Problem Definition

The systems we analyse, design, and develop today are characterized by an ever growing complexity. Modelling and Simulation (*M&S*) become increasingly important enablers in the development of such systems. Building models, often abstractions, of the system, that can be checked, simulated, and optimized before any realization of the system is made allows for a tremendous decrease in cost, while increasing the overall quality of the delivered system. Furthermore, it is often also possible to synthesize parts of the system from these models.

The M&S approach can only be successful if the modeller (often a domain expert, such as an automotive engineer) has access to advanced tools which enable the creation of models and provide the necessary framework for performing simulation and deployment of models onto hardware. Core activities include the initialization of the system with a set of values, results collection (after simulation has been performed), and acceptance or rejection of simulation results according to predefined simulation objectives. This framework, called the Experimental Frame (*EF*), was introduced by Zeigler [1]. Traoré and Muzy [2] use EFs to make a clear distinction between what is inherent to the model, which is an abstraction of a (sub)system to be analysed, and its context, which contains all information needed to run the simulation. Their technique, using EFs, allows one to reuse a single context in which to simulate different models, and conversely, simulate a model in different contexts.

This separation of concerns is a necessary feature for modelling environments, to allow the rapid development and simulation of a number of candidate models. The environment should also allow the modeller to have sufficient *control* over the simulation execution. When the model being simulated is rejected by the EF for not meeting a simulation goal the modeller will be interested in a more detailed view on the execution of the simulation. In traditional, code-based, software development, a developer has access to various tools for analysing a piece of code (also a model, or specification), such as a debugger. Debuggers allow to pause the execution of a program at certain points, step through instructions, inspect variables, and much more [3]. Mannadiar and Vangheluwe [4] survey the current state-of-the-art in debugging and explore how these concepts can be translated to the realm of Domain-Specific Modelling.

Many modelling languages are in common use today. One the one hand, general-purpose languages, such as Petri nets [5], Causal Block Diagrams (also known as Synchronous Data Flow) [6], Statecharts [7], and Modelica [8] have well-defined semantics and can be used for modelling a wide variety of systems. On the other hand, domain-specific languages are created by a language designer for a specific domain. Their semantics are often defined by mapping onto a formalism with known semantics.

There is a need for a set of general debugging concepts, methods, techniques, tools, and processes that can be applied to general-purpose, as well as domain-specific modelling languages. Furthermore, there is an industrial need for advanced experimentation and debugging environments that are based on these debugging concepts.

Strategic Goals

The high-level goal of this project is to provide techniques for tackling the ever growing *complexity* of *multi-formalism modelling, simulation, and deployment environments* as found in *industrial* applications. This can be broken down into a number of sub-goals:

- I will investigate current *best-practices* in debugging traditional software development, as well as techniques used for experimentation and (model) simulation.
- I will provide *mechanisms* to debug and control the model simulation process. In particular, I want to give the same level of control a developer has when debugging a traditional software component. The debugging of software components differs significantly from debugging the simulation of a model, however. A model is created in a modelling formalism, or in the case of multi-paradigm modelling, a number of formalisms. The semantics of these formalisms include non-determinism (for example, Petri nets), concurrency (for example, Statecharts), and different notions of time: some formalisms are based on discrete-time, some on continuous-time. Furthermore, the simulation itself can be run in (scaled) real time, or as-fast-as-possible. As such, I want to create specific debugging concepts that account for these semantics.
- I will develop *prototypes* that implement these mechanisms. These prototypes will be developed for different formalisms (and combinations of formalisms), different users (*i.e.*, to give different levels of control to different types of users) and platforms (including as-fast-as-possible simulation, scaled real-time simulation, hardware-in-the-loop, software-in-the-loop, and deployed systems).
- I will *validate* the developed techniques on industrial cases, working together with industrial partners. The focus will be on tool builders, such as LMS and Triphase.

My approach will build on the concept of experimental frames, which was introduced in [1], further developed in [2], and implemented in a simulation environment in [4]. There has been some work on creating debuggers for domain-specific languages [9, 10] and one paper that focuses on debugging model simulation by explicitly modelling the debugging environment [11]. This, and anecdotal evidence from industry, shows that there is a clear need for advanced debugging and simulation environments. Research on this subject is still in an early stage.

