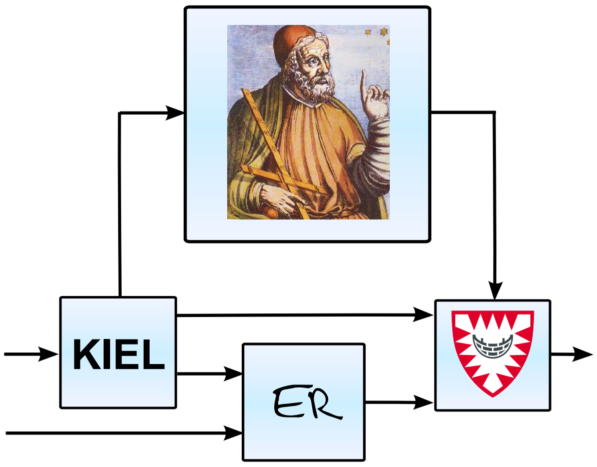
Semantics and Execution of Domain Specific Models

KlePto and an Execution Framework

Diploma Thesis of Christian Motika



Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

19th December 2009
Advised by: Dipl.-Inf. Hauke Fuhrmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

In this diploma thesis a two-level approach is presented that allows to extend the abstract syntax of models in a way to provide simulations with low effort. On a first level, a generic strategy for implementing simulation engines of behavior models is illustrated by two case studies. Semantic model specifications and a runtime interfacing to the Ptolemy II tool suite used as a simulation back-end is shown in detail. This is done by combining already existing technologies while not introducing any new kind of language or notation.

On a second level, a light-weight execution framework for iterable models with a generic user interface allows the tool smith to provide and use arbitrary execution and visualization engine implementations for his or her Domain Specific Language (DSL). Various reusable implementation examples are given.

All parts of this project are contributions to the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. Hence they are open source extensions to the Eclipse modeling projects.

Key words modeling language, domain specific, DSL, semantics, execution, simulation, Eclipse, KIELER, Ptolemy II

Contents

Abbreviations

ABRO	The *hello world* synchronous paradigm
API	application programming interface
ATL	Atlas Transformation Language
BFS	breadth-first search algorithm
CASE	Critical Applications & SystEms Development
CAN	Controller Area Network
CT	Continuous Time
DE	Discrete Events
DSL	Domain Specific Language
DTD	Document Type Definition
EMF	Eclipse Modeling Framework
FSM	Finite-State-Machine
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GUI	graphical user interface
IDE	integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
I/O	input/output
JSON	Java Script Object Notation
KlePto	KIELER leveraging Ptolemy
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager

M2M	model-to-model
M2T	model-to-text
MDSD	Model Driven Software Development
ME	Micro Edition
MoC	model of computation
MOF	Meta Object Facility
MOML	modeling markup language
MVC	Model-View-Controller
oAW	openArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
QVT	Query/View/Transformations
PHP	PHP Hypertext Preprocessor
PN	Process Networks
RCP	Rich Client Platform
RTI	runtime infrastructure
SCADE	Safety Critical Application Development Environment
SDF	Synchronous Dataflow
SR	Synchronous Reactive
SSM	Safe State Machines
TCP	Transmission Control Protocol
TMF	Textual Modeling Framework
TTP	Time Triggered Protocol
UI	user interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier

VHDL	Very High Speed Integrated Circuit Hardware Description Language
XML	Extensible Markup Language

List of Figures

Listings

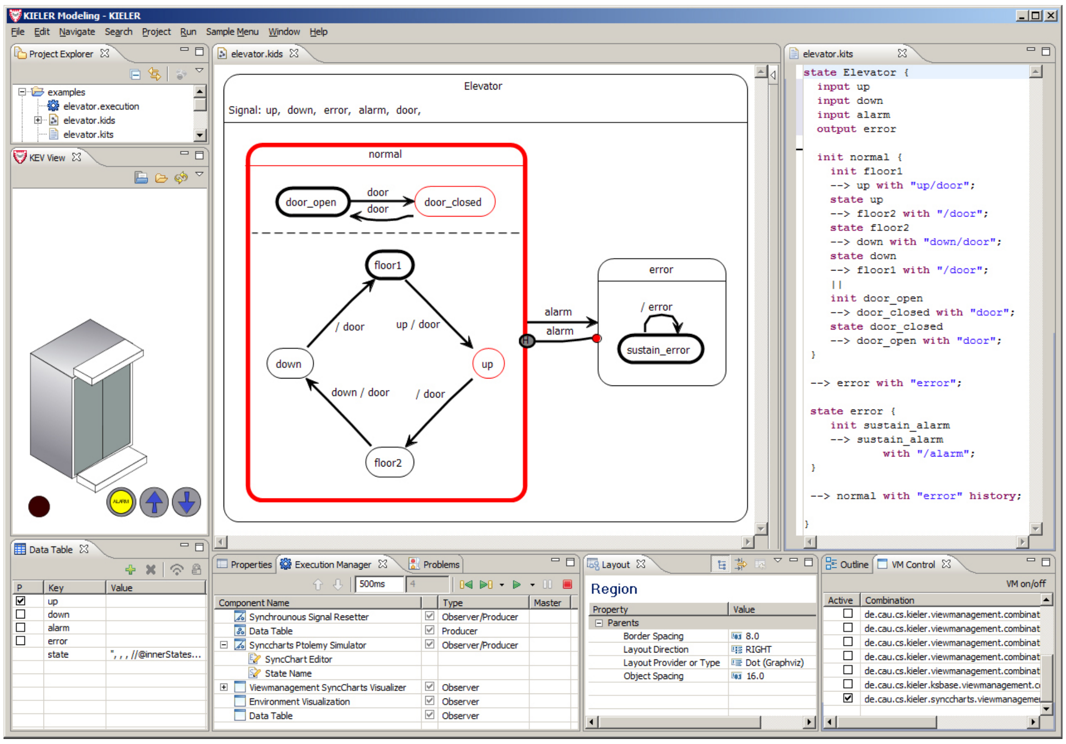
List of Tables

1 Introduction

Computer simulations [22] 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 (e. g., weather forecasts or flight simulators). On the other hand one aspires to emulate computer systems—often embedded ones—themselves prior to their physical integration in order to increase safety and cost effectiveness (e. g., airbag controller or mars robot).

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.



elevator.kits

```
state Elevator {
  input up
  input down
  input alarm
  output error

  init normal {
    init floor1
    --> up with "up/door";
    state up
    --> floor2 with "/door";
    state floor2
    --> down with "down/door";
    state down
    --> floor1 with "/door";
    ||
    init door_open
    --> door_closed with "door";
    state door_closed
    --> door_open with "door";
 }

 --> error with "error";

 state error {
    init sustain_alarm
    --> sustain_alarm
            with "/alarm";
 }

 --> normal with "error" history;

}
```

P	Key	Value
☑	up	
☐	down	
☐	alarm	
☐	error	
	state	", , , //@innerStates...

500ms 4

Component Name		Type	Master
Synchrounous Signal Resetter	☑	Observer/Producer	
Data Table	☑	Producer	
Syncharts Ptolemy Simulator	☑	Observer/Producer	
SyncChart Editor			
State Name			
Viewmanagement SyncCharts Visualizer	☑	Observer	
Environment Visualization	☑	Observer	
Data Table	☑	Observer	

Region

Property	Value
Parents	
Border Spacing	8.0
Layout Direction	RIGHT
Layout Provider or Type	Dot (Graphviz)
Object Spacing	16.0

Active	Combination
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.ksbase.viewmanagement.c
☑	de.cau.cs.kieler.syncharts.viewmanageme

Figure 1.1: GUI of KIELER and the KIEM Eclipse plug-in during a simulation run

1.1 KIELER Framework

The approach of this work is implemented in the context of 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 [23].

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 Eclipse Modeling Framework (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 general, generic interfaces and specification possibilities for semantics themselves need to be found. This is the purpose of the KIELER Execution Manager (KIEM) and KIELER leveraging Ptolemy (KlePto), the execution and simulation approach presented in this work.

Fig. 1.1 shows the KIELER GUI during a simulation run with KlePto and KIEM. The latter provides a user interface, shown in the bottom Eclipse View of Fig. 1.1. The former provides visual feedback about computed simulation details. These are presented both in the graphical model editor itself and in separate Eclipse Views like the data table (or optional environment visualizations).

The context of both projects is depicted in Fig. 1.2 that illustrates the Model-View-Controller (MVC) pattern used in KIELER, where the simulation of models is part of the controller aspects.

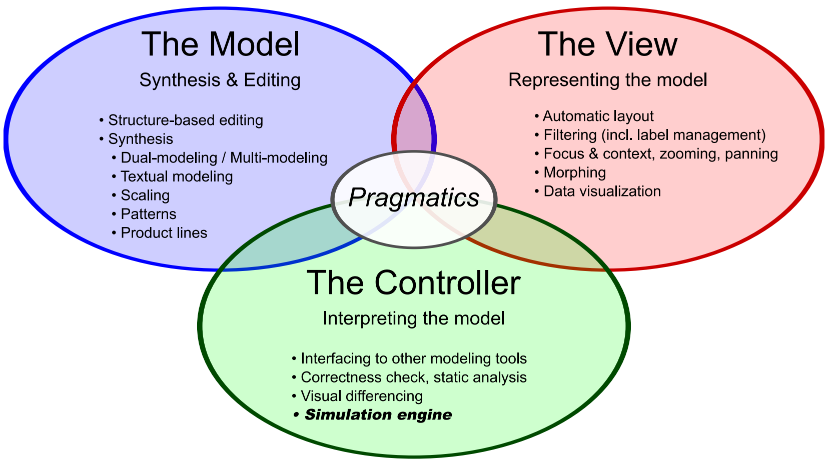


Figure 1.2: MVC in KIELER: The Execution Manager belongs to the controller part (from [23])

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

1.2 Problem Statement

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 sometimes they are too general and complex to be widely accepted.

As a recent development, Domain Specific Languages (DSLs) 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, such as model transformations, validation, persistence, and versioning. The designer of tools regarding such a DSL, 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. This 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 also has to be defined in order to let a computer execute[2] such models. For the specification of the latter no well established approaches exist yet. But as semantics often exists at least implicitly in the mind of the DSL-tool-designer there is a need to provide a way for making it explicit. Additionally, there should be a common way to simulate and validate models once the semantical aspects are covered.

The contribution of this work is twofold: On the one hand it is a proposal on how DSL semantics can be defined flexibly and in a denotational sense by utilizing existing *semantic domains* without introducing any new kind of language or notation. On the other hand, a light-weight and extensible execution framework is presented that allows a common way to specify components in order to simulate, validate, and dynamically visualize models of a DSL.

The following claims can be made for the solutions to both parts:

Model semantics:

1. Simplicity: Avoid new language or notation
2. Flexibility: Avoid concrete language
3. Usability: Favor common technologies
4. Extensibility: Avoid closed specifications, favor modular ones

[2]Model simulation requires an executable model. A model is executable if a mappable representation exists that a computer is able to execute (e. g., a generated and compiled program).
In other words a model is executable if an execution semantics exists for it (s. Sec. 2.2).

5. Comprehensibility: Concern target group, no too formal notations
6. Abstraction: Keep abstraction level as high as possible

Model execution:

1. Simplicity: Keep interfaces simple
2. Flexibility: Develop tailored data representations
3. Extensibility: Use standards for interfaces
4. Comprehensibility: Emphasize traceable framework behavior
5. Uniformity: Avoid usage of numerous languages and technologies
6. Interactivity: Focus on discrete, stepwise executions

1.3 Outline of this Document

In Chap. 2 some major model simulation frameworks that partially influenced this work are presented, including Esterel Studio, Ptolemy II, SCADE, Matlab/Simulink, Topcased, and KIEL. Additionally, different approaches to specify execution semantics and model transformations are compared, while reasoning about design decisions.

Chap. 3 introduces the technologies used throughout the implementation. Firstly, it gives an overview of Eclipse, its workbench and the utilized plug-in mechanism. Secondly, it introduces fundamental Eclipse technologies, such as EMF, GMF and transformation languages like Xpand and Xtend. An outline of Ptolemy, its computational strengths and an association to the Eclipse world follows. Finally, the JSON standard as an efficient data exchange format is sketched.

Two case studies are presented in Chap. 4, exemplarily illustrating the common KlePto approach to denotational specify execution semantics. In the first, a language to describe railway controllers is covered. The second case study concerns a subset of a graphical synchronous modeling language (SyncCharts). For both case studies, the domain is presented first, followed by a language analysis. As this chapter emphasizes the Model Driven Software Development (MDSD) by utilizing the presented DSL toolchain, the automatic generated model editors are inspected next. Afterwards the transformation ideas are described and finally implementation details are given.

In Chap. 5 the KIEM architecture is studied. First a motivation and an overview of the component parts is given. The basic building blocks of this architecture, interfacing ideas, and the GUI are discussed. Ideas about the scheduling mechanism, data management, and features like extensibility and flexibility follow. Implementation details of all ideas are outlined afterwards. This chapter closes with several example implementations using the KIEM architecture.

Finally, Chap. 6 gives an outlook on future work and summarizes the results of this diploma thesis.

2 Related Work

As this work splits up into two major parts, this also applies to the related work chapter. In the first part, some major model simulation frameworks are presented. In the second one, different approaches on how to define execution semantics and model transformations are analyzed and compared in short, in order to justify design decisions made in this work.

2.1 Simulation Frameworks

There exist a lot of modeling tools that provide discrete simulation runs for their domain models. This means that the (graphical) model is executed stepwise. There are two classifications of models, fitting to discrete execution semantics:

- Data-flow models usually are communicating components linked by channels. This is where the *data flow*. In a synchronous communication setting, each component may interact once during an execution step. In an asynchronous communication setting, an execution step could mean: one interaction of each distinct component (e. g., Kahn process networks [34]).
- Control-flow models usually are states linked by transitions. Control changes by state-to-state transitions. This is where the *control flows*. There are several ways to compute this stepwise, such as finding a fixed point or taking at most one enabled transition per execution step (e. g., Ptolemy ModalModel domain or SR domain, see Sec. 3.2).

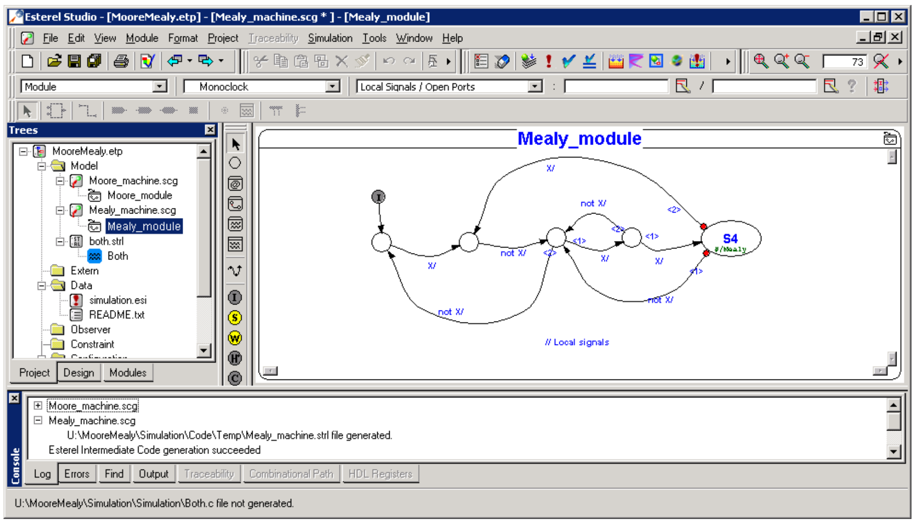
In the following some popular tools supporting simulations are mentioned.

2.1.1 Esterel Studio

Based on the Esterel [11] language, Esterel Studio [20] is an environment to design control-flow models. Mainly it was used to synthesis hardware, e. g., by exporting models to Very High Speed Integrated Circuit Hardware Description Language (VHDL) code. Formal verification is well integrated into this tool.

Simulation is also a built-in feature. In early compiler versions (Esterel 4), a VHDL based soft-prototype was generated out of the graphical control-flow model (Safe State Machines (SSM), a Statechart dialect) and then simulated with a generic hardware simulation engine.

Because SSMs can be seen as the commercial version of SyncCharts, the simulation visualization of active states inspired the visualization component of SyncCharts (see Sec. 4.4) used in this work.


Esterel Studio - [MooreMealy.etp] - [Mealy_machine.scg *] - [Mealy_module]
File Edit View Module Format Project Traceability Simulation Tools Window Help
Module
Monoclock
Local Signals / Open Ports
Trees
MooreMealy.etp
Model
Moore_machine.scg
Moore_module
Mealy_machine.scg
Mealy_module
both.strl
Both
Extern
Data
simulation.esi
README.txt
Observer
Constraint
Project Design Modules
Mealy_module
X/
not X/
<2>
<1>
S4
// Local signals
Console
Log Errors Find Output Traceability Combinational Path HDL Registers

```
⊞ Moore_machine.scg
⊟ Mealy_machine.scg
    U:\MooreMealy\Simulation\Code\Temp\Mealy_machine.strl file generated.
Esterel Intermediate Code generation succeeded
```

```
U:\MooreMealy\Simulation\Simulation\Both.c file not generated.
```

Figure 2.1: Esterel Studio GUI

2.1.2 Ptolemy II

Ptolemy II [36] is a framework that supports heterogeneous modeling, simulation, and design of concurrent systems. For integrated simulation purposes Ptolemy contains a graphical editor called *Vergil*. But there also exists the possibility to embed the execution of Ptolemy models into arbitrary Java applications. As Ptolemy II is the most current development, it will just be called *Ptolemy* throughout the rest of this work and implicitly the Java based Ptolemy II is meant, *not* it's C based Ptolemy Classic predecessor.

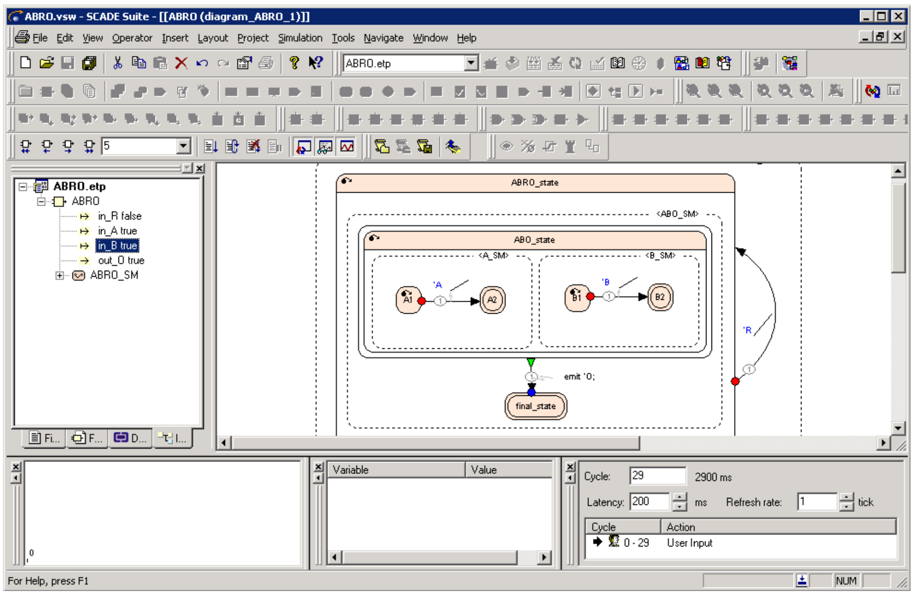
As the Ptolemy framework has not only inspired this work but also is used as a simulation back-end, it is discussed in more details in Chapter 3.

Most interesting is the combination of Java and the ability to *model* complex concurrent systems under several (combinable) model of computations (MoCs).

2.1.3 SCADE

SCADE [21] (Safety Critical Application Development Environment) is a tool for graphically defining synchronous data-flow models. Originally Lustre [26] was the underlying data-flow language and the SCADE language was a graphical representation of it [16]. By now, SCADE evolved to an own, independent language where control-flow and data-flow parts are can be arbitrarily mixed with each other.

The SCADE suite mainly offers a certified code generator that for example can produce C code. Compiled, this code can also be used to simulate a SCADE model within the tool, as shown in Fig. 2.2. As the data-flow components are very modular, each component (that also may contain others) can be simulated. Inputs of the component are treated as environmental inputs, outputs are treated as environmental


ABRO.vsw - SCADE Suite - [[ABRO (diagram_ABRO_1)]]
File Edit View Operator Insert Layout Project Simulation Tools Navigate Window Help
ABRO_state
<ABO_SM>
ABO_state
<A_SM>
<B_SM>
A1
A2
B1
B2
'A
'B
'R
emit 'O;
final_state
For Help, press F1
NUM

ABRO.etp
ABRO
in_R false
in_A true
in_B true
out_O true
ABRO_SM

Variable	Value

Cycle: 29 2900 ms
Latency: 200 ms Refresh rate: 1 tick

Cycle	Action
0 - 29	User Input

Figure 2.2: Simulation in the SCADE suite

outputs. In order to interface with the SCADE simulation, the Transmission Control Protocol (TCP) and the internal data representation can be used.

In the context of this work, the KIEM execution framework tries to simplify interfacing by using a standardized data exchange format and common Eclipse technologies. TCP communication can be bridged but is not mandatory. The simulation handling of the stepwise executing gave much inspirations for this work. This was extended by also allowing the user to make execution steps backwards (to already computed values) in a generic manner.

2.1.4 Matlab/Simulink/Stateflow

Like SCADE, Matlab [42]/Simulink and Stateflow [43] is a toolchain already used a lot in industry, but mostly in the automotive area. Matlab is mainly used for technical calculations and the Simulink extension (see Fig. 2.3) brings in graphical data-flow models where components represent more or less complex mathematical calculation functions.

The simulation here works different to other tools as the user first defines the simulation duration and later may inspect, interpret or analyze the resulting values. The advantage is that not only discrete time models but also continuous ones can be simulated. The drawback is that a stepwise execution (that allows user interaction) may only be emulated on already computed results.

