INSTITUT FÜR INFORMATIK

Semantics and Execution of Domain Specific Models

Christian Motika, Hauke Fuhrmann,
Reinhard von Hanxleden

Bericht Nr. 0923
Dezember 2009

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Semantics and Execution of Domain Specific Models

Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden

Bericht Nr. 0923
Dezember 2009

e-mail:
cmot@informatik.uni-kiel.de,
haf@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de

Technical Report

Abstract

In this paper we present a two-level approach to extend the abstract syntax of models with concrete semantics. First, a light-weight execution interface 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 Modeling Language (DSML). 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 DSML 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. After introducing basic concepts, the paper explains how simulations are currently being integrated into the framework and presents some illustrative case studies also covering UML approaches.

Contents

List of Figures

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 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 Modeling Languages* (DSML) 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 DSML 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 DSML, the tool smith, faces less efforts in developing his or her modeling environment. This is achieved by the 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 DSML. This also 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 DSML there is a need to provide a way for making it explicit.

The contribution of this paper is a proposal on how DSML semantics can be defined flexibly by using existing *semantic domains* without introducing any new kind of language or notation. Fig. 1.1 shows an example setup of the architecture and gives an overview of the following sections. 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

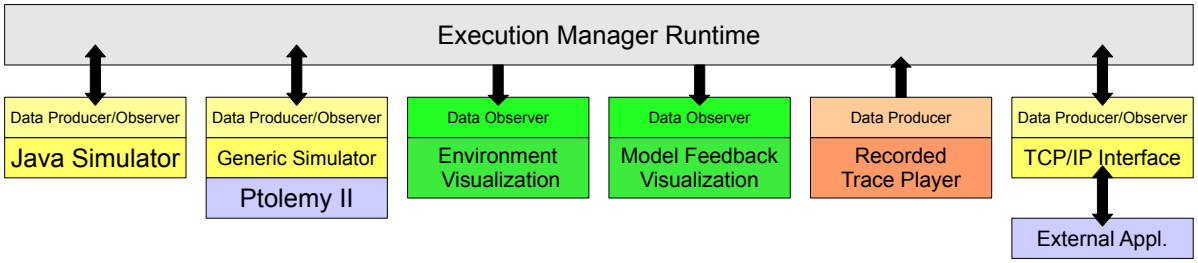

Figure 1.1: Schematic overview of the Execution Manager infrastructure

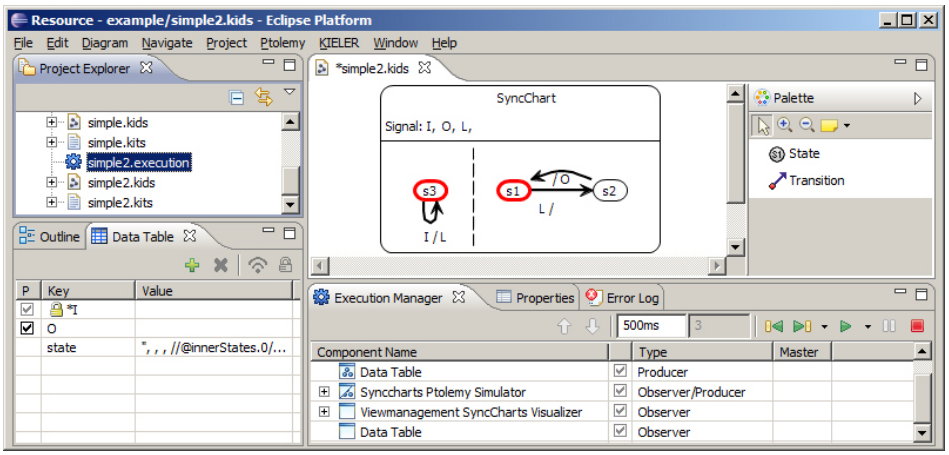

Figure 1.2: GUI of KIELER and the KIEM Eclipse plug-in during a simulation run

Sec. 1.3. In Sec. 2 we present how we define semantics for a DSML with the Ptolemy II suite (cf. Fig. 1.1). In this context two case studies are examined, one about simulating SyncCharts, and another about simulating UML state machines in both by leveraging Ptolemy. An implementation overview about our general approach of integrating simulations in the Eclipse platform is given in Sec. 3—the Execution Manager Runtime in Fig. 1.1. Additionally we show that this solution is extendable and has open support for e.g., model analysis and validation or co-simulations. Sec. 4 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* of model-based system design, i. e., the way the user interacts with models [7].
The KIELER framework is a set of open source Eclipse plug-ins that integrate with com-

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

mon 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. 1.2 shows a simulation run with KIEM in KIELER which provides a user interface, shown in the bottom view in Fig. 1.2, and visual feedback about simulation details, both in the graphical model view itself and in separate views like data tables or an environment visualization.

