INSTITUT FÜR INFORMATIK

Executing Domain-Specific Models in Eclipse

KLEPTO - KIELER leveraging Ptolemy

Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden, Edward A. Lee

Bericht Nr. 1214
October 2012

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

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

Executing Domain-Specific Models in Eclipse KLEPTO - KIELER leveraging Ptolemy

Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden, Edward A. Lee

Bericht Nr. 1214
October 2012

e-mail:
cmot@informatik.uni-kiel.de,
haf@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de,
eal@eecs.berkeley.edu

Technical Report

Contents

List of Figures

Abstract

We present a two-level approach to extend the abstract syntax of domain-specific models with concrete semantics in order to execute such models. First, a light-weight execution infrastructure for executable models with a generic user interface allows the tool smith to provide arbitrary execution and visualisation engine implementations for a DSL. Second, as a concrete but nevertheless generic implementation of a simulation engine for behaviour models, we present semantic model specifications and a runtime interfacing to the Ptolemy II tool suite as a formally founded backbone for model execution. We present our approach as an open source extension to Eclipse modelling projects.

Key words: Semantics, Eclipse, Modeling, EMF, KlePto, KIELER, Execution Manager

1 Introduction

Computer simulations are an established means to analyse the behaviour 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 hand one wishes 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 semantic analysis and execution. The notation of a model instance is a concrete textual or graphical syntax.

Traditionally, all model editing, parsing, and processing facilities were manually implemented with little generic abstractions that inhibit interchangeability. Standardised languages, e. g., the Unified Modelling 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 applications, offering tailored abstractions and complying to the exact needs of developers within certain domains. On the one hand, there are already well established tool kits like the *Eclipse Modeling Framework* (EMF) or Microsoft's DSL tool kit to define an abstract syntax of a DSL in a model-based way. They provide much infrastructure, such as a meta-model 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 a modelling 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

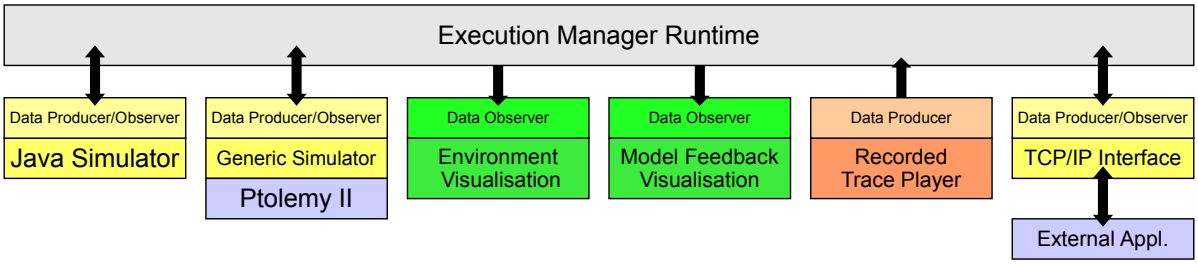


Figure 1.1: Schematic overview of the Execution Manager infrastructure with various kinds of interacting DataComponents (from Motika et al. [16]).

latter no common mechanism 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.

Contribution 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. The focus is on iterative models. We show how simulations of such models can easily be integrated into the Eclipse Modeling Framework using the KIELER Execution Manager (KIEM) architecture which is illustrated in the example setup in Fig. 1.1.

Outline In Sec. 1.1, a survey about the KIELER framework, which is the context of the work presented in this paper, is given. Fundamental concepts are introduced in Sec. 1.2. 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. 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 shown 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. 5 concludes and gives an outline of future work.