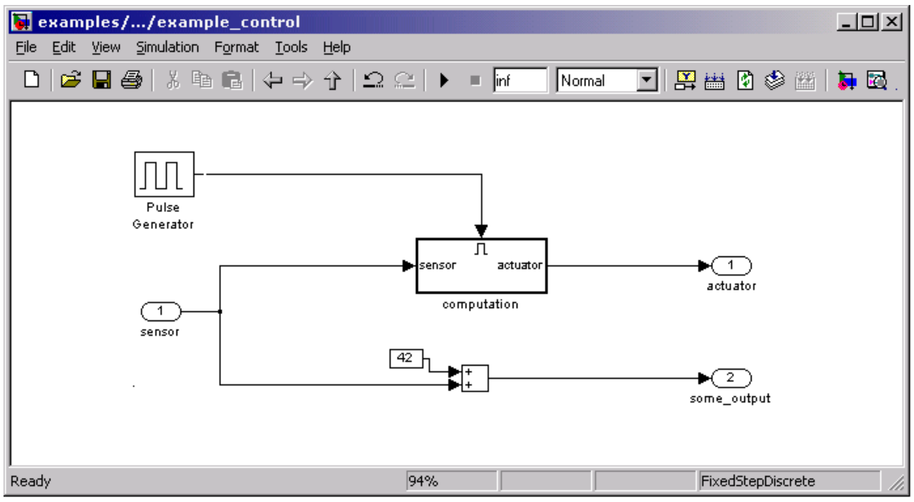


Figure 2.3: Simulation GUI of Matlab/Simulink tool

2.1.5 Topcased

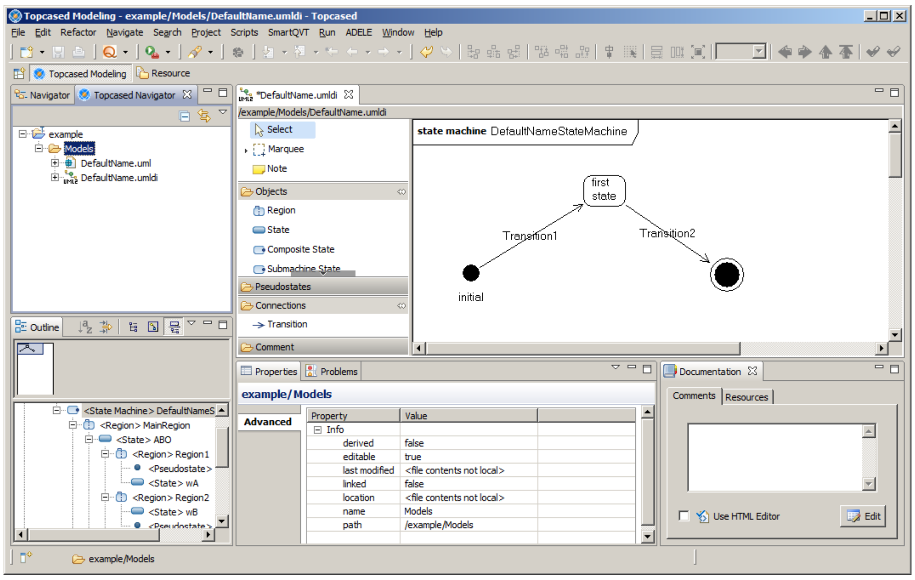
The Topcased[1] project is a software environment primarily for constructing critical embedded systems. It is used in the aviation industry and offers hardware/software co-design. Topcased is based on the Eclipse framework and targets the model driven development with simulation as the key feature for validating models [17]. It focuses on the development of a modular and generic Critical Applications & SystEms Development (CASE) environment. For this, the toolkit is based on the Eclipse Modeling Project. Topcased can not only be used to model a system but also to model the environment and scenarios.

This inspired the execution framework presented in this work significantly, as it will try to map all three entities to interacting components with one and the same underlying exchange data model. In Fig 2.4, the Eclipse based Topcased GUI is shown.

2.1.6 Modelica

Modelica[2] is an environment to describe and simulate physical models. It is based on object-oriented equations to describe these kinds of systems. Modelica is increasingly being used in the automotive industry. Simulations are based on a mathematical representations that can be solved by Modelica simulation environments, such as MathModelica or SCICOS. This is comparable to the Simulink approach and there exist possibilities to exchange models between both tools.

[1]http://www.topcased.org
[2]http://www.modelica.org/



Property	Value
Info	
derived	false
editable	true
last modified	<file contents not local>
linked	false
location	<file contents not local>
name	Models
path	/example/Models

Figure 2.4: Eclipse GUI of Topcased tool

2.1.7 KIEL

The KIEL project [50] presents an alternative paradigm for visualizing Statecharts [27] during simulation. It follows the basic idea to dynamically construct a view of the system model that includes all active states (the focus) and their parent states (the context). The project's implementation is a good but monolithic Java tool. As a successor, the KIELER project tries to strengthen extensibility and allow for more generic constructs.

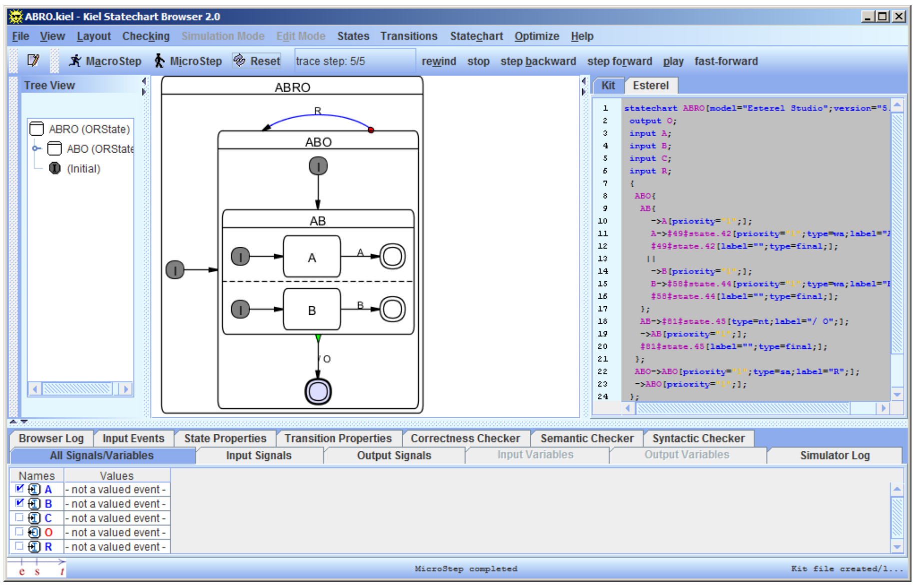
Hence as much ideas, strategies, and technology as possible have been reused from the KIEL project. Fig. 2.5 shows the simulation of a Statechart within the KIEL tool.

2.2 Execution Semantics

According to Webster's, *semantics* is the "study of meanings".

The term *execution semantics* (of a programming language) is usually understood as the definition of the implicated program behavior based on various language constructs.

Hence, in order to simulate a model, semantics must be clear. Usually semantics is defined for model languages (or tools supporting them) either informal or formal. For an informal description, natural language is often sufficient. For a formal


ABRO.kiel - Kiel Statechart Browser 2.0
File View Layout Checking Simulation Mode Edit Mode States Transitions Statechart Optimize Help
MacroStep MicroStep Reset trace step: 5/5 rewind stop step backward step forward play fast-forward
Tree View
ABRO (ORState)
ABO (ORState
(Initial)
ABRO
R
ABO
AB
A
A
B
B
O
Kit Esterel
Browser Log Input Events State Properties Transition Properties Correctness Checker Semantic Checker Syntactic Checker
All Signals/Variables Input Signals Output Signals Input Variables Output Variables Simulator Log
e s t
MicroStep completed
Kit file created/1...

```
statechart ABRO[model="Esterel Studio";version="5.
 output O;
 input A;
 input B;
 input C;
 input R;
 {
  ABO{
   AB{
     ->A[priority="1";];
     A->$49$state.42[priority="1";type=wa;label="
     $49$state.42[label="";type=final;];
    ||
     ->B[priority="1";];
     B->$58$state.44[priority="1";type=wa;label="
     $58$state.44[label="";type=final;];
   };
   AB->$81$state.45[type=nt;label="/ O";];
   ->AB[priority="1";];
   $81$state.45[label="";type=final;];
  };
  ABO->ABO[priority="1";type=sa;label="R";];
  ->ABO[priority="1";];
 };
```

Names	Values
✓ A	- not a valued event -
✓ B	- not a valued event -
C	- not a valued event -
O	- not a valued event -
R	- not a valued event -

Figure 2.5: Simulation in the KIEL GUI

definition, one distinguishes between:

1. Axiomatic semantics
2. Operational semantics
3. Denotational semantics

In the first, logical axioms are used to represent model or language constructs where Hoare logic [29] is a common paradigm. In the second, the execution of a model or language construct is described directly (often in a mathematical way). The lambda calculus [31] or an abstract state machine are examples here. In the third, each model or language construct is transformed into a denotation [55], a target language, often a mathematical one.

In the area of semantics w.r.t. programming or model languages numerous works exist. In the following, exemplarily two different approaches are compared that refer to this work. Before this, it is shown how semantics affects the simulation of current modeling tools presented in the last section. Because this work is based on a denotational semantic definition using model transformations, finally some M2M transformation approaches are compared.

2.2.1 Simulation Tools

Most of the tools presented in Sec. 2.1 are specific and follow clear semantics. This enables such tools to come up with a tailored simulation engine that is able to execute the models following this concrete semantics.

For example, SCADE defines a graphical notation for the synchronous data-flow and control-flow SCADE-language. For this, formal semantics exist and the C code generation delivers an executable simulation of the model. Similar assertions apply to Esterel Studio.

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, it was investigated further as a generic semantic back-end in combination with the Eclipse modeling projects as elaborated in Chap. 4.

2.2.2 Different Approaches

As mentioned in Scheidgen and Fischer [51], two fundamentally different concepts (relating to the classifications from above) can be emphasized for specifying model semantics primarily in the context of DSLs:

1. Model transformation into a *semantic domain*. In this 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 or/and because there exist formal semantics like in the case of Ptolemy). This relates closely to a denotational semantic definition.

2. Provision of a new action language, i. e., an operational semantic definition. This 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 given specifications).

In Chen at al. [14] the latter is a compositional approach for specifying the model behavior. The *M3Action* framework for defining operational semantics[3] is illustrated by Eichler at al. [28].

Although defining a new high-level action language for transforming a *runtime model* during execution is a very interesting field and retains a stricter separation between the different abstraction levels, in the context of this work it was decided to follow the more natural approach. That is, leveraging an existing semantic domain and specifying model transformations with necessary inter-abstraction-level mapping links to the model in question.

Some advantages can be identified:

1. There is no need to define any new language to express semantics on the meta model abstraction level.

[3]`http://www.metamodels.de`

2. There is a quite direct connection for meta model elements and their counterparts in the semantic domain that allows easy traceability.

3. The expressiveness is not limited by a concrete language due to the flexibility of conceptionally using any semantic domain (e. g., also well-known ones).

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).

2.2.3 Model Transformations

According to the presented and compared approaches and for the above reasons, the model transformation approach was chosen in this work. Hence a way to describe such transformations is essential.

These play a key role in generative MDSD. They 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.

Because the context of this work, namely KIELER, is based on Eclipse and EMF, a model-to-model (M2M) transformation must be chosen that is or can be well integrated into both.

Czarnecki and Helsen [18] give a feature based survey of model transformation approaches by comparing several transformation languages in the sense of for example:

- Domain (e. g., string patterns or typed variables)
- Syntactic separation of parts in a rule (e. g., for the source and for the target model)
- Multidirectionality (e. g., execute a rule in both directions)
- Parametrization (e. g., allow control parameters for specifying alternative patterns)

These and other features are explained and illustrated in the context of a selection of transformation languages. They can be used in order to help with the decision which language may be suited best for a project.

Prominent Eclipse and EMF based transformation languages are:

Atlas Transformation Language (ATL): ATL [12] is part of the Eclipse M2M Project[4] combining imperative and declarative programming aspects.

The Object Constraint Language (OCL)[5] is used to query models. The transformation consists of a set of rules applying to the elements of the specific source metamodel.

[4]`http://www.eclipse.org/m2m/`
[5]`http://www.omg.org/spec/OCL/2.0/`

Query/View/Transformations (QVT): This is part of the Eclipse M2M Project and also part of the Meta Object Facility (MOF)[6]. It conforms to the Object Management Group (OMG) standard. It also has a declarative and an imperative part.

Xpand/Xtend: These are both parts of the Eclipse Modeling Project. Xpand is a template language to perform M2T transformations. Xtend is a functional language to extend metamodels and to perform M2M transformations. Querying models is done in an OCL-like manner.

To sum up, ATL differs from QVT mainly by the syntax where QVT complies with the OMG standard [45]. ATL is the one that has been used for a longer time in the Eclipse context while QVT is not fully implemented yet.

Xpand and Xtend evolved from the openArchitectureWare (oAW) project and are totally different approaches. They recently have been seamlessly integrated into Eclipse. Because Xtend is not only a transformation language, it can be used to extend metamodels. This way for example Xpand templates can be simplified using Xtend constructs. Xtend can also be extended by using Java wherever limits of the functional expressiveness may be reached. These languages have also a long and successful history within the oAW project already. They are based on the same type system and the same expression language (e. g., to query models).

In this work both M2T and M2M transformations are used. Additionally, in the context of KIELER several parts (e. g., metamodel extensions) should be reusable. This leads to the decision to use Xpand and Xtend for model transformations.

Nevertheless, the Eclipse based approach presented in this work is conceptually open to use any transformation language that supports EMF meta models.

2.2.4 Simulation High-Level Architecture

Our architecture is related to the IEEE standard for modeling and simulation high-level architecture [33]. The KIELER Execution Manager (KIEM) presented in Sec. 5 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. Fig. 5.1 on page 78 gives an overview of this infrastructure provided by KIEM.

[6] `http://www.omg.org/mof/`

3 Used Technologies

Before diving into the explanation of the main ideas or even any implementation details of this work, it is necessary to introduce some key technologies used throughout the next chapters. Because these cannot all be discussed in their full depth in the context of this work, often only a sketch of the most important ideas and eligible use cases are given with some references for further reading.

3.1 Eclipse

Because KIELER is an Eclipse based framework and the work of this project is a part of the former, a short introduction to Eclipse seems in order.

Eclipse is a platform, well known as *the* Java IDE. It itself is implemented in the Java language but by now has evolved to be a development tool for various other languages (e. g., C++, PHP Hypertext Preprocessor (PHP), Extensible Markup Language (XML)). It can also be seen as a framework for *building* IDEs. This is summarized by the common principle that Eclipse is "an IDE for anything, and nothing in particular" [47].

3.1.1 Plug-ins

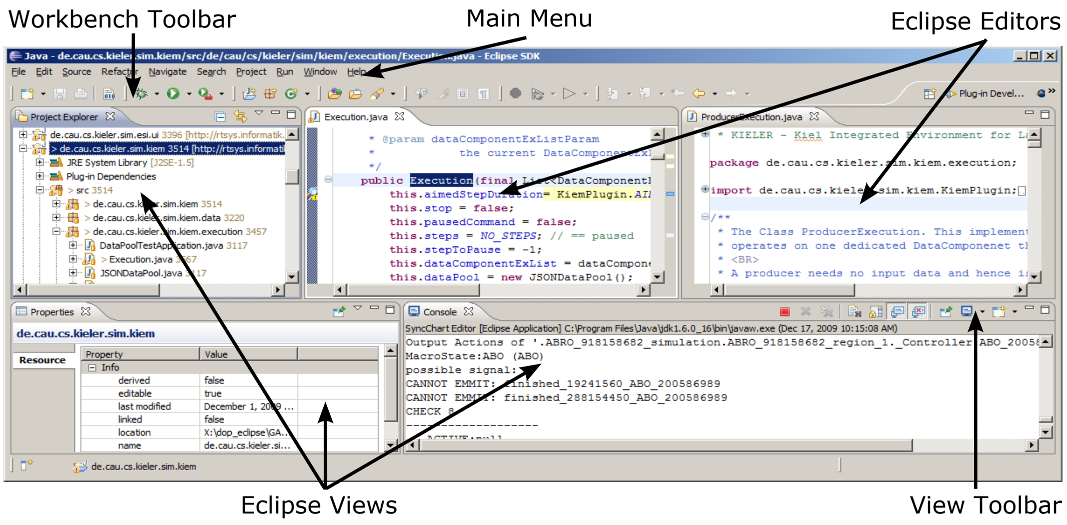
The building blocks of Eclipse are components called *plug-ins*. The basic Eclipse platform consists of a small number of such plug-ins and is extended by other plug-ins. This architecture is flexible, modular, and extendable at the same time because plug-ins can be replaced, plug-ins can extend other plug-ins and every plug-in can be seen as a more or less independent puzzle piece of the overall application.

For each plug-in, the following information is necessary:

- What other plug-ins does it depend on, e. g., what functionality of other plug-ins does it use?
- What functionality does it add?
- What of this functionality may be extended by yet other plug-ins, e. g., what *extension points* does it offer?

This kind of *meta* information is not part of the Java language. It is used to describe the interrelation within the overall architecture. The *platform runtime engine* needs this information because it loads and runs plug-ins. This runtime engine exists on top of the OSGi[1] Service Platform, where plug-ins (also called *bundles*) can be installed and removed.

The necessary meta information of a bundle is stored in the XML based file `plugin.xml`. In addition to the dependency information, it contains a name and an id of the respective plug-in.



```
     * @param dataComponentExListParam
     *            the current DataComponentExL
     */
    public Execution(final List<DataComponentE
        this.aimedStepDuration= KiemPlugin.AIM
        this.stop = false;
        this.pausedCommand = false;
        this.steps = NO_STEPS; // == paused
        this.stepToPause = -1;
        this.dataComponentExList = dataCompone
        this.dataPool = new JSONDataPool();
```

```
 * KIELER - Kiel Integrated Environment for La

package de.cau.cs.kieler.sim.kiem.execution;

import de.cau.cs.kieler.sim.kiem.KiemPlugin;

/**
 * The Class ProducerExecution. This implement
 * operates on one dedicated DataComponenet th
 * <BR>
 * A producer needs no input data and hence is
```

```
SyncChart Editor [Eclipse Application] C:\Program Files\Java\jdk1.6.0_16\bin\javaw.exe (Dec 17, 2009 10:15:08 AM)
Output Actions of '.ABRO_918158682_simulation.ABRO_918158682_region_1._Controller ABO_20058
MacroState:ABO (ABO)
possible signal:
CANNOT EMMIT: finished_19241560_ABO_200586989
CANNOT EMMIT: finished_288154450_ABO_200586989
CHECK 8
-------------------
```

de.cau.cs.kieler.sim.kiem

Property	Value
Info	
derived	false
editable	true
last modified	December 1, 2009 ...
linked	false
location	X:\dop_eclipse\GA...
name	de.cau.cs.kieler.si...

Figure 3.1: Eclipse Workbench with Editors and Views

3.1.2 Rich Client Platform and Workbench

The Eclipse Rich Client Platform (RCP) allows to build IDEs using the plug-in architecture described above. To reuse plug-ins in another context (e. g., their *own* context), it describes a minimal set of bundles needed for an independent Eclipse based RCP application.

The *Workbench* is among this minimal set of basic plug-in components. It can simplified be seen as the *main window* or *desktop* of an Eclipse application. All parts of the Workbench may look similar at first glance, but are basically of two different types:

Eclipse Editor: Editors are integrated into the Workbench, meaning that all contributions (e. g., toolbar icons) are embedded into the overall workbench, when a specific editor is *active* (i. e., it has the focus). Editors cannot be decoupled (e. g., dragged out) from the Workbench. There exists an editor pane that all

[1]Open Services Gateway initiative: `http://www.osgi.org/`

editors share. They can be arranged within this pane but not outside of it. Editors can be registered to file types and hence appear automatically if the user opens a file of a respective type. Several Editor instances (with different contents) may be opened at the same time. In Fig. 3.1 two editors can be seen in the upper right editor pane area.

Eclipse View: Views are not as integrated into the Workbench as Editors are. They usually present additional or different information about the contents of the currently active editor (or even a selected object within). Typically there exists at most one instance of a View. Views usually have their own toolbar (but can also contribute to the Workbench's toolbar and Main Menu). The position of Views can be arranged by the user more freely. He or she may also drag them out of the Workbench. In this case they are displayed in their own window. In Fig. 3.1 several Views (e. g., the Console View, Properties View or the Project Explorer View) are arranged below the editor pane or left to it.

To retrieve additional and more detailed information about Eclipse, one should visit the Eclipse Website[2] or be referred to some common Eclipse literature [15].

3.1.3 The Eclipse Modeling Framework (EMF)

A basic prerequisite for MDSD are models that are based on metamodels. Metamodels define the abstract syntax of models and hence allow the specification of languages as object-oriented structure models. The Meta Object Facility is such a metamodeling framework defined by the OMG [46], which has been taken shape for the Eclipse world as the Eclipse Modeling Framework (EMF).

The EMF is *the* commonly accepted modeling framework for Eclipse. Because there are few comparable competitors and the EMF is a basis for several other technologies, it is sensible to use it in an Eclipse based project that wants to utilize the interchangeability within the Java, Eclipse, and EMF based world.

Metamodel

EMF unifies Java, XML, and UML and lets the user define an EMF model in all of these forms while it is able to generate the other representations out of it. This makes it easy to start with EMF as one can choose to:

- Directly use an EMF tree editor,
- Use a graphical editor,
- Import an XML schema definition, or
- Extract an EMF model from a Java implementation.

[2]`http://www.eclipse.org`

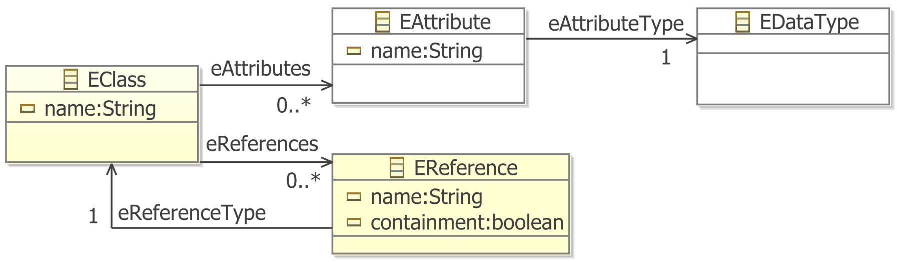


Figure 3.2: Simplified Ecore metamodel subset

An EMF model is represented by *Ecore* (see. Fig 3.2). Because EMF bootstraps itself, Ecore is again an EMF model.

The term metamodel and meta-metamodel might always seem a bit confusing but it simply means that some model is a model for other models. Because an EMF model (in Ecore) is often used to describe a set of other models, for these, it serves as a *metamodel*. Hence, the other models can also be considered *metamodel instances*.

Fig. 3.2 shows a simplified subset of Ecore with the following four classes:

1. **EClass** represents model classes having a name and optionally having attributes and references,

2. **EReference** represents an association between classes, has a name, may be of type containment and has exactly one target class,

3. **EAttribute** represents an attribute having a name and being of exactly one type, and

4. **EDataType** represents the type of an attribute, which may be for example int or String.

This (almost) is sufficient to define EMF models that are supposed to serve as metamodels. Hence one can use Ecore to define the abstract syntax of a modeling language.

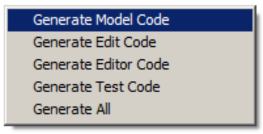
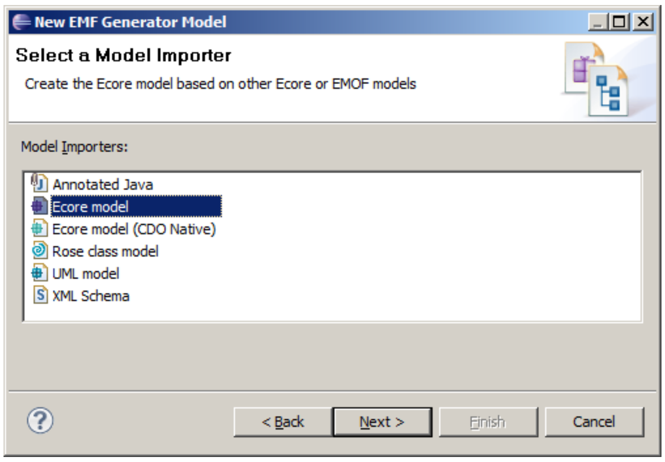


Figure 3.3: EMF generator context menu options

The EMF Generator

EMF can take the Ecore representation and generate (e. g., Java) code out of it. This is why the Eclipse Modeling Framework can also be considered a code generation facility. The generated Java implementation already contains getter and setter methods and other complete code to load (parse) and save (serialize) models (i. e., metamodel instances of the original EMF model). Fig. 3.3 shows the context menu options of an EMF generator model. The latter is an EMF model, enriched with additional information such as name spaces or package names. With the code generator one can not only generate the other representations (Generate Model Code) but also the loading and saving features (Generate Edit Code) and a ready to use tree editor (Generate Editor Code) or even code to test models (Generate Test Code).

