Explicit Modelling of Statechart Simulation Environments

Sadaf Mustafiz[†] **and Hans Vangheluwe**[‡,†]
[†]**School of Computer Science, McGill University, Canada**
[‡]**University of Antwerp, Belgium**
sadaf@cs.mcgill.ca, hans.vangheluwe@ua.ac.be

Keywords: interactive simulation environments, experimentation, Statecharts, model transformation

Abstract

In this paper, we propose an experimentation environment for the interactive simulation of Statechart models. We choose the Statecharts formalism as the most appropriate formalism to model and synthesize the environment. We take inspiration from software debugging as well as from simulation experimentation to explicitly model the detailed reactive behaviour of our environment. We map program debugging techniques such as execution modes, steps, and breakpoints to the simulation domain. We further explore how to integrate the notion of simulation time for the purpose of (scaled) real-time visualisation. Finally, we provide support for a (browser)client-server architecture, again making use of the features of Statecharts. We build the experimentation model on top of the model to be simulated by instrumenting it using model transformation techniques. The entire Statechart modelling, simulation, and experimentation environment described in this work is supported by our tool, AToMPM.

1. INTRODUCTION

Over the past decades, numerous interactive modelling and simulation environments have been developed. This has greatly simplified the design and development of complex systems. Most of these environments are however hand-crafted, complex software systems. The hand-crafting is labour-intensive and uses abstractions (namely software) that are far from optimal. In particular, it is hard to understand and evolve simulation experimentation environments as they are reactive (to both user and simulation engine events) and time-dependent.

The software engineering community has recently developed various techniques and tools for model-driven engineering of complex systems. Meta-modelling, model transformation, and code synthesis now support the rigorous and rapid engineering of modelling languages with accompanying (visual) modelling environments, simulators, and code generators.

There is a need to integrate model-driven engineering (and in particular modelling language engineering) and simulation. This will enable the rigorous and rapid development of integrated environments for modelling and simulation of models in formalisms such as Statecharts, DEVS, Causal Block Diagrams, and Petri nets. In this paper we will focus on the single, popular formalism Statecharts. Simulation experimentation environments for multi-formalism models are even more complex than single-formalisms environments. We believe the work in this paper is a first step to the future model-based development of multi-formalism simulation experimentation environments.

Our goal is to explicitly model and subsequently synthesize an interactive simulation experimentation environment for models in the Statechart formalism by bringing together inspiration from software debugging and simulation experimentation domains.

We choose to explicitly model the simulation experimentation environment with Statecharts due to the expressiveness of this formalism. It allows us to model interrupts, timeouts, hierarchy, and history in a modular and intuitive fashion. In this paper, we restrict ourselves to the simulation of Statechart models. This simplifies the (code) synthesis, but is by no means limiting. Simulation models in a discrete-event formalism such as DEVS pose no problems as such formalisms have similar expressiveness. Early experiments have shown that models in the rather different Causal Block Diagram (also known as Synchronous Data Flow) formalism can equally well be integrated in a simulation experimentation modelled using Statecharts. This is the subject of our future work.

Explicitly modelling the simulation environment has many advantages. It acts as a form of documentation for the user as well as for the developer of the environment. It helps in understanding, is easier to modify, and allows for formal analysis and testing. It allows for example modellers to guarantee that the environment always responds to user request within certain time bounds. It also is possible to explicitly model the user leading to a fully autonomous simulation environment.

We seek inspiration in techniques used in the debugging of software programs. Debugger primitives such as execution modes, step, and breakpoints, have proved to be of utmost use in the programming domain. In our work, we lift these concepts to the modelling and simulation realm to allow us to experiment with models through simulation in much the same way that we can with code.

We also take inspiration in the theory of modelling and simulation, and in particular, in the notion of Experimental Frame [10, 11]. The Experimental Frame describes experimental conditions and environment under which the system under study and corresponding models will be used. It reflects the objectives of the experimenter who performs the experiment on a real system or, through simulation, on a model. Experimentation is the act of carrying out an experiment. An experiment may interfere with system operation (influence its input and parameters). As such, the experimentation environment may be seen as a system in its own right and it may be modelled explicitly.

This paper is organised as follows: Section 2. gives background information on Statecharts and model transformation. Section 3. discusses our approach of explicit modelling of an experimentation environment. Section 4. describes the implementation of the Statecharts simulation environment. Section 5. presents related work and section 6. draws some conclusions and presents directions for future work.