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) [14], 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[3] is a functional 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.

1.3 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

[2] http://wiki.eclipse.org/Xpand
[3] http://wiki.eclipse.org/Refactorings_for_Xpand_/_Xtend_/_Check

integrated simulation purposes Ptolemy provides the *Vergil* graphical editor. There also exists the possibility to embed the execution of Ptolemy models into arbitrary Java applications [5].
With Matlab/Simulink/Stateflow and SCADE 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 [3]. Other simulation supporting frameworks are Scilab/Scicos, Hyperformix Workbench, Sildex, StateMate, or Uppaal.
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 DSMLs, 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. 2.1.
As outlined in [15] two fundamentally different concepts can be emphasized for specifying model semantics: (1) Model-Transformation into a *semantic domain* or (2) provision of a new action language. 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 (e.g., because models in this domain are executable). The second concept applies semantics by extending the metamodel with semantical information on the same abstraction level for which 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 latter in [2] is a compositional approach for specifying the model behavior. The *M3Action* framework for defining operational semantics[4] is illustrated in [10, 15].
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 the following 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) The expressiveness is not limited by a concrete language but unlimited due to the flexibility of conceptionally using any semantic domain. (4) 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).
Our architecture is related to the IEEE standard for modeling and simulation high-level architecture [11]. The execution manager presented in Sec. 3 follows the ideas of the

[4] http://www.metamodels.de

runtime infrastructure (RTI). However, our approach does not follow the standard in detail, but is meant to be a light-weight approximation.

2 Generic Semantic Specification

As introduced above, there are two possible ways to specify semantics of a DSML, 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 concrete case studies.

2.1 Ptolemy

The Ptolemy II project studies heterogeneous modeling, simulation, and design of concurrent systems with a focus on systems that mix computational domains [5].
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 as *Actor-Oriented-Design*. These actors interact under a model of computation (MoC) which specifies the semantics encapsulated in a special *director actor*. Ptolemy models are hierarchically layered allowing different MoCs for each layer. 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 MoC that defines the overall in- and output behavior.
There exist several built-in directors (i. e., concurrency implementations) that come 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 easily adapt or define new Ptolemy II directors in Java that implement their own more specialized semantical 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. 2.1. For the sake of brevity we cannot go into much technical details here and refer to the technical Ptolemy documentation, in particular about the *charts (pronounced starcharts) principle [8]. 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 very different Statechart dialects, thus specifying their semantics and producing executable representations.

2.2 SyncCharts

The *Statecharts* formalism of David Harel [9], which extends *Mealy machines* with hierarchy, parallelism, signal broadcast, and compound events, is a well known approach for modeling control-intensive tasks. *SyncCharts*, the natural adoption of Statecharts to the synchronous world, were introduced almost ten years later [1], evolving from ARGOS [12].
As an example of the flexibility of our approach we implemented a model-to-model (M2M) transformation that synthesizes Ptolemy models from Eclipse models which abstract syntax was defined using the EMF tool chain. This transformation was defined using *Xtend* (cf. Sec. 1.2).
The latter operates on EMF metamodel instances, hence the transformation itself must be defined for two metamodels. The *source metamodel* stems from the EMF tool chain that already exists after defining the DSML's abstract syntax. The *target metamodel* describes the language of all possible Ptolemy models and is common for all DSMLs in this approach.

We define the semantics of a SyncChart EMF model by mapping each element to a Ptolemy actor. A simple example mapping is illustrated in Fig. 2.1. The main idea is to represent the hierarchical layers as Ptolemy composite actors that are connected by data links incorporating the broadcast mechanism together with the SR semantics. The actors contain FSM nodes that are either refined in case of original *macro states* or not refined in case of original *simple states*. Special care needs to be taken to introduce an equivalence to the *normal termination* concept defined in SyncCharts [1].
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. 3, as depicted in Fig. 2.2. It will process input and output signals and also collect additional output information such as the currently active states. The information and the signal data are used to visualize the simulation and feed a Data Table, as shown in Fig. 1.2.