I will perform my research in the Modelling, Simulation and Design Lab (MSDL), part of the Antwerp Systems and Software Modelling (AnSyMo) group. This research unit has a world-class reputation in Model-Based Systems Engineering (MBSE). MSDL develops methods, techniques, and tools for the development of complex, software-intensive systems in a wide variety of domains (including automotive, mechatronics, and mobile applications). MSDL focuses on model transformation, (visual) syntax-direct modelling environments, distributed simulation, and co-simulation. This project is complementary to their activities; it investigates a current challenge in the field: how to develop advanced simulation environments. Solutions are developed using core competencies (such as model transformation) of the group. MSDL has experience in (co-)simulation and collaborates with industrial partners, including international world leaders such as The MathWorks, General Motors Research, and IBM Research. As such, MSDL is the ideal environment for me to carry out my research.

Project Description

The goal of this project is to create advanced environments for the debugging and simulation of models, as described in the previous section. In the sequel, these goals will be broken down into sub-goals, and possible solutions will be proposed.

Defining Debugging Concepts for Model Simulation

A first goal is to construct a set of debugging concepts for a number of well-known general-purpose modelling formalisms: Petri nets, Statecharts, Causal Block Diagrams (CBDs), DEVS [1], Modelica, and rule-based model transformations (MoTif [12] in particular). I will start from the well-known debugging concepts in traditional, code-based software development, such as breakpoints, step into/over/out, pause, and resume [3]. I will also include simulation-specific concepts, in particular the notion of time. A simulation can be run in (scaled) real-time, or as-fast-as-possible. Furthermore, different formalisms have different notions of time: continuous, or discrete.

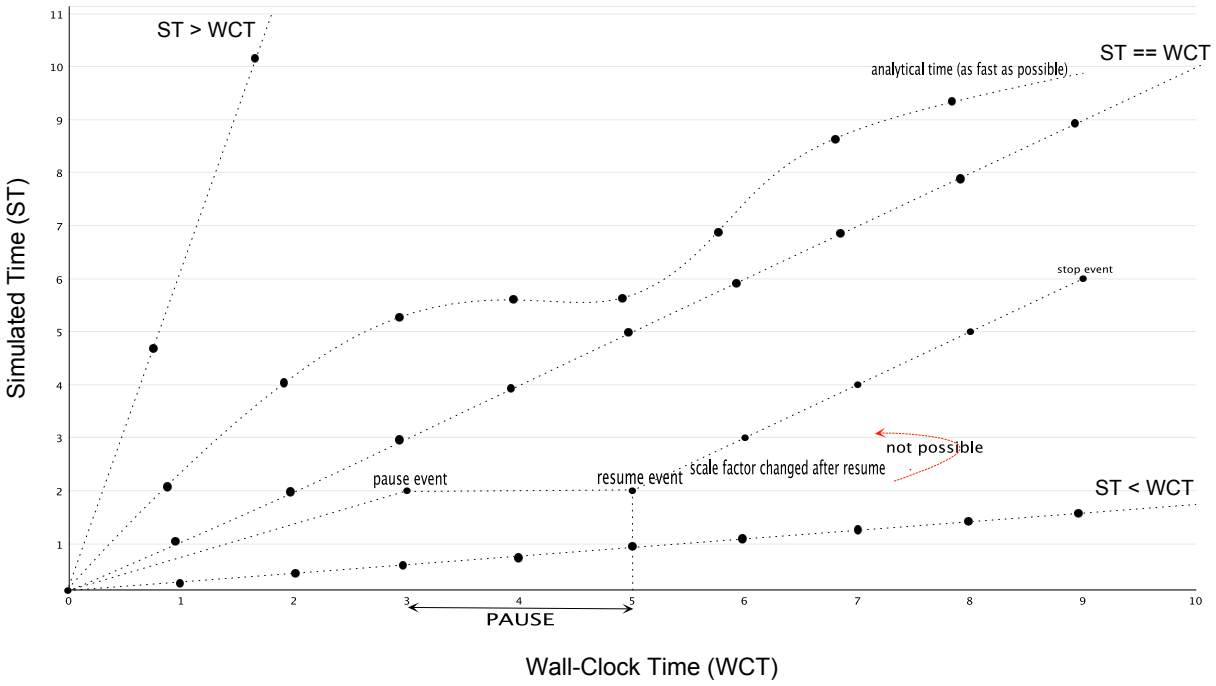


Figure 1: Different notions of simulated time.

