SCCD Debugger: a Debugger for Statecharts and Class Diagrams

Francisco Simões
Miguel Goulão
Vasco Amaral
fd.simoes@campus.fct.unl.pt
mgoul@fct.unl.pt
vma@fct.unl.pt
NOVA School of Science and Technology
Lisbon, Portugal

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

ABSTRACT

Model-driven development (MDD) is increasingly relevant in the software development landscape. However, its adoption in the industry remains challenging. According to several studies, inadequate tool support, combined with insufficient expertise in the workforce and organisational and social factors, is part of the problem. One area for improvement in tool support is the implementation of adequate debugging mechanisms for models and software systems generated from those models. This paper introduces a debugger for models specified in the SCCD (SCXML extended with class diagrams) formalism, which combines statecharts with class diagrams. The debugger, a crucial tool in the context of MDD, supports debugging model-generated applications at the model level rather than at the level of the synthesized code. The debugger integrates with an open-source modelling and simulation tool for the SCCD formalism. Such debugging mechanisms are a stepping stone towards wider modelling adoption.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **System modeling languages**; *Unified Modeling Language (UML).*

KEYWORDS

statecharts, class diagrams, model-driven development, SCCD

ACM Reference Format:
Francisco Simões, Miguel Goulão, Vasco Amaral, Joeri Exelmans, and Hans Vangheluwe. 2024. SCCD Debugger: a Debugger for Statecharts and Class Diagrams. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3652620.3687792

1 INTRODUCTION

As the complexity of software systems continues to grow, we need appropriate methods and tools to cope with this complexity. Adopting Model-Driven Development (MDD) offers several potential development improvement opportunities, including development automation and better systems understanding.

There are still several challenges to Model-Driven Engineering adoption in general [3], and MDD in particular. These challenges are of diverse kinds, including foundation, domain, tool, community and social. In this paper, we focus on improving tool support for MDD, more specifically by improving analysis capabilities. We present the SCCD Debugger for the SCCD formalism [14], which combines statecharts with class diagrams. This tool makes it easier to assess models, by adding a previously missing debugging mechanism to the existing SCCD formalism tool support, moving a step closer to the maturity exhibited by traditional IDEs and their support for debugging and contributing to the development of higher quality models.

This paper is organised as follows: section 2 briefly introduces the SCCD formalism. Section 3 presents the SCCD modelling environment architecture, focusing on its debugging environment. Section 4 describes the main debugger tool features we implemented. Section 5 presents usage scenarios for the SCCD debugger. Section 6 presents a pilot evaluation of our tool. Section 7 discusses related work. Finally, section 8 concludes the paper and outlines directions for future work.

2 BACKGROUND

The SCCD formalism [14] combines the Statecharts and Class Diagrams formalisms.

Statecharts are behaviour diagrams used in the specification of (possibly complex) reactive timed systems [6]. A reactive system is event-driven and reacts to internal and external stimuli. The specification of reactive systems requires an easy to understand and precisely defined formalism in which to develop behavioural specifications. Statecharts are an appropriate formalism with precisely defined syntax and semantics. Class diagrams are structure diagrams. They describe invariants over the (possibly changing) structure of a system in terms of classes with their attributes and operations, associations, and multiplicities between them [5]. At runtime, the structures are objects (instances of the classes) in memory.

By combining statecharts and class diagrams, SCCD supports modelling complex, timed, autonomous, reactive, and dynamic-structure systems. SCCD has a syntax in an XML format based on the W3C SCXML recommendation. Models may be synthesised into several programming languages (currently Python, JavaScript and C#) and several runtime platforms implemented in those languages.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687792

SCCD tooling does not currently come with a visual modelling environment. It consists only of a (command line) parser and code generator. In contrast, other tools usually include a graphical interface allowing the creation of models. To model with SCCD, one must write the model using its XML format syntax.

An SCCD model consists of several classes and associations. Each class has an associated statechart which specifies the behaviour of its object instances at runtime. Objects can only communicate with each other via events. The entry point of an SCCD model is a default class from which a single instance is created and started when the system runs. Instances can then *create*, *start*, and *delete* other instances of classes, as well as *create* and *delete* instances of associations.