As stated before, there are various import options to start from. The import dialog of EMF is depicted in Fig. 3.4 and shows these numerous alternatives.


New EMF Generator Model

Select a Model Importer

Create the Ecore model based on other Ecore or EMOF models

Model Importers:
- Annotated Java
- Ecore model
- Ecore model (CDO Native)
- Rose class model
- UML model
- XML Schema

Figure 3.4: EMF generator model import

Metamodels and DSLs

In the context of this work, all models are based on EMF. To describe the abstract syntax of a DSL, one needs to define an EMF model in Ecore. This is done for the case studies presented in Chap. 4. As graphical diagram representations of EMF models (like the one in Fig. 3.2) are easy to read, such diagrams will be presented in due course.

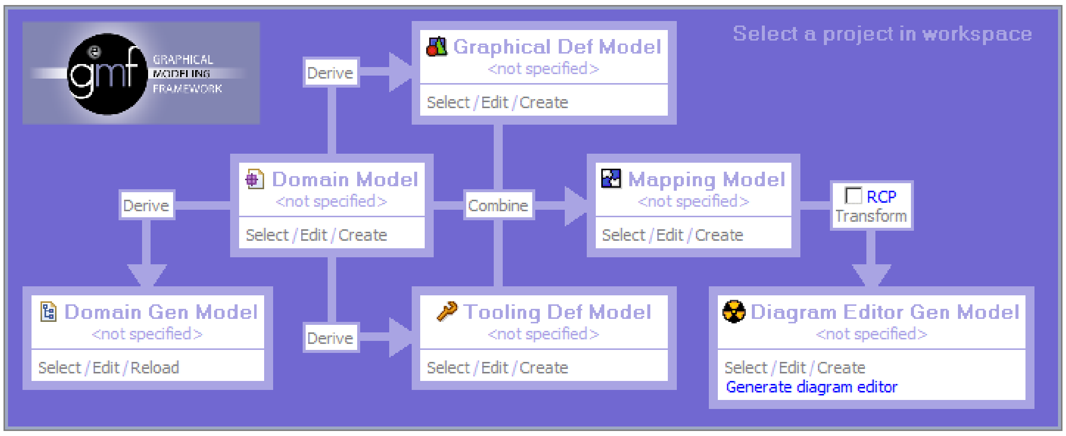


Figure 3.5: GMF Dashboard View for diagram editor generation

To gain detailed knowledge about the Eclipse Modeling Framework, the reader should visit the EMF Website[3] or refer to common EMF literature [54].

3.1.4 The Graphical Modeling Framework (GMF)

The Graphical Modeling Framework (GMF) is another framework built on top of Eclipse and EMF. It also exploits the Graphical Editing Framework (GEF) that facilitates the development of rich graphical editors. With the help of GEF and EMF, GMF provides a generative component and runtime infrastructure for developing rich graphical model editors.

The GMF Dashboard of Fig. 3.5 gives an overview of the generative toolchain that comes with that framework. Given an EMF model describing an abstract syntax, it can be used to generate a graphical editor out of the box. This editor serves the same purpose as the tree editor that can be generated by the EMF toolchain. It operates on models that are metamodel instances (i. e., they conform to the original metamodel in EMF). Because a graphical representation comes up with additional information such as the position and size of a node, this information is stored in a separate *notation* model.

Nodes and edges of the graphical model may represent `EClasses` of the EMF model. The latter is referred to as the *Domain Model* in Fig. 3.5. The graphical representation may be derived from it and is referred to as the *Graphical Definition Model*. In a graphical editor, additionally there are tools that allow the user to create new nodes of different types or connect some nodes with edges. These tools are defined in the *Tooling Definition Model*. Finally, the *Mapping Model* combines all three models and specifies their interrelations.

[3]`http://www.eclipse.org/modeling/emf/`

The *Diagram Editor Generator Model* is comparable to the EMF generator model. It is a copy of all four models, combined and enriched by some information that allows GMF to generate the diagram editor (i. e., the graphical model editor) out of it.

Graphical GMF editors can further be customized by modifying the Mapping Model or by using techniques like it was done for the SyncCharts editor of Fig. 4.8 by Schmeling [52]. For further information about GEF[4] and GMF[5], the Websites or adequate literature [25] should be consulted.

3.1.5 Xpand

Like EMF and GMF, Xpand is part of the Eclipse Modeling Project[6]. It is a statically-typed template language that focuses on the generation of textual artifacts from models. Xpand was originally developed as part of the oAW[7] project before all parts of oAW have been moved to Eclipse.

Xpand can be used together with various types of metamodels. Supporting EMF models natively, with Xpand textual artifacts (e. g., Java or C code) can easily be generated out of a model conforming to an EMF based metamodel.

The language back-end is based on Java and fully integrated into Eclipse, meaning that several adequate plug-ins exist for all purposes of Xpand, such as reading components, transformation components, pretty printer components, and writing components.

Type System and Expression Language

Xpand is based on an expression language and type system that is also shared with Xtend (s.b.). It has built-in types such as `Void`, `Integer`, `String` or `Boolean`. Additionally, it is dynamically extended by the `metamodel types` (i. e., class instances of the metamodel). These can also occur as collections, lists or sets (e. g.,`Collection[my::Type], List[my::Type], Set[my::Type]`).

The expression language knows about:

- Basic arithmetic and boolean operations (e. g.,`1+2*3, !(true && true)`)
- Operators (e. g.,`==, ! =, <, ...`)
- Strings (e. g.,"`this is a string`")
- Integers, Reals (e. g.,`10, 3.4`)
- Collection operations (e. g.,`collection.select(i | i > 3),...`)

[4]`http://www.eclipse.org/modeling/gef/`
[5]`http://www.eclipse.org/modeling/gmf/`
[6]`http://www.eclipse.org/modeling/`
[7]`http://www.openarchitectureware.org/`

Listing 3.1: Xpand example template

```
1  « IMPORT simpleecore »
2
3  « DEFINE main FOR SimplifiedEcore -»
4     « FILE "implementations.java" -»
5
6  /* some comment */
7
8  « FOREACH EClass AS class ITERATOR i -»
9     class « class.name -» {
10       //implementation of class number « i.counter1 -»
11    }
12 « ENDFOREACH-»
13
14 « ENDFILE -»
15 « ENDDEFINE »
```

Templates

Xpand templates are stored in `*.xpt` files. These are textual files that combine raw template text and commands/expressions of the template language. The commands and expressions operate on the model, based on the given metamodel. Special escape characters ("«" and "»") are used to differ between template text and commands or expressions. Commands and expressions must always be encapsulated by these characters.

Listing 3.1 presents a very simple Xpand template file. In the first line the metamodel `simpleecore.ecore` is imported using the `IMPORT()` command enclosed by the escape characters. Lines 3 and 4 create a destination (text) file `implementations.java` for every instance of `SimplifiedEcore` within a model conforming to the metamodel from line 1. Line 5 and following lines define the contents of this text file. This text file's second line will be the java comment of line 6. This is a raw template text (not encapsulated by escape characters) that is just copied into the destination file. The `FOREACH()` block of line 8 to 12 will result in class definitions for each instance of `EClass`. This will be customized using the expression of line 9, where the name of such an instance is extracted. Additionally, the comment of line 10 includes an expression with a reference to the counter (`i`) of the `FOREACH()` block.

For further information and examples of the Xpand template language and code generation, the reader should refer to the given websites (s.a.).

3.1.6 Xtend

Xtend is another language, once arising from the oAW project. It is also now well integrated into the Eclipse Modeling Project. Originally it was used to define *exten-*

sions[8] to a metamodel used in the Xpand template language.

Xtend is based on the same type system and expression language as Xpand is and as it was explained in the previous section.

In addition to extending a metamodel, Xtend can also be used to define a model-to-model (M2M) transformation [56].

Metamodel Extensions

Xtend extensions are stored in `*.ext` files. Listing 3.2 shows how simple extension functions can be defined. Both of these functions have `EClass` as its first element. This way they extend the operations on instances of class `EClass` within the metamodel `simpleecore.ecore`. These extensions (i. e., functions on `EClass` objects) can then be used within the Xtend file and within an Xpand file that imports the Xtend file.

Considering an object called `myClass` of type `EClass`, there are two possibilities to call and make use of the defined Xtend functions:

1. `myClass.upperCaseName` or
   `myClass.setUpperCaseNameAndId("new name", "new id")`
   to call them in a member-style syntax.

2. `upperCaseName(myClass)` or
   `setUpperCaseNameAndId(myClass, "new name", "new id")`
   to use them like static functions.

This Xtend file of Listing 3.2 shows some common and basic structures that can be found in most extension definitions. The function defined in line 3 has a `String` return value. The value of a function results from the evaluation of its last statement. In this case it is the statement `class.name.toUpperCase()` of line 4, where the function `toUpperCase()` is built-in for the basic `String` type. As explained above such basic types are already supported by the type system and expression language.

The second function defined in line 8 has no return value as indicated with the `Void` return type. It introduces a new *local variable* `nameCopy` that the parameter `name` is just assigned to. The static type system makes sure that the type of `nameCopy` will also be `String`. In line 11 a statement (with no return value) to set the id is executed. This statement has side-effects, namely the id assignment. Because this is the last statement of the method, the (absent) return value (of type `Void`) is inherited from it and matched to the defined one (also `Void`). In line 10 another assignment statement is executed. The sequence operator "`->`" delimits sequential, imperative statements as the one of line 10 and line 11. This way initializations, which are inherently imperative and not side-effect-free, can be realized.

[8]The name Xtend is derived from the fact that it serves as an extension to the metamodel in the context of Xpand.

Listing 3.2: Xtend example for model extensions

```
import simpleecore;

String upperCaseName(EClass class):
  class.name.toUpperCase()
;


Void setUpperCaseNameAndId(EClass class, String name, String id):
  let nameCopy = name:
  class.setName(nameCopy.toUpperCase()) ->
  class.setId(id)
;
```

Model Transformations

As indicated above, Xtend can not only be used to extend a metamodel but also to transform models. Basically one distinguishes between two types of model transformations:

Model to Model In this case there are two different metamodels. One of them serves as the source and the other as the target metamodel. The model transformation describes how instances of the target metamodel can be constructed out of instances of the source metamodel. As the implication is directed from the source metamodel instance to the target metamodel instance this is called a unidirectional[9] transformation description.

Inplace In case there is just one metamodel and only modifications on metamodel instances are defined, this is called an inplace model transformation.

Using Xtend, both transformation types are possible. Listing 3.3 shows two Xtend functions, the first method `transform()` of line 4 defines a M2M transformation, the other method `inplace()` of line 9 defines an inplace model transformation.

In the first transformation the keyword `create` instantiates a new entity of type `otherecore::MainType` defined in the target metamodel. It takes a parameter `main` that belongs to the source metamodel. In line 5 it simply sets the name of the newly created entity (`this`) to the name given by the object from the source metamodel instance. It then returns the new entity, as all `create` functions do.

In the second transformation of line 9, the source metamodel instance is modified and hence can be seen to be the target metamodel instance at the same time. The name of the entity `main` is modified to the same, but upper case letters.

[9] Apart from Xtend, there also exist bidirectional transformation descriptions that not differ between a source and a target metamodel, but these descriptions tend to be a lot more restrictive in their expressiveness.

Listing 3.3: Xtend example for model transformations

```
import simpleecore;
import otherecore;

create otherecore::MainType this transform(simpleecore::MainType main):
   this.setName(main.name)
;


Void inplace(simpleecore::MainType main):
   main.setName(main.name.toUpperCase())
;
```

Listing 3.4: Xtend example with escape to Java

```
import simpleecore;

String upperCaseName(simpleecore::EClass class):
  class.getNameInJava.toUpperCase()
;


String getNameInJava(simpleecore::EClass class) :
   JAVA myUrl.myPackage.myJavaClass.myStaticMethod(myUrl.simpleecore.EClass)
;
```

Escape to Java

Whenever for example the functional style of Xtend or the absence of global variables seems to limit the user in any way, he or she can easily escape to the Java language instead. This is done by using the `JAVA` keyword.

Listing 3.4 shows a function called `getNameInJava()` that uses this keyword to escape its computation to the static Java method `myStaticMethod()` defined in the package `myUrl.myPackage` and shown in Listing 3.5. The computation itself takes place in line 6 of this listing and is nothing extraordinary. As the Java code requires the EMF implementation code for the given metamodel `simpleecore.ecore`, all getter and setter methods for the defined classes/types are easily accessible.

For further information and examples of the Xtend extension and transformation language, the reader should refer to the given websites (s.a.).

3.2 Ptolemy

The Ptolemy II project studies heterogeneous modeling, simulation, and design of concurrent systems with a focus on systems that mix computational domains [19].

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

Listing 3.5: Java code called by Xtend code

```java
package myUrl.myPackage;
import myUrl.simpleecore.EClass;

public class myJavaClass {
   static String myStaticMethod(EClass class) {
      return class.getName();
   }
}
```

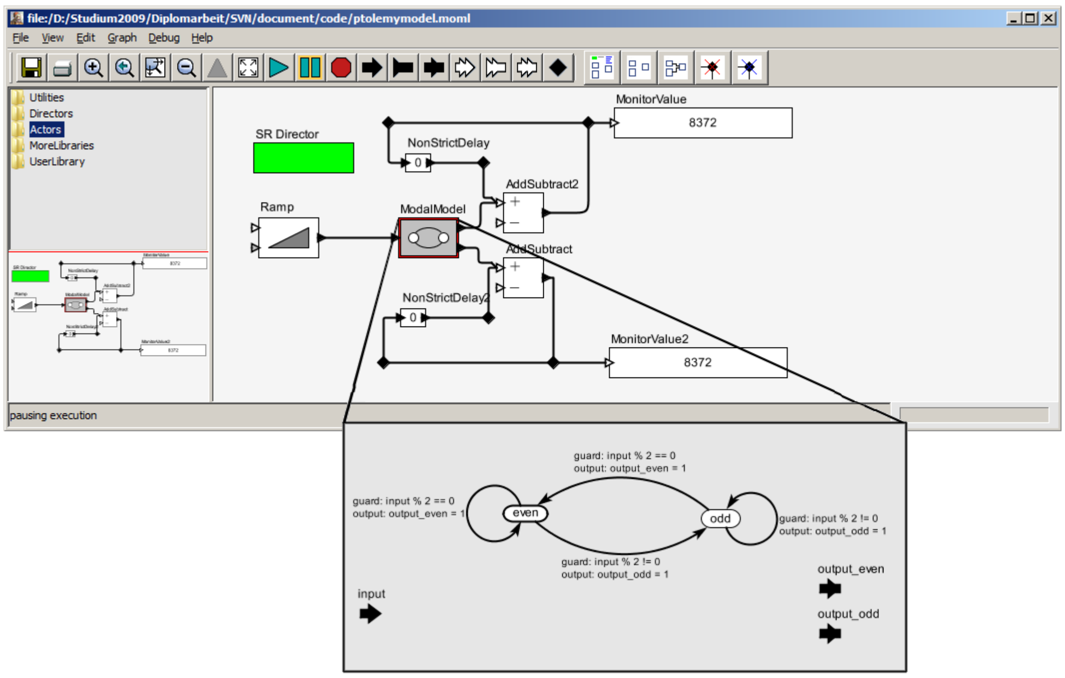


Figure 3.6: Ptolemy model using SR and ModalModel domain to sum-up even and odd values

with executable models. The latter consists of interacting components called *actors*. Hence this approach is referred to 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 are allowed to consist of one of the following:

1. Pure Java code that may produce output for some input during execution.

2. Other Ptolemy actors composed together under a separate MoC that defines the overall in- and output behavior.

3.2.1 Domains

There exist several built-in directors (i. e., concurrency and communication 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-Machine (FSM). Table 3.1 gives a short overview of these and describes their domains. Whenever this seems to limit the developer, he or she may easily adapt or define new Ptolemy II directors in Java that implement their own (e. g., more specialized) semantical rules of actor 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.

3.2.2 A Heterogeneous Model Example

An example Ptolemy II model is presented in Fig. 3.6. In this example the two domains SR and FSM (or rather the ModalModel domain as a new and enhanced version of the older FSM, see [37]) are mixed. On the top-level there is a synchronous data-flow actor-to-actor communication. At a given rate, execution steps occur where each actor is fired once (i. e., its `fire()` method is called). Inside this, the actor may react to given (or absent) input tokens with output tokens.

In this example the ramp actor produces an ascending sequence of integer numbers, starting with `0`. Each firing, one token arrives at the input of the ModalModel. This has two output ports and due to its implementation shown in the expanded view below, it alternately produces a token `1` on each output port as a result of this specific input token stream.

Looking closer into its control-flow implementation, each time the ModalModel is fired, it may change its state by taking an enabled transition. Due to the nature of these transition guards in Fig. 3.6, there always is an enabled transition. If the state is `even` and the input token (referred to as `input`) is even, the `even_output` will produce a `1` token, the other will not produce any token. The state is not changed in this case. If the state is `even` and the input token is odd, the `odd_output` will produce a `1` token, the other will not produce any in this case. The transition leads to the other state `odd` now. This means that for the next firing, the `odd` state's outgoing transitions are inspected and define a potential reaction of the actor. In this example, the behavior in state `odd` is (inverse) analogue to the one in state `even`.

On the SR data-flow layer each out coming token of the two ModalModel output ports is counted in a delayed loop where the value of the loop is monitored by a special *sink* actor. According to the ramp input and the ModalModel implementation, these values will not differ by more than one. Note that the AddSubtract actor has *multiport* input ports, meaning that all tokens arriving are summed-up and result in just one output token of this value. Also note that according to the synchronous semantics of the SR domain, the delaying of the values is mandatory, because instantaneous loops are prohibited.

Domain	Description
Continuous Time (CT)	Interaction via continuous time signals, actors compute differential relations between inputs and outputs, differential equation solvers for simulation
Process Networks (PN)	Interaction via asynchronous message passing, actors react to input messages and compute new output messages
Synchronous Dataflow (SDF)	Special PN domain with fixed number of messages each firing
Synchronous Reactive (SR)	Interaction via synchronous message passing at a fixed rate, messages can be absent
Finite-State-Machine (FSM)	Entities are not conventional actors but states here, execution is strictly ordered sequence of state transitions, built-in expression language to evaluate transitions guards

Table 3.1: Overview of some of Ptolemy's built-in domains

3.2.3 Simulations

Fig. 3.6 shows a Ptolemy model in its graphical representation, opened in the *Vergil*[10] editor that comes bundled with the Ptolemy framework. In addition to create and modify Ptolemy models, this editor also allows to simulate them. According to the used actors and domains, simulation results and even debugging results may be visible right away in the editor. For example, in Fig. 3.6 the monitored values are displayed.

But the simulation of Ptolemy models is not restricted to Vergil only. Ptolemy is built on the Java language and all components are implemented in Java. In the memory, a Ptolemy model is nothing more than (cascaded) instances of various Ptolemy classes (mainly actors with parameters). The framework additionally offers possibilities to execute Ptolemy models programmatically and stepwise from and within Java programs. But stepwise executions are not generally possible using Vergil.

Compared to for example SCADE, Ptolemy models cannot have input or output ports on their outermost hierarchy level that could serve to interact with the environment. Thus, Ptolemy models can be considered *closed models* where all interaction is limited to the participating actors. In order to communicate with an environment, special actors need to provide this functionality.

[10]Note that Fig 3.6 shows two hierarchy layers while Vergil can only display one layer per window.

3.2.4 Technical Details

For the sake of brevity no advanced technical details can be given here and the reader may refer to the technical Ptolemy documentation, in particular about the *charts (pronounced starcharts) principle [24]. It illustrates how hierarchical Finite-State-Machines can be composed using various concurrency models leading to arbitrarily nested and heterogeneous model semantics.

In addition to this also the Ptolemy documentation[11] or other literature concerning the Ptolemy project [19, 13, 38, 35, 36] might give some useful additional and more detailed technical information.

3.2.5 Ptolemy EMF Model

The graphical representation of Ptolemy models shown in Fig. 3.6 is just one, used by the bundled Vergil editor. If you save a Ptolemy model into a file, an XML based format is used. This is called the modeling markup language (MOML). The MOML XML file content is used just as another representation of one and the same Ptolemy model. Parsers and serializers exist to transform a Ptolemy model in Java to its MOML equivalent and vice versa. Hence a Ptolemy model can be specified in both ways. The advantage of specifying it using MOML is that the compiled Ptolemy classes are not needed for constructing the model.

Because MOML is a *language* describing (syntactically valid) Ptolemy models, it can be considered an abstract syntax. The latter is defined by a Document Type Definition (DTD). This can also be represented in the XML schema[12] definition language.

As seen in Sec. 3.1 a XML schema definition file can be imported by the EMF framework in order to construct an EMF based metamodel. Such a metamodel type is useful in the context of Eclipse as discussed in this section.

Fig. 3.7 shows a simplified version of the Ptolemy EMF based MOML metamodel. Note that most Ptolemy components (e. g., actors) are represented as `EntityType` instances that specify the implementing Java class and a name. These may have named ports represented as `PortType` instances. All links (hierarchically) inside a component are contained by it and consist of two `LinkType` and one `Relation-Type` instance. A relation may optionally be a vertex (shown by Vergil as a diamond). Components usually have a number of named properties of type `Property-Type`. The latter sometimes contain values but always are typed by their class attribute. Additionally, a director of type `DirectorType` is a special component to express the MoC. Because this is a rather simplified version of the MOML metamodel, only basic components are visible. The reader should refer to the Ptolemy documentation for a complete metamodel.