2. BACKGROUND

Within the context of this paper, we follow the terminology presented in [4]. A *model* is completely described by its abstract syntax (its structure), concrete syntax (its notation, or how it is presented to the user – textually and/or visually) and semantics (its unique and precise meaning). A modelling *language* is a possibly infinite set of (abstract syntax) models. This set can be concisely described by means of a grammar or a meta-model. No semantics or concrete syntax is given in meta-models. When the language is combined with concrete syntax semantics, we call it a *formalism*. Our work in this paper is based on the Statecharts formalism. The use of model transformation is an integral part of our work. We give a brief background on the two topics in this section.

The Statecharts formalism is an extension of Deterministic Finite State Automata with hierarchy, orthogonality and broadcast communication introduced by David Harel in 1987 [5]. It is a popular formalism for the modelling of the behaviour of reactive systems. It has an intuitive yet rigourously defined visual notation and semantics. It is the basis for documentation, analysis, simulation, and code synthesis. A statechart model is usually described with the following basic elements: states (basic, orthogonal, composite), transitions (event-based or time-based), enter/exit actions, history state, guards, and actions. Please refer to [5] for details. Many variants of Statecharts exist, including the one in the UML standard. In our modelling environment, we support the DCharts [3] variant and the UML syntax. Our simulator supports the Statemate semantics introduced by Harel [5].

Model transformation [2] concerns the mapping of source models in one or more formalisms to target models in one or more formalisms, using a collection of transformation rules. In this work, we use *rule-based graph transformation* as well as template-based model-to-code generation as the means for model transformations. The former requires (meta-)models to be stored as typed, attributed graphs, thus allowing model manipulations to be defined as graph transformation rules.

3. MODELLING AN EXPERIMENTATION ENVIRONMENT USING STATECHARTS

In this section, we outline how we blend different aspects: (1) debugging concepts, (2) simulation experimentation concepts, (3) animation techniques and (4) means to support a client/server architecture. We start by describing each aspect, and then discuss how we integrate them in our environment.

3.1. Debugging

As with the adoption of debugging concepts in domain specific modelling proposed in [7], we describe a conceptual mapping between key debugging concepts in the software programming world and their simulation domain counterparts. We integrate several debugging concepts into our simulation experimentation environment: step, breakpoint, and input.

3.1.1. Steps

In analogy with the stepping through code line by line, simulations can be run step by step, pausing for user input after every step. In the case of the Statechart formalism, this implies that we want to single step from state to state. In programming, it is possible to *step* through code in three possible ways: *step into*, *step over*, and *step out*. We address the need for such debugging primitives in our experimentation environment.

Step into A software function is an encapsulion unit. It encapsulates local data on the stack as well as control flow. *Step into* refers to stepping through instructions including any child functions defined within. The analog of an encapsulation unit in a Statechart is a hierarchical state. Stepping through a model in such a case means stepping through each composite and orthogonal state at the level of basic states.

Step over Step into and step over are each other's dual. Simulation in *step over* mode implies that we step through the model at the composite state level. Step over can be seen as a filter. When debugging, we only have control over pausing and resuming. Step over does not mean that we do not execute the underlying functions, we only hide the underlying details. Events can still trigger change at a low level, but that is done transparently. Hence, stepping over several units of encapsulation only animates the states at the composite state level. This is helpful for didactic purposes since it gives a high-level (trace) view of the model. Note that we only apply this to hierarchical components and not to orthogonal components. While it is technically possible to do the same for concurrent states, it does not logically make sense as it would create un-balance between *and* states. It makes sense to make them either all visible or invisible.

Step out This makes it possible to start stepping through a function and then when requested by the user switch to continuous mode. In Statechart, when simulated in *step into* mode, the user has the option, at every step, to switch to simulating in the *step out* mode. At that point, the animation is applied at the composite state level and the details inside the state are hidden.

The inputs for simulation can be given in several ways: (1) we can start a dialog to get inputs from the user in real-time; (2) the user can give a model of the input; or (3) we can have a trace-driven input (a special kind of input model).

3.1.2. Breakpoint

Steps in debugging is orthogonal to *breakpoints*. Breakpoints in programming are in essence assertions. Once the assertion becomes true, the execution halts. This may be triggered by a particular (algebraic) relationship holding over variables, or by reaching a particular line (in source code). In analogy, we support breakpoints in our experimentation environment to pause or terminate the simulation. The environment allows users to set a breakpoint and when the condition holds, the execution pauses. Breakpoints can be of three kinds.