The concept of simulated time is visually represented in Figure 1. Simulated Time (ST) is related to the Wall-Clock Time (WCT), which is the time of the physical reality. Four types of relationships between WCT and ST are distinguished:

- Real-Time: this is the case when the simulated time advances in lockstep with the wall-clock time. This implies that the simulation steps have a hard real-time deadline: *i.e.*, the values of the runtime variables have to be computed before the WCT reaches the ST.

- Scaled Real-Time: when the simulation needs to run faster or slower than the WCT (for instance, simulating the growth of the world population over a period of years), a multiplication factor ensures that the ST proceeds faster or slower than the WCT. ST and WCT remain linearly related, however.
- As-Fast-As-Possible: in this case, the simulation is run as fast as possible. As a result, there is no predefined relationship between ST and WCT. The ST is just a variable in the simulation.

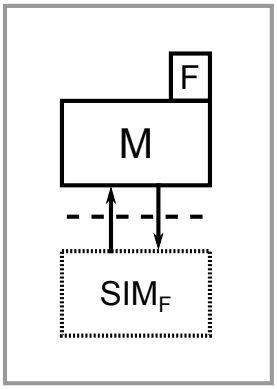
As can be seen from the figure, the ST might be paused (when we pause the simulation). When resuming, the user might want to change the simulated time (by changing the scale factor, for instance). We cannot go back in time in the physical world. We can go back in simulated time, however. A use-case is the boundary-crossing problem, known in hybrid systems, where we look for the point in time when a function crosses a certain value. As time goes on, the simulator checks whether the value is larger than the defined threshold. When the value is larger, it will be necessary to go 'back in time' to look for the actual point at which the value crossed the boundary. Both these operations on simulated time, of course, add to the complexity of the debug operations.

Once this set of debugging concepts is defined, they will be transposed to each formalism, *i.e.*, the semantics of each debugging operation will be defined for each modelling formalism whose models we wish to debug. The constructed set of debugging operations is the same for all formalisms, which leads to the construction of an single explicit model of the reactive behaviour of a debugger. This debugger will react to user requests and will control the state of the simulator. Of course, this model is meaningless on its own. It has to be combined with the original simulator to create an instrumented simulator that implements the debugging functionality.

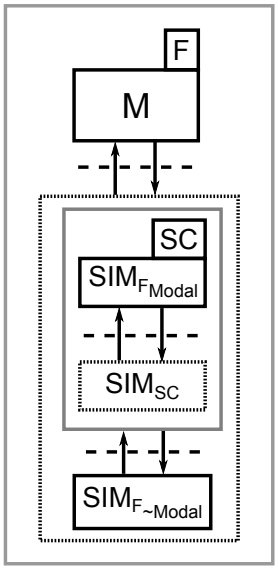
De/Reconstructing the Simulator

Current simulators are most often code-based. The result of the previous goal is a single explicit model of the reactive behaviour of a general debugger for simulation. This model has to be combined with the the implementations of simulators for the different formalisms (Petri nets, CBDs, Statecharts, DEVS, Modelica, and rule-based model transformation). Relying on traditional, code-based, software development processes makes this a challenging task, as concurrency, non-determinism, and different notions of time all have to be taken into account. In this part of the project, I will deconstruct the simulator for each formalism, extracting the "modal" part. In general, each simulator has a "main simulator loop". This comprises a number of states and transitions between them, performing some action that updates the state of the model. In the most naive case, there is a single state which represents the main loop of the simulator and a transition going from and to that state, each time performing one simulation "step". The most appropriate formalism to describe this behaviour is a state machine, more specifically a Statechart. This, as it allows the description of both reactive and autonomous (timed) behaviour.

