# Semantics and Execution of Domain Specific Models

Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden
Christian-Albrechts-Universität zu Kiel
{cmot,haf,rvh}@informatik.uni-kiel.de

**Abstract:**

In this paper we present a two-level approach to extend the abstract syntax of models with concrete semantics in order to execute such models. First, a light-weight execution infrastructure for iteratable models with a generic user interface allows the tool smith to provide arbitrary execution and visualization engine implementations for his or her Domain Specific Language (DSL). We discuss how the common execution manager runtime allows co-simulations of different model types and engine implementations to provide a flexible framework in the diverse DSL scenery. Second, as a concrete but nevertheless generic implementation of a simulation engine for behavior models, we present semantic model specifications and a runtime interfacing to the Ptolemy II tool suite. As a project in the area of model simulation, the latter provides a mature sophisticated and formally grounded backbone for model execution.

We present our approach as an open source Eclipse integration to be an extension to the Eclipse modeling projects.

## 1 Introduction

Computer simulations are an established means to analyze the behavior of a system. On the one hand one wants to be able to predict and better understand physical systems and train humans to better interact with them, for example weather forecasts or flight simulators. On the other one aspires to emulate computer systems—often embedded ones—themselves prior to their physical integration in order to increase safety and cost effectiveness.

The basis for such a simulation is usually a model, an abstraction of the real world, carrying sufficient information to specify the relevant system parameters necessary for the

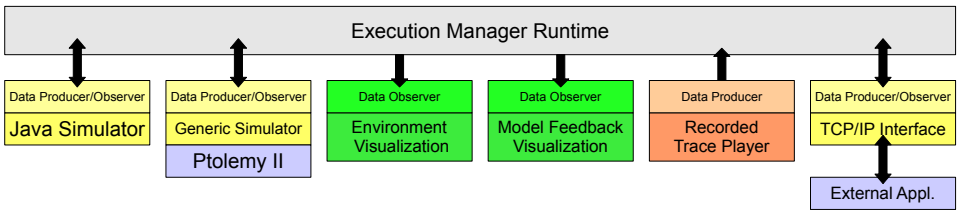


Figure 1: Schematic overview of the Execution Manager infrastructure

semantical analysis and execution. The notation of a model instance is a concrete textual or graphical syntax.

In the past all model editing, parsing, and processing facilities were manually implemented with little generic abstractions that inhibit interchangeability. Standardized languages, e. g., the Unified Modeling Language (UML), try to alleviate this, but they are sometimes too general and complex to be widely accepted.

As a recent development, *Domain Specific Languages* (DSL) target only a specific range of application, offering tailored abstractions and complying to the exact needs of developers within such domains. On the one hand, there are already well established toolkits like the *Eclipse Modeling Framework* (EMF) or Microsoft's DSL toolkit to define an abstract syntax of a DSL in a model-based way. They provide much infrastructure, such as a metamodel backbone, synthesis of textual and graphical editors, and post-processing capabilities like model transformations, validation, persistence, and versioning. The designer of tools for such a DSL, the tool smith, faces less efforts in developing his or her modeling environment. This is achieved by sophisticated tool assistance and possibly a generative approach. The latter provides, e. g., generated implementations for simple model interactions automatically and in a common and interchangeable way.

On the other hand there is the semantics of such a DSL. This additionally has to be defined in order to let a computer execute such models. For the specification of the latter no common way exists yet. But as such a semantics often exists at least implicitly in the mind of the constructor of a new DSL there is a need to provide a way for making it explicit.

The contribution of this paper is a proposal on how DSL semantics can be defined by using existing *semantic domains* and existing model transformation mechanisms without introducing any new kind of language or notation focusing on an Eclipse integration. Fig. 1 shows an example setup of the architecture. In Sec. 1.1, a survey about the KIELER framework, which is the context of the work presented in this paper, is given. A short overview about existing technologies in the area of simulations and semantic specifications is given in Sec. 2. In Sec. 3 we present how we define semantics for an example DSL with the Ptolemy II suite (cf. Fig. 1). In this context a case study about simulating SyncCharts by leveraging Ptolemy is presented. An implementation overview about our general approach of integrating simulations in the Eclipse platform is given in Sec. 4—the Execution Manager Runtime in Fig. 1. Additionally we show that this solution is extendable and has open support for, e. g., model analysis and validation or co-simulations. Sec. 5 concludes and gives an outline of future work.