3 SCCD DEBUGGER ARCHITECTURE

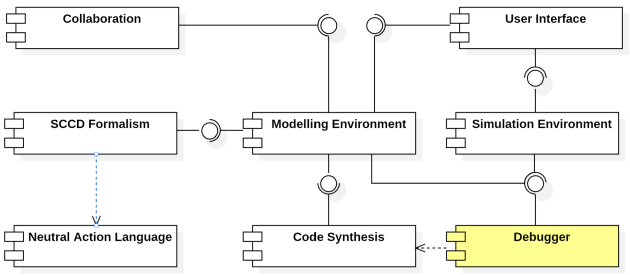

Figure 1: Architecture of the SCCD Modelling, Simulation, Code Synthesis and Debugging Environment

This work is part of an international collaboration between the NOVA School of Science and Technology (Portugal) and the Universiteit Antwerpen (Belgium).

The developed tool supports the following usage flow, supported by the components in Figure 1: a model is created in the SCCD formalism using the *Modelling Environment* component, and code is synthesized from it using the *Code Synthesis* component. The generated code may then be run, resulting in a working application. This application may be debugged using the *Debugger* component (yellow-coloured in the figure since it is the focus of this work). The debugging process is made visible at the model level while simulating the model's execution using the *Simulation Environment* component. This makes it easier to reason about the system's behaviour and to find the defects of the model and the system. In the future, the tool will also support collaboration mechanisms via its *Collaboration* component. All the interactions between the user and the tool use the *User Interface* component.

This tool is an open-source project that can serve as a basis for further research into improving MDD tools. Our implementation and working methodology has independent tool components (such as the debugger, the modelling environment, and so on), creating much-desired modularity, as each component may be updated without changing the rest of the implementation.

4 IMPLEMENTED FEATURES

4.1 Play/Pause

This feature allows the user to pause a simulation and resume it at the point it stopped.

4.2 Simulation Modes

There are three different simulation modes available:

- **(Soft) Real-Time Simulation** - This is the default which executes the model specification.
- **Scaled Real-Time Simulation** - This mode is similar to the **Real-Time Simulation** but applies a speed scale factor to all of the time dependencies in the model. *Example*: If the user gives a speed scale factor of 2 to a simulation, then the simulation will be 2x faster. If there is a time delay of 10s specified on a transition, it will only take 5s using a scale factor of 2.
- **As-Fast-as-Possible Simulation** - This mode uses simulated time (just a variable). The relationship of the simulated time to the wall clock time is no longer linear.

The simulation mode is selected through command line arguments when launching the simulation on a console.

4.3 Breakpoints

There are three different breakpoint types available:

- **State Breakpoint** - This type of breakpoint may be placed on a specific state. When the simulation enters the specified state, it is paused.
- **Variable Breakpoint** - This type of breakpoint sets a condition regarding a variable of the model. The user can define a value for the variable and, when the variable reaches that value, the simulation is paused.
- **Time Breakpoint** - This type of breakpoint can be set to a particular time from the start of the simulation. When this time is reached, the simulation is paused.

The user may place breakpoints using a breakpoints file, in XML format. This file contains an XML element called *breakpointsList* which may contain several *breakpoint* elements. Depending on the breakpoint type the user wishes to place, it may take one of the following 3 attributes: *state*, *timestamp* and *variable*.

4.4 Steps

When using code debuggers, stepping execute chunks of code without pausing during their execution. In model debuggers, steps should allow one to execute transitions between states. There are 2 different transition types in statechart diagrams: time-based transitions and event-based transitions. Time-based transitions depend on how long it has been since the current state was entered, whereas event-based transitions are not dependent on time. Steps are generated for time-based transitions so the user does not have to wait for the required time to pass. Event-based transitions do not need steps as the user simply needs to write the event name for the transition to be performed. Both types of transitions can be made conditional through a guard condition.

4.5 Tracing

Tracing is a debugger feature that records all the events that occurred during a model simulation in a text file. The recorded events are the *start of the simulation*, the *entry* and *exit* of states and the *end* of the simulation. Each recorded event has an associated timestamp of its occurrence, its name and a list of the values of all the model attributes at that moment (a "snapshot"). Besides the events, some other useful metrics are recorded, including the total time of simulation, execution time, and debugging time. No events are recorded after the user sends the *stop* event.

4.6 Help

This feature is not specifically related to a debugger or models, however, it is an important one for tools to have. It was included to provide a more user-friendly experience and to help new users get used to the tool. At any point in the execution, the user may type *help* on the console, sending the *help* event to the model. This will display a small menu to the user with available commands and actions that the user can perform.

5 USING THE DEBUGGER

To use the debugger, it is first necessary to compile the model to be simulated and debugged. The model in Figure 2 specifies the behaviour of a very simplified phone system. It is used to exemplify a model simulation. An XML file containing this model is provided to the compiler so it can generate code. The name of the file is passed as an argument on the command line when running the compiler alongside other arguments, such as the target language and a flag to indicate whether the user wants debugging mechanisms on or off.