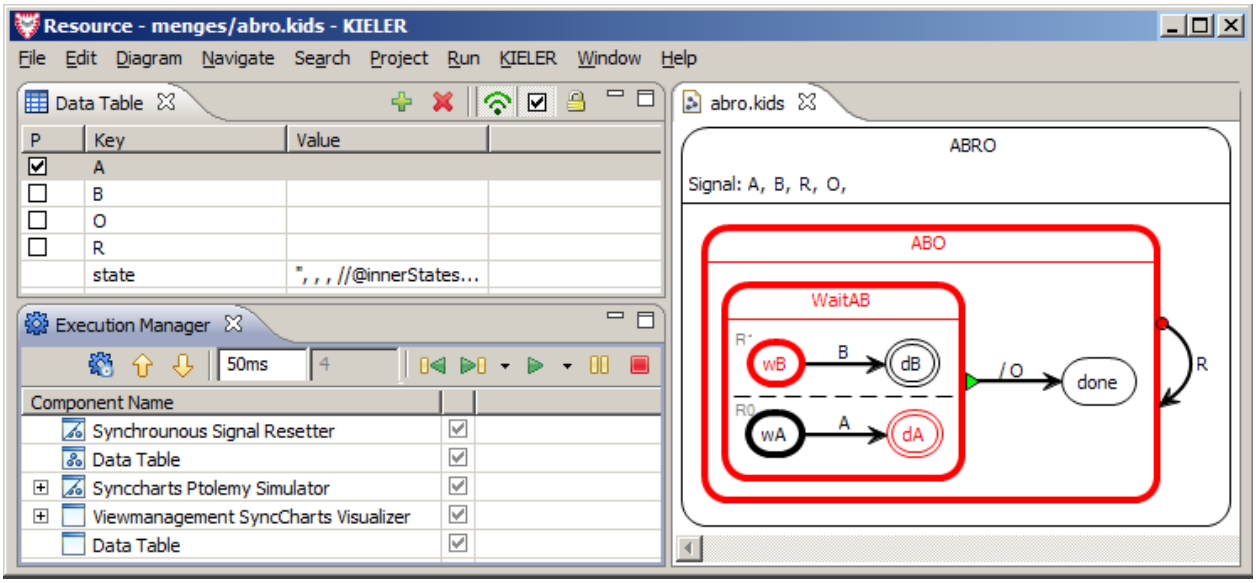


Figure 1.2: The GUI of KIELER and the KIEM Eclipse plug-in during a simulation run executing the synchronous ABRO program implemented as a SyncChart (from Motika et al. [16]).

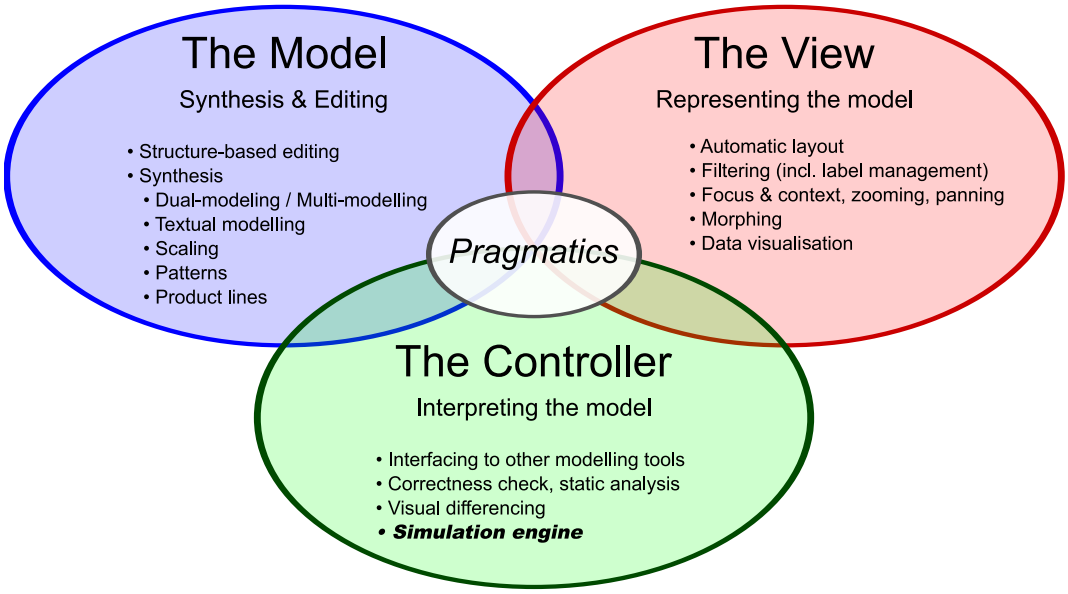


Figure 1.3: MVC in KIELER: The Execution Manager and the Ptolemy-based DSL simulation in this paper belong to the controller part (adapted from Fuhrmann et al. [8]).