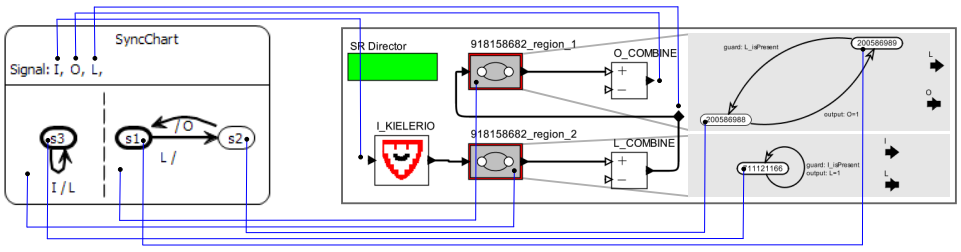

Figure 2.1: A SyncChart model (left) and a generated Ptolemy model (right) with their transformation relations (blue)

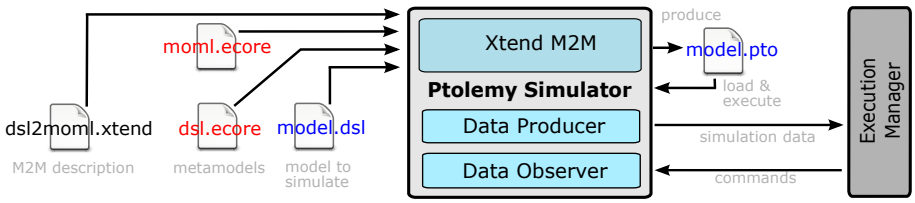

Figure 2.2: Abstract transformation and execution scheme

2.3 UML State Machines

SyncCharts have a very precise formal semantics, which recommend them for the design of safety-critical applications. Unlike UML for example, which mainly focuses on the standardization of syntax, the UML specification [13] does specify semantics, but with informal prose only that leads to ambiguities and unclarities [6].
This has prompted several approaches to formalize parts of the UML. If we look for example at UML state machines in contrast to SyncCharts, Crane and Dingel compare 26 different approaches to the semantics of UML state machines [4]. Most of them cover only parts of the UML features and hence there does not exist *the* formalism, but one has to choose one suitable for the application at hand—which obviously makes UML tools less interchangeable.
An implementation of an execution engine for UML state machines in KIELER can be done at different levels of abstraction, in four different ways:
First, it could be a pure manual implementation in Java, or any other language that could be interfaced with Java. It then could be interfaced with the execution manager by directly subclassing the DataComponent, as described in Sec. 3. This "escape" to the programing language level is always possible and will at least result in possibilities to interchange simulation data with other KIEM components. However, it is likely to be effort prone. Being the author of one of the 26 approaches to UML semantics or of another implementation, one could simply interface it with KIEM to get access to all the other mentioned features.
Second, a custom domain, i. e., a *Director*, could be implemented for Ptolemy in Java that implements the execution of that domain, which would execute UML state machines. This is close to the first approach, but would transfer the semantics implementation into the Ptolemy framework. As it was originally designed for these kinds of execution strategies, this approach would benefit from Ptolemy's internal infrastructure.
Third, a transformation in one of the most expressive pre-defined Ptolemy II models of computation could be created, as described in Sec. 2.1. This also is always possible, because some of the pre-existing domains such as PN are Turing-complete. However, such transformation might be quite complex as well and maybe not very efficient for execution.
As a fourth approach it is often possible to reuse existing domains in Ptolemy, mixed in nested hierarchy levels. Such an approach appears as the most promising, as it avoids to massively reduce the abstraction level by a manual transformation. As a feasibility

study for this approach, we realized basic UML state machine concepts in Ptolemy. The remainder of this section elaborates this further.

The run-to-completion (RTC) principle in the UML is not really an implementation of concurrency, because "such semantics are quite subtle and difficult to implement" [13]. Although Ptolemy's major strength is the composition of many different concurrency paradigms for clear implementations, we used Ptolemy to emulate the "not-quite-concurrent" behavior of UML state machines. This is done by composing FSMs with the DE semantics [8].

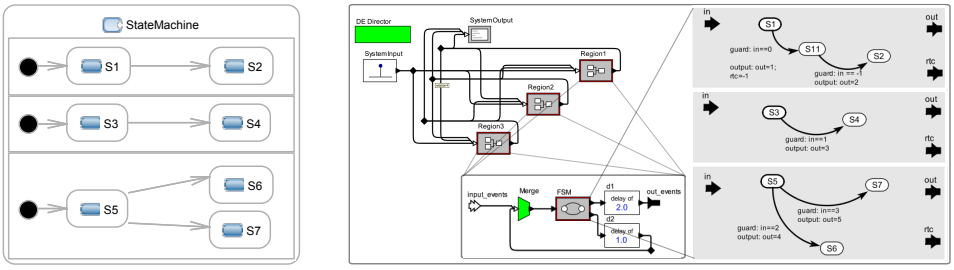

Figure 2.3: Expressing the run-to-completion principle in Ptolemy II by composing FSMs with DE