After the model is compiled, the user obtains a file containing the code to simulate the given model. The user will then run an auxiliary file using the previously generated file to start the simulation. When running this auxiliary file, the simulation mode and speed scale factor can be given as arguments in the command line. If these are not specified, the simulation will be **Real-Time Simulation**, and the scale factor will be 1, by default.

We will use a **Real-Time Simulation** (using a speed scale factor of 1) with no breakpoints. After starting the simulation, the screen in Listing 1 will be shown to the user. The user can see which states were entered and the current state. Available transitions coming out of the current state are also shown to the user, including the name of the transition, its type (event-based or time-based), its target state, its timer if it is time-based and finally, the guard condition if it has one. In this case, the user may send one of the 2 events: **press_power_button** or **plug_charger**. In this simulation, the user will charge their phone before turning it on.

When the charger is plugged,a new time-based transition is available with a timer of 15s, which can be seen in Listing 2. This timer may be skipped using the **step** event. When the phone has enough battery power, the user will turn it on. A parallel region is entered there, so it is possible to see that the current states are */powered_on/charge/not_charging* and */powered_on/apps/homescreen*. In the first active state, the user may plug the phone charger as before; in the second active state, the user may choose to use the apps available on the phone.

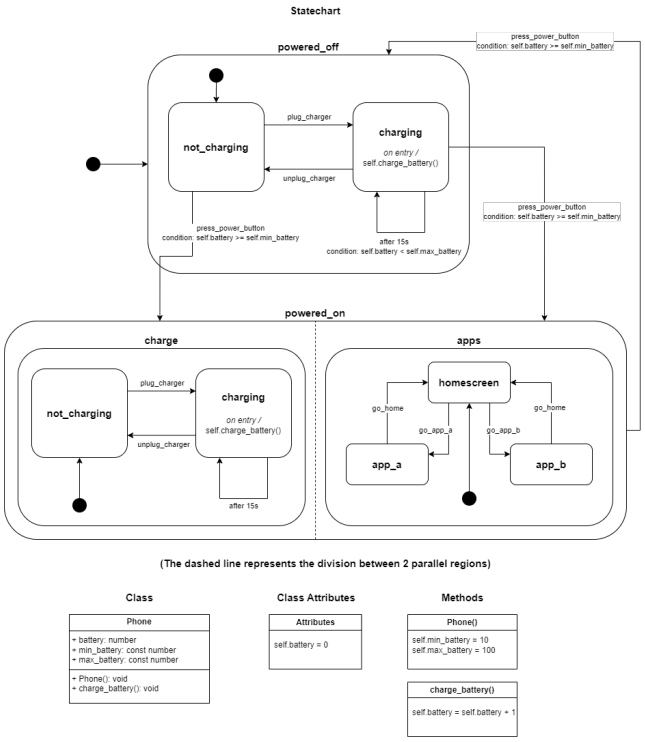

Figure 2: Example Model

6 PILOT EVALUATION

We performed a pilot evaluation to assess our tool with 10 computer science Masters students. 9 were male, and 1 was female. Participants were between 23 and 24 years old, with basic knowledge of statecharts (8 worked with them for 6 months, 1 for 1 year, 1 for 2 years) and class diagrams (4 worked with them 1 year, 5 for 2 years, 1 for more than 2 years). They performed 5 tasks using the debugger and then answered questions about the tasks and the tool, including questions to evaluate the usability of the tool and task load, using the standard questionnaires of the System Usability Scale (SUS) [2] and NASA-TLX [7]. Each participant also answered a set of demographic questions. Every user test session was performed online. The user shared their screen with the researcher while performing the tasks. Each session was recorded, allowing usage precision and recall scores (with respect to gold standard solutions developed by the first author) to be calculated. For each task, we defined the necessary steps for its successful completion and what steps could be considered mistakes. Some actions were redundant for our assessment, but not considered mistakes (e.g. consulting the help menu). Most mistakes consisted of using commands that did not exist or inappropriately using the ones that existed.

Each task exercised different features of the debugger. Some had specific instructions to guide participants through the different functions that can be performed using the debugger and teach them how to use them. Others had less detailed instructions on how to achieve general goals using the debugger.