[11]http://ptolemy.eecs.berkeley.edu/ptolemyII/designdoc.htm
[12]http://www.w3.org/XML/Schema

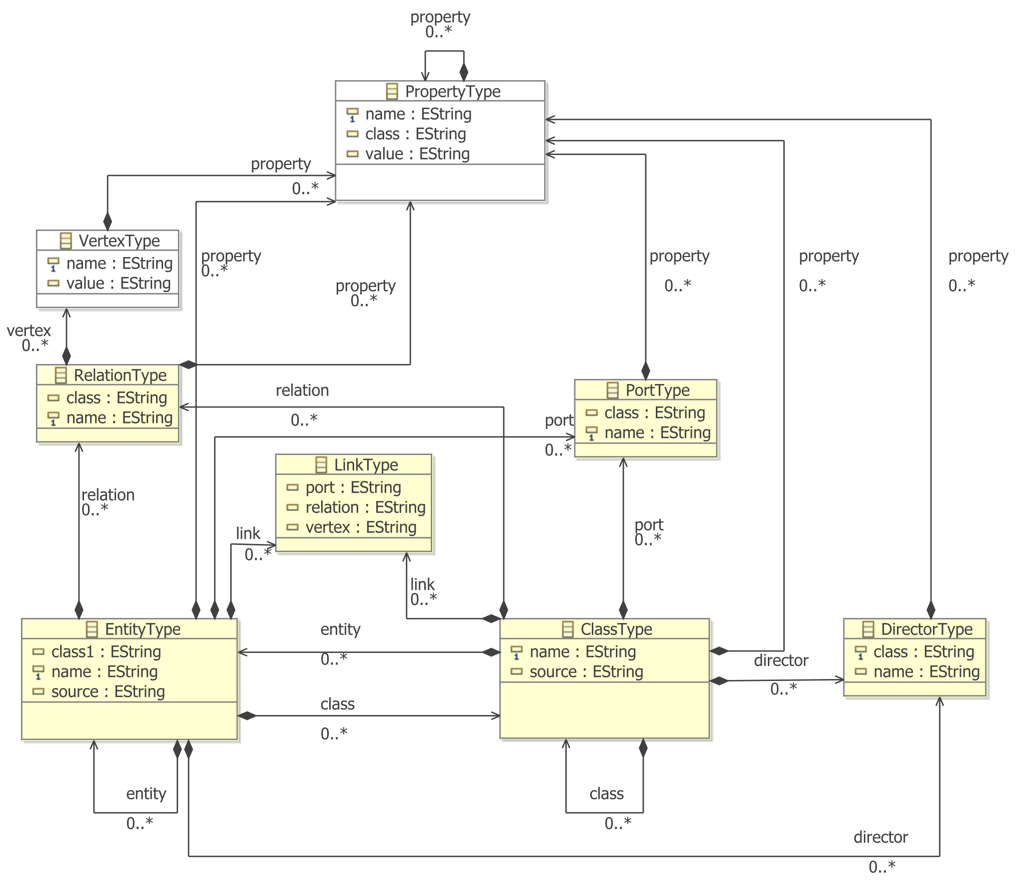


Figure 3.7: Simplified metamodel of MOML file structure

3.3 The Java Script Object Notation (JSON)

The Java Script Object Notation (JSON) is a lightweight format for saving and interchanging data. Although it is based on the JavaScript programming language, JSON itself is completely language independent. Besides its efficient representation (compared for example to XML), JSON is a lot more human readable. It is based on two building blocks:

1. Key/value pairs
2. Ordered lists

The first one is often implemented as objects, structs, or hash tables, the second often as arrays or lists.

3.3.1 Implementations

Because these basic constructs can be represented in most modern higher programming languages, there exist various JSON implementations. Just to numerate some:

- C, C++, C#
- Delphi
- Eiffel, Erlang, Haskell, Lisp
- Java, JavaScript
- Perl, PHP, Phython, OpenLaszlo
- Ruby, Visual Basic

A complete list of implementations can be found on the official JSON website[13].

3.3.2 Structure

The Java implementation[14] used in the context of this work basically consists of the `JSONObject` class. An instance of this class is an unordered collection of key/value pairs. Its serialized form is wrapped into curly braces having colons separating keys from their values and commas separating several of such key/value pairs.

For example such an object with two keys *key1* and *key2* and with two respective values *value1* and *value2* would have the following string representation:

```
{ key1:value1 , key2:value2 }
```

While the key is of type `String`, a value can have various types. For example:

- `boolean`
- `int`, `double`
- `String`
- `Object`

With the last generic `Object` type, it is also possible to cascade `JSONObjects` into another. Consider the example from above, where the second value is another `JSONObject` instance consisting of the two key/value pairs *key2a*, *value2a* and *key2b*, *value2b*:

```
{ key1:value1 , key2:{ key2a:value2a, key2b:value2b } }
```

[13]http://www.json.org
[14]http://www.json.org/java

The usage of the JSON implementation for Java is well documented on the Website. Basically there exist the following priorly used methods in order to:

- Accumulate values under a key: `accumulate()`
- Get the value of a key: `get()`
- Determine the `JSONObject` contains a specific key: `has()`
- List all contained keys: `keys()`
- Remove key/value paires: `remove()`
- Serialize the `JSONObject`: `toString()`
- Parse a serialized representation: `JSONObject()`

4 Semantics

As introduced in Chap. 2, there are two possible ways to specify semantics of a DSL:

1. Model transformation into a semantic domain

2. Provision of a new action language

Both approaches have their pros and cons as already discussed. In this work the first one is used in leveraging Ptolemy II as a flexible and extensible simulation back-end. For this, a model transformation is utilized to automatically generate valid Ptolemy models (i. e., meta model instances are transformed into MOML files). The meta model describes the specific DSL for which two different case studies are presented:

- First, a language describing simple controllers for a model railway system is examined in Sec. 4.1.

- Second, a transformation following the semantics of SyncCharts, a special synchronous Statechart dialect, is given in Sec. 4.2.

For each case study a short introduction into the domain is given, followed by a description of the metamodel and an outline of the transformation.

Common to both DSLs is the idea of the simulation and visualization component that is interfaced with the Execution Manager presented in Chap. 5. The simulator engine is described in Sec. 4.3. All of this is covered in the KIELER leveraging Ptolemy (KlePto) subproject that also comes up with the implementation counterpart for the presented transformation and simulation component ideas.

4.1 Model Railway Controller Language

In a first case study a controller language for a model railway system is presented. This language is similar to a more or less simple state machine dialect but with the additional concept of concurrency built-in. For this study a model transformation into the Ptolemy model language is given in addition to a code generation approach for C that is I/O equivalent.

The train service on a railway system is a paradigm of a system with much dynamic behavior. Often, only a limited number of tracks is available, passing of trains is only possible in certain constellations, and collisions must be avoided as well as blocking. Possibly there are additional constraints such as time tables to meet or special directions and speeds to maintain. The task of a controller is to operate such a railway system while meeting a collection of such requirements.

4.1.1 Model Railway Installation

The model railway installation (see. Fig 4.1) of the department of computer science at the Christian-Albrechts-Universität zu Kiel is a concrete and real live example of such a dynamic system. In this rather complex laboratory there are more than 200 sensors and actuators installed that can be operated over several distinct bus systems, namely Controller Area Network (CAN), Time Triggered Protocol (TTP), and Ethernet., as depicted in Fig. 4.2. The (simplified) track scheme of the installation is given by Fig. 4.3.

Every track segment that is visually separated from the others can be controlled separately (i. e., the speed of a locomotive is influenced by controlling the specific voltage power of such a segment). The latter is addressed by its name (e.g, `KH_ST_5`). Nearly all segments are equipped with two *reed* contacts (mechanical switches). There is one at each end of a track segment, denoted as a small blue bar in Fig. 4.3. These contacts are triggered when passed by a train and hence they serve as sensors to derive train positions. Additionally, power electronics installed under the baseplate is able to detect trains occupying a track by measuring potential differences. All 29 switch points and every signal light can also be addressed and set by peripheral devices.

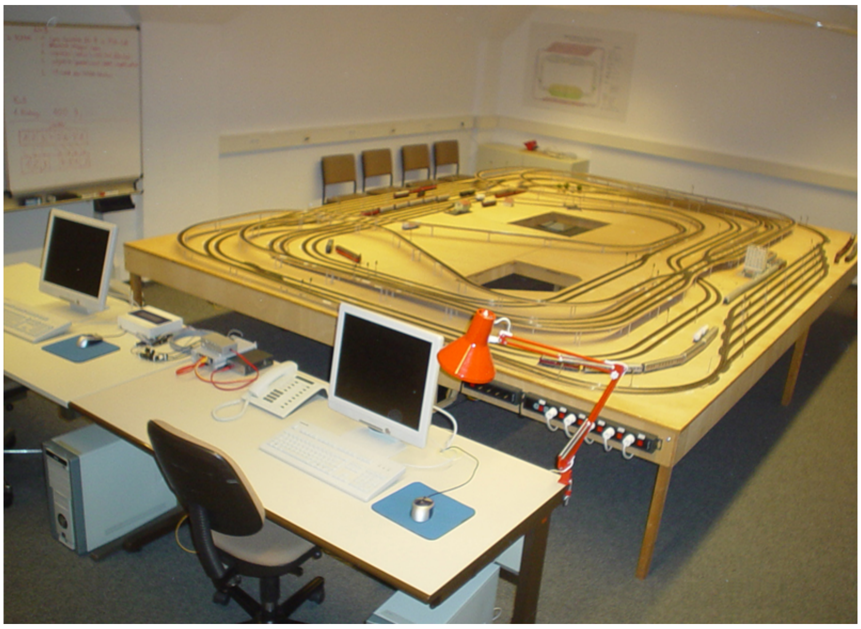


Figure 4.1: Model railway installation

The tracks can be divided into three circles:

1. Kicking Horse Pass (red)
2. Outer Circle (green)

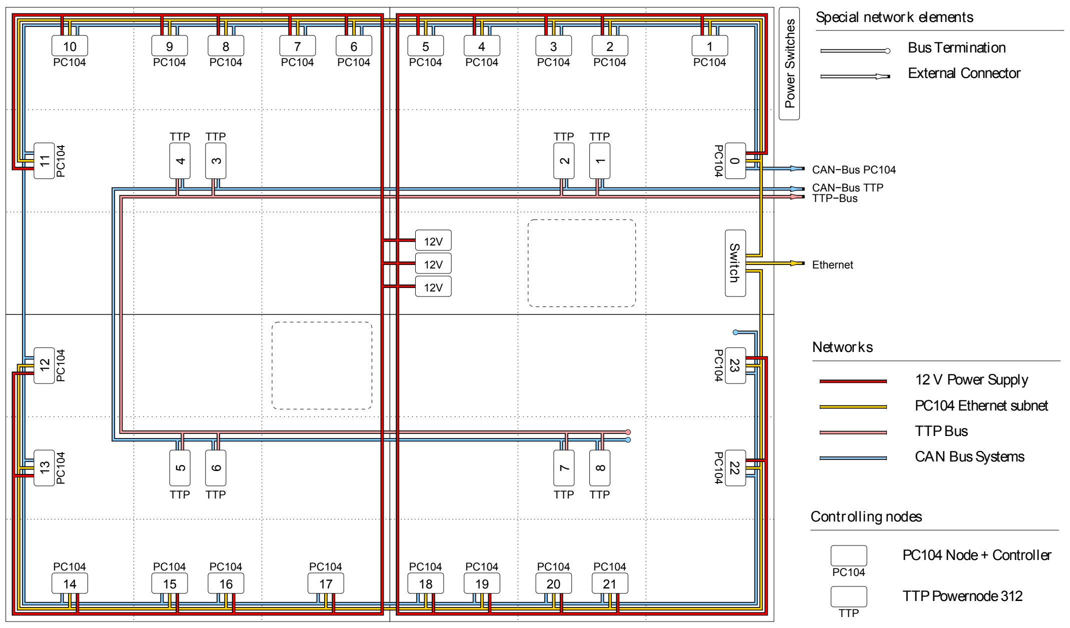


Figure 4.2: Model Railway Bus Diagram (from [30])

3. Inner Circle (orange)

Trains may transit from one circle into another using the blue turnaround loops or the approach tracks of the Kicking Horse Pass on the left and right side.

A more detailed and more technical description of the model railway is given on the website[1] dedicated to it or in the diploma thesis of Stephan Hörmann [30]. There also exists more information concerning the C interface that is used in Sec. 4.1.6. Details about the Java interface and the simulation used in Sec. 4.1.5 can be found elsewhere [44].

4.1.2 Domain Specific Language

Considering the above railway system as the domain, a language was defined to model simple controllers that are able to fulfill various tasks. To summarize, such a controller should be able react to three possible kinds of input events:

- Reed contact was triggered
- Timeout event occurred
- Track is detected to be occupied

[1]http://informatik.uni-kiel.de/~railway/

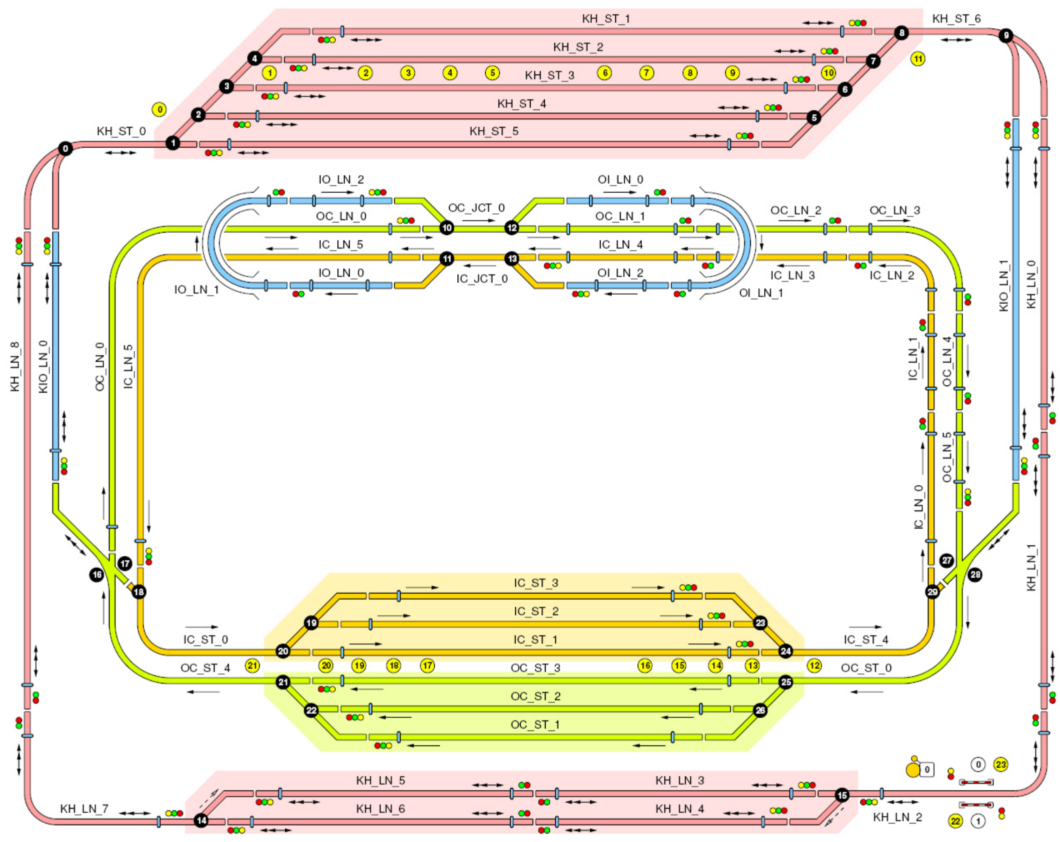


Figure 4.3: Schematic track scheme of the model railway (from [30])

Additionally, this controller should be able to generate the following kinds of actions:

- Set the speed of a track segment
- Set a switch point to straight or branch position
- Set a signal light to a specific color

A controller can be seen as a reactive system, which responds to input events with output actions where the pace is limited by the environment (i. e., the physics of the model railway system). As mentioned earlier, concurrent reactive tasks should be possible.

A state-based language is sufficient to describe such controllers. It treats the input events as triggers of transitions and covers the different output actions as three types of nodes representing states. The specific action of a node type will be taken as soon as it is entered. Transitions (i. e., directed edges) lead from one node to another.

To cope with the concurrency aspects, several nodes may be marked with an initial tag. The computation of all strongly connected components can then take place concurrently.

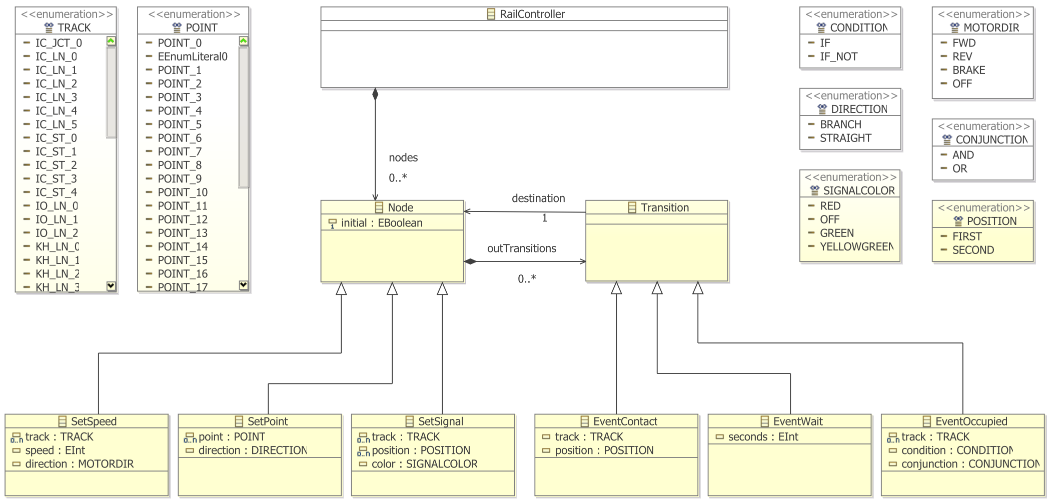


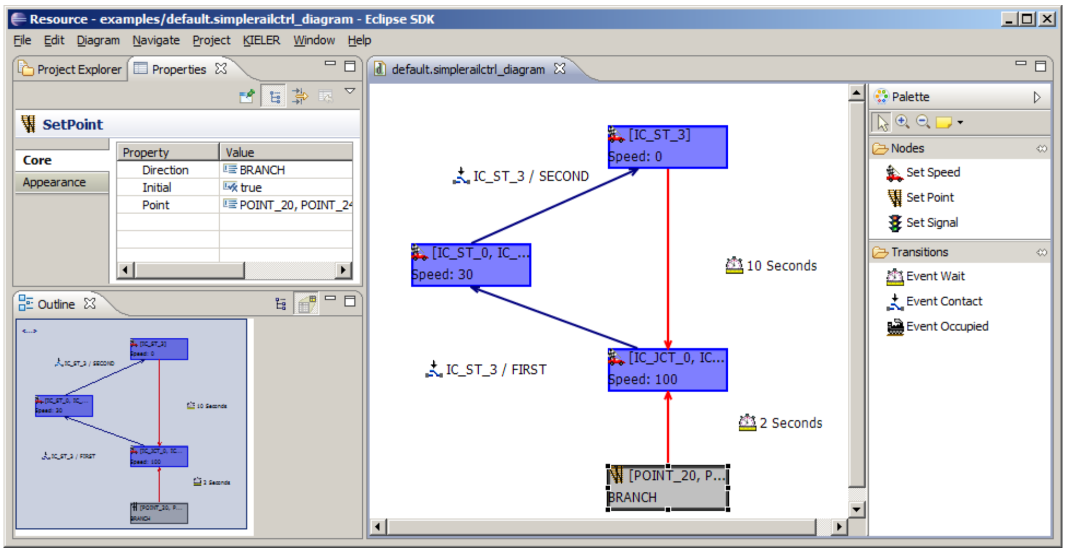
Figure 4.4: Metamodel of the railway controller language

Fig. 4.4 shows the metamodel representing such a model railway controller language. The `RailController` class owns an arbitrary number of `Node`s. These can have an initial tag as explained above. A `Node` is an abstract class that is derived by the `SetSpeed`, `SetPoint`, and `SetSignal` class, each having specific properties required by their distinct actions. `Node`s also contain their outgoing `Transition`s. The latter link them to exactly one possibly other destination `Node`. The `Transition` class is abstract as well and substantiated by the classes `EventContact`, `EventWait`, and `EventOccupied`. These types represent the different triggers that may enable each `Transition`, configured by the properties of each child class.

4.1.3 Domain Specific Editor

The metamodel described in Sec. 4.1.2 represents the abstract syntax of railway controller models. Using the Eclipse EMF/GMF toolchain introduced in Sec. 3, it is relatively straightforward to produce a graphical Eclipse based editor. Using such, one is able to create and modify graphical representations of the controller models where, e. g., `Node`s are represented as boxes (each type having a different color) and `Transition`s are drawn as connecting edges between those boxes (again, each type having its own color). Such an out of the box editor can be seen in Fig. 4.5.

In the end the additional graphical information (i. e., the layout) is stored apart from the domain model information.



Property	Value
Direction	BRANCH
Initial	true
Point	POINT_20, POINT_2

Figure 4.5: Railway controller example model in Eclipse GMF editor

4.1.4 Transformation

The domain model information contains only enough information to specify the unambiguous semantics of a model railway controller, having the implicit knowledge of the defined domain language in mind. To make this knowledge about the domain language explicit and to give the controller models an execution semantics, a model transformation into a semantic domain was chosen, for the already discussed reasons. If the semantics of the target domain is clearly defined and formal proven[2], an unambiguous transformation is sufficient. Further formal proofs just need to concern the transformation. Hence using Xtend as a M2M transformation language from the railway controller domain into a Ptolemy semantic domain is adequate.

In addition to this, a M2T transformation is given using the Xpand template language to generate runnable C code. This code is input/output (I/O) equivalent, where not a complete reasoning about this is provided, but evidence for some implementation details that support this thesis. This also applies to the M2M transformation description. In the next two sections both approaches are discussed in more detail.

4.1.5 Xtend M2M Transformation

Within the Xtend transformation description all parts of the domain language must to be considered, leading to the following specification requirements:

- Behavior must be deterministic,

[2]This is the case for the Ptolemy framework, as outlined in Sec. 3.2.

- `Nodes` must be represented as states (where control holds),
- `Transitions` need to connect states,
- The type of a `Node` and the type of a `Transition` must be represented accordingly, evaluating the events and taking actions respectively, and
- Concurrent strongly connected `Nodes` should be able to react concurrently:
  - All concurrent nodes must have the same input,
  - All concurrent actions must be considered (in any order).

In order to partially reason about the correctness of the transformation, first a simple example is surveyed and later the collected requirements and their concrete implementations within the transformation are discussed.

Example

Addressing the example railway controller model that can be seen in Fig. 4.5, a transformation that fulfills these requirements leads to a Ptolemy model as depicted in Fig 4.6. This controller has the following behavior:

1. Start with a (gray) initial node to set some points of the (orange) Inner Circle (see. Fig. 4.3) in order to route a train positioned on track `IC_ST_3` around.
2. Wait for 2 seconds.
3. Set the speed of all tracks of the Inner Circle to `100`.
4. Wait until the first reed contact of track `IC_ST_3` is triggered.
5. Set the speed of `IC_ST_0` and `IC_ST_3` down to `30` in order to slow down the train.
6. Wait until the second reed contact of track `IC_ST_3` is triggered.
7. Stop the train by setting the speed of `IC_ST_3` to `0`.
8. Wait for 10 seconds before starting in 3. again.

Note that in this simple example there is no concurrency present because there exists just one strongly connected component. In a concurrent setting with several such components, one would have to create a ModalModel actor for each strongly connected component. Its outputs must be connected to the merging actors and the inputs to the delay split relations.