### 1.1 The KIELER Framework

The execution and semantics approach presented in this paper is implemented and integrated in the *Kiel Integrated Environment for Layout Eclipse Rich Client* (KIELER)[1] framework. It is a test-bed for enhancing the *pragmatics*, i. e., the user interaction, of model-based system design as described elsewhere [FvH10].

[1] http://www.informatik.uni-kiel.de/rtsys/kieler/

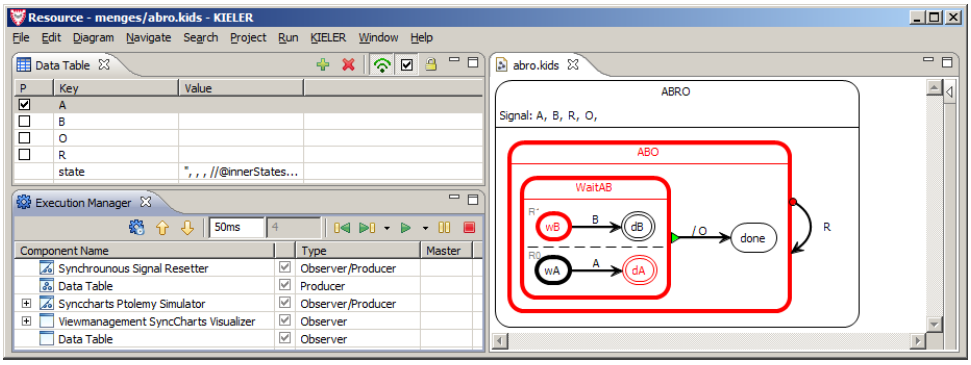


Figure 2: GUI of KIELER and the KIEM Eclipse plug-in during a simulation run

The KIELER framework is a set of open source Eclipse plug-ins that integrate with common Eclipse modeling projects, such as the Graphical Modeling Framework (GMF), the Textual Modeling Framework (TMF), and especially the modeling backbone EMF.

While Eclipse handles model syntax in a common and generic way, this is not yet done for semantics. Hence, before handling pragmatics of simulations for models in a generic way, we need to find generic interfaces and specification possibilities for semantics themselves. This is the purpose of the *KIELER Execution Manager* (KIEM), the execution approach presented in this paper.

Fig. 2 shows a simulation run with KIEM in KIELER which provides a user interface, shown in the bottom view in Fig. 2, and visual feedback about simulation details, both in the graphical model view itself and in a separate data table view.

## 1.2 Basic Concepts

A basic prerequisite for Model Driven Software Development (MDSD) are models that base on metamodels. The latter define the abstract syntax of models and hence allow the specification of languages as object-oriented structure models. The Meta Object Facility (MOF) is such a metamodeling framework defined by the Object Management Group (OMG) [Obj06], which has been taken shape for the *Eclipse* world as the Eclipse Modeling Framework (EMF) with its *Ecore* metamodel language that we use in the context of this project.

Model transformations play a key role in generative software development. These describe the transformation of models (i. e., metamodel instances) that conform to one metamodel into models which then conform to another or even the same metamodel. There are several model transformation systems available today that are well integrated into the Eclipse platform. Xpand[2] realizes a model to text template based approach, Xtend is a functional

[2] http://wiki.eclipse.org/Xpand

metamodel extension and transformation language based on Java with a syntax borrowed from Java and OCL, and there exist other transformation frameworks such as QVT or ATL.

In our implementation we will use the Xtend language, as it is a widely used and was refactored for a seamless integration into the Eclipse IDE. Additionally, its extensibility features allow to escape to Java for sequential or complex transformation code fragments. Nevertheless our approach is conceptually open to use any transformation language which supports EMF meta models.