Listing 1: Console on Simulation Start

```
Real-time Simulation
Scale Factor: 1.0
Type help to see the available commands.

Entered /powered_off
battery: 0

Entered /powered_off/not_charging
battery: 0
Available Transition Options:
[event-based] type press_power_button to perform the transition to ['/powered_on'] with the guard condition self.battery >= self.min_battery
[event-based] type plug_charger to perform the transition to ['/powered_off/charging'] with the guard condition None
[ /powered_off/not_charging ] >
```

Listing 2: Console on Charger Plugged

```
[/powered_off/not_charging ] > plug_charger

Entered /powered_off/charging
battery: 1
Available Transition Options:
[time-based] type step to skip the transition to ['/powered_off/charging'] which has a duration of 15 seconds and the guard condition
self.max_battery > self.battery
[event-based] type press_power_button to perform the transition to ['/powered_on'] with the guard condition self.battery >= self.min_battery
[event-based] type unplug_charger to perform the transition to ['/powered_off/not_charging'] with the guard condition None
[ /powered_off/charging ] >
```

Table 1 summarises the usability and task load scores. The results were encouraging. The tool obtained a mean SUS score of 76.5 points (the average SUS score is 68 [9]). The mean TLX result was 24.8 out of 100 (where 100 is the most negative score) denoting a low effort level.

Table 1: User Test Usability and Task Load Scores

Test	Mean	Std Deviation	Min	25%	50%	75%	Max
SUS	76.5	16.6	45.0	65.0	80.0	89.4	97.5
NASA-TLX	24.8	7.3	15.0	19.2	24.2	29.6	38.3

As for the precision and recall results seen in Table 2, both have very high scores, which indicates how easy it may be to get used to the tool. The worst precision rate was on task 3, with a mean of 88.25%. This was due to the **step** event being used in debug mode, which was not allowed in the implementation of the debugger when the user tests were made. This was suggested by the participants of the user tests a few times as an improvement to the tool and would surely be a very important upgrade.

Table 2: Precision and Recall Statistics per Task

Task	Mean	Std Dev	Min	25%	50%	75%	Max
T1 Prec	97.89	4.46	88.89	100.00	100.00	100.00	100.00
T1 Rec	98.89	3.51	88.89	100.00	100.00	100.00	100.00
T2 Prec	98.33	5.27	83.33	100.00	100.00	100.00	100.00
T2 Rec	100.00	0.00	100.00	100.00	100.00	100.00	100.00
T3 Prec	88.25	9.74	75.00	81.82	89.45	97.50	100.00
T3 Rec	98.89	3.51	88.89	100.00	100.00	100.00	100.00
T4 Prec	95.75	5.53	87.50	90.00	100.00	100.00	100.00
T4 Rec	95.56	9.39	77.78	100.00	100.00	100.00	100.00
T5 Prec	98.09	4.03	90.00	100.00	100.00	100.00	100.00
T5 Rec	99.00	3.16	90.00	100.00	100.00	100.00	100.00

7 RELATED WORK

T here are some important features that should be included in a statechart modelling and simulation tool. We may evaluate and compare different tools based on the following features:

Graphical User Interface. A graphical user interface can be useful to create or debug a model. A prototype visual modelling and simulation environment was developed in Sylvain Elias' Masters thesis [4].

Statechart Modelling and Code Generation. The SCCD compiler takes models in SCCDXML, a textual syntax (based on SCXML) as input and produces code in different languages (currently Python, Javascript and C#) for different platforms. Visual modelling environments may export SCCDXML.

Debugging and Testing Mechanisms. Different debugging and testing mechanisms can be added to SCCD, including variations of mechanisms used in the classic programming IDEs, such as breakpoints. These mechanisms were chosen based on Simon Van Mierlo's PhD research [11].

Collaboration Mechanisms - There are several possible ways in which collaboration mechanisms might be used. There may exist a shared repository where models are stored and from which they may be pulled or pushed (the same way git repositories work), or there might exist some live model sharing or screen sharing feature, allowing concurrent live modelling.

State Machine Simulation. Given an SCCD model, it can be simulated by running the generated code.

Multi State Machine Simulation. Multi-state machine refers to multiple state machine objects that function independently of each other but can also communicate between them to form a more complex system. The multi-state machine system must be able to be simulated. The advantage of having multi-state machine interactions is that each state machine is simpler than one single, very complex state machine.

Table 3: Statechart modelling advanced feature

Tool	Graphical Interface	Statechart Modelling	Code Generation	Debugging Mechanisms	Collaboration Mechanisms	State Machine Simulation	Multi State Simulation	Dynamic Structure Models
Itemis	X	X	X	X		X	X	
Sparx	X	X	X					
Umple	X	X	X					
Stateflow	X	X	X	X	X	X		
SCCD		X	X	X		X	X	X

Dynamic Structure Models. Dynamic Structure Models entails creating and deleting new objects with their associated state machines at runtime.