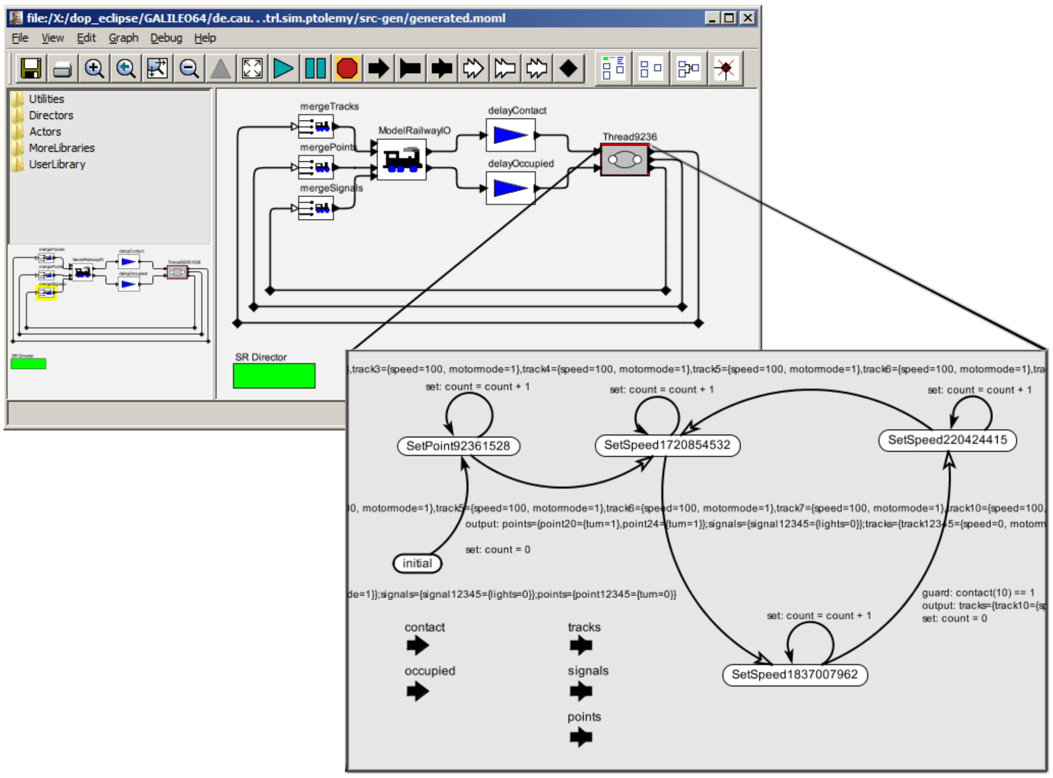

file:/X:/dop_eclipse/GALILEO64/de.cau. . .trl.sim.ptolemy/src-gen/generated.moml
File View Edit Graph Debug Help
Utilities
Directors
Actors
MoreLibraries
UserLibrary
mergeTracks
mergePoints
mergeSignals
ModelRailwayIO
delayContact
delayOccupied
Thread9236
SR Director
},track3={speed=100, motormode=1},track4={speed=100, motormode=1},track5={speed=100, motormode=1},track6={speed=100, motormode=1},tra
set: count = count + 1
set: count = count + 1
set: count = count + 1
SetPoint92361528
SetSpeed1720854532
SetSpeed220424415
0, motormode=1},track5={speed=100, motormode=1},track6={speed=100, motormode=1},track7={speed=100, motormode=1},track10={speed=100,
output: points={point20={turn=1},point24={turn=1}};signals={signal12345={lights=0}};tracks={track12345={speed=0, motorm
set: count = 0
initial
de=1}};signals={signal12345={lights=0}};points={point12345={turn=0}}
guard: contact(10) == 1
output: tracks={track10={s
set: count = 0
set: count = count + 1
contact
occupied
tracks
signals
points
SetSpeed1837007962

Figure 4.6: Railway controller embedded in a Ptolemy model

Deterministic behavior

To achieve deterministic behavior the SR domain of Ptolemy is chosen (see Sec. 3.2 or [57]). With this domain, all events and actions are represented as tokens, which have a clear and distinct value at each Tick. This is because of the fixed point iteration that takes place in the background.

Representing `Nodes` and `Transitions`

In order to get a natural representation of `Nodes` and `Transitions` in Ptolemy, the ModalModel domain is chosen as it already supports a simple kind of state machine.

Representing events and actions

A ModalModel needs input- and output-ports according to the events to react to and actions to take. Actions are intended to take place when a node is entered and ModalModel states do not support this kind of *entry actions* natively. But this can be modeled as output actions of ModalModel transitions that lead to the specific typed `Node`. Hence an extra initial ModalModel node is needed to take care of

Listing 4.1: Main tasks of Ptolemy transformation

```
create EntityType this createBaseEntity(simplerailctrl::RailController rc):
   //set static name, create SR director and RailwayInterfaceActor
   this.setName("RailController") ->
   this.setClass1("ptolemy.actor.TypedCompositeActor") ->
   this.property.add(createMainSRDirector()) ->
   this.entity.add(createModelRailwayIOActor()) ->
   //create concurrent threads (and connect them)
   createThreads(rc.nodes, this) ->
   //delay operators to avoid instantaneous loop
   this.entity.add(createDelay("delayContact")) ->
   this.entity.add(createDelay("delayOccupied")) ->
   //merge concurrent output actions togehter
   this.entity.add(createModelRailwayMerge("mergeTracks")) ->
   this.entity.add(createModelRailwayMerge("mergePoints")) ->
   this.entity.add(createModelRailwayMerge("mergeSignals")) ->
   //split input events for all concurrent threads
   this.relation.add(createVertexRelation("contactsGLOBAL")) ->
   this.relation.add(createVertexRelation("occupiedGLOBAL")) ->
   //connect (static) actors
   connectActors(this, "ModelRailwayIO.contact",
                   "delayContact.input", "delayContactRelation") ->
   connectActors(this, "ModelRailwayIO.occupied",
                   "delayOccupied.input", "delayOccupiedRelation") ->
   connectActors(this, "ModelRailwayIO.tracks",
                   "mergeTracks.output", "mergeTracksRelation") ->
   connectActors(this, "ModelRailwayIO.points",
                   "mergePoints.output", "mergePointsRelation") ->
   connectActors(this, "ModelRailwayIO.signals",
                   "mergeSignals.output", "mergeSignalsRelation") ->
   addLink(this, "delayContact.output", "contactsGLOBAL") ->
   addLink(this, "delayOccupied.output", "occupiedGLOBAL")
;
```

the (unconditioned) action associated with the initial `Node` of the railway controller domain language. Events of this language are handled by guard expressions of the ModalModel transitions that lead from one node to another. Special attention has to be payed to the case in which two of such expressions (belonging to different transitions) evaluate to `true` at the same tick instant. This case is handled with a deterministic choice where all events are processed in the order they were created, giving precedence to transitions created earlier. In the transformation this is handled by adding the negated expression of a transition in a conjunction to all transitions with lower priority according to their time of creation.

Concurrency aspects

As not only one initial `Node` is allowed, all strongly connected `Nodes` starting at an initial node, are transformed into separate distinct ModalModels. Note that models with other than exactly one initial node per strongly connected component

Listing 4.2: Example transformation code fragment: Traversing inner states

```
Void createInnerStates(EntityType modalController,
                       simplerailctrl::Node sourceState,
                       List[simplerailctrl::Transition] transitionList) :
   let currentTransition = transitionList.last():
   let doneDestinationState = isMarked(currentTransition.destination):
   createSimpleStateEntity(modalController, currentTransition.destination) ->
   //first look up other target states (of this source state)
   if (transitionList.size > 1) then
      createInnerStates(modalController,
              sourceState,
              transitionList.withoutLast()) ->
   //then create transitions to these
   addTransition(modalController, currentTransition) ->
   //then go on with the target state (as new source state)
   if ((!doneDestinationState)
    && (currentTransition.destination.outTransitions.size > 0)) then
      createInnerStates(modalController,
              currentTransition.destination,
              currentTransition.destination.outTransitions)
;
```

are considered invalid. This could be enforced by using a syntax validation language[3].

Strongly connected components, each represented as a ModalModel, are run concurrently by Ptolemy.

To fulfill the requirement that all concurrent ModalModels need to see the same consistent events, the input tokens are split up on the SR layer. This is also where the merging of output actions takes place. For each type of action there is a merging actor that gives precedence to later connected ModalModels on its multiport. Not conflicting actions (on different tracks, points, or signals) can be merged by taking the union of all.

Further implementation details

Listing 4.1 shows the main Xtend function used for the transformation. In the first line the `RailController` class instance of the source controller model is passed as a parameter. This function creates a Ptolemy entity (see Fig. 3.7), i. e., a `TypedCompositeActor`, and sets its name and class type in the second and third line.

To chose Ptolemy's SR domain, the specific director is added in line 4 as a property of the top-level entity. The communication with the environment (i. e., the real model railway or its simulation) is handled by the special actor `ModelRailwayIOActor`. It takes actions as input tokens and produces events (as they occur on the real model railway or its simulation) as output tokens. In line 7 all threads (i. e., strongly connected components) are created as described. The following delay operators are

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

mandatory to breakup the instantaneous loop. For each kind of output signal a merge operator is needed in order to handle multiple threads (ModalModels). The relations created in lines 16 and 17 split the input events of the railway system for all concurrent ModalModels as explained above. Eventually all actors on the top-level need to be connected accordingly, which is done in lines 19 to 31 and 8.

As another example fragment of this transformation, Listing 4.2 shows the traversal through all inner states of a strongly connected component that starts at its initial node. This is a recursive function following all outgoing transitions to their target states if those have not already been marked as visited. The recursion ends iff there are no unvisited states left that can be reached from the initial node. Other nodes are not of interest for this ModalModel, either because they may belong to another thread, or because they are not reachable by any initial node and hence they would never be processed at all and can safely be omitted in the transformation. This recursion is started for all nodes marked as initial.

4.1.6 Xpand M2T Transformation

As introduced above, the Xpand transformation generates C code. Xpand is a template based approach (see Sec. 3.1.5) to transform metamodel instances into arbitrary text or code. Because the generated C code should behave exactly like the Ptolemy model, in the following, the implementation concerning the same specifications stated in Sec. 4.1.5 is outlined.

Deterministic behavior and concurrency aspects

Concurrency is naturally implemented by using Pthreads. To maintain a deterministic behavior, a notion of a *tick* is introduced in the C code. This leads to a barrier synchronization scheme as described by Andrews [7, p. 115], where concurrent state machines are implemented as worker threads and a master is the coordinator thread. Example code for the latter can be seen in Listing 4.3. All worker threads block on private semaphores (using the flag synchronization principle [7, p. 118]). The coordinator increases the global tick when all threads have done so. It also clears the semaphores, which gives the concurrent threads the opportunity to proceed to the next tick. In order to avoid a deadlock situation, it has to be ensured that all threads make progress, i. e., eventually flag that they have proceeded.

All threads need to grab a global lock for communicating with the railway interface, because concurrency needs to be avoided for the interface access. The notion of ticks ensures fairness: Eventually all threads get the chance to communicate within one tick at the latest if all others flagged that they are done for this tick. Because at least one node is able to make progress and will eventually flag, this guarantee can be proven by a simple induction reasoning.

As a simplification, terminated threads can flag their termination separately and the master will not wait for them again. This assumes that once a thread has terminated, i. e., has reached a state with no outgoing transitions, it will stall forever.

Listing 4.3: Xpand template code fragment of coordinator thread

```
void* ThreadFunctionMaster(void* port)
{
  int track;
  while (!MasterShutdown) {
         //do not wait for terminated threads
« FOREACH nodes AS n ITERATOR i -»
   « IF n.initial -»
            if (localTERMINATED«i.counter1 -») {
            localLOCK«i.counter1-» = 1;
         }
   « ENDIF -»
« ENDFOREACH -»

      //wait for all localTicks to be increased
      if (« FOREACH nodes AS n ITERATOR i -»
                        « IF n.initial -»(localLOCK« i.counter1 -» == 1)
                        &&« ENDIF -»« ENDFOREACH -»(1)) {
         //fill ContactMem and OccupiedMem array
         for (track = 0; track < 48; track++) {
           ContactMem[track][0] = getcontact(railway,track,FIRST,1);
           ContactMem[track][1] = getcontact(railway,track,SECOND,1);
           OccupiedMem[track] = trackused(railway,track);
         }

         //increase a tick counter (useless)
         TICK++;
         //if (DEBUGCONTROLLER) printf("TICK %d\n",TICK);
« FOREACH nodes AS n ITERATOR i -»
   « IF n.initial -»
         localLOCK«i.counter1-» = 0;
   « ENDIF -»
« ENDFOREACH -»
      }//end if
      usleep(1000); // sleep for 1ms
  }
  return (void *)1;
}
```

Representing `Nodes` and `Transitions`

Using a so-called *state machine programming pattern* [1], strongly connected component nodes (each such component in its own thread) are represented by a loop construct with an additional switch-case statement incorporating the different states in its case arms. The initial node is the start value of a state variable that changes when transitions are taken. Hence transitions are just the change of this variable, conditioned by its trigger.

Representing events and actions

Reed contact and occupied events are represented by the global read only variable fields `ContactMem` and `OccupiedMem`. The wait event is handled individually in each thread. To ensure that all threads see the same input events, after all threads have proceeded and before the resetting of their private semaphores takes place, the coordinator reads in all contact and occupied events from the interface. Multiple or concurrent reads of the interface would either result in inconsistent views on the input events, or in access errors as the interface itself is not thread safe.

The merging of output actions in this implementation is sequentialized, due to the use of global locks, in an arbitrary access order. Hence in the non-conflicting case this results in the same output as the Ptolemy model would produce. In the conflicting case, the Ptolemy model output is one special case of an execution trace output of this implementation, where the order is given by the thread creation id.

4.2 SyncCharts

The Statecharts formalism, proposed by of David Harel in 1987 [27], extends Mealy machines with hierarchy, parallelism, signal broadcast, and compound events and is a well known approach for modeling control-intensive tasks. SyncCharts [2], the natural adoption of Statecharts to the synchronous world, were introduced almost ten years later.

A M2M transformation that synthesizes Ptolemy models from Eclipse EMF models (representing SyncCharts) is presented as a second case study. In the following, first semantical aspects of SyncCharts are studied, followed by a short overview of the SyncChart editor and a discussion of the transformation. The transformation already covers a major subset of the SyncCharts language. A sketch of implementing additional parts will be given later in Sec. 6.2.

4.2.1 Domain Specific Language

A SyncChart can also be seen as the graphical notation for the synchronous Esterel language [11]. SyncCharts and Esterel are used to describe reactive systems and so are SyncCharts. Following the *synchrony hypothesis* [49] the reaction is computed tick wise and the computation itself does not take any time. This means that the generated output computed to an input in one tick instance is visible in this same instance. Because the output might have an impact on the computation itself, the computation proceeds until the system's variables reach a defined value.

In the following only a subset of SyncCharts is presented that can be handled by the transformation. A metamodel for this subset can be seen in Fig 4.7. André [5, 4, 6] presents and analyzes additional features and semantical aspects of SyncCharts.

Signals

`Signals` are a very important concept of SyncCharts. Signals can be either *present* or *absent* during an instant but not both at the same time. The presence status of input signals is defined by the environment. The presence status of local and output signals is computed by the simulation depending on the current state and the signal status of other input and local signals (but not on output signals). `Signals` are

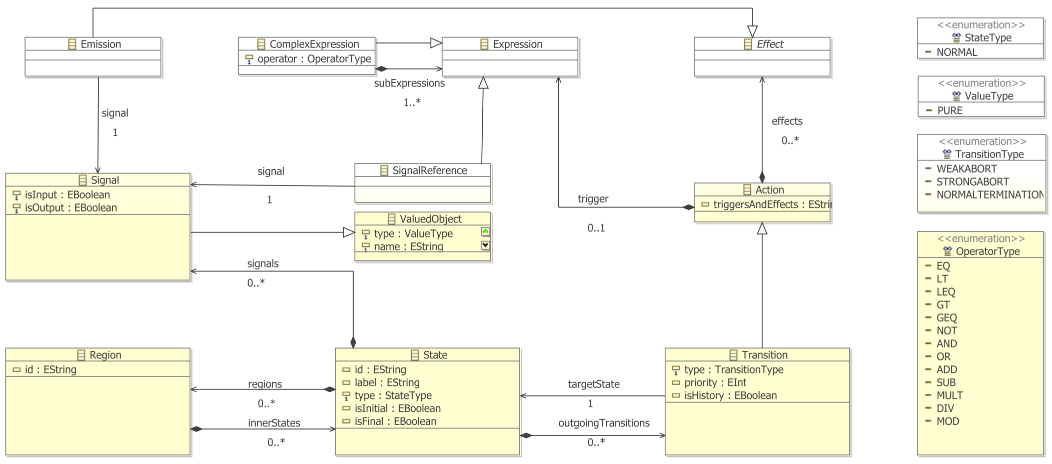


Figure 4.7: Simplified metamodel of SyncChart language

defined to be input signals and/or output signals, or alternatively local signals. They are defined within a state. As signals are `ValuedObjects`, they additionally have a name and a type. SyncCharts provide two types of signals. *Pure signals*, which just have a presence status, and also *valued signals*, which in addition to the presence status carry a value. At this point, the transformation presented here supports only pure signals, hence the type must always be `PURE`. Using signals, a broadcast communication between concurrent states can be implemented. The presence status of signals must always be clear and unambiguously defined within a tick. Otherwise the SyncChart is invalid (i. e., not constructive, see [9]) and must be rejected.

Expressions

`Expressions` are boolean terms that can be either of simple or of complex type. In the first case they only consist of a `SignalReference` evaluated to the presence-value of the specific `Signal`. In the second, they carry an additional operator that indicates how to link sub-expressions within this *ComplexExpression* together. This way any boolean combination of `SignalReferences` can easily be represented.

States

`States` are one of the building blocks of SyncCharts. They have an id and a label that do not need to be equivalent but must comply to some formatting rules. The type of a `State` is always `NORMAL` because other types, such as conditional or textual [2], are not supported by the transformation.

Hierarchy is supported by SyncCharts meaning that `States` may contain children `States`. These must be grouped in one or more `Regions`. A State that contains other regions/states is also called *macrostate*. It is called *simple-state* if does not contain anything. Additionally, a state can be tagged to be initial or to be final. When control enters its parent, immediately the initial state is entered. When a final state is entered, either the control stalls or a so-called *normal termination* defined for its parent is taken (s.b.).

Transitions

Transitions lead from one state to another and are contained by the source state. Inter-level transitions are not permitted by SyncCharts. Transitions can take one of the following types:

- `WEAKABORT`
- `STRONGABORT`
- `NORMALTERMINATION`

The `STRONGABORT` means *preemption* of a macrostate if it is taken, i. e., no signals can be emitted within the macrostate in this instance. In this property it differs from the `WEAKABORT` that will allow such signal emissions in the same instance it is taken. The `NORMALTERMINATION` is *syntactical sugar* because it can be represented using a `WEAKABORT` (see [5]). It is triggered from inside a macrostate when all parallel regions have reached a final state.

Transitions have labels, which are represented by the `Action` class and optionally contain an `Expression` and/or an `Effect`. The latter is a sequence of pure signal emissions. The `Expression` represents the *trigger* and hence infers whether the transition is ready to be taken (i. e., is *enabled*) or not.

Several (outgoing) transitions are allowed for a state. A transition can be assigned a priority, i. e., a positive integer value where `0` means the transition has the highest precedence. Additionally, a transition can be marked to be linked to a *history connector* by setting the boolean `isHistory` flag.

Entering a macrostate over a history connector means that possibly not the initial states within are enabled. Instead, other states may be active that were last active, when this macrostate has been left before.

Regions

`Regions` introduce a notion of parallelism. They are contained in one (parent) macrostate. Regions must contain at least one state but can contain several ones. Within `Regions` there must be exactly one initial state. `Signals` can be used to implement *broadcast* communication between parallel regions, i. e., a transition in one region may emit a `Signal` in one tick instance that enables a *Transition* in a parallel `Region`, where this *Signal* is a part of the trigger in the same tick instance.

4.2.2 Domain Specific Editor

The KIELER SyncCharts editor (see Fig. 4.8) was created by Schmeling [52] using the Eclipse EMF/GMF toolchain. Additional modifications to the standard GMF editor were made. The structure of the complete EMF model, i. e., the SyncCharts metamodel, where the metamodel of Fig. 4.7 is a direct sub-metamodel, can also be found in Schmeling's study thesis [52].

Because the transformation currently does not support the complete metamodel, some restrictions apply that will be discussed in Sec. 4.3. Nevertheless, this editor is capable of creating models that conform to the structure as it was introduced in the section above.

Fig. 4.8 shows a running simulation. Active states are colored in red (see Sec. 4.4). Details of the simulation engine are discussed in Sec. 4.3.

`Regions` are surrounded or separated by a dotted line. States are represented as boxes carrying an optional textual label. Macrostates have an additional child area, where `Regions` can be inserted. Transition labels are displayed as their textual representation and also entered and parsed as such. In addition to weak abortion transitions, strong abortion transitions have a red dot symbolizing the preemption. Normal termination transitions are indicated by a green triangle. The history connector is displayed as a circled "H". Finally, all signals are enumerated in each macrostate's header bar.

4.2.3 Transformation

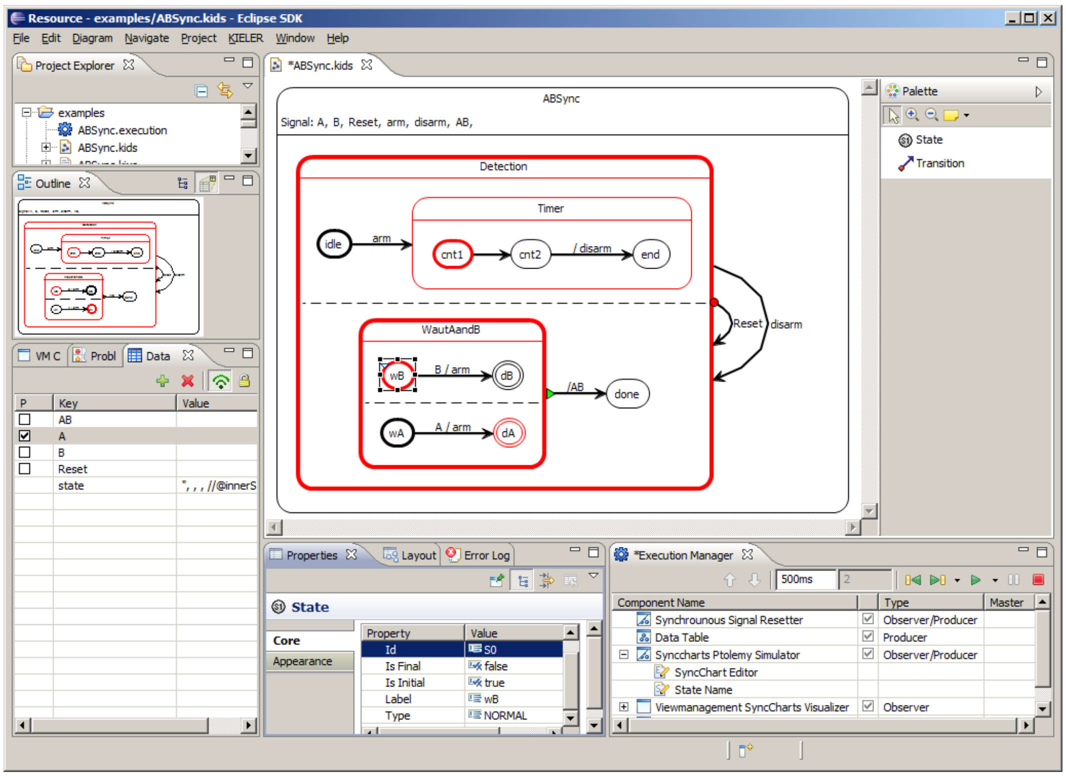
To make the knowledge about the SyncCharts language explicit and to give the syntactically correct SyncChart models an execution semantics, a model transformation into the Ptolemy semantic domain was chosen, as already done in the first case study.