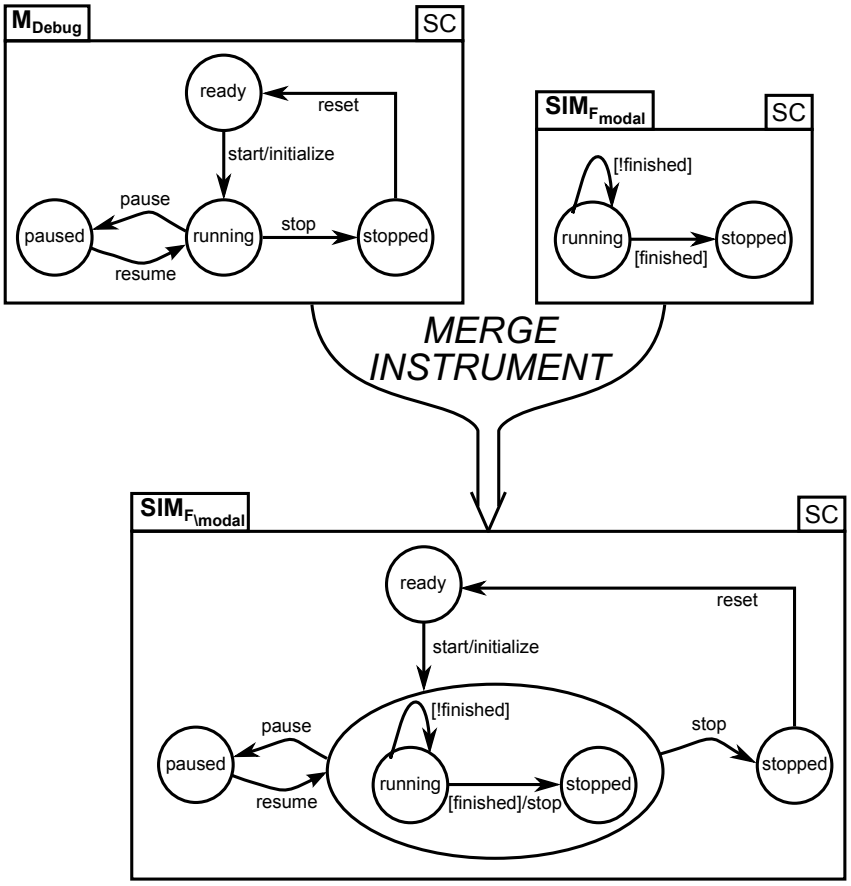
The workflow for de/reconstructing the simulator is shown in Figure 2.
In Figure 2a, a model created in a formalism F and the simulation kernel for formalism F are



(a) A model in formalism F and a simulation kernel for F.



(b) De/Reconstructing the simulator.



(c) Merging the debugging concepts with the modal behaviour of the simulator.

Figure 2: The workflow for explicitly modelling the simulator's behaviour.

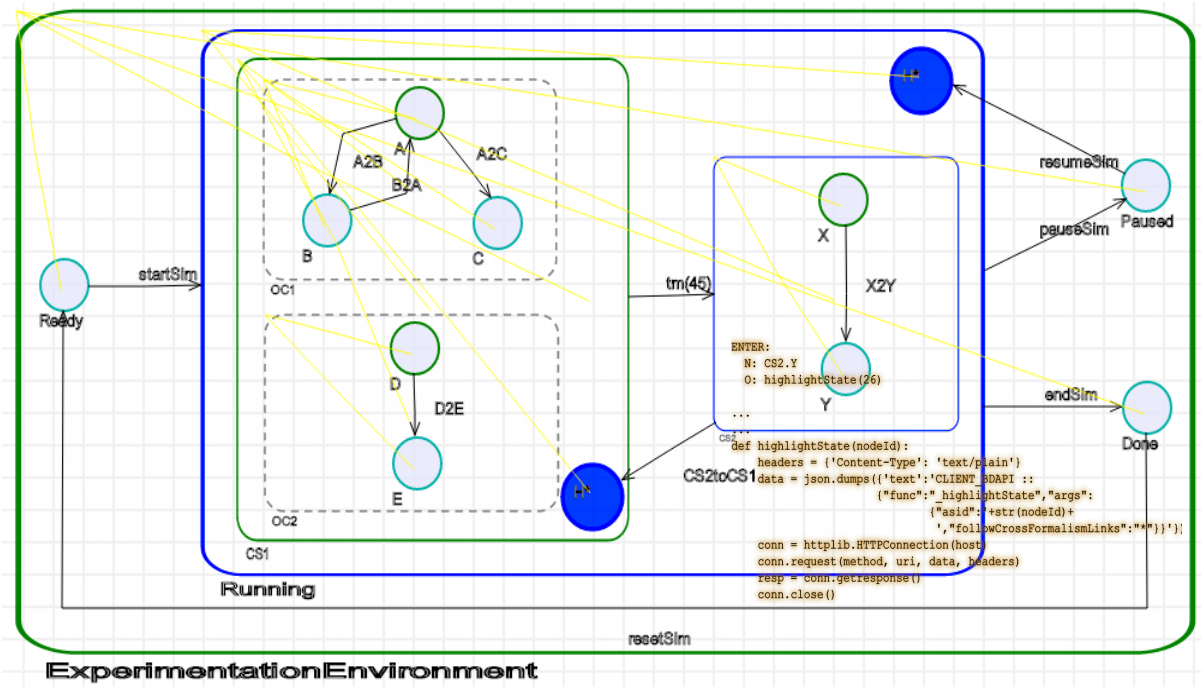


Figure 3: The instrumented Statechart model [11].