## 2 Related Work

There exists a range of modeling tools that also provide simulation for their domain models. To just mention some of the popular ones: *Ptolemy II* is a framework that supports heterogeneous modeling, simulation, and design of concurrent systems. For integrated simulation purposes Ptolemy provides the *Vergil* graphical editor. But there also exists the possibility to embed the execution of Ptolemy models into arbitrary Java applications as described by Eker et al. [EJL+03].

With Matlab/Simulink/Stateflow of Mathworks and SCADE of Esterel Technologies the user is able to integrate control-flow and data-flow model parts in their own Statecharts dialect and data-flow language.

The Topcased project is based on the Eclipse framework and targets the model driven development with simulation as the key feature in validating models [CCG+08]. Other simulation supporting frameworks are Scilab/Scicos from INRIA, Hyperformix Workbench from Hyperformix, StateMate from David Harel, or Uppaal from Uppsala University.

Most of these tools are specific and follow a clear semantics. This allows such tools to provide a tailored simulation engine that can execute the models according to this concrete semantics. Ptolemy supports heterogeneous modeling and different semantics for and within the same model. However, Ptolemy has fixed concrete and abstract syntax and hence cannot be used directly to express arbitrary DSLs, where one reason to create them is to get a very specific language notation. Hence, we investigated it further as a generic semantic backend in combination with the Eclipse modeling projects as elaborated in Sec. 3.3.1.

As outlined by Scheidgen and Fischer [SF07] two fundamentally different concepts can be emphasized for specifying model semantics: (1) Model-Transformation into a *semantic domain* (denotational) or (2) provision of a new action language (operational). In the first case semantics is applied to a metamodel by a simple mapping or a more complex transformation into a domain for which there already exists an explicit semantical meaning. The second concept applies semantics by extending the metamodel with semantical operations on the same abstraction level. For this a meaning additionally has to be defined, e.g., in writing generic model simulators that interpret this information based on formal or informal specifications. The *M3Action* framework for defining operational semantics is illustrated by Eichler et al. [ESS06] or Scheidgen and Fischer [SF07]. Chen et al. [CSN07] presents a compositional approach for specifying the model behavior.

We follow the first approach by utilizing existing languages only and describe structure based transformations as explained in more details elsewhere [Mot09].

Although defining a new high-level action language for transforming a *runtime model* during execution retains a stricter separation between the different abstraction levels, we decided to follow the more natural approach. That is, leveraging a semantic domain and specifying model transformations with necessary inter-abstraction-level mapping links to the model in question. We identified some advantages: (1) There is no need to define any new language to express semantics on the meta model abstraction level. (2) There is a quite direct connection for meta model elements and their counterparts in the semantic domain which allows easy traceability. (3) Abstraction levels can be retained by carefully choosing an abstract semantic domain and advanced techniques for model transformation (e.g., a generative approach for the transformations as well).

## 3 Semantic Specification

As introduced above, there are two possible ways to specify semantics of a DSL, where we use the second approach in leveraging Ptolemy II as a flexible and extensible simulation backend. Therefore, in the following we will give a short introduction into Ptolemy, which we use as an example semantic domain, and afterwards give some brief overview of a concrete case study.

### 3.1 Ptolemy

The Ptolemy II project studies heterogeneous modeling, simulation, and design of concurrent systems with a focus on systems that mix computational domains [EJL+03].

The behavior of reactive systems, i.e., systems that respond to some input and a given configuration with an output in a real-time scenario, is modeled in Java with executable models. The latter consists of interacting components called *actors*, hence this approach is referred to as *Actor-Oriented-Design*. These actors can be interconnected at their ports. Ptolemy actors can be encapsulated into composed actors introducing a notion of hierarchy. Ptolemy models strictly try to separate the syntax and the semantics on one modeling layer. The first is given by the structural interconnection of all used actors. The second is encapsulated in a special and mandatory *director* actor that specifies the way of actor interaction and scheduling. Ptolemy allows models to mix different models of computations (MoCs) on different hierarchy layers. Actors consist of (1) pure Java code that may produce output for some input during execution or (2) other Ptolemy actors composed together under a separate model of computation (MoC) that defines the overall in- and output behavior.