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 by Fuhrman et al. [9].

The KIELER framework is a set of open source Eclipse plug-ins that integrate with common Eclipse modelling projects, such as the Graphical Modeling Framework (GMF), the Textual Modeling Framework (TMF), and especially the modelling 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 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 left bottom Eclipse View in Fig. 1.2, and visual feedback about simulation details, both in the graphical model itself and in a separate Data Table Eclipse View.

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

1.2 Basic Concepts

A basic prerequisite for Model Driven Engineering (MDE) are models that base on meta-models. 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 meta-modelling framework defined by the Object Management Group (OMG)[2], which has taken shape for the *Eclipse* world as the Eclipse Modeling Framework (EMF) with its *Ecore* meta-model 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., meta-model instances) that conform to one meta-model into other models which conform to another or even the same meta-model. There are several model transformation systems available today that are well integrated into the Eclipse platform. Xpand[3] realises a model to text template-based approach, Xtend is a functional meta-model 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 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] `http://www.omg.org/spec/MOF/2.0/PDF`
[3] `http://wiki.eclipse.org/Xpand`

2 Related Work

As outlined by Scheidgen and Fischer [17] two fundamentally different concepts can be distinguished for the specification of 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 meta-model by a simple mapping or a more complex transformation into a domain for which there already exists an explicit semantic meaning—the *semantic domain*. The second concept applies semantics by extending the meta-model with semantic operations on the same abstraction level. For this, additionally a meaning 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. [5] or Scheidgen and Fischer [17]. Chen et al. [3] presents a compositional approach for specification of model behaviour. We follow the denotational approach by utilising existing languages and describing transformations. Such an approach is explained in more detail by Motika [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. This approach is to leverage a semantic domain and to specify model transformations with necessary inter-abstraction-level mapping links to the DSL model in question. This has 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) 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.

There exists a range of modelling 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 modelling, 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. [6]. With Simulink/Stateflow[1] and SCADE[2] the user is able to integrate control-flow and data-flow model parts in a dedicated Statecharts dialect and data-flow language. The Topcased[3] project is based on the Eclipse framework and targets the MDE with simulation as the key feature in validating mod-

[1] `http://www.mathworks.de`
[2] `http://www.esterel-technologies.com`
[3] `http://www.topcased.org`

els Combemale et al.([4]). Other simulation supporting frameworks are Scilab/Scicos[4], the Hyperformix Workbench[5], or StateMate([12]). 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 modelling 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 modelling projects as elaborated in Sec. 3.3.1.

Our architecture is related to the IEEE standard for modelling and simulation high-level architecture([13]). The execution manager presented in Sec. 4 follows the ideas of the *runtime infrastructure* (RTI). However, our approach does not follow the standard in detail, but is meant to be a light-weight approximation.

Motika et al. [16] also presented an approach to augment the abstract syntax of models with concrete semantics in order to execute such models. We extend that work in several areas, including extensions to Ptolemy using the case of immediate transitions to show Ptolemy's general extensibility features, optimisations in the semantic domain that improve traceability, and a detailed description of an execution infrastructure.

[4] `http://www.scilab.org`
[5] `http://www.mmsolutions.com`

3 Semantic Specification

As introduced above, there are two possible ways to specify semantics of a DSL, where we follow the second denotational approach to leverage 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 modelling, simulation, and design of concurrent systems with a focus on systems that mix computational domains Eker et al.([6]).