shown. The simulation kernel interacts with the model through an interface (shown as a dashed line in the figure), modifying and querying its state. This combination of model and simulation kernel can be seen as a black-box, which is given input signals and produces output signals.
The de/reconstruction process is shown in Figure 2b. The first step, deconstrucing the simulator, extracts the modal part of the simulator in a Statechart (SC) model called $SIM_{F_{modal}}$. This model is combined with a Statechart simulator, interpreter, or compiler called SIM_{SC} to give it operational semantics. The Statechart together with its executor interface with the non-modal part of the simulator for formalism F ($\text{SIM}_{F_{\setminus modal}}$, which, in this case, consists of the coded functions to run the simulator). To implement this interface, I will use the FMI standard [13]. The combination of the modal and non-modal part of the simulator results in a behaviourally equivalent simulator to SIM_F. From the user's point of view, the black-box containing the model to be simulated and its simulator is unchanged.
In Figure 2c, the last step in creating an instrumented simulator is shown. We *merge* the modal part of the simulator for F with the general behavioural model of the debugger. This results in an instrumented model of the modal behaviour of the simulator. The last step is to replace $SIM_{F_{modal}}$ in Figure 2b with this instrumented model. Again, this should not change the behaviour of the simulator in any way if the user does not make use of the debugging functionality. Extra behaviour has been added, but running the simulator as before is still possible. In the example shown, the debugger includes the concepts of *start*, *pause*, *resume*, and *stop*. The simulator only has two states: *running*, and *stopped*. It runs the main loop of the simulator until the *finished* condition is satisfied, signalling that the simulation is done. These are trivial examples; the models that will be used in the project result from the previous steps described above: defining a set of debugging concepts for the debugging model, and extracting the modal part of the simulator (for each formalism).

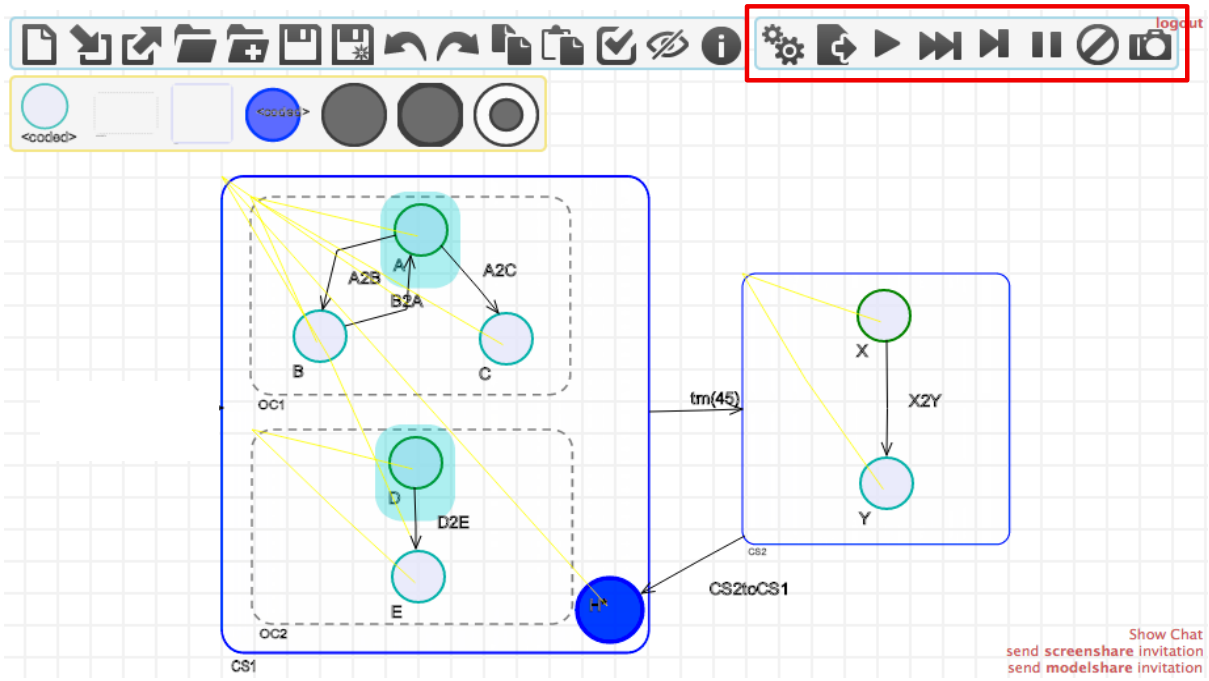


Figure 4: The generated Statechart debugging environment [11].