There exist several built-in directors that come along with Ptolemy II, such as Continuous Time (CT), Discrete Events (DE), Process Networks (PN), Synchronous Dataflow (SDF), Synchronous Reactive (SR) and Finite-State-Machines (FSM). Whenever this seems to limit

the developer, one may adapt or define new Ptolemy II directors in Java that implement their own more specialized semantic rules of component interaction. The combination of these various, extendable domains allows to model complex systems with a conceptually high abstraction leading to coherent and comprehensible models. An example Ptolemy II model is presented in Fig. 4.

For the sake of brevity we cannot discuss the technical details here and refer to the Ptolemy documentation, in particular about the *charts (pronounced starcharts) principle [GLL99]. It illustrates how hierarchical Finite-State-Machiness can be composed using various concurrency models leading to arbitrarily nested and heterogeneous model semantics. We employed this technique to emulate a Statechart dialect, thus specifying the semantics and producing executable model representations.

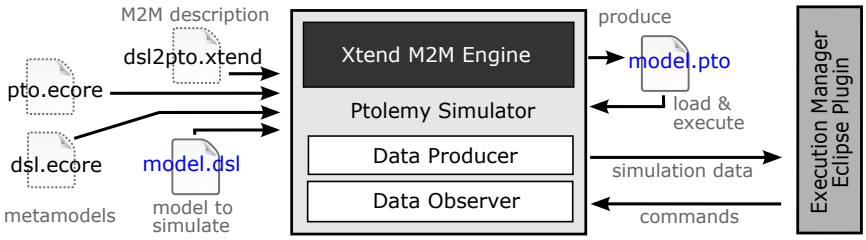


Figure 3: Abstract transformation and execution scheme

### 3.2 Concept

Fig. 3 shows the underlying concept where the description of the transformation is defined in the file dsl2pto.xtend using the *Xtend* language (cf. Sec. 1.2). The latter operates on EMF metamodel instances, hence the transformation itself must be defined referencing two metamodels. The *source metamodel* dsl.ecore stems from the EMF tool chain that already exists after defining the DSL's abstract syntax. The *target metamodel* pto.ecore describes the language of all possible Ptolemy models and is common for all DSLs.

The actual transformation of the source model model.dsl into the target Ptolemy model model.pto is done by the Xtend transformation framework. Instrumentation code is injected during the Ptolemy model composition that allows to easily map Ptolemy actors back to their corresponding original model elements. The Ptolemy Simulator itself is part of the execution runtime interface and can load and run Ptolemy models and interact with the Execution Manager as described in the next chapter.

### 3.3 SyncCharts

The *Statecharts* formalism of David Harel [Har87], which extends *Mealy machines* with hierarchy, parallelism, and signal broadcast, is a well known approach for modeling control-

intensive tasks. *SyncCharts* were introduced almost ten years later [And96] as an adoption to the synchronous world. They serve as a graphical representation of the Esterel language [BC84] following the same execution semantics.

SyncCharts simplify the modeling of complex reactive systems because they allow to model deterministic concurrency and preemption. However, they are more difficult to execute compared to other state machine models. As a challenging example we defined the semantics of SyncChart EMF models in a model-to-model (M2M) transformation, mapping each element to Ptolemy actors utilizing the combination of the Synchronous Reactive and Finite-State-Machines domains.

### 3.3.1 Transformation

The main idea is to represent the hierarchical layers by Ptolemy composite actors. These are connected by links incorporating the signal broadcast mechanism.

The SR fixed-point semantics guarantees finding a fixed point for the signal assignment w.r.t. the signal coherence rule. The latter means that each SyncCharts signal can either be present or absent in a synchronous tick instant but not both at once. For each tick, the fixed point computation in SR starts with unknown signal states on all data links.