We compared our tool to the following alternatives:

Itemis Create. Itemis [1] is a commercial statechart tool allowing model creation and simulation through a graphical user interface. It can generate code for a model, and it has debugging mechanisms that may be used during a simulation.

Sparx Enterprise Architect. Enterprise Architect [13] targets professional corporate usage, allowing users to create and simulate models while benefiting from high performance and effective global collaboration mechanisms. It can also generate model code and supports debugging for model simulations.

Umple. Umple [8] is an open-source modelling tool and programming language family which enables Model-Oriented Programming. It adds UML-derived abstractions, such as attributes and state machines, to object-oriented programming languages such as Java, C++ or PHP. It also allows the creation of state machine diagrams and class diagrams textually.

MATLAB Stateflow. Stateflow [10] is a modelling tool developed by MathWorks, providing a graphical language combining mainly state machine diagrams and flow charts. It is possible to model systems using combinatorial and sequential decision logic that can be simulated as a block within a Simulink (modelling and simulation tool also developed by MathWorks) model or executed as an object in MATLAB. It provides executing and debugging mechanisms for the created models, and it distinguishes itself for what it can do alongside the MATLAB language: Stateflow charts may use the capabilities of MATLAB, and they may be used as MATLAB objects in applications which require state machine and timing logic.

SCCD differs from the other tools in that it uses a formalism which combines statecharts with class diagrams thus supporting both multi-state machine simulation and dynamic structure modelling. A feature comparison between SCCD and other tools can be found in Table 3.

8 CONCLUSION

With this project, we developed a command-line interface tool capable of model simulation, debugging, and performing multi-state machine simulations and dynamic structure modelling. Despite having a large room for improvement, in its current state, this tool can be seen as a stepping stone in modelling and simulation tools and their usability as it allows both multi-state machine simulation and dynamic modelling and combine these features with debugging mechanisms.

CODE REPOSITORY AND DEMO VIDEO

The code repository is available at: https://github.com/FranciscoSimao11/sccd-debug. The code repository includes links to video tool demos showcasing the most relevant debugger features. More detailed documentation on the debugger can be found in [12].

ACKNOWLEDGMENTS

The authors would like to thank NOVA Laboratory for Computer Science and Informatics with the reference UIDB/04516/2020: DOI 10.54499/UIDB/04516/2020 and UIDP/04516/2020: DOI 10.54499/UIDP/04516/2020.

REFERENCES

[1] Itemis AG. 2018. *Itemis Create.* Itemis AG. https://www.itemis.com/en/products/itemis-create/

[2] John Brooke. 1995. SUS: A quick and dirty usability scale. *Usability Eval. Ind.* 189 (11 1995).

[3] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19 (2020), 5–13.

[4] Sylvain Elias. 2021. Model-Based Development of a Modelling and Simulation Environment for the Statecharts and Class Diagrams (SCCD) Formalism. Master's thesis. Univerity of Antwerp.

[5] Object Management Group. 2017. OMG UML Specification. https://www.omg.org/spec/UML/2.5.1/About-UML.

[6] David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

[7] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Human Mental Workload*, Peter A. Hancock and Najmedin Meshkati (Eds.). Advances in Psychology, Vol. 52. North-Holland, North-Holland, 139–183. https://doi.org/10.1016/S0166-4115(08)62386-9

[8] Timothy C. Lethbridge, Andrew Forward, Omar Badreddin, Dusan Brestovansky, Miguel Garzon, Hamoud Aljamaan, Sultan Eid, Ahmed Husseini Orabi, Mahmoud Husseini Orabi, Vahdat Abdelzad, Opeyemi Adesina, Aliaa Alghamdi, Abdulaziz Algablan, and Amid Zakariapour. 2021. Umple: Model-driven development for open source and education. *Science of Computer Programming* 208 (2021), 102665. https://doi.org/10.1016/j.scico.2021.102665

[9] James R Lewis and Jeff Sauro. 2018. Item benchmarks for the system usability scale. *Journal of Usability Studies* 13, 3 (2018), 158–167.

[10] MathWorks. 2018. *Stateflow.* MathWorks. https://www.mathworks.com/products/stateflow.html

[11] Simon Van Mierlo. 2018. *A multi-paradigm modelling approach for engineering model debugging environments.* Ph. D. Dissertation. University of Antwerp.

[12] Francisco Simões. 2024. *Debugging Statecharts Extended with Class Diagrams.* Master's thesis. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

[13] Sparx Systems. 2021. *EnterpriseArchitect.* Sparx Systems. https://sparxsystems.com/products/ea/

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