From this instrumented model, a simulation environment can be automatically synthesized. This allows the user to control the simulation process by sending the correct events to the instrumented simulator. A first prototype, validating the approach for debugging the simulation of Statechart models, was created by Mustafiz and Vangheluwe in [11]. They instrument the Statechart model with debugging concepts and automatically generate a simulation environment in the AToMPM tool. The instrumented model is shown in Figure 3. Four states are added: the *Running* state, which contains the Statechart model, and the *Paused*, *Done*, and *Ready* states. A deep history state is added to keep track of the state the simulator is in when the user pauses/resumes. Their approach only works for Statechart models, as a model in any other formalism cannot simply be embedded in this way. The approach proposed for this project is a generalization of their work. I propose to embed not the model to be simulated, but a Statechart model of the modal part of the simulation kernel. This will work for any formalism.
The generated visual experimentation environment is shown in Figure 4. As can be seen, a toolbar exposing the debugging actions to the user is added at the top-right. The model that is shown in the environment is the original model the user has created, but behind the screens the instrumented model is executed. Thus, exposing the implementation of the debugger to the user is avoided.

The instrumented simulator, because it is explicitly modelled, becomes analysable. It is thus possible to prove properties such as liveness and deadlock. I plan to do this using the tool UPPAAL [14].

Up to this point, I have only explained how I plan to implement debugging techniques for simulating models created in a single formalism. Once the debugging techniques have been validated for single-formalism models, I want to extend them to multi-formalism models. In

that case, multiple simulators (one for each formalism) are in charge of simulating the model. These simulators will have to be combined, and embedded in the debugger.

Instrumenting the Model

Once the de/reconstruction of the simulator is successful, the user is able to control the simulation process, but the model that is simulated needs to be instrumented further for two reasons:

1. Certain debugging concepts are formalism-dependent and the user should be able to add these constructs to the model to control the debugging process further. A prime example is the breakpoint. The concept of a breakpoint needs to be defined at the level of the formalism and exposed to the user.

2. While controlling the debugging process is important, visual information on the state of the simulator should also be presented to the user. This can for example be realised by highlighting pertinent parts of the model (in case of explicit discrete states) or showing continuous signal plots in case of CBDs.

The first point will be solved by introducing debugging concepts for each formalism, as a result of the previous work on defining debugging concepts for model simulation. This will allow the user to, at the level of the formalism, define breakpoints. The semantics of the breakpoint should be implemented in the state machine that describes the behaviour of the debugger.

The second point was included in the prototype by Mustafiz and Vangheluwe. The instrumented Statechart model contained (Statechart ENTER) action code to highlight the active state (in the original model) of the Statechart model. This can be seen in Figure 4, where the active states *A* and *D* are highlighted. Whenever the state changes, the code is executed to highlight the new active state. This same technique will be used to instrument models in different formalisms. Even further, I will generalize this concept to the following cases:

1. Domain-specific languages (DSLs). The semantics of a DSL can be defined by mapping it onto a general-purpose modelling formalism (which, in the limit, can be action code such as Java or C++). A naive approach to debugging models in a DSL would be to debug the model in the underlying semantic domain. The user of a DSL, however, wants to debug at the level of the DSL. In [9, 10], the authors look into the challenge of debugging at the level of the DSL. By generating traceability links, the underlying semantic model can be executed or simulated, while calling back to the DSL concepts.

2. The semantics of a DSL can also be defined as an in-place rule-based model transformation, effectively creating a simulator that executes the model. This model transformation can be debugged at the level of its rule-based definition (which may be appropriate while creating the transformation, to validate its correctness). A user executing the transformation, however, wants to track the state of the model during the simulation on the level of the DSL. Again, traceability information has to be generated as part of the model transformation, such that there is a callback to the DSL concepts.