The SyncChart example of Fig. 4 shows a broadcast communication between two parallel regions R1 and R2. R1 emits the signal L by taking an enabled transition from initial state S0 to state S1 guarded by an implicit *true* trigger. R2 waits in its initial state S2 for the signal L to be present in order to take the transition to state S3.

The structural transformation ensures that for each parallel region, every signal is represented as an input and output port (e.g., Li and Lo) because conceptually each region can emit a signal or may react to a present signal, or even both. The latter implies the requirement of a feedback structure for each signal using a special combine actor. In the Ptolemy model, the presence of a signal is represented by a data token traveling across the dedicated link. The absence of a signal is equivalent to a special clear operation on a channel. If the combine actor receives a token of any parallel region, it immediately outputs a token to the feedback loop. If the combine actor on the other hand notices a clear on each connected

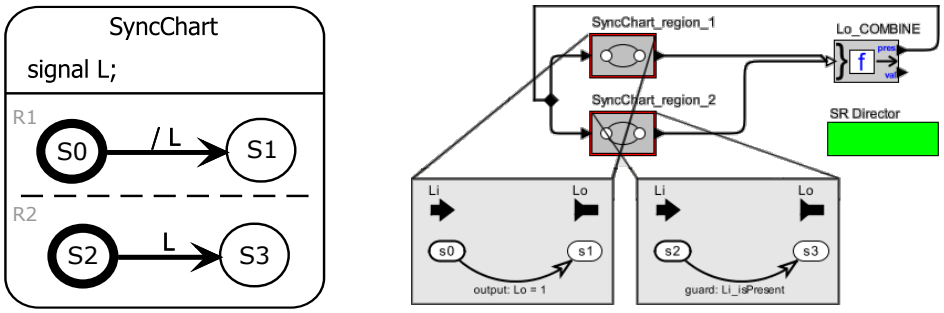


Figure 4: A SyncChart model (left) and the generated Ptolemy model (right)

incoming channel, it also clears its output. This reflects the fact that a signal is present iff it is emitted in a tick instance and a signal is absent iff it is not emitted anywhere.

In the generated Ptolemy model of Fig. 4, the concurrent actor SyncChart_region_1 will produce an output token that will be received and forwarded by the combine actor. Finally, a duplicate of this token reaches the actor SyncChart_region_2 triggering a state transition.

Further concepts of the transformation description consider the aspect of hierarchy: Concurrent Ptolemy actors that represent parallel regions contain FSM nodes that are either refined in case of original SyncCharts *macro states* or not refined in case of original SyncCharts *simple states*. The refinements can again contain concurrent actors representing regions within such a *macro state*. Because signals within a SyncCharts state can be emitted in any inner state, their dedicated ports are replicated in the transformation process for all lower hierarchy layers.

The Ptolemy expression language allows evaluation of complex triggers that for example are a boolean combination of signal presence values. This makes it straight forward to support the last special SyncChart concept of compound events in Ptolemy models.

The mapping takes also place during the model transformation process as we link the EMF model elements with attributes of the generated Ptolemy model elements. This simulation engine is interfaced with the Execution Manager presented in Sec. 4, as depicted in Fig. 3. It will process input and output signals and also collect additional output information such as the current active states. The information and the signal data are used to visualize the simulation and feed a Data Table, as shown in Fig. 2.

## 4 Execution Integration

As a subproject of KIELER the Execution Manager (KIEM) implements an infrastructure for the simulation and execution of domain specific models and possibly graphical visualizations. It does not do any simulation computation by itself but bridges simulation components, visualization components and a user interface to control execution within the KIELER application, as indicated in Fig. 1. These components can simply be constructed using the Java language implementing some commonly defined interfaces.

An approach on how to implement such simulation engines themselves using model transformations and Ptolemy as a simulation backend has just been presented in Sec. 3. The simulator component depicted in Fig. 3 loads Ptolemy models using Ptolemy internal loading mechanisms. Ptolemy models are instances of Java classes. Thus they can easily be accessed by the simulator component for example to inject input values coming from KIEM or extract output values and current state information to send them to KIEM.