Again Xtend suffices as a M2M transformation language from the (reduced) SyncChart domain model of Fig. 4.7 on page 46 into the Ptolemy domain model of Fig. 3.7. Additional effort has to be taken to bridge the synchronous semantics and Ptolemy's compositionality. This is done by the implementation of a new SyncChart director as described in Sec. 4.2.7.

In the following, all the main ideas of the transformation process are discussed. Because the overall transformation is rather large, implementation details will only be given in extracts in the following sections.

Simple-States

Simple-states are transformed into Ptolemy kernel states, which can be seen in Fig. 4.9. These belong to and are contained by a ModalModel actor that implements a flat finite state machine as described in Sec. 3.2. State names in Ptolemy are restricted to a smaller alphabet and need to be unique on different hierarchy levels (s.b.). Hence an additional hash-value is added to its possibly slightly adapted original name. The latter is usually relevant only to allow tracing between the two



P	Key	Value
☐	AB	
☑	A	
☐	B	
☐	Reset	
	state	", , , //@innerS

Property	Value
Id	S0
Is Final	false
Is Initial	true
Label	wB
Type	NORMAL

Component Name		Type	Master
Synchronous Signal Resetter	☑	Observer/Producer	
Data Table	☑	Producer	
Syncharts Ptolemy Simulator	☑	Observer/Producer	
SyncChart Editor			
State Name			
Viewmanagement SyncCharts Visualizer	☑	Observer	

Figure 4.8: Simulation of a SyncChart example model in the Eclipse GMF editor

models, e. g., if trying to simulate erroneous SyncCharts where an exception message from Ptolemy should help with debugging of the original SyncChart.

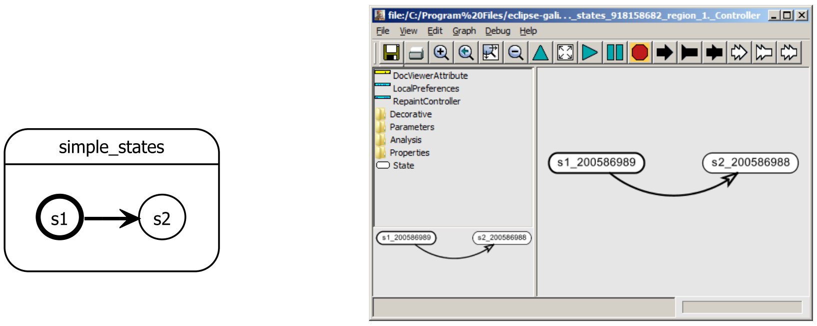


Figure 4.9: Transformation of simple-states

Transitions and Signals

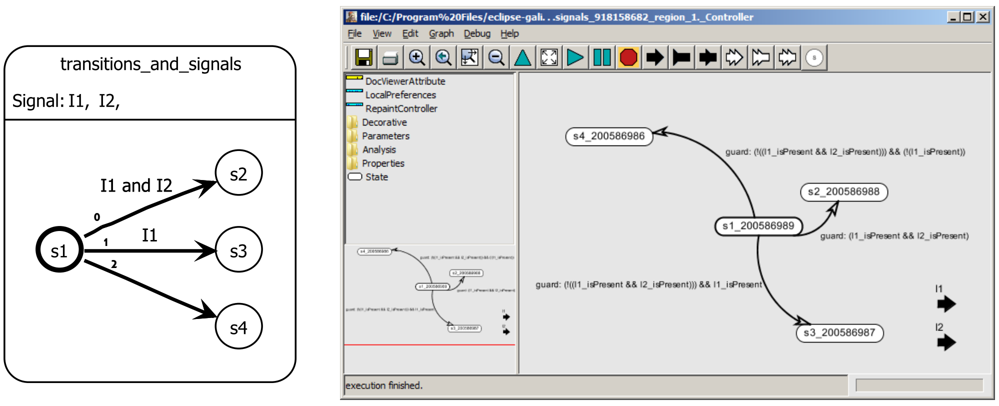
Transitions are basically supported by the ModalModel actor, unless some specialties of SyncChart transitions exist that need some extra effort to be expressed in Ptolemy. Fig. 4.10 shows three transitions from one initial source state to three different target states. Within the SyncChart these transitions have distinct priorities where the upper most transition has the highest priority. The ModalModel only distinguishes deterministic and non-deterministic transitions, but not priorities.

To address this, the transformation applies the following rule *trans*(): Assume a state with n outgoing transitions e_i with triggers t_i, priorities p_i and $i \in I := \{1..n\}$. For transition e_c with $1 \leq c \leq n$, the transformed trigger $trans(t_c)$ is computed as follows:

$$trans(t_c) = (!t_{k1} \wedge ... \wedge !t_{km}) \wedge t_c,\ k_1, ..., k_m \in \{k | k \in I \wedge p_k > p_c\}$$

This means that every trigger of one outgoing transition has an impact on all transformed triggers of outgoing transitions with a lower priority. An empty transition trigger is interpreted as a `true`.

Additionally, what can be seen in Fig. 4.10 is the declaration of two input signals `I1` and `I2` for which ports exist in the ModalModel actor to connect the signals to an outer data-flow layer.



```
guard: (!((I1_isPresent && I2_isPresent))) && (!(I1_isPresent))
guard: (I1_isPresent && I2_isPresent)
guard: (!((I1_isPresent && I2_isPresent))) && I1_isPresent
```

Figure 4.10: Transformation of transitions with signal triggers

Parallelism and Signals

Ptolemy's built-in concurrency is the main feature of using it as a semantic domain for SyncCharts, which also offer parallelism.

Fig. 4.11 shows all hierarchy layers of a Ptolemy model. The outermost layer, an implicit state of the SyncChart, contains two regions, separated by the dotted line. In the topmost state, a local signal `L` is defined that is used by the regions to communicate. Parallel regions are represented as concurrent ModalModel actors. All

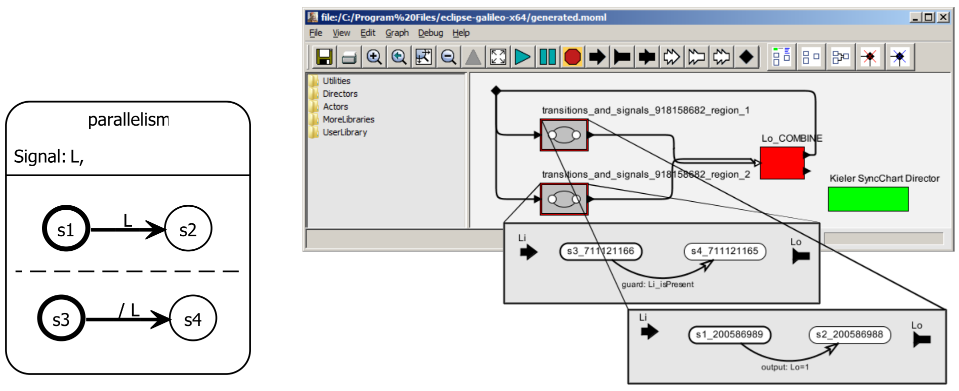

parallelism
Signal: L,
s1
L
s2
s3
/ L
s4
file:/C:/Program%20Files/eclipse-galileo-x64/generated.moml
File View Edit Graph Debug Help
Utilities
Directors
Actors
MoreLibraries
UserLibrary
transitions_and_signals_918158682_region_1
Lo_COMBINE
transitions_and_signals_918158682_region_2
Kieler SyncChart Director
Li
s3_711121166
s4_711121165
Lo
guard: Li_isPresent
Li
s1_200586989
s2_200586988
Lo
output: Lo=1

Figure 4.11: Transformation of parallel regions and local signals

concurrent ModalModels need to be connected to the same signals, defined or visible on the data-flow layer, which represents the parent state of the regions. This is because all regions need to have a consistent view on the signals and the ModalModels could possibly produce the same signals. All output tokens must be merged instantaneously with a (*non-strict*) merging combine actor. The latter is presented in Sec. 4.2.7.

A ModalModel can have incoming and outgoing ports. On the containing data-flow layer these are connected to other actors, e. g., other ModalModels. There, a token will be generated and *flow* iff a signal is present. Iff a signal is absent, no token is generated. The generation takes place initially at the specific port of the ModalModel where the signal is emitted as a consequence of an enabled and taken transition. The token may traverse several Ptolemy actors during the fixed point iteration.

The computation of the fixed point, to find a clear signal status for all local and all output signals, takes place in the SyncChart director that is explained in full detail in Sec. 4.2.7.

Local Signals

To compute a clear signal status for local signals, as seen in Fig. 4.11, is more involved than for input and output signals. This is because they can be used to communicate between parallel regions also across hierarchy levels. Therefore the flow of a possible token, indicating the presence of such a signal, must be feasible to cover all possible control-flows. Thus every ModalModel actor, which represents a region, has two ports for every local signal `L`, one input port named `Li` and one output port `Lo`. Both require an adequate renaming of transition triggers and emissions within the region.

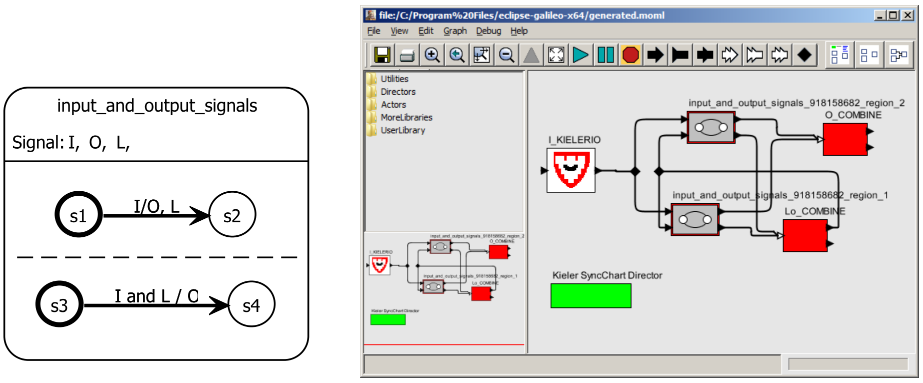

input_and_output_signals
Signal: I, O, L,
s1
I/O, L
s2
s3
I and L / O
s4
file:/C:/Program%20Files/eclipse-galileo-x64/generated.moml
File View Edit Graph Debug Help
Utilities
Directors
Actors
MoreLibraries
UserLibrary
I_KIELERIO
input_and_output_signals_918158682_region_2
O_COMBINE
input_and_output_signals_918158682_region_1
Lo_COMBINE
Kieler SyncChart Director

Figure 4.12: Transformation of input and output signals

Output Signals

Output signals are not allowed to appear within transition triggers. This makes it relatively easy to compute them. Possible tokens need to be passed to the outermost layer where they can be merged and interpreted by a simulation engine, see. Sec. 4.3. This is done by adding output signal ports to a ModalModels where they appear in some transition effect, see Fig. 4.13. For parallel regions, i. e., concurrent ModalModels, output signals are merged with the same Combine Actor as local signals.

Fig. 4.12 shows an output signal `O` that is emitted by two regions at the same time when input signal `I` is present.

Input Signals

Ptolemy models are closed models. Therefore a special actor is needed to dynamically insert tokens (representing present signals) during the execution. This is done by the IO Actor presented in Sec. 4.2.7. The simulation engine (see. Sec. 4.3) is able to inject tokens, representing input signals, if the environment sets them to a present status. Otherwise they are set to an absent status.

Fig. 4.12 shows an input signal `I` that is contained in triggers within two parallel regions.

Hierarchy

Hierarchy is another important feature of SyncCharts that Ptolemy natively supports. States can have an optional refinement. This allows a state to have another data-flow layer as its refinement. On this, several concurrent ModalModel actors may represent parallel inner regions of the original SyncChart macrostate.

Fig. 4.13 shows an example of hierarchy where macrostate `S2` contains a region with another simple-state `S3`. On the Ptolemy side a state with a refinement is

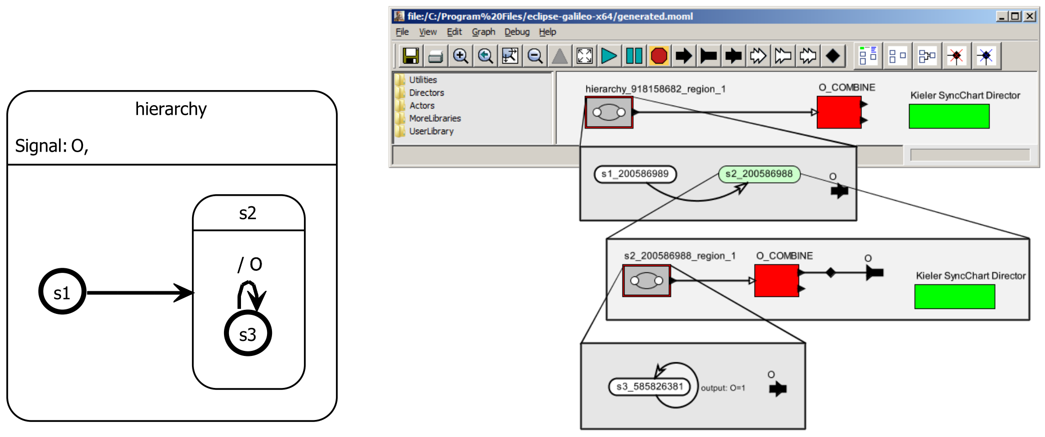

hierarchy
Signal: O,
s1
s2
/ O
s3
file:/C:/Program%20Files/eclipse-galileo-x64/generated.moml
File View Edit Graph Debug Help
Utilities
Directors
Actors
MoreLibraries
UserLibrary
hierarchy_918158682_region_1
O_COMBINE
Kieler SyncChart Director
s1_200586989
s2_200586988
O
s2_200586988_region_1
O_COMBINE
O
Kieler SyncChart Director
s3_585826381
output: O=1
O

Figure 4.13: Transformation of hierarchy

colored green, as it is the case for state S2. Signals need to be passed on correctly between all layers as explained above.

Normal Termination

The notion of a normal termination transition of SyncCharts is not natively supported by Ptolemy. A normal termination transition of a macrostate is triggered, if all inner regions of this macrostate have reached a state tagged as final. A normal termination has no other triggers but can have signal emissions in its effects.

The transformation uses the approach described by André [5]. In this approach a normal termination transition is replaced by a weak abort. The trigger of this abort is the conjunction of auxiliary signals, one for every inner region of the originating macrostate. These auxiliary signals must also be added to the emissions of all transitions that lead to final states in an inner region. Additionally, an auxiliary self-transition for each final state sustains the emission of these signals.

Pseudocode for this transformation is given in Listing 4.4. First, for all regions within the topmost state, `search_region()` is called. In a recursive manner for all states within the inspected regions, `search_state()` is invoked. Secondly, for all states, outgoing transitions are handled by `inspect_transitions()`. If a transition has the type normal termination, `build_triggers_and_effects()` takes care of the transformation details and the transition's type is changed to a weak abort.

In line 24 `build_triggers_and_effects()` retrieves the current macrostate, where an outgoing normal termination transition is going to be transformed, and the normal termination transition as well. For all inner regions, corresponding signals are created, added to the macrostate and to the trigger of the outgoing transition.

For all regions their final states are focused. For such a final state a new self-

Listing 4.4: Pseudocode to transform normal termination transitions

```
search_region(region r):
  for all states s of r
    if s is MACROSTATE then
       search_state(s)
    fi
    if s has OUTGOING_TRANSITIONS then
       inspect_transitions(s)
    fi
  rof

search_state(state s):
  for all regions r of s
    search_region(r)
  rof

inspect_transitions(state s):
   for all transitions t of s
     if (t is NORMALTERMINATION) then
        build_triggers_and_effects(s, t)
        set type of t to WEAKABORT
     fi
   rof

build_triggers_and_effects(state s, transition t):
   for all regions r of s
      signal_r := new signal
      add signal_r to s
      add signal_r to trigger of t

      for all FINAL states f of r
         transition_f := new transition
         set traget of transition_f to f
         add transition_f to transitions of f

         for all INCOMING transitions i of f
             set effect of i to emit signal_r
         rof
      rof
   rof
```

transition is created and then for all incoming transitions, an output emission is added. The latter consists of the added signal for the respective region.

Fig. 4.14 shows a macrostate with two regions that have two final states `S3` and `S6`. Whenever one of these states is reached in the transformed Ptolemy model, a token signalizing this is produced. If eventually both regions of `S1` have reached their final states, the weak abortion transition that has a conjunction of all auxiliary region signals as its trigger, is enabled. This finally leads to taking the normal termination transition and emitting the signal `O`.

A more detailed description of the Xtend implementation for this transformation is given in Sec. 4.2.4.

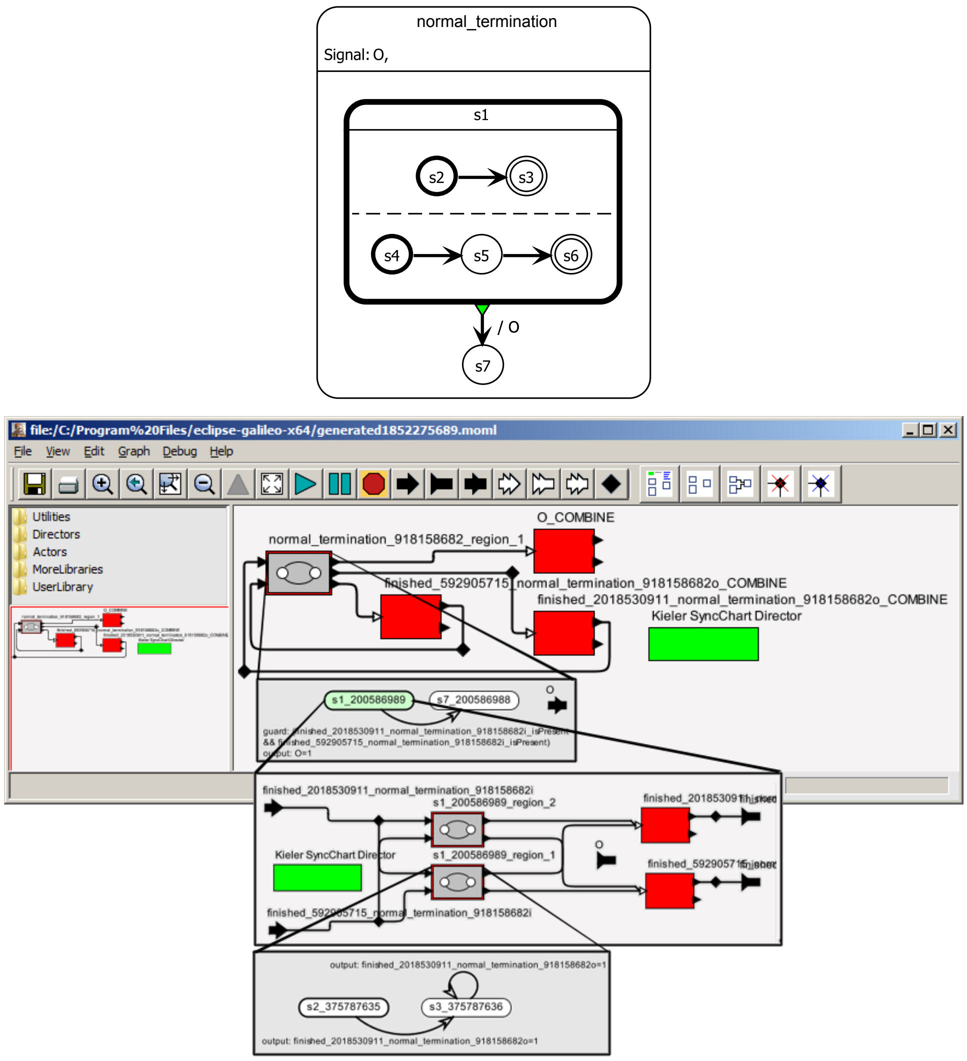

normal_termination
Signal: O,
s1
s2
s3
s4
s5
s6
/ O
s7
file:/C:/Program%20Files/eclipse-galileo-x64/generated1852275689.moml
File View Edit Graph Debug Help
Utilities
Directors
Actors
MoreLibraries
UserLibrary
O_COMBINE
normal_termination_918158682_region_1
finished_592905715_normal_termination_918158682o_COMBINE
finished_2018530911_normal_termination_918158682o_COMBINE
Kieler SyncChart Director
s1_200586989
s7_200586988
guard: (finished_2018530911_normal_termination_918158682i_isPresent && finished_592905715_normal_termination_918158682i_isPresent)
output: O=1
finished_2018530911_normal_termination_918158682i
s1_200586989_region_2
s1_200586989_region_1
finished_592905715_normal_termination_918158682i
output: finished_2018530911_normal_termination_918158682o=1
s2_375787635
s3_375787636

Figure 4.14: Transformation of normal termination transitions (omitting states S4, S5, and S6 in the Ptolemy model for clarity reasons)

Preemption and the History Connector

Strong abortion and the history connector, i. e., a transition marked as *preemptive* and not as a *reset* transition, are features directly supported by Ptolemy ModalModels. In the first case, if the trigger is true and the transition is taken, no signals that would be emitted inside a possible refinement are actually emitted, i. e., all signal emissions and state transitions inside are preempted. In the second case, a transition that is not marked to reset its target state, will cause this state to remain in its last configuration when it is reentered. In other words, if such a macrostate is reentered

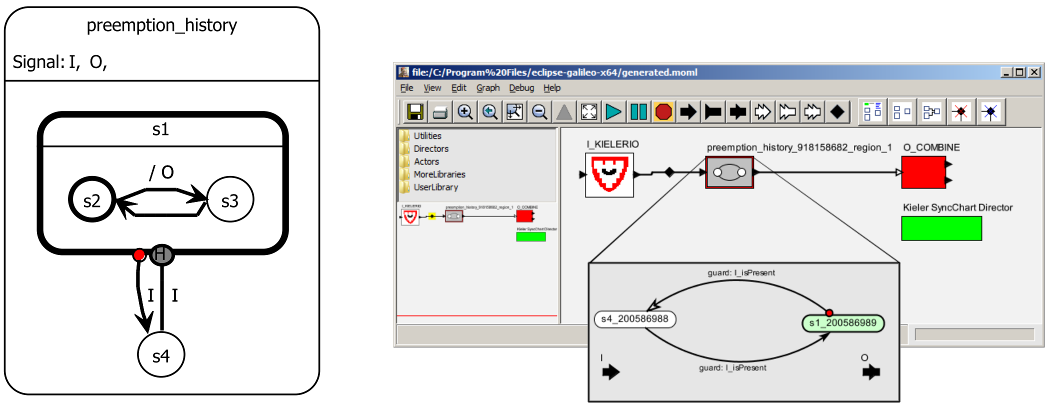


Figure 4.15: Transformation of preemption and history transitions

over a history transition, its last enabled states are still enabled, instead of its initial states as in the normal reset transition case.