(Modal) state This specifies a particular state (or set of states). When these are reached, the execution of the model is paused.

Store This specifies a particular condition (e.g., $x+y>10$, where x and y are variables within the model) which needs to be met for the execution to be paused.

Time This specifies a particular simulation time. When this time is reached, the execution is to be paused (as is the simulation time). The aspect of time and the issues related to it are discussed in more detail in Sec 3.3..

3.1.3. Execution modes

Just as it is possible to stop or pause and then resume the execution of code during debugging at any point in time, in our Statecharts environment we allow simulations to be paused or stopped by the user at any given moment during the execution of the model.

We build the experimentation environment on top of the model to be simulated using graph transformations. The environment is by default in a **Ready** mode. When the user starts the simulation, the environment switches to the **Running** mode. In the **Running** mode, the modeller can switch between the different simulation modes: *step into*, *step over*, and *step out* (described in Sec 3.1.1.). While the simulation is running, the user can choose to pause the simulation at any time, thus switching to **Pause** mode. From the pause mode, the user can resume simulation. This leads to returning to the **Running** mode and to exactly the state(s) in the model prior to pausing the simulation. The environment retains the history of the model and can resume from where the user had left off. Terminating a simulation puts the environment in a **Stopped** mode. A timed *reset* event is triggered to reset the environment and return to the initial **Ready** mode from the **Stopped** mode. The different states and modes are illustrated in Figure 6.

3.1.4. Inspect/Modify Variables

Debugging environments allow the user to inspect and/or modify parameter and variable values. In the Statechart environment, the user can choose to keep track of a list of variables and any changes to these variables due to actions being executed when transitions are triggered are displayed to the user. Our environment does not support this feature at the moment, but we plan on implementing it in the near future.

3.2. Experimentation

We borrow notions from the theory of modelling and simulation and in particular the concept of Experimental Frames to build our simulation environment for the Statecharts formalism.

This tells us we have the model on the one hand and the experimentation environment on the other. Variables (states) need to be assigned initial values (conditions), as well as do parameters (constants), both for the model and for the environment. When we set up an experiment, we must customize the model by first choosing a particular model, giving it initial conditions by setting initial values for variables, giving it parameters, and also providing a model for the environment (that specifies the inputs). In case of an autonomous system, there are no inputs. For example, in a Statechart, if the model contains orthogonal components and time delays it may never deadlock, even without any events coming from the environment.

In our domain of interest, namely Statecharts, the parameters and initial conditions are specified within the model. The initial states are set using a visual special syntax (such as pseudo states in UML) or by setting an initial state attribute to *true* (such as in DCharts); the variables and parameters can be set in the enter/exit actions of states or in the actions triggered by transitions. For other formalisms, it might be necessary to set the parameters and variables outside the model.

Structural changes change the structure of the model and requires recompilation. *Parameteric changes* usually only affect the behaviour of the model. However, structural parameters may be defined which leads to structural changes but is based on a parameter which determines the structure. For example in the equation, $Y=Ax+B$, changing A or B changes the behaviour of the model whereas adding an extra parameter, Cx^2 changes the structure of the model.

A simulation runs until some condition is satisfied (possibly over the variables, or some time is reached) or until a user intervenes. Time can increase in fixed steps or variable steps, and we continue and update time accordingly till the simulation terminates. The experiment can possibly output some performance results. Based on the results, it might be necessary to refine model parameters and initial conditions and repeat the experiment one or more times. Figure 1 shows the activities in model-based system analysis. Our focus is on the simulation phase.

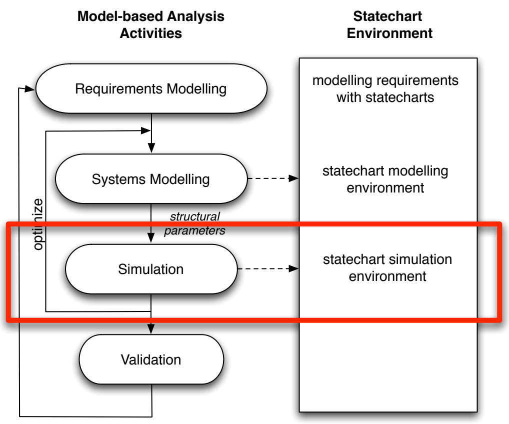
Model-based Analysis Activities
Statechart Environment
Requirements Modelling
modelling requirements with statecharts
Systems Modelling
statechart modelling environment
optimize
structural parameters
Simulation
statechart simulation environment
Validation

Figure 1. Modelling and Simulation Activities

3.3. Time/Animation

Time in simulated environments can be defined in three ways (see Figure 3).