The behaviour of reactive systems, i.e., systems that respond to some input and a given configuration with an output in a real-time scenario, is modelled 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 modelling layer. The first is given by the structural interconnection of all used actors. The second, the model of computation (MoC), is encapsulated in a special and mandatory *director* actor that specifies the way of actor interaction and scheduling. Ptolemy allows models to mix different 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 possibly different MoC that defines the overall in- and output behaviour.

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-Machine (FSM). Whenever this seems to limit the developer, one may adapt or define new Ptolemy II directors in Java that implement their own more specialised semantic rules of component interaction and scheduling. 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. 3.2.

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 as described by Girault et al. [10]. It illustrates how hierarchical Finite-State-Machines can be composed using various concurrency models leading to arbitrarily

nested and heterogeneous model semantics. We employed this technique to emulate a Statecharts dialect, thus specifying the semantics and producing executable model representations.

3.2 Concept

Fig. 3.1 shows the underlying concept where the description of the transformation is defined in the file dsl2pto.xtend using the *Xtend* language (Sec. 1.2). The latter operates on EMF meta-model instances, hence the transformation itself must be defined referencing two meta-models. The *source meta-model* dsl.ecore stems from the EMF tool chain that already exists after defining the DSL's abstract syntax. The *target meta-model* pto.ecore describes the language of all possible Ptolemy models and is common for all DSLs.

The 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 Sec. 4.

3.3 SyncCharts

The *Statecharts* formalism of Harel [11], which extends *Mealy machines* with hierarchy, parallelism, and signal broadcast, is a well known approach for modelling control-intensive tasks. *SyncCharts* as evolving from ARGOS([14]) are the natural adoption of Statecharts to the synchronous world and were introduced almost ten years later by Andr [1]. They serve as a graphical representation of the Esterel([2]) language following the same execution semantics.

SyncCharts simplify the modelling 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

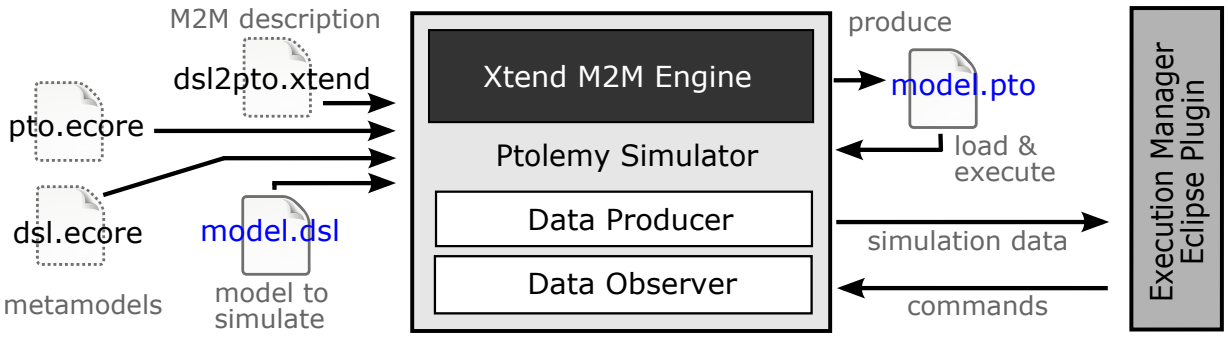


Figure 3.1: The transformation and execution scheme of a Ptolemy-based DSL model simulation (from Motika et al. [16]).