Fig. 4.15 shows the two additional transition types and their corresponding transformation results in Ptolemy. Both strong abortion transitions in the SyncChart model and the preemptive transition in the Ptolemy model are indicated by a red circle. Note that the option *reset* can be chosen for a Ptolemy transition in its parameter settings but is not reflected in the graphical view.

4.2.4 Xtend M2M Transformation

The previously given transformation concepts are implemented using the Xtend transformation language. This language was introduced in Chap. 3. To give an impression of the rather direct implementation strategy that can be used in order to realize these transformations, the following presents an overview and a detailed example.

Implementation approach

In order to retain a better structure, the overall transformation resolves into the following three main parts:

1. Preprocessing transformation steps,
2. Recursive main transformation, and
3. Postprocessing transformation steps.

In the preprocessing steps, e. g., all normal termination transitions are transformed into an equivalent representation with a weak abort, as described in Sec. 4.2.3. Additionally, all local signals are renamed according to their hierarchy definition in order to make their names globally unique. In another preprocessing step, all local signals, defined within any inner macrostate, are raised up to the one and only topmost macrostate. The latter is done to guarantee that the simulation in Ptolemy later is able to find a signal assignment. This step is described in the following sections.

After preprocessing, the recursive transformation starts with the outermost state. This contains at least one region. For every region a ModalModel actor is created. Every region contains at least one state. For every state of a region that is a macrostate, a Ptolemy state with a refinement is created. For its inner regions this process continues recursively. If this state is a simple-state, then just a simple Ptolemy kernel state with no refinement is created. All hierarchy levels in the given SyncChart is directly mappable to the hierarchical structure of the constructed Ptolemy model.

During the creation of ModalModels, all local signals and all input and output signals that are referred to from within, result in input or output ports being constructed accordingly. These are in addition named and annotated to the name and the type of the signal they belong to. For every transition of the original SyncChart, a transition is constructed during the recursive main transformation steps. The trigger and effect of it is computed in Java escaped primarily imperative code.

After the second main transformation has completed, postprocessing steps finalize the transformation. They take advantage of the auxiliary annotations made during the transformation. One of these postprocessings connects all ports within Ptolemy state refinements; this is where original SyncChart regions are represented by ModalModels having several input and output ports. As another postprocessing step, all nondeterministic transitions are resolved by taking their original priorities into account. These have also been added as auxiliary annotations.

Normal Termination Transformation using Xtend

As an implementation example of a transformation step described in the previous paragraph, the normal termination transformation postprocessing is presented here.

The Xtend code implementing the pseudocode of Listing 4.4 is given in Listing 4.5. The transformation ideas were already discussed in Sec. 4.2.3.

The transformation step starts in line 1 with a call of `n2wRegion()` with the outermost region's states as a list parameter. This region contains a single macrostate representing the overall SyncChart. In line 2, the first state of the list (in this case there is only one) is saved in the `state` variable. In line 3, if there are other states, the recursive call takes place without the already considered one. This construction implements a recursive walk through a whole list of states and is also used in the other functions below. If this state is a macrostate, then in line 6 the `n2wState()` function is called that handles all regions of this macrostate and calls `n2wRegion()` again for their inner states. In line 7 the state is inspected for any outgoing transitions that are of interest because they might be normal termination transitions to be transformed.

If the latter is the case, `n2wTransition()` is called and handles all transitions of this macrostate. The state is also handed over as a parameter because it is needed

Listing 4.5: Xtend code to transform normal termination transitions

```
Void n2wRegion(List[syncharts::State] stateList) :
  let state = stateList.first():
  (stateList.withoutFirst().size > 0) ?
     n2wRegion(stateList.withoutFirst()) : null ->
  (state.isMacroState()) ?
   n2wState(state.regions) : null ->
  (!state.outgoingTransitions.isEmpty) ?
   n2wTransition(state.outgoingTransitions, state) : null;

Void n2wState(List[syncharts::Region] regionList) :
  let region = regionList.first():
  (regionList.withoutFirst().size > 0) ?
     n2wState(regionList.withoutFirst()) : null ->
  (region.innerStates.size > 0) ?
    n2wRegion(region.innerStates) : null;

Void n2wTransition(List[syncharts::Transition] transitionList, syncharts::State state) :
  let transition = transitionList.first():
  let complexExpression = new syncharts::ComplexExpression:
  (transitionList.withoutFirst().size > 0) ?
     n2wTransition(transitionList.withoutFirst(), state) : null ->
   complexExpression.setOperator(OperatorType::AND) ->
  (transition.type.toString().matches("NORMALTERMINATION") && (state.regions.size > 0)) ?
    transition.setTrigger(complexExpression) : null ->
  (transition.type.toString().matches("NORMALTERMINATION") && (state.regions.size > 0)) ?
    n2wBuildTriggerState(state.regions,
                   state.parentRegion.parentState,
                   transition) : null ->
  (complexExpression.subExpressions.size == 1) ?
     transition.setTrigger(complexExpression.subExpressions.get(0)) : null ->
  (transition.type.toString().matches("NORMALTERMINATION")) ?
    transition.setType(syncharts::TransitionType::WEAKABORT) : null;

Void n2wBuildTriggerState(List[syncharts::Region] regionList,
                          syncharts::State mainState,
                          syncharts::Transition weakAbortTransition) :
  let region = regionList.first():
  let regionSignalName = "finished_" + hash(region.innerStates.flatten().toString()):
  let regionSignal = new syncharts::Signal:
  let signalReference = new syncharts::SignalReference:
  let trigger = weakAbortTransition.trigger:
  regionSignal.setName(regionSignalName) ->
  regionSignal.setType(ValueType::PURE) ->
  mainState.signals.add(regionSignal) ->
  signalReference.setSignal(regionSignal) ->
  ((ComplexExpression)trigger).subExpressions.add(signalReference) ->
  (regionList.withoutFirst().size > 0) ?
    n2wBuildTriggerState(regionList.withoutFirst(), mainState, weakAbortTransition) : null ->
  (region.innerStates.size > 0) ?
   n2wBuildTriggerRegion(region.innerStates, mainState, weakAbortTransition, regionSignal) : null;

Void n2wBuildTriggerRegion(List[syncharts::State] stateList, syncharts::State mainState,
                           syncharts::Transition weakAbortTransition, syncharts::Signal regionSignal) :
  let state = stateList.first():
  (stateList.withoutFirst().size > 0) ?
    n2wBuildTriggerRegion(stateList.withoutFirst(), mainState, weakAbortTransition,regionSignal) : null ->
  (state.isFinal) ?
    n2wBuildTriggerFinalState(state, mainState, weakAbortTransition,regionSignal) : null;

Void n2wBuildTriggerFinalState(syncharts::State finalState, syncharts::State mainState,
                               syncharts::Transition weakAbortTransition, syncharts::Signal regionSignal) :
   let newTransition = new syncharts::Transition:
   newTransition.setTargetState(finalState) ->
   finalState.outgoingTransitions.add(newTransition) ->
   n2wBuildTriggerFinalStateTransitions(finalState.incomingTransitions(), weakAbortTransition, regionSignal);

Void n2wBuildTriggerFinalStateTransitions(List[syncharts::Transition] transitionList,
                                          syncharts::Transition weakAbortTransition,
                                          syncharts::Signal regionSignal) :
    let transition = transitionList.first():
    let signalEmission = new syncharts::Emission:
    signalEmission.setSignal(regionSignal) ->
    transition.effects.add(signalEmission) ->
    (transitionList.withoutFirst().size > 0) ?
          n2wBuildTriggerFinalStateTransitions(transitionList.withoutFirst(),
                             weakAbortTransition,
                             regionSignal) : null;
```

later when new auxiliary signals have to be added to it. `n2wTransition()` will walk through all (outgoing) transitions of the state in lines 17 to 32. A `Complex-Expression` is created (line 19) in case this is a normal termination. This is done because a normal termination does not have an explicit trigger but its transformation, the weak abortion, needs such a trigger. Because later the conjunction of all auxiliary signals is needed, the operator is set to `AND` in line 22. The further processing for all regions of the macrostate (e. g., the creation of the auxiliary signals) is done in `n2wBuildTriggerState()` that is called in line 26.

The function `n2wBuildTriggerState()` of line 34 walks through all regions of the macrostate. It creates a new signal in lines 38-39 and 42-43. This is added to the containing macrostate in line 44. In line 46, it is also added to the trigger of the considered outgoing weak abortion transition. Function `n2wBuildTrigger-Region()` of line 52 then searches for final states of a region contained in the macrostate. These final states are then handled by `n2wBuildTriggerFinal-State()` of line 62. A new self-transition is created in line 66 and added to the final state in line 68. Finally, all incoming transitions of the final state (including the just created self-transition) need to get an additional signal emission. The latter is done by function `n2wBuildTriggerFinalStateTransitions`. It walks through all incoming transitions of the final states and adds such a `Emission` in line 77.

4.2.5 Constructiveness

So far, the structural syntax and additional semantical aspects as well as the transformation and implementation ideas have been outlined. Now a short introduction into the constructive behavioral semantics as it is also found in the synchronous language Esterel [9] will be given, to help understanding the implementation of the Ptolemy SyncChart director.

A SyncChart with a cyclic dependency can be seen in Fig 4.16 on the left side. For such a SyncChart it cannot be decided whether signal `A` and signal `B` are present or not. If all kinds of cyclic SyncCharts would be rejected, also good-natured ones as shown in Fig 4.17 were not allowed. The constructive approach helps to be less restrictive.

The Signal Coherence Law

Within one tick, a signal can either be present or absent, but not both at the same time. A *signal assignment* (i. e., a collection of signals projected on the present or the absent state for a tick instance) conforms the coherence law, iff all (local) signals that are evaluated to be present are emitted in any taken transition, respecting scopes.

Logically Reactive SyncCharts

A SyncChart is called to be *reactive*, iff there is at least one signal assignment conforming to the coherence law. This must be true for every input signal assignment and every possible configuration. The SyncChart on the left side of Fig. 4.18 is not

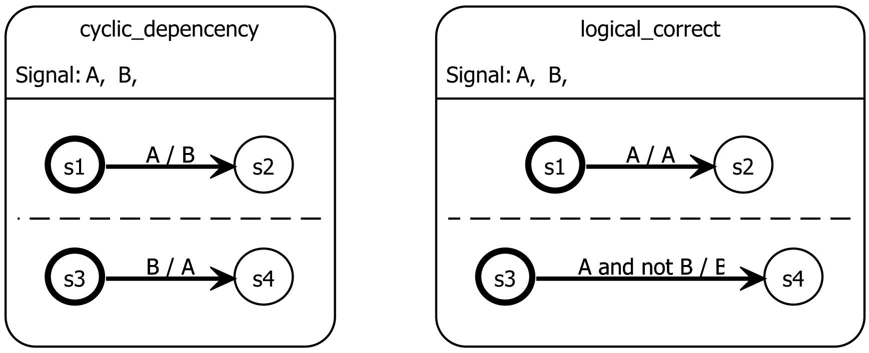


Figure 4.16: A cyclic (left) and a logical-correct (right) SyncChart

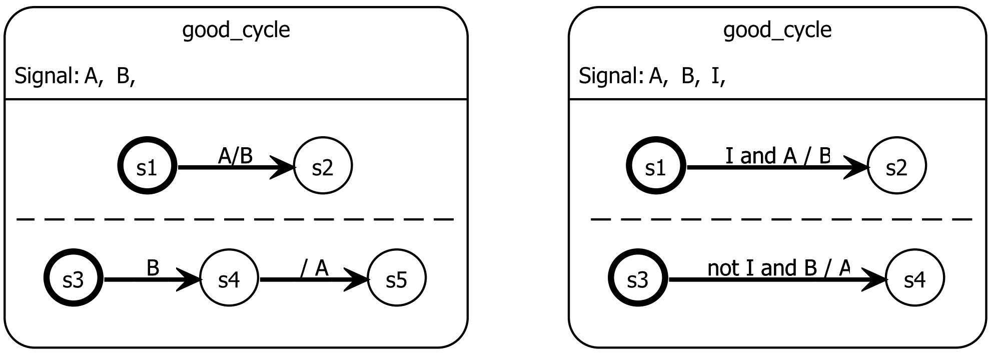


Figure 4.17: Examples of good-natured cylclic SyncCharts

reactive. This is because if on the one hand the signal A is assumed to be present, the upper transition is taken. But then the signal would never be emitted, which contradicts the coherence law. If on the other hand the signal A is assumed to be absent, the lower transition is taken. But then the signal would be emitted and the upper transition with its higher priority must have been taken instead. This means there is no signal assignment that conforms to the coherence law.

Logically Deterministic SyncCharts

A SyncChart is called *deterministic*, iff there is at most one signal assignment conforming to the coherence law. This must be true for every input signal assignment and every possible configuration. The SyncChart on the right side is not logically deterministic. This is because if on the one hand the signal A is assumed to be present, the upper transition is taken (because of its precedence). Also signal A is

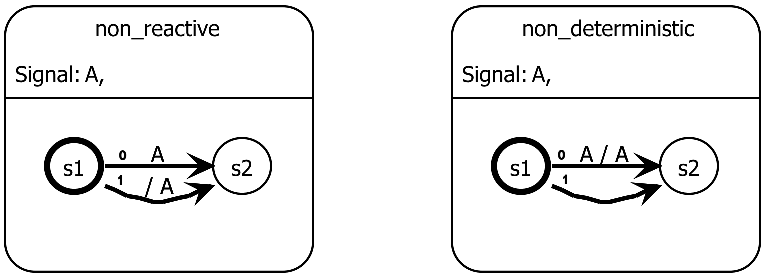


Figure 4.18: A non-reactive (left) and a non-deterministic (right) SyncChart

emitted making this assignment coherent. But if on the other hand the A is assumed to be absent, the lower transition is taken (because the upper one is not enabled) and no A is emitted. This makes the second assignment also conform to the coherent law, ending up in two such signal assignments.

Logically Correct SyncCharts

A SyncChart is called *logically correct*, iff there is exactly one signal assignment conforming to the coherence law. This must be true for every input signal assignment and every possible configuration. In other words, a SyncChart that is reactive and deterministic is logically correct. An example for such a SyncChart is shown in Fig. 4.16 on the right side.

Constructive SyncCharts

Because SyncCharts like the one on the right side of Fig. 4.16 are very difficult to analyze in the general case and make use of *self justification* and *speculative computing* [9], such SyncCharts should also be rejected.

According to Berry [10] and Ohlhoff [48], a signal assignment for constructive SyncCharts can be established following several rules:

1. All signals have the status *unknown* initially.

2. Input signals are set to their present status, i. e., to either present or absent.

3. During the computation, the status of local and output signals is computed considering the following rules:
    - If a signal *cannot be emitted*, its status must be set to absent.
    - If a signal *must be emitted*, its status must be set to present.
    - A signal *cannot be emitted*, if there is no *possibly enabled* transition that would emit it.

Listing 4.6: Pseudocode to find possibly emitted signals

```
returnList = new List

for all actorsAllowedToFire as actor
   if actor.type is ModalModel then
      currentState = ((ModalModel)actor).currentState()
      outTransitions = curentState.getOuttransitions()

  for all outTransitions as transition
     if (isPossiblyEnabled(transition))
        signalList = transition.getEmissionSignalList()
   returnList.append(signalList)
          fi
       rof
   fi

   if actor.type is StateRefinement then
      stateRefinement = (StateRefinement)actor
      nestedDirector = (SyncChartDirector)stateRefinement.getDirector()
      nestedSignalList = nestedDirector.getPossibleSignals()
      returnList.append(nestedSignalList)
    fi
rof
```

- A transition is *possibly enabled* if it is clearly enabled or if it cannot be clearly evaluated to `false`, e. g., because some signals in the trigger have an *unknown* status.
- A transition can be taken and possible signals must be emitted, if the trigger evaluates to `true` and if no other transition with a higher precedence exists, or all existing clearly evaluate to `false`.

If any signals remain *unknown* after all possible stepwise computations have been done, i. e., a fixed point for the signal assignment has been found, then this SyncChart has to be rejected as being non-constructive.

4.2.6 The Token Ring Arbiter

The Token Ring Arbiter presented by Berry [10] is an example for cyclic circuits. It models a token ring network, which naturally leads to a cyclic dependency of signals. This is because of its circular connection, where neighboring stations directly dependent on each others signals.

Fig. 4.19 shows a simplified version of the Token Ring Arbiter with three stations. Note that this can be extended to any number of stations. Each station in the ring has an input request signal (`R1`, `R2`, or `R3`) and an output grant signal (`G1`, `G2`, or `G3`). Additionally there exist local network signals, for synchronization between all stations. That are signals to pass on a *token* (`P1`, `P2`, and `P3`) and signals that represent the token itself (`T1`, `T2`, and `T3`).

At all times, exactly one member of the ring carries the token, i. e., exactly one of the signals `T1`, `T2`, and `T3` must be present. Whenever several stations try to access

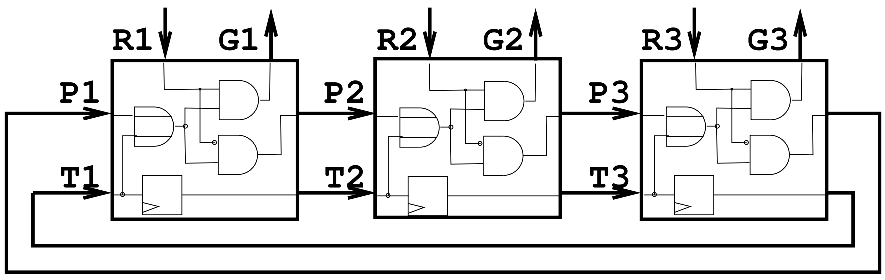


Figure 4.19: Simplified version of Token Ring Arbiter (from [39])

the network, i. e., request the bus using their signals `R1`, `R2`, or `R3`, only one station may succeed to ensure mutual exclusion. This is the station closest to the token (in an unidirectional way). The token is propagated on in the same unidirectional way to the next station using the signals `T1`, `T2`, or `T3` respectively. This ensures fairness.

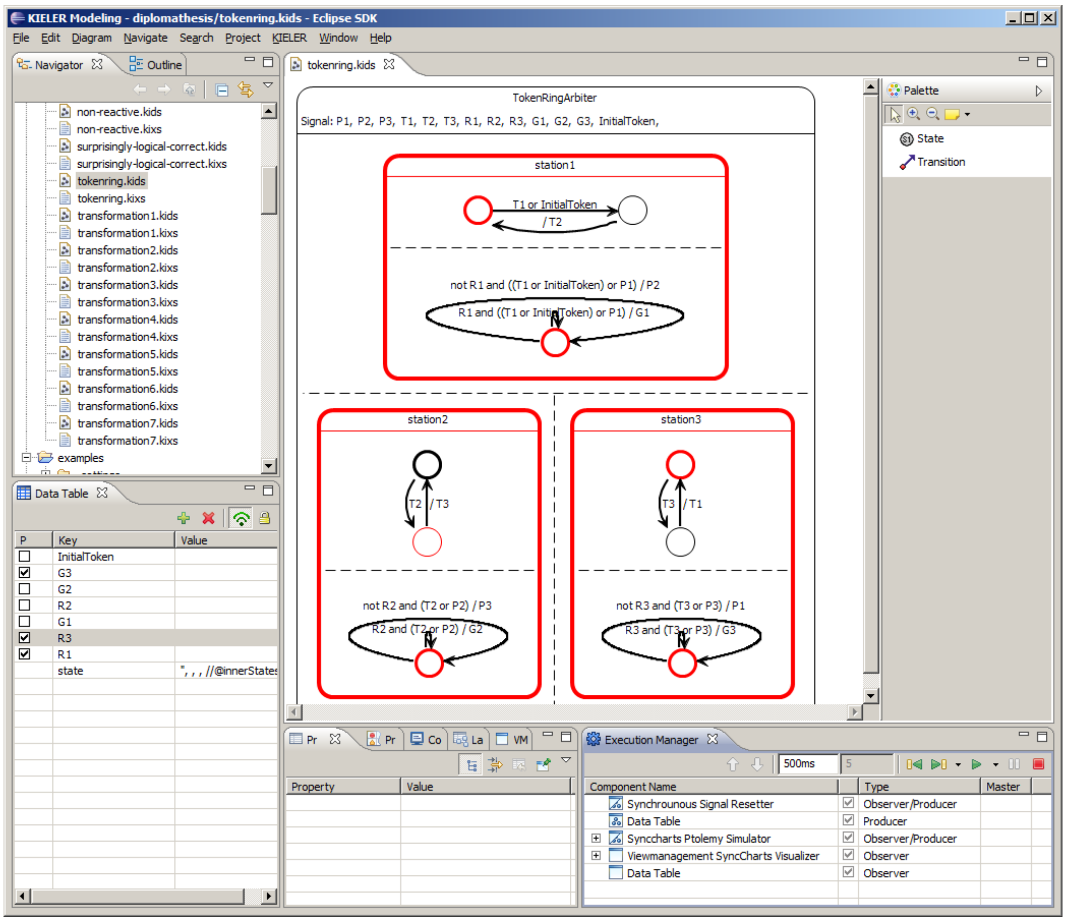
The Token Ring Arbiter is a paradigm for a system with a static cycle that only dynamically can be broken up, by the circulating token. Most Esterel compilers cannot handle this kind of cycles, as they are difficult to detect. This difficulty is explained in more details by Lukoschus and von Hanxleden [40].

Fig. 4.20 shows the Token Ring Arbiter modeled in SyncCharts and simulated with KlePto in KIELER. All network signals are modeled as local signals. The upper regions of all stations implement the token circulation where `station1` passes the token on to `station2`, `station2` passes the token on to `station3`, and `station3` passes the token on to `station1`.

The lower regions are the more interesting ones. Here the granting takes place. The reaction is as follows and matches to the one of figure 4.19:

- If a station gets a token or a pass signal, it is allowed to react:
    1. If it has an open request (signal), it grants access.
    2. If there is no present request signal, it will pass on the access right using the pass signal of the next station in order.

These two possible reactions are modeled with two transitions for each station. Note that `station1` has an additional input signal `InitialToken` it reacts to. This is because some station *must* have an initial token. Otherwise this would lead to non-determinism because of cyclic dependencies as outlined by Berry [10]. Thus, in order to simulate this in KIELER, the `InitialToken` input signal must be set to be present, before the first simulation step is allowed to take place. Afterwards it should be absent.