- **Real-time** means to run the simulation in synch with the wall-clock time.
- **Scaled real-time** means to run the simulation faster or slower than real-time, but with a constant (scale factor) relating them.
- **As-fast-as-possible simulation** (known as "analytical" in the HLA community) implies that there is no linear relationship between simulated and wall-clock time. Rather, the simulated time is updated in leaps and bounds.

Time is what sets simulation apart from the execution of code. In code, a statement execution changes the state and then the control flow determines which statement should be executed next. Though causality must be preserved, the actual time value is irrelevant. In the case of simulation, a much more detailed notion of time is used. We need to consider times at which events occur, times at which simulation actions happen internally, and times when users give inputs. There are several issues relating to time that we need to address.

- Our environment allows us to pause the simulation at any time, but this can lead to ambiguous situations for which design decisions need to be made. In our environment, the pause event during timed transitions would interrupt the transition and put the model in the originating state. Figure 2 shows a possible case which requires us to return to the source state if a pause event occurs. This would be strange if the timed transition is a few hours long which would mean we interrupt and restart the transition. In such simulations we usually use scaled time, which lessens the error. A correct approach is not only to keep track of which state the system was in, but also how long it was in that state for (the *elapsed time* of the DEVS formalism). It is possible to instrument our models such that elapsed time can be measured.

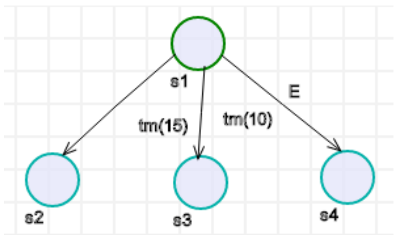

Figure 2. Statechart Example: Issues in the Pause Mode

- While rollbacks are possible in as-fast-as-possible simulation, with real-time systems, such methods cannot be applied. The inexorable passage of time does not allow us to go back in time and to insert inputs at a time less than the current time. This is depicted in Figure 3.

3.4. Client/Server

We move on to the more technical aspects of our work which involves being able to model and simulate the Statecharts in a browser. Therefore, we need to address and adapt our implementation to work in a client-server architecture. Since our intention is to experiment with Statecharts within a browser, it is necessary to transform and instrument our model and the generated code with backward links to enable the model, when simulated on a server, to be animated in the browser. Figure 4 shows a snapshot of a sample model being simulated and animated in a browser in *step into* mode.

3.5. Modelling Method

The experimentation environment is modelled explicitly (described further in Section 4.). We model the experimentation model on top of the model to be simulated using Statecharts. The model artifact itself is a Statechart (see sample model in Figure 5). We use model transformation to generate the experimentation model which adds extra states and transitions to model the simulation environment. The initial Statechart model is moved to a composite state, *Running*, and