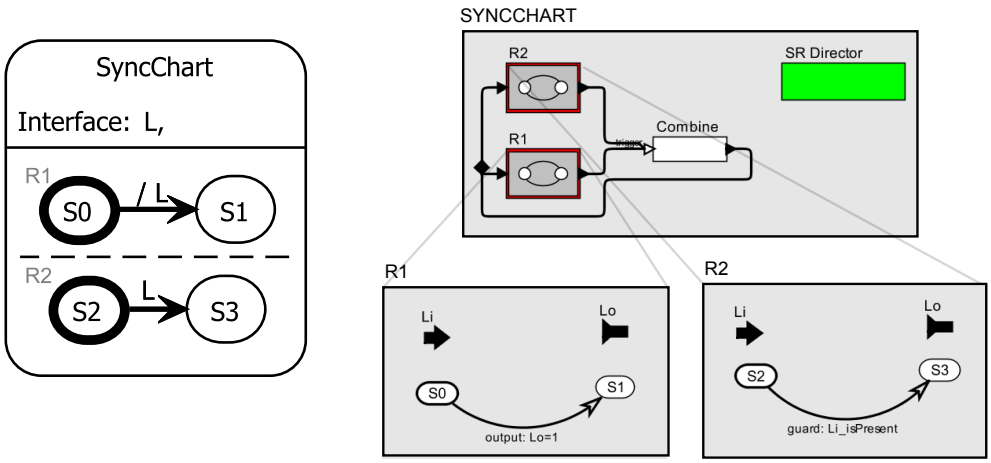


Figure 3.2: A SyncChart model (left) and the generated Ptolemy model (right) (adapted from Motika et al. [16]).

the semantics of SyncCharts EMF models in a model-to-model (M2M) transformation, mapping each element to Ptolemy actors utilising the combination of the Synchronous Reactive and Finite-State-Machine domains. More specifically we took the *modal model* domain as the successor of the FSM domain also allowing nested arbitrary Ptolemy models as state refinements. For the sake of simplicity in the following we will refer to both FSMs and modal models as FSMs.

3.3.1 Transformation

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

The SR fixed-point semantics guarantees finding a fixed-point for the signal status 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 statuses on all data links.

The SyncCharts example of Fig. 3.2 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.

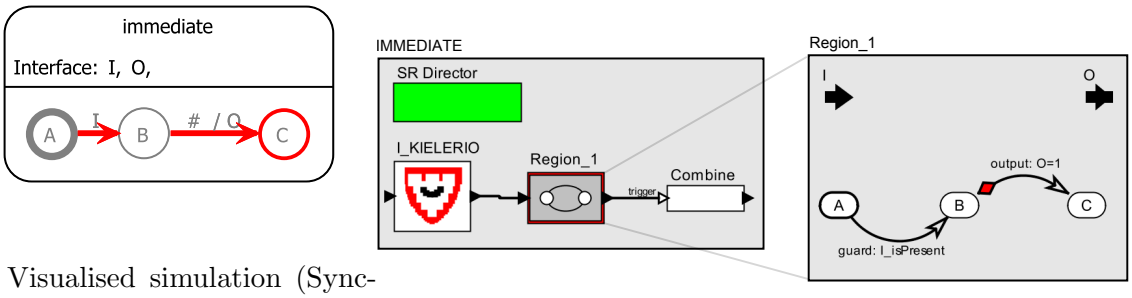
The structural transformation ensures that for each parallel region, every signal is represented as an input and output port (e.g., Li and Lo) because conceptually each region can emit a signal or may react to a present signal, or even both. The latter implies the requirement of a feedback structure for each signal using a special *combine actor*. In the Ptolemy model, the presence of a signal is represented by a data token travelling 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 incoming channel, it also clears its output. This reflects 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. 3.2, the concurrent actor R1 will produce an output token when taking the transitions from S0 to S1 that will be received and forwarded by the Combine actor. Finally, a duplicate of this token reaches the actor R2 triggering a state transition from S2 to S3.

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 straightforward to support the SyncCharts concept of *compound events* in Ptolemy models.

The mapping takes 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. 3.1. It will process input and output signals and also collect additional output information such as the current active states. The information and the signal status data are used to visualise the simulation and feed a Data Table, as shown in Fig. 1.2.



(a) Visualised simulation (SyncChart)

(b) Ptolemy immediate transitions

Figure 3.3: New immediate transitions in Ptolemy for simulating SyncCharts immediate transitions and conditional states. The visualisation shows that both transitions are taken in the same synchronous tick when the input signal I becomes present. Hence the output signal O is emitted instantaneously.

