z

[6] Joeri Exelmans. 2020. *A Study and Implementation of Semantic Variability in Statecharts.* Master's thesis. University Of Antwerp.

[7] Joeri Exelmans, Simon Van Mierlo, Yentl Van Tendeloo, and Glenn De Jonghe. 2022. SCCD source code. https://msdl.uantwerpen.be/git/jexelmans/sccd Accessed: 2022-07-29.

[8] David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274.

[9] Zhaoyi Luo and Joanne M Atlee. 2016. BSML-mbeddr: integrating semantically configurable state-machine models in a C programming environment. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 105–117.

[10] Florence Maraninchi and Yann Rémond. 2001. Argos: an automaton-based synchronous language. *Computer languages* 27, 1-3 (2001), 61–92.

[11] Amir Pnueli and Michal Shalev. 1991. What is in a step: On the semantics of Statecharts. In *International Symposium on Theoretical Aspects of Computer Software*. Springer, 244–264.

[12] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. 2016. SCCD: SCCDXML extended with class diagrams. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*. 1–6.

[13] Michael von der Beeck. 1994. A Comparison of Statecharts Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems - ProCoS, Lübeck, Germany, September 19-23, Proceedings (Lecture Notes in Computer Science, Vol. 863)*, Hans Langmaack, Willem P. de Roever, and Jan Vytopil (Eds.). Springer, 128–148.