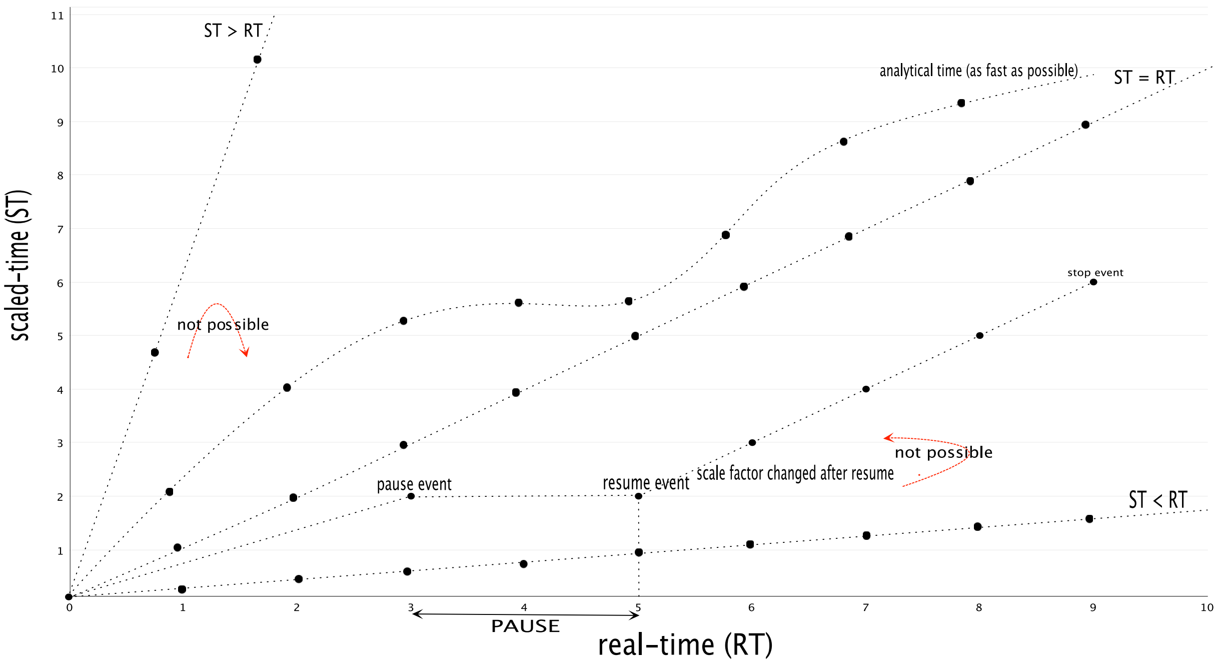

Figure 3. Different notions of time

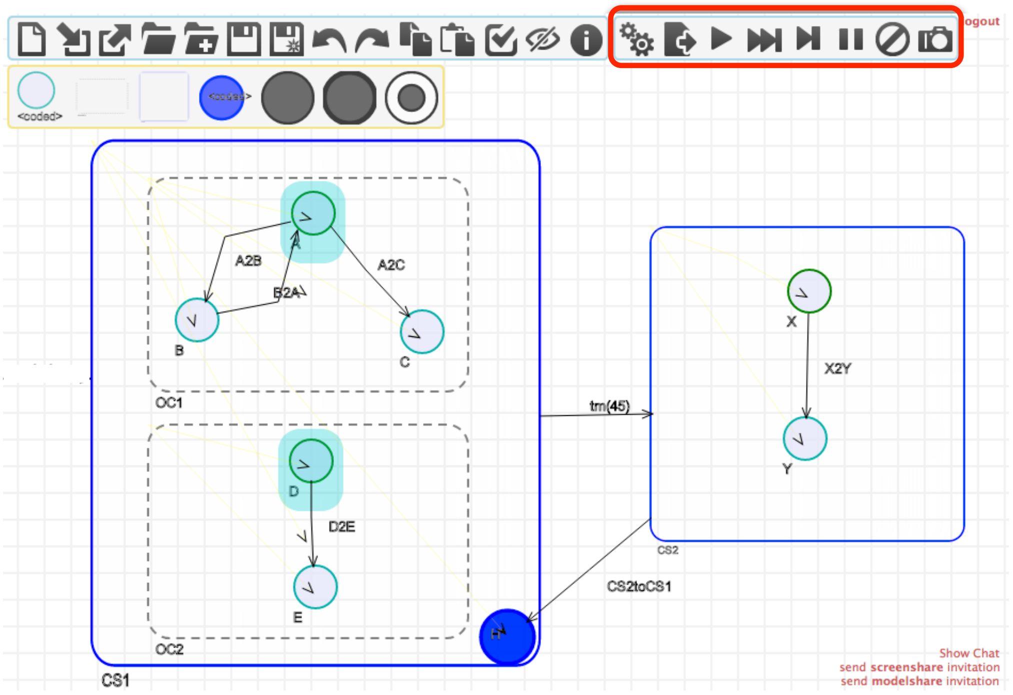

Figure 4. Simulating Sample Model in Experimentation Environment

extra states *Ready*, *Pause*, and *Done* are added to the instrumented model. Figure 6 shows the instrumented Statechart for the model in Figure 5. The instrumented model is subsequently processed by a compiler which generates the running system.

An alternate way of modelling the experimentation environment would be to model the states of the simulator (ready, running, paused) as an orthogonal component, with the model contained in a second orthogonal component. Events can be broadcast based on received pause/resume events which triggers the model to move to the next state. This allows the experimentation environment to have a clear separation from the actual model artifact, and makes it easier to plug in the environment with other formalisms. More details of the implementation are discussed in Section 4.