3.3.2 Extending Ptolemy

The transformation is already implemented for the main SyncCharts concepts. However, we further investigated on some special SyncCharts features initially not supported by Ptolemy FSMs like *immediate* transitions. Immediate transitions are outgoing transitions possibly immediately, i. e., in the same synchronous tick, taken when the trigger is true and their source state is entered. In SyncCharts such transitions are denoted by a hash mark in the transition trigger label.

As explained in Sec. 3.3.1, whenever predefined Ptolemy domains, i. e., Ptolemy's predefined directors, seem to limit the expressiveness of the DSL in question one can freely choose to (1) add special actors to the Ptolemy framework as done for IO purposes by Motika [15], (2) combine existing Ptolemy domains and actors as we did in this approach, or (3) design a derived or even completely different director.

Immediate transitions can also be attractive to a modelling tool like Ptolemy because they allow to structure combinatorial chains of transitions following the write-things-once (WTO) principle. We applied this feature to the modal model domain with the advantage of having it available for expressing SyncCharts immediate transitions in most cases. Fig 3.3 shows both a SyncChart during simulation with an immediate transition that is taken immediately when the state B is entered and a corresponding Ptolemy model visualising the fact that the transition is marked to be immediate by the red diamond.

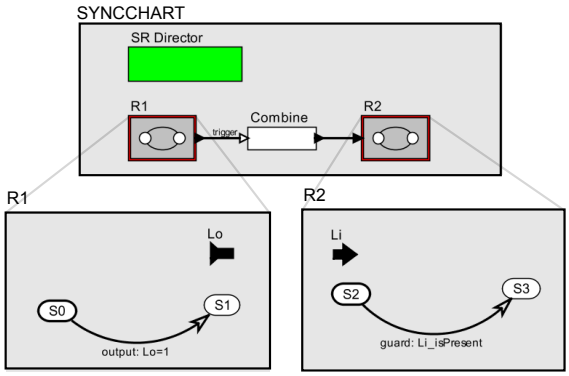


Figure 3.4: An optimised Ptolemy model.

3.3.3 Transformation Tracing and Optimisation

Considering Fig. 3.2 it may seem relatively easy to trace elements of a generated Ptolemy model back to their corresponding elements of the SyncChart source. Unfortunately for larger SyncCharts this becomes problematic.

To improve tracing possibilities we adapted the transformation to consider labels of SyncCharts elements.

Input and Output Signals To allow the emission of input signals or the test of output signals within transition triggers, a transformation layer is used that adds regions for each signal to the top level Ptolemy model in order to support this in the general case. But if there are no emitted input signals or tested output signals, then these regions are superfluous and can be eliminated. We extended the transformation to handle both cases.

Actor Signature and Feedback Loop Optimisation In Fig. 3.2 the input Li is present in the actor R1 although this actor does not do anything with this input. In general this input must be available to the actor that implements the SyncChart region R1. However in case inputs are not needed, unnecessarily they add complexity to the generated Ptolemy model. We extended the transformation to optimise the Ptolemy model in such way that superfluous inputs are removed. We also did this optimisation for output signals.

This results in a generally clearer actor structure with minimal actor interfaces and data-flow channels. Also unnecessary feedback loops are removed automatically by this procedure. In Fig. 3.2 an unoptimised Ptolemy model is shown where the feedback of L is clearly superfluous because the signal L is only emitted in region or actor R1 and tested in region or actor R2. An optimised version of this Ptolemy model is shown in Fig. 3.4.