### 4.1 DataComponents

*DataComponents* are the building blocks of executions in the KIEM framework. They use data in order to interact with each other. Hence they may produce data addressed for other DataComponents or observe data from other components or even both at once. See again Fig. 1 for an example setup. It shows several example data components like a basic Java simulator or a more abstract Ptolemy simulator and also components that only visualize data either in the model itself or in a separate view of the model's environment.

DataComponents can be classified according to their type of interaction into multiple categories: *Pure observer* DataComponents do not produce any data which for example is the case for simulation visualizations. *Pure producer* DataComponents like user input facilities do not observe any data. Hence they are data independent of others. Often there are *observing and producing* DataComponents like simulation engines that react to input with some output.

```
1  public interface IDataComponent {
2
3      void initialize() throws KiemInitializationException;
4      void wrapup() throws KiemInitializationException;
5
6      boolean isProducer();
7      boolean isObserver();
8
9      JSONObject step(JSONObject jSONObject) throws KiemExecutionException;
10
11 }
```

Figure 5: DataComponent Interface

Fig. 5 shows the simple and self-explanatory interface for DataComponents. A component needs to declare whether it is an observer or a producer of data. It should declare some initialization and wrapup code. The step method is most significant in this interface. It should implement the execution behavior of the DataComponent. The parameter value holds all input data in case of an observer component. The return value should hold all output data in case of a producer component.

### 4.2 User Interface

Fig. 2 shows the Graphical User Interface (GUI) of the Execution Manager. Listed are all DataComponents that take part in the execution. The order of the DataComponents in the list is the one in which they are scheduled. Together with (optional) property settings the list of DataComponents forms a savable *execution setting*. The execution can be triggered by the user by pressing one of the active control buttons (e. g., step, play, or pause). The step button allows a stepwise, incremental execution while in each step all DataComponents are executed at most once (see below). The lower bound on a step duration can be set in the UI, while the upper bound depends on the set of all producer DataComponents.

### 4.3 Data Pool and Scheduling

Data are exchanged by DataComponents in order to communicate with each other. The Execution Manager collects and distributes sets of data from and to each registered (w.r.t. the Eclipse Rich Client Platform (RCP) plug-in concept) DataComponent. Therefore it needs some kind of memory for intermediate storage to reduce the overhead of a broadcast, and to restrict and decouple the communication providing a better and more specific service to each single DataComponent.

This storage is organized in a data pool where all data are collected for later usage. The Execution Manager only collects data from components that are producers of data. Whenever it needs to serve an observer DataComponent, it extracts the needed information from its data pool, transparent to the component itself.

All components are called by the Execution Manager in a linear order that can be defined by the user in an *execution setting*. Because the execution is an iterative process—so far only iteratable simulations are supported—all components (e.g., a simulation engine or a visualizer) should also preserve this iterative characteristic. During an execution, KIEM will stepwise activate all components that take part in the current execution run and trigger them to produce new data or to react to current data. As KIEM is meant also to be an interactive debugging facility, the user may choose to synchronize the iteration step times to real-time. However, this might cause difficulties for slow DataComponents as discussed below.

All components are executed concurrently. This means that they are executed in their own threads. For this reason, DataComponents should communicate (e. g., synchronize) with each other via the data exchange mechanism provided by the Execution Manager only to ensure thread safety. There are also additional scheduling differences between the types of DataComponents listed above. These concern two facts: First, DataComponents that only produce data do not have to wait for any other DataComponent and can start their computation immediately. Second, DataComponents that only observe data, often do not need to be called in a synchronous blocking scheme since no other DataComponents depend on their (nonexistent) output.

### 4.4 Further Concepts

Besides the described basic concepts of the Execution Manager there are some facilities and improvements that are summarized in the following.

**Analysis and Valdiation:** For analysis and validation purposes it is easy to include *validation DataComponents* that observe special conditions related to a set of data values within the Data Pool. These components may record events in which such conditions hold or may even be able to pause the execution to notify the user.