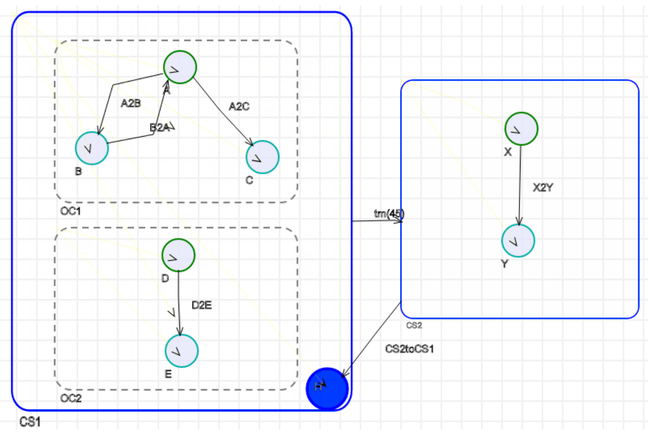

Figure 5. Sample Statechart Model

4. DESIGN AND IMPLEMENTATION OF AN EXPERIMENTATION ENVIRONMENT IN ATOMPM

We use metamodelling and model transformations to implement our modelling and simulation environment using our tool, AToMPM [6], A Tool for Multi-Paradigm Modelling. AToMPM rigorously applies the “model and conforming meta-model” workflow to all facets of domain-specific modelling. It allows modelling of language syntax (abstract and concrete) and semantics. The tool supports rule-based graph transformations and pre- and post-condition pattern languages to allow specification of model transformations. AToMPM consists of clients running in a web browser and a server and provides support for real-time, distributed collaboration.

4.1. Experimentation Model

The experimentation environment itself is a Statechart model that conforms to our Statechart language. Our Statecharts modelling environment has been meta-modelled in AToMPM by explicitly modelling the abstract syntax and the concrete syntax. The model to be simulated is first modelled using this environment. For simulating the model, it is transformed to an instrumented experimentation model by rule-based graph transformation. The model is instrumented with the states: Ready, Running, Paused, and Done (as discussed in Section 3.5.). The model to be simulated is incrementally added to the container, Running, and a deep history state (H*) is included inside the container since we need to return to the prior state when a user pauses and resumes the simulation. Due to space reasons, details of the transformation and rules are not included here.

4.2. Statechart Simulation

We transform our Statechart model in AToMPM to the DCharts Textual Syntax (DES) [3] using model-to-code generation techniques. We have created a plugin for AToMPM to generate the DCharts description file. A code snippet of the generated DES is presented here. We then use SCC (StateChart Compiler) [3] to generate Python source code from the Statechart descriptions in DES format. SCC generates efficient Java, Python, and C++ source code from DES. The Python source is then executed to simulate the model. The above outlined steps are all carried out at the server side. The user does not need to be aware of the technicalities of DES and SCC.

```
...
OPTIONS:
  InnerTransitionFirst = 1


STATECHART:
  CS1 [HS*]
    OC1 [CS]
      B
      A [DS]
      C
    OC2 [CS]
      D [DS]
      E
  CS2
    X [DS]
    Y
...
```

```
...
ENTER:
  N: CS1
  O: [DUMP("CS1 entered")]

EXIT:
  S: CS1
  O: [DUMP("CS1 exited")]
...

TRANSITION:
  S: CS1.OC1.A
  N: CS1.OC1.B
  E: A2B
  C: 1
...
```

4.3. Modelling and Simulation Environment

4.3.1. Toolbar and Execution Modes

We have modelled the toolbar as a buttons model in AToMPM. The toolbar provides an interface via which the user can give the commands: start simulation (in step into mode), pause simulation, resume simulation, switch to step over mode, switch to step out mode, and stop simulation. The toolbar also includes buttons to generate DES code and to export models as SVG images.
Figure 4 presents a snapshot of our simulation environment in AToMPM. The Statecharts toolbar is shown within a red-bordered rectangle.

4.3.2. Steps and Breakpoints

For debugging, we instrument the DES code with extra *Enter* and *Exit actions* to animate states in each simulation mode (details in Section 4.4.). In the *step into* mode, only the basic states are animated which requires the Enter/Exit actions of the basic states to be instrumented with extra code. In the *step over* and *step out* mode, the composite states are animated and the details within are made invisible.