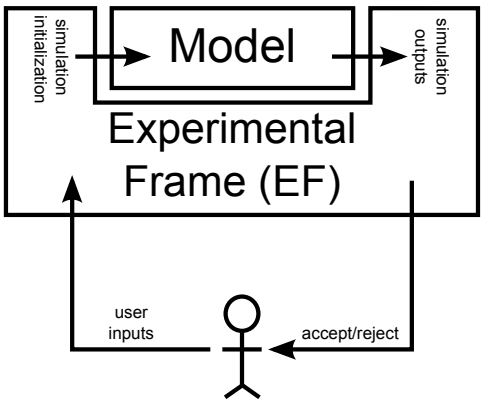


Figure 5: A high-level view of the Experimental Frame. Adapted from [1].

3. A model, once validated, will be deployed to a physical system. This deployment might involve a compilation to binary code, or the physical system might contain a simulator that is capable of simulating the model as-is. In this case, we need to generate traceability information as well, to be able to control and follow the simulation process in our simulation environment.

This instrumentation will be performed by automatic, explicitly modelled model transformations.

Experimental Frames

A high-level view of simulation activities is given by the Experimental Frame (EF) [1]. An EF contains all information needed to run a sequence of experiments on a model. It contains the goals of an experiment (*i.e.*, the criteria for a successful experiment) and all environmental specifications that are not part of the simulated model. The task of the EF is to initialize the model at the beginning of each simulation run, let the simulator execute the model, and check whether the outputs meet the experiment goals. A visual representation of the EF is given in Figure 5.

The user interacts with the EF by defining goals, and initializing the EF with a model to be simulated. He then gets an accept/reject message from the EF, depending on whether the simulated model meets the simulation goals.

The concept of EFs has been explored in the past, and some attempts have been made to implement such a system [4]. However, EFs need to be further explored, and in particular, their semantics have to be defined rigorously. This work on debugging can be used to enhance EFs with a debugging framework. I then assume that the reactive behaviour of the EF is explicitly modelled. In a sense, it is a simulator in its own right, with its own time-base, where each step of the simulator is the initialization and simulation of the model, and output checking. I plan to transpose my debugging concepts to the EF formalism.

Schedule

The schedule is divided according to the research activities distinguished in the previous section. Each activity leads to a *work package* that consists of a number of *phases*, each phase having a deliverable or milestone. The result is a repeated process of a number of steps, resulting in a number of publications and reports, which are combined at the end to form the PhD thesis. In each work package, there is a phase dedicated for collaboration with the industry. This is to understand the challenges the industry has with respect to model simulation. This knowledge will guide me during the other phases.

I distinguish two main work packages: first, there's the definition of concepts for simulation debugging. Then, I will implement those concepts in a series of prototypes. Each prototype will implement the techniques for a specific formalism. At regular intervals, I plan to write workshop or conference papers, as well as journal papers. For details, refer to Table 1.

Work Package	**Phase**	**Time Estimate**
Defining Concepts for Simulation Debugging	Industrial Input	**1 month**
	Foundations and Formalisation	**3 months**
	Prototype Implementation	**4 months**
	Workshop/Conference Paper	**1 month**
De/Reconstructing the Simulator and Instrumenting the Model	Industrial Cooperation	**2 months**
	Foundations and Formalisation	**3 months**
	CBD Implementation	**2 month**
	Statecharts Implementation	**2 month**
	DEVS Implementation	**4 months**
	Petri nets Implementation	**4 months**
	Workshop/Conference Paper	**1 month**
	Journal Paper	**1 month**
	Modelica Implementation	**4 months**
	Workshop/Conference Paper	**1 month**
	Journal Paper	**1 month**
	Model Transformation Implementation	**4 months**
	Workshop/Conference Paper	**1 month**
	Journal Paper	**1 month**
	Industrial Validation and Refinement	**5 months**
	Workshop/Conference Paper	**1 month**
Writing Thesis		**2 months**
	Total	**48 months**

Table 1: Work Packages and Time Table

Applications

The result of this project will be a collection of methods, techniques, and tools for creating advanced simulation environments, by explicitly modelling their behaviour. There are two types of users that are interested in my results:

1. Tool builders, that can incorporate the techniques in their tools. The techniques will decrease the development cost of creating and maintaining visual debugging and experimentation environments, and increase the functionality they offer. Interested companies include:

   (a) The MathWorks, a world leader in modelling and simulation software for the model-based design of software-intensive systems. They provide debugging capabilities for their Simulink tool, but have discovered the limitations of a code-based approach. They need techniques for the model-based development of simulation and experimentation environments.