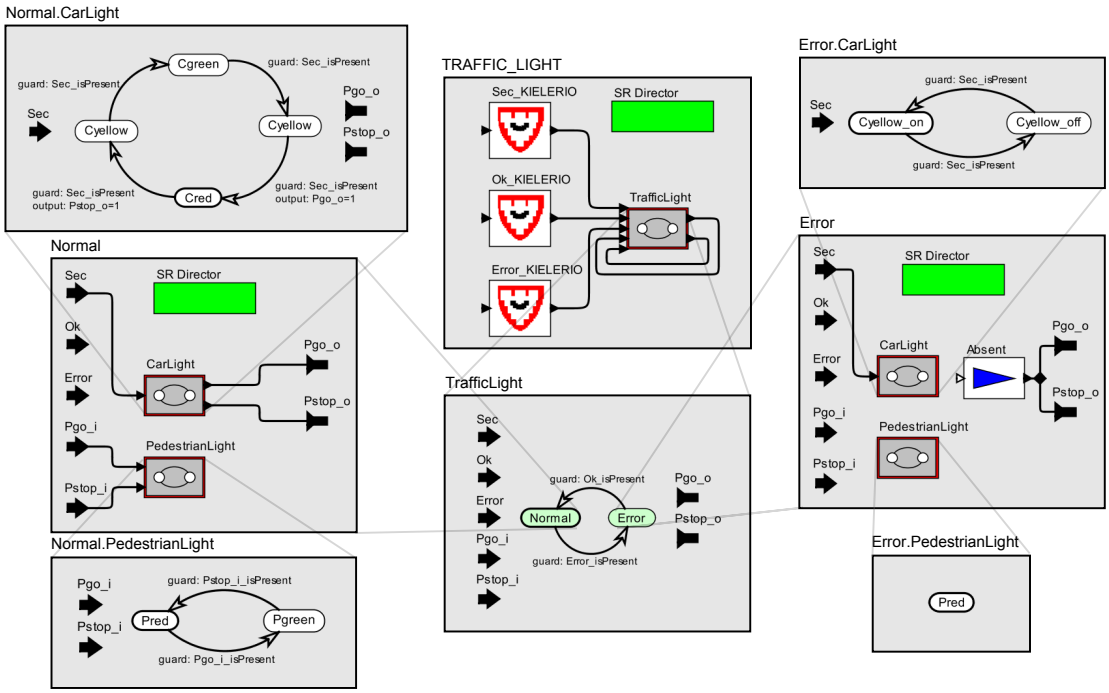


Figure 3.5: The traffic light example generated Ptolemy model using traceable actor labels. Inputs and outputs of an actor are optimised according to their internal usage.

Consider a more sophisticated SyncChart, the traffic light example shown in Fig. 3.6 together with its generated corresponding Ptolemy model shown in Fig. 3.5. The signals Pgo and Pstop are used to communicate from the Car controller region to the Pedestrian controller region. Because they are defined on the top level in the Ptolemy model there are necessary feedback loops for both signals within the TRAFFIC_LIGHT actor. But the inputs and outputs of each modal model implementing the region, i. e.,CarLight and PedestrianLight, are optimised and restricted to only used/referred signals.

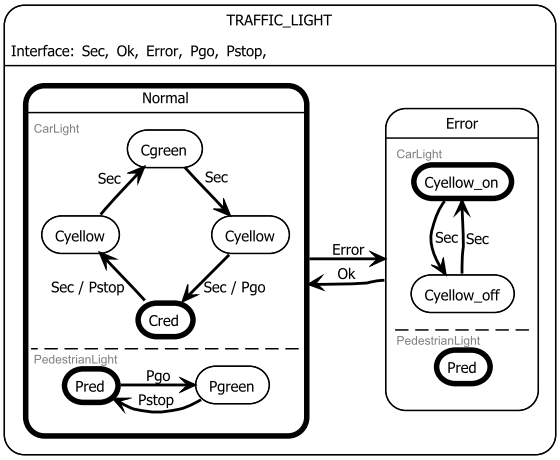


Figure 3.6: The traffic light example SyncCharts model.

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 visualisations. It does not do any simulation computation by itself but links together simulation components, visualisation 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.