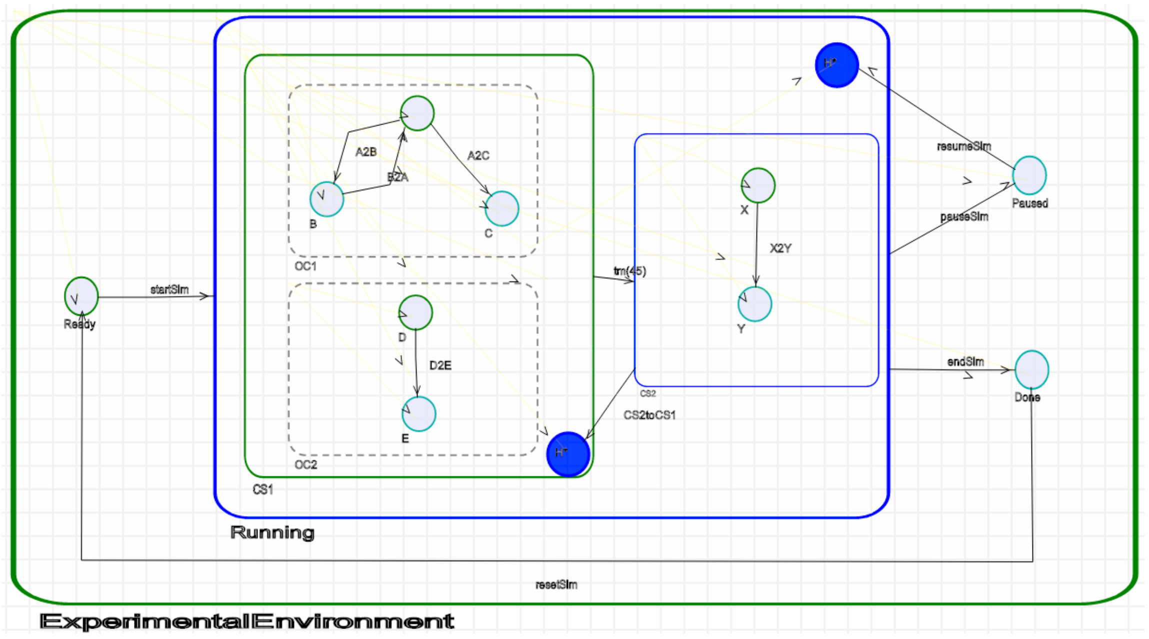

Figure 6. Instrumented Statechart Model

Users can also choose to run the simulation in continuous mode (step into) and specify a breakpoint to terminate or pause the simulation. The breakpoint is input via a dialog box on the client-side which is displayed to the user when he presses the simulate button in the toolbar. The user has the option of specifying a breakpoint as a state set, a condition, or a specific time, or "none". When the condition is satisfied, the simulator state changes to "paused".

4.3.3. Time

When the simulate command is given, a user dialog is started to take inputs for scaling time. In the instrumented DES model, the timed transitions are modified and the time is multiplied by a scale factor (t * scale factor).
It is also possible to change the scale factor after a pause event. The instrumented code gets the scale factor from a variable, and when we pause we tweak that variable.
For the time being, we only accept user input in real-time.

4.4. Animation in Client Server Environment

We use remote APIs defined in AToMPM to animate the models remotely which forwards specially formatted HTTP queries targeted at the back-end to the client. A code snippet from the instrumented DES code is shown here.

```
INITIALIZER:
  import httplib, json
  method = 'PUT'
  host = 'localhost:8124'
  uri = '/GET/console?wid=65'
  timeout = 5000

  def highlightState(nodeId):
    headers = {'Content-Type': 'text/plain'}
    data = json.dumps({'text':'CLIENT_BDAPI
      :: {"func":"_highlight","args":{"asid":'+str(nodeId)
      + ',"followCrossFormalismLinks":"*"}}'})
    conn = httplib.HTTPConnection(host)
    conn.request(method, uri, data, headers)
    resp = conn.getresponse()
    conn.close()
...
ENTER:
  N: CS1.OC1.B
  O: highlightState(3)

ENTER:
  N: CS1.OC1.A
  O: highlightState(4)

ENTER:
  N: CS1.OC1.C
  O: highlightState(5)
...
```