   (b) LMS, a multi-national company located in Leuven that specializes in virtual simulation of systems in different application domains, such as mechatronics, automotive, and physical systems. They provide advanced modelling languages such as AMESim and can benefit from the developed techniques to enable the debugging of simulations.

   (c) TriPhase, located in Heverlee, specializes in tools for Rapid Control Prototyping (RCP), such as the Real Time Target, and Hardware in the Loop (HiL) simulation.

2. Companies that use modelling and simulation tools, and are able to build extensions to those tools according to their needs. They will greatly benefit from the techniques developed during this project, as they can use them to create simulation environments according to their specific needs. Examples of such companies are Punch Powertrain, Dana Spicer Brugge, Verhaert, Luperco, and Bombardier Brugge.

3. Other domains that benefit are home automation (with the Antwerp-based company Fifthplay) and mobile application development.

From the very onset of the project, the goal is *adoption* of the developed methods, techniques, (prototype) tools and processes in the Flemish industry. During each phase of the project, interaction with companies is planned as to align industrial needs and scientific developments.

Using the work of Broy et al. [15], an estimate of the quantitative impact of this project can be given. 20% of the budget developing an embedded system is spent on testing/debugging. This project will have a direct impact on those development processes. The debugging of models of the embedded system will be facilitated, enabling rapid prototyping and as such, realizing a decrease in cost.

This project will also greatly contribute to the work of MSDL. The ultimate goal of this research group is to turn MBSE into a true engineering discipline with appropriate methods, techniques, tools and processes. This requires a rigorous and complete development of all aspects of the

MBSE process, including the now missing techniques for debugging, simulation, and experimentation.

I will explicitly model the experimentation environment to support the debugging of model simulation. The advantages of this approach are manifold:

1. The semantics of the simulator and/or the experimenter are documented. This facilitates adapting the simulator to changing requirements.

2. The simulator, because it is explicitly modelled, becomes analysable. It is thus possible to prove properties such as liveness and deadlock.

3. The debug environment is automatically generated from its model.

For the technical implementation, I will use a prototype-based, incremental, and iterative development process. This reduces the risk, as this process is highly adaptive to change. It is also exhaustive, as we iterate over all goals set for this project and for each goal have a fixed work plan (see Schedule).

References

[1] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*. Academic Press, 2 ed., 2000.

[2] M. K. Traoré and A. Muzy, "Capturing the dual relationship between simulation models and their context," *Simulation Modelling Practice and Theory*, vol. 14, no. 2, pp. 126 – 142, 2006.

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

[4] T. Daum and R. G. Sargent, "Experimental frames in a modern modeling and simulation system," *IIE Transactions*, vol. 33, no. 3, pp. 181–192, 2001.

[5] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[6] F. E. Cellier, *Continuous system modeling*. New York: Springer-Verlag, 1991.

[7] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, June 1987.

[8] P. Fritzson and P. Bunus, "Modelica – a general object-oriented language for continuous and discrete-event system modeling," in *IN PROCEEDINGS OF THE 35TH ANNUAL SIMULATION SYMPOSIUM*, pp. 14–18, 2002.

[9] H. Wu, J. Gray, and M. Mernik, "Grammar-driven generation of domain-specific language debuggers," *Software: Practice and Experience*, vol. 38, no. 10, pp. 1073–1103, 2008.

[10] M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju, "TIDE: A generic debugging framework — tool demonstration —," *Electron. Notes Theor. Comput. Sci.*, vol. 141, pp. 161–165, Dec. 2005.

[11] S. Mustafiz and H. Vangheluwe, "Explicit modelling of statechart simulation environments," in *Summer Simulation Multiconference*, pp. 445 – 452, Society for Computer Simulation International (SCS), July 2013. Toronto, Canada.

[12] E. Syriani and H. Vangheluwe, "A modular timed graph transformation language for simulation-based design," *Software and Systems Modeling (SoSyM)*, vol. 12, no. 2, pp. 387–414, 2013.

[13] "The Functional Mockup Interface." https://www.fmi-standard.org/. Accessed: 2013-09-12.

[14] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems," in *Hybrid Systems III* (R. Alur, T. Henzinger, and E. Sontag, eds.), vol. 1066 of *Lecture Notes in Computer Science*, pp. 232–243, Springer Berlin Heidelberg, 1996.

[15] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz, *What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?*, ch. 13, pp. 343–369. IGI Global, 2011.