Sec. 4 presented an approach on how to implement such simulator components themselves using model transformations and Ptolemy as a simulation backend. The simulator component depicted in Fig. 3.1 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 to 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. 1.1 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 visualise data either in the model itself or in a separate Eclipse View. Additionally, a trace recorder may record and later play back simulation data in order to analyse a simulation run in detail.

```
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 4.1: The KIEM DataComponent Interface: In the `step()` method a component specifies its step-wise behaviour and may react to input data with some output data.

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 visualisations. *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.

Fig. 4.1 shows the fairly self-explanatory simple interface for DataComponents. A component needs to declare whether it is an observer or a producer of data. It should declare some initialisation and wrap-up code. The step method is most significant in this interface. It should implement the execution behaviour 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. 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 saveable *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 step-wise, 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

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 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 organised 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, transparently to the component itself.

As denoted by Fishwick [7] *internal* and *external* events of a model are distinguished, where with the internal events the inputs come from a lower (abstraction) level of the model and not from outside with respect to the model itself as for the external events. In the case of the Data Pool, external input events can be considered the data changes observed by an observer DataComponent. External output events can be considered the

Figure 4.2: Mobile Data Table application visualising the synchronous ABRO program (see Fig. 1.2) simulation results (from Motika [15]).

modifications of data values in the Data Pool as the effect of the returned data by a producer DataComponent.

4.4 Linear Scheduling

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 iterative simulation executions are supported—all components (e.g., a simulation or visualisation engine) should also preserve this iterative characteristic. During an execution, KIEM will step-wise 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 synchronise 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., synchronise) 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 (non-existent) output.

4.5 Further Concepts

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

Analysis and Validation 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[1] application (see Fig. 4.2) that can fully interact with the Execution Manager.

Flexible Data Representation Synchronous signal data are easily represented within the data model of KIEM. In a synchronous setting, a signal has not only a value, but additionally a status, which denotes it to be present or absent. For this purpose one just needs to find a common representation within the used JSON format. Because signal presence is made explicit in the Data Pool of the Execution Manager, data components need to make sure to reset signals to be absent for a next synchronous tick. This can be done by introducing a separate DataComponent which resets all present signals in the beginning of a step-computation. In special cases this may also be done by the communicating DataComponents itself. We introduced such a specification as used in the example of Section 4.1 and built several DataComponents that are able to interact following the synchronous semantics of their external signals (see Sec. 4.3) within an execution of KIEM.

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/cyber-physical systems.

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

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 analysing situations better. For example, when a validation observer DataComponent pauses the execution because a special condition holds, one may want to analyse 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 behaviour models. We gave a short introduction into used concepts and into the Ptolemy suite as a multi-domain, highly flexible and extensible modelling environment with formally founded semantics. In order not to reinvent the wheel, we propose to utilise these existing features in an integrated way for the specification of denotational DSL 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 executions are seamlessly integrated into KIELER and therewith into the Eclipse platform.

We plan to further evaluate 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 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, e. g., for the Esterel synchronous language, into Eclipse using the presented KIEM infrastructure.

Bibliography

[1] C. André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[2] G. Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner.*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

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

[4] B. Combemale, X. Cregut, J.-P. Giacometti, P. Michel, and M. 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.

[5] H. Eichler, M. Scheidgen, and M. Soden. *A Meta-Modelling Framework for Modelling Semantics in the contect of Existing Domain Platforms.* Department of Computer Science, Humboldt-Universität zu Berlin, 2006.

[6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[7] P. A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

[8] H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, pages 116–140, 2010.

[9] H. Fuhrmann and R. von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 196–210, 2010.

[10] A. Girault, B. Lee, and E. 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.

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

[12] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, Apr. 1990.

[13] 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.

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

[15] C. Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf`.

[16] C. Motika, H. Fuhrmann, and R. von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010*, pages 891–896, 2010.

[17] M. Scheidgen and J. Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Model Driven Architecture-Foundations and Applications*, volume 4530 of *LNCS*. Springer-Verlag, 2007.