As depicted in Fig. 2.3, FSMs represent parallel regions of a state machine. They are embedded in a data-flow model, which explicitly models the signal communication between the different regions. Unlike the solution of Sec. 2.2, DE semantics with some specialities implements the UML RTC behavior. The example shows broadcast communication where every region sends its signals to all other regions. The FSMs themselves are wrapped in another data-flow composite actor that also comprises a merge and delay actors. The delays put the new signal events into the right logical order, while the merge puts them in this order onto the same input signal event queue which emulates the UML event pool. This is necessary because FSMs in Ptolemy are seen by the DE director as zero delay operators and hence an ordering of new signal emissions would not be possible. Therefore multiple sequenced actions in an UML transition get expanded to multiple sequential FSM transitions that execute in succeeding microsteps that all together compose one UML RTC step. Delays have to be computed in the transformation process. The ordering of events then guarantees the right order in which the transitions are taken.
Other UML particularities can also be handled in the transformation. Keeping hierarchy is no problem, because Ptolemy also supports refinements of states and also transitions. The implicit transition priorities in hierarchical states can be applied by adding explicit boolean expressions such that inner transitions always have priority over outer levels; the default behavior in Ptolemy is, like preemption, the other way round.

3 Generic Execution Integration

As a subproject of KIELER the Execution Manager (KIEM) implements an interface for the simulation and execution of domain specific models and possibly graphical visualizations. Itself it does not do any simulation computation but bridges simulation components, visualization components and a user interface to control execution within the KIELER application, as indicated in Fig. 1.1. These components can simply be constructed using the Java language implementing some commonly defined interfaces. A generic approach on how to implement such simulation engines themselves has just been presented in Sec. 2.

DataComponents DataComponents are the building blocks of executions in the KIEM framework. As the name suggests, DataComponents handle (simulation) data and use these to interact with each other. Hence they may produce data addressed for other DataComponents or observe data values from other components or even both at once. See again Fig. 1.1 for an example setup.
We can therefore classify DataComponents according to their type of interaction into four categories: (1) *Initializer* DataComponents usually neither observe nor consume any data. Examples are the running of a web server during the execution or the synchronization of a data base before and after a simulation run. (2) *Pure observer* DataComponents do not produce any data which e.g., is the case for simulation visualizations. (3) In contrast *pure producer* DataComponents do not observe any data. This makes such components (e.g., user input facilities) data independent of others. (4) Finally there are *observing and producing* DataComponents like simulation engines that react to input with some output.

User Interface Fig. 1.2 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 as described below.

Data Pool 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.

Linear Scheduling All components have in common that they 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 ask them to produce new data or to react to older data. As KIEM is meant to be also 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.

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 Extensible Markup Language (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[1] 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 an UML 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, e. g., one may not want a recording component to observe (i.e, to record) any data 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 very well during interactive debugging sessions.

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

4 Conclusions and Outlook

The usage of DSMLs 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 DSMLs. We gave a short introduction into how we used the concepts of EMF, Xtend M2M transformations and the Ptolemy suite as a multi-domain, highly flexible and extensible modeling environment with a formally founded semantics, in particular concerning aspects of concurrency. We proposed to utilize these features for specifying the semantics of a DSML w.r.t. an adequate simulation in order to not reinvent the wheel. As an example two case studies have been discussed. The first illustrated how to conceptually leverage Ptolemy for simulating SyncCharts models. The second example covered UML state machines, outlining additional approaches that should cover most imaginable DSMLs. While the SyncCharts simulation is almost complete and needs only small refinements about some special SyncChart features, the UML experiments illustrate the high potential of Ptolemy as an execution backend.
Further, it was outlined how iterative executions are currently integrated into the KIELER project 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. Finally, we hope to advance the generative model-based approach, e. g., to come up with a generated transformation of some higher order mapping specifications in the future.

Bibliography

[1] 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. http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf.

[2] 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.

[3] 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.

[4] Michelle L. Crane and Jürgen Dingel. On the semantics of UML state machines: Categorization and comparison. Technical Report 501, School of Computing, Queen's University, Kingston, Ontario, Canada, 2005.

[5] 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.

[6] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 State Machines. In *ICFEM*, volume 3785 of *LNCS*, pages 52–65. Springer, 2005.

[7] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[8] 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.

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

[10] Hajo Heichler, 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.

[11] IEEE. Ieee standard for modeling and simulation (m&s) high level architecture (hla)—framework and rules. *IEEE Std 1516-2000*, pages i–22, Sep 2000. http://ieeexplore.ieee.org/servlet/opac?punumber=7179.

[12] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, October 1991.

[13] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, Aug 2005. http://www.omg.org/docs/formal/05-07-04.pdf.

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

[15] 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.