5. RELATED WORK

Many simulation environments and tools currently exist. However, their development was mostly adhoc. The simulation tools are often independent of the modelling environments and are implemented using some programming language. Moreover, very few works have addressed the need for an experimentation environment for carrying out simulations. There remains a need to have explicitly modelled simulation environments integrated with the modelling environment.
Traore and Muzy [9] discuss the need to be able to differentiate between the model and its context, and propose to formalize the context using experimental frames. Similar to the separation of concerns in modelling and simulation, the authors suggest that experimental frames should also be defined

according to the activity involved. They form a coupled relationship between the model and the frame, and between the simulator and the experimenter, and then define a mapping of the framed model to the simulator context. Similarly, Daum and Sargent [1] adapt the experimental frame concept to fit discrete event simulation, and implement it in their modelling and simulation system, HiMASS-j. Their modelling environment allows for structural and parametric changes to be made during simulation without the need to recompile the model. While we have been inspired by experimental frames and consider them in our work, we have not looked into formalizing experimental frame in such a manner. Our focus has been on creating a simulation environment based on the experimental frame idea, and to extend it with debugging facilities.

Shang and Wainer [8] extend traditional static-structure simulation and implement an experimental environment that allows dynamic structure changes to DEVS models during the simulation process. While our high-level goals are similar, namely to be able to experiment with models during simulation, our simulation model is of static nature and the purpose of our simulation environment is primarily to be able to run and test or debug the model in different modes.

Mannadiar and Vangheluwe [7] addressed the need for debugging model transformations and synthesized applications, and proposed the mapping of debugging activities from the programming domain to model-based design. They focus on domain-specific modelling, and do not address simulation-level concerns and needs.

6. CONCLUSION AND FUTURE WORK

We have proposed a simulation environment for Statecharts using Statecharts in a client-server environment. We have described our mapping of debugging and experimental frame concepts to the simulation domain.

We have explicitly modelled the Statechart language, the modelling environment, and the experimentation environment using AToMPM. To begin with, the abstract syntax and the concrete syntax of Statecharts have been meta-modelled. Once a model is created, it can be simulated by first transforming it to an instrumented model using graph transformation. This creates the experimentation model, which needs to be executed to start the simulation process. The interface allows users to pause, resume, and stop simulations. The environment also allows users to simulate the model in several modes, *step into*, *step over*, and *step out*, and also provides the option of setting breakpoints. The simulation can be run in real-time or scaled real-time as specified by the user. We use the SCC compiler and simulator to simulate our models. SCC runs together with the AToMPM kernel. We also have a dedicated user interface on the client side. The Statechart model to be simulated is instrumented with extra actions to allow the Statechart in the browser to be animated during simulation.

We believe our experimentation environment can be useful if applied to other formalisms, such as Causal Block Diagrams and DEVS. We intend to provide support for simulating models in formalisms other than Statecharts (and their combinations) within our environment. Moreover, we are also interested in looking at extending the work to allow tools instead of users to interact remotely with the environment. Ultimately, it should be possible to model our experimentation environment as a means to do tool integration.

REFERENCES

[1] T. Daum and R. Sargent. Experimental frames in a modern modeling and simulation system. *IIE Transactions*, 33(3):181–192, 2001.

[2] H. Ehrig and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999.

[3] T. H. Feng. DCharts, a formalism for modeling and simulation based design of reactive software systems. M.Sc. dissertation, School of Computer Science, McGill University, 2004.

[4] H. Giese, T. Levendovszky, and H. Vangheluwe. Summary of MPM 2006 Workshop. In *MoDELS'06*, pages 252–262, 2006.

[5] D. Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, May 1988.

[6] R. Mannadiar. *A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling*. PhD thesis, McGill University, June 2012.

[7] R. Mannadiar and H. Vangheluwe. Debugging in domain-specific modelling. In *Software Language Engineering*, SLE'10, pages 276–285, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] H. Shang and G. Wainer. Dynamic structure DEVS: Improving the real-time embedded systems simulation and design. In *ANSS*, pages 271–278, 2008.

[9] M. K. Traoré and A. Muzy. Capturing the dual relationship between simulation models and their context. *Simulation Modelling Practice and Theory*, 14(2):126 – 142, 2006.

[10] B. Zeigler. *Theory of Modelling and Simulation*. A Wiley-Interscience Publication. John Wiley, 1976.

[11] B. Zeigler, H. Prähofer, and T. Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.