**Extensibility:** The data format chosen in the implementation relies on the Java Script Object Notation (JSON). This is often referred to as a simplified and light-weight XML.

It is commonly used whenever a more efficient data exchange format is needed. Due to its wide acceptance many implementations for various languages exist, thus aiding the extensibility of the Execution Manager.

Although DataComponents need to be specified in Java, the data may originally stem from almost any kind of software component, e. g., an online-debugging component of an embedded target. With this approach the Java DataComponents do not need to reformat the data and can simply act as gateways between the Execution Manager and the embedded target.

As an example we have developed a mobile phone Java ME[3] application that can fully interact with the Execution Manager.

**Co-Simulation:** Co-operative simulation allows the execution of interacting components run by different simulation tools. For each different simulation tool a specific interface DataComponent just needs to be defined. This way Matlab/Simulink for example could co-simulate with a SyncCharts model and an online-target debugging interface to get a model- and hardware-in-the-loop setup, which is useful for designing embedded/cyberphysical systems.

**History:** Together with the Data Pool the built-in history feature comes for free. This enables the user to make steps backwards into the past. DataComponents need to explicitly support this feature: For example one may not want a recording component to observe/record any data again when the user clicks backwards. This feature may help analyzing situations better. For example, when a validation observer DataComponent pauses the execution because a special condition holds, one may want to analyze how the model evolved just before. This assists during interactive debugging sessions.

## 5 Conclusions and Outlook

The usage of DSLs gains more and more importance when it comes to system specifications intended to be done by domain experts as opposed to computer experts.

In this paper we presented a two-level approach into the simulation and semantics of domain specific behavior models. We gave a short introduction into used concepts and into the Ptolemy suite as a multi-domain, highly flexible and extensible modeling environment with formally founded semantics. In order to not reinvent the wheel we proposed to utilize these existing features in an integrated way for specifying denotational semantics w.r.t. an adequate simulation. As an example a case study discussed how to conceptually leverage Ptolemy for simulating SyncCharts models. Further, it was outlined how iterative executions are seamlessly integrated into KIELER and therewith into the Eclipse platform.

We plan to evaluate further the needs and requirements for a transformation language used in the context of semantic definitions as described in this paper. Also we hope to advance

[3] Java Micro Edition Framework: http://java.sun.com/javame

the generative model-based approach, e. g., to come up with a generated transformation of some higher order mapping specifications in the future. Finally, there are other ongoing efforts to integrate additional simulation engines into Eclipse using the presented KIEM infrastructure.

## References


[And96] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[BC84] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 389–448. Springer-Verlag, 1984.

[CCG+08] Benoit Combemale, Xavier Cregut, Jean-Patrice Giacometti, Pierre Michel, and Marc Pantel. Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In *Proceedings of the 4th European Congress EMBEDDED REAL TIME SOFTWARE (ERTS '08)*, 2008.

[CSN07] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Compositional specification of behavioral semantics. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*, pages 906–911, San Jose, CA, USA, 2007.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming Heterogeneity—The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[ESS06] Hajo Eichler, Markus Scheidgen, and Michael Soden. A Meta-Modelling Framework for Modelling Semantics in the contect of Existing Domain Platforms. Technical report, Department of Computer Science, Humboldt-Universität zu Berlin, 2006.

[FvH10] Hauke Fuhrmann and Reinhard von Hanxleden. Taming Graphical Modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, LNCS, Oslo, Norway, October 2010. Springer.

[GLL99] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18:742–760, 1999.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Mot09] Christian Motika. Semantics and Execution of Domain Specific Models—KlePto and an Execution Framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification, v2.0, January 2006. http://www.omg.org/spec/MOF/2.0/PDF/.

[SF07] Markus Scheidgen and Joachim Fischer. Human Comprehensible and Machine Processable Specifications of Operational Semantics. In *Model Driven Architecture- Foundations and Applications*, volume 4530 of *LNCS*. Springer-Verlag, 2007.