P	Key	Value
☐	InitialToken	
☑	G3	
☐	G2	
☐	R2	
☐	G1	
☑	R3	
☑	R1	
	state	", , , //@innerStates

Component Name		Type	Master
Synchrounous Signal Resetter	☑	Observer/Producer	
Data Table	☑	Producer	
Synccharts Ptolemy Simulator	☑	Observer/Producer	
Viewmanagement SyncCharts Visualizer	☑	Observer	
Data Table	☑	Observer	

Figure 4.20: Token Ring Arbiter defined in SyncCharts

4.2.7 Ptolemy Extensions

SyncChart Director

The SyncChart director is the heart of the Ptolemy SyncChart simulation. It is based on the Ptolemy `FixedPointDirector`. As such, it performs a fixed point calculation for one tick instance in order to find a signal assignment for constructive SyncCharts as introduced in the section before. Because Ptolemy is designed to be very modular but the constructive semantics is not[4], the SyncChart director has to

[4] Huizing and Gerth [32] showed that the combination of the synchronous hypothesis with causality and modularity cannot be done with any simple semantics. Hence, because SyncCharts are based on the synchronous hypothesis and are causal, they cannot be modular.

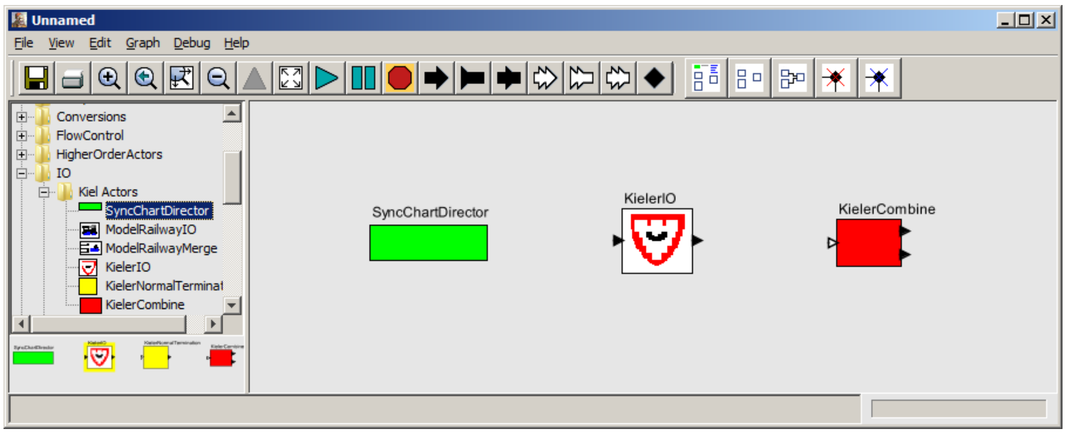


Figure 4.21: Ptolemy extension actors necessary for the supported subset of SyncCharts

bridge this gap. It does so during the execution only because the modular design in Ptolemy should be preserved as far as possible.

No other predefined director that comes bundled with Ptolemy fulfills these tasks. This is why the SyncChart director is needed to reasonably compute signal values following the SyncCharts semantics.

To cope with the constructive rules presented in Sec. 4.2.5, the SyncCharts Director needs to do a *must-cannot analysis* (see [9] or [53]) w.r.t. local signals. Therefore it needs to know which signals could possibly be emitted within one fixed point calculation. The pseudocode of Listing. 4.6 addresses this issue. All actors need to be inspected of their type:

ModalModel: In this case all outgoing transitions of the current ModalModel state are inspected. If an outgoing transition is found to be possibly enabled then the signals appearing in the emission of its effect are added to a return list, representing all possibly emitted signals.

State with refinement: In this case a recursion gaps the modularity and hierarchy to preserve the constructive semantics. The SyncCharts Director gets all signals that are possibly emitted from the inside of a macrostate. These are also added to the return list.

The function that returns a boolean value depending on the fact whether a transition is possibly enabled or not is shown in Listing 4.8.

A transition in Ptolemy can be asked whether it is enabled or not by calling the `isEnabled()` method. It is enabled, iff the trigger of this transition can clearly be evaluated to `true`, where *unknown* signals are considered under the logical inference rule of neutrality. For example, let signal `P` be present and signal `A` be absent. The

Listing 4.7: Pseudocode extending the Ptolemy fixed point iteration

```
schedule = orderIOActorsfirst(schedule)

do
   do
      // normal fixed point iteration of Ptolemy
   while not iterationConverged()

   possibleSignals = getPossibleSignals()
   stateSignals = getStateSignals()

   for all (stateSignals as signal)
      if not possibleSignals.contains(signal) then
         setToAbsent(signal)
      fi
   rof

while not iterationConverged()
```

status of signal `U` is *unknown*. Then the following can be inferred for triggers t_1 and t_2 of two transitions e_1 and e_2:

$$t_1 := \texttt{P} \lor \texttt{U}$$
$$t_2 := \texttt{A} \land \texttt{U}$$

$$enabled(e_1) = \texttt{P} \lor \texttt{U} = \texttt{true} \lor \texttt{U} = \texttt{true}$$
$$enabled(e_2) = \texttt{A} \land \texttt{U} = \texttt{false} \land \texttt{U} = \texttt{false}$$

If the enabledness of a transition cannot be inferred, as it would be in cases with triggers $t_3 = \texttt{P} \land \texttt{U}$ or $t_4 = \texttt{A} \lor \texttt{U}$, then an exception is thrown by Ptolemy. In this case the transition cannot yet be clearly declared neither to be enabled nor to be disabled. But it still has the status *possibly enabled* and `true` must be returned (see Listing 4.8). In all other cases `false` can safely be returned, as the transition can clearly be determined to be disabled.

With the above considerations, the idea of extending the fixed point iteration can be discussed. It is illustrated in the pseudocode of Listing 4.7. The normal fixed point iteration stops, when no further input ports of the considered (scheduled) actors have either been *cleared* (i. e., indicating the absence of a token) or got a new token.

All possibly emitted signals are retrieved by calling the `getPossibleSignals()` method described above. The outermost SyncChart director carries a list of all local signals that can be accessed by calling the `getStateSignals()` method.

Because only transitions on the current hierarchy level or lower levels are inspected, to deal with possible emissions in higher levels, the approach here is to compute the possible emissions starting at the highest level. Hence, all local signal declarations need to be raised to the highest hierarchy level (after making their names unique). For the restricted subset of SyncCharts considered here, this is not a problem because

Listing 4.8: Java code to decide whether a transition is *possibly* enabled

```java
public boolean isPossiblyEnabled(Transition transition)
                                              throws IllegalActionException {
    try {
        //if we for sure know this transition is enabled we can return true
        if (transition.isEnabled()) {
            return true;
        }
    } catch(UndefinedConstantOrIdentifierException e) {
        //if we cannot evaluate the transition trigger because of a missing
        //signal status, we must return also true here
        return true;
    }
    //if we for sure know this transition is disabled we can return false
    return false;
}
```

Listing 4.9: `fire()` method of IO Actor

```java
public void fire() throws IllegalActionException {
    // dispose an optional trigger token
    if (trigger.getWidth() > 0) {
        if (trigger.hasToken(0)) {
            trigger.get(0);
        }
    }

    // if signal is present, send out an integer token with the signal's value
    if (present.getValueAsString().equals("true")) {
        signal.send(0, new IntToken(Integer.valueOf(value.getValueAsString())));
    }
    // else send clear because signal is absent
    else {
        signal.sendClear(0);
    }
    super.fire();
}
```

it is not possible to access signal states/values of other ticks (e. g., by using `pre` or `suspend`).

For all these local signals it is determined whether they are possibly emitted or not. If such a signal is not in the list of the possibly emitted signals, then it *cannot* be emitted under any circumstances in a constructive setting. Hence it can be set to absent, i. e., the respective input ports of scheduled actors can be *cleared*. If any new local signal has been set to absent in this manner, the fixed point iteration has to continue.

After a fixed point has been found, hopefully, all signals have a *known* status and either are present or absent. If this is not the case, an exception is raised in the `postfire()` method, just like the Ptolemy `FixedPointDirector` handles this. It indicates a causality loop in the original SyncChart model, making it non-constructive.

IO Actor

The IO Actor, as shown in Fig 4.21 and used for example in Fig 4.15, is necessary to inject tokens into a running Ptolemy model. This actor of the type `TypedAtomicActor` has 4 parameters:

1. A signal name that can be used to map signals in the Ptolemy model to a signal in the original SyncChart,
2. A boolean present status,
3. An integer value of the signal, and
4. An optional permanent flag, indicating that a possible present signal should also be emitted in further ticks.

Additionally, this actor and its parameters can fully be controlled programmatically with respective getter and setter methods. Its relatively simple and self-explanatory `fire()` method is illustrated in Listing. 4.9.

Combine Actor

The Combine Actor as shown in Fig. 4.11 is responsible for merging the signals of parallel regions in the SyncChart and hence for merging tokens of concurrent ModalModels in the Ptolemy model. It is already equipped to handle *valued signals* (i. e., signals that in addition to their present status also carry an integer value). These are currently not supported by the SyncChart simulation (see. Sec. 4.2.1).

Therefore it computes a combined value using one of the predefined commutative and associative functions shown in Table 4.1. It provides two output ports, one that states whether the signal is present or absent, and an additional one that states the combined value of the signal. This actor has a *multiport* for the inputs and is not strict. The latter means that it can produce output tokens as soon as any input token arrives on any input channel. It will only clear its output, if it cannot expect any input tokens, i. e., all input channels have been cleared.

Its `fire()` method is illustrated in Listing. 4.10. In the upper part of this method the combined value is calculated based on all available input tokens. In the lower part it is tested whether all input channels are cleared. This is the case if there has not any token been received yet. But if there was a token, or several tokens, the combined value is sent out.

Listing 4.10: `fire()` method of Combine Actor

```java
public void fire() throws IllegalActionException {
    super.fire();

    // check if any ports have known inputs
    for (int i = 0; i < input.getWidth(); i++) {
        if (input.isKnown(i) && input.hasToken(i)) {
            _present = true;
            // get the token
            IntToken in = (IntToken) input.get(i);
            if (in != null) {
                // apply commutative+associative combine function
                _value = _updateFunction(in.intValue(), _value);
            }
        }
    }

    if (!_present) {
        // check if all ports are cleared (known w/o any token)
        boolean allKnown = true;
        for (int i = 0; i < input.getWidth(); i++) {
            allKnown &= input.isKnown(i);
        }
        // if no token can arrive, clear the output
        if (allKnown) {
            output.sendClear(0);
            value.sendClear(0);
        }
    } else {
        // send out combined integer token if presentToken
        output.send(0, new IntToken(1));
        value.send(0, new IntToken(_value));
    }
}
```

Function	**Output**
CONST	A constant integer value
ADD	The sum of all received tokens
MULT	The product of all received tokens
MAX	The maximum value of all received tokens
MIN	The minimum value of all received tokens
AND	Logical conjunction interpreting `1` as `true` and `0` as `false`
OR	Logical disjunction interpreting `1` as `true` and `0` as `false`

Table 4.1: Combine functions used by the Combine Actor

Listing 4.11: Method for initializing loading and running the Ptolemy Model

```java
public synchronized void executionInitialize() {
    // create a file url
    URI momlFile = URI.createFileURI(new File(PtolemyModel).getAbsolutePath());
    // create new MoML parser
    // make sure Ptolemy is in dependencies
    MoMLParser parser = new MoMLParser();
    // create lists for iterating inputs and outputs
    kielerIOList = new LinkedList<KielerIO>();
    modelOutputList = new LinkedList<ModelOutput>();
    // parse the Ptolemy moml file
    NamedObj ptolemyModel = null;
    ptolemyModel = parser.parse(null, new URL(momlFile.toString()));
    parser.reset();
    // start executing the model
    if (ptolemyModel != null && ptolemyModel instanceof CompositeActor) {
        // check if the parsed model is of correct type
        modelActor = ((CompositeActor) ptolemyModel);
        // get the manager that manages execution
        manager = modelActor.getManager();
        // go thru the model and add fill the kielerIOList (Inputs)
        fillKielerIOList(kielerIOList, extracted());
        // go thru the model and add fill the kielerCombine (Outputs)
        fillModelOutputList(modelOutputList, extracted());
        // run the model
        if (manager != null) {
            // run forest, run!
            manager.initialize();
        }
    }// end if
}
```

4.3 KIELER leveraging Ptolemy Simulation Component

The KlePto simulation component handles three activities:

1. Transform the DSL model into a Ptolemy model,
2. Execute the transformed model with Ptolemy, and
3. Handle the interfacing of the model.

As the transformation in general is unique w.r.t. to the DSL, the execution and interfacing can be quite similar, at least for DSLs of the same nature, e. g., state based ones. Hence, in this section the focus is on the execution and interfacing activities.

The general idea of transforming, executing, and interfacing, with the Execution Manager presented in Chap. 5, is depicted in Fig. 4.22. For the upper part, a M2M transformation needs two metamodels and a model as its input. The target metamodel is the metamodel of Ptolemy, the metamodel of the DSL in question is the source metamodel. The model should conform to the source metamodel. The transformation creates a new model that then conforms to the target metamodel,

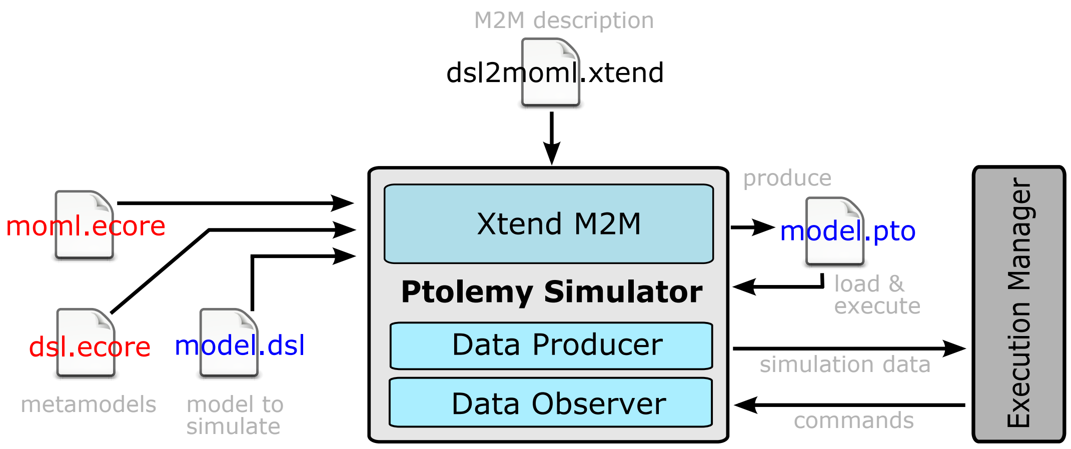


Figure 4.22: Abstract transformation and execution scheme of KlePto

i. e., is a Ptolemy model. This can be loaded and executed as described in the following sections. Additional care needs to be taken to maintain mappings between both models. A DataObserver and DataProducer part of the simulation component then interacts with the Execution Manager. The DataObserver reads in possible commands (input signals) from the environment. The DataProducer writes out resulting simulation data, e. g., the current state or emitted signals. Details of these components will be discussed in the next chapter.

Execution of Ptolemy Models

Before executing the Ptolemy model produced by the M2M transformation, it must be read into memory from a MOML file and parsed. Listing 4.11 shows the corresponding Java code. Lists of actors for handling the input and outputs are created in lines 8 and 9. The actual parsing takes place in line 12. The Ptolemy model is contained in the `ptolemyModel` object that should be of type `CompositeActor`. A Ptolemy manager is extracted (see line 19), which is necessary to iterate over the model.

The execution of a Ptolemy model step, equivalent to a synchronous tick because of the SyncChart director, then takes place in several computation steps illustrated in Fig 4.23. First all output signals are reset to be absent. Then the list of IO Actors (see Sec. 4.2.7) that was filled during the initialization is filled with the present status of all input signals defined in the environment. How signals are represented in the environment and how this concretely can be done is explained in Chap. 5.

Now the Ptolemy manager's `iterate()` method is called. This triggers one fixed point iteration for the given set of input signals and the current configuration, i. e., the current states of the ModalModels.

Because the simulation wants to extract the currently active states, this is done in a recursive fashion. In Ptolemy, all ModalModels have a current state, even the

Listing 4.12: Setting model input signals

```java
// iterate thru all kielerIOs = set the input signals
// assuming inputData is a JSONObject containing environment signals
for (KielerIO kielerIO : kielerIOList) {
    // calling the signal name getter method
    String signalName = kielerIO.getSignalName();
    // resolving environmental signal status
    boolean isPresent = false;
    if (this.inputData.has(signalName)) {
       Object object = this.inputData.get(signalName);
       isPresent = (JSONSignalValues.isPresent(object));
    }
    // calling the present status setter method
    kielerIO.setPresent(isPresent);
}
```

Listing 4.13: Getting model output signals

```java
// returnObj should afterwards contain all present output signals
JSONObject returnObj = new JSONObject();
// iterate thru all Combine Actors = get the output signals
for (int c = 0; c < this.modelOutputList.size(); c++) {
   // if the signal status of the Actor is present
   if (this.modelOutputList.get(c).present) {
       // retrieve the name
       String signalName = this.modelOutputList.get(c).signalName;
       // create a new signal with this name and a present status true
       JSONObject signalObject = JSONSignalValues.newValue(true);
       // add this to the returnObject
       returnObj.accumulate(signalName, signalObject);
   }
}
```

ones that do not appear as refinements of other current states. To adapt this interpretation, the recursion needs to proceed only in depth for ModalModels appearing within current states.

Also the present status of all output signals is of interest. Therefore it is iterated through all Combine Actors (see Sec. 4.2.7) that merge output signals on the topmost hierarchy layer and know about their present statuses.

Interfacing and Mapping

The interfacing with the environment, i. e., with KIEM, uses the JSON data format (see Sec. 3.3). Input and output signals are addressed using the original signals names, which must be unique. The mapping is strongly linked to the transformation process, as the IO Actor used for input signals and the Combine Actor used for output signals, must store an additional *signal name* value. This value is set during the transformation because only there the mapping is implicitly clear by construction. It can be made explicit in the transformation only. This is done by annotating the

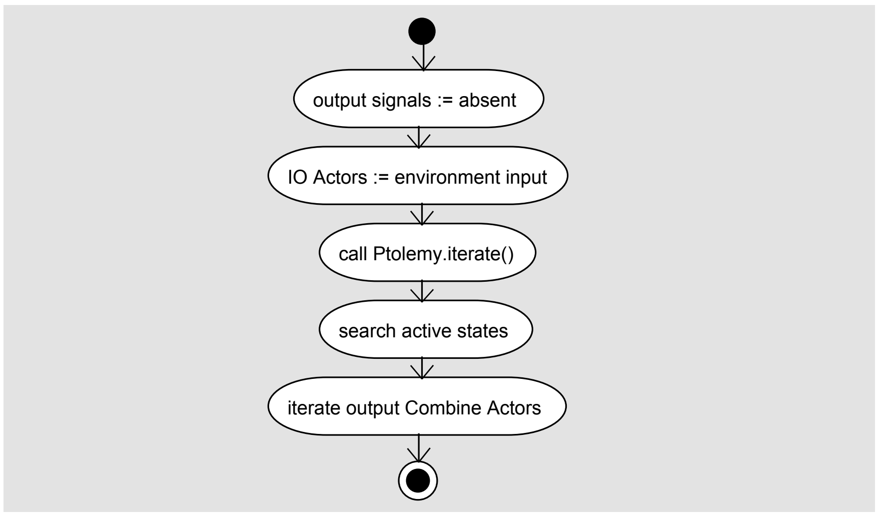


Figure 4.23: Computing an execution step with KlePto

Ptolemy model (e. g., by adding such parameters). The same is true for the states of ModalModels. At the time they are constructed by the transformation, it is clear to which original SyncChart state the new state belongs. Because names of states are not uniquely defined across hierarchy levels, another option to address EMF objects within a model is used for this purpose: *FragmentURIs*, similar to the *XPath*[5] notation, serve to identify states. Their string representation makes it easy to attach them as an additional parameter to created Ptolemy ModalModel states.

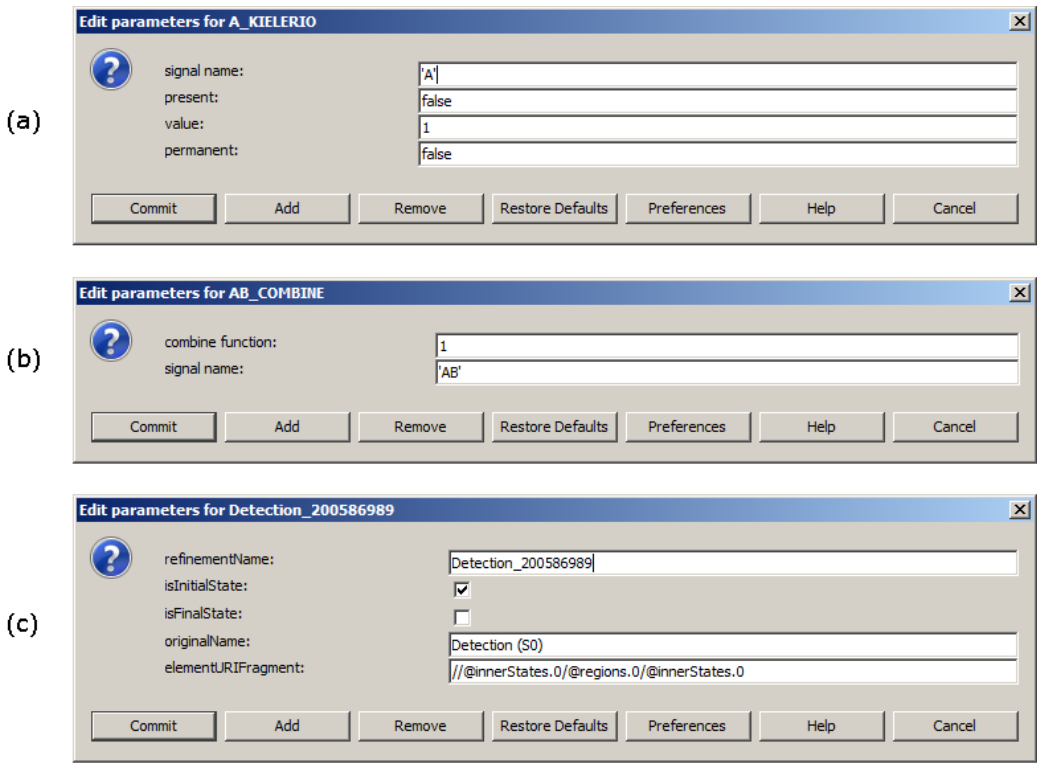
Listing 4.12 shows how input signals can be injected using the IO Actor introduced in Sec. 4.2.7, where the signal name serves as a unique identifier. In line 9 a JSON implementation method is used to retrieve a `JSONObject` for a given signal name. The present status is tested in line 10, with a special implementation that conforms to a convention for representing synchronous signals in JSON (see Chap 5). In Listing. 4.13 the corresponding part for extracting the signal status of output signals using the Combine Actor (see Sec. 4.2.7) is shown. For every present signal a `JSONObject` of the same name is created and merged into an overall return value.

In Fig. 4.24 the mapping of signals (w.r.t. their names) and of states (w.r.t. their FragmentURI) is shown according to the example of Fig. 4.8.

4.4 Visualization Simulation Component

The visualization of SyncCharts (as seen in Fig. 4.8) takes place directly in the GMF model editor. Hence it can be called a *model visualization*.

[5]`http://www.w3.org/TR/xpath20/`



(a) Edit parameters for A_KIELERIO

signal name:	'A'
present:	false
value:	1
permanent:	false

(b) Edit parameters for AB_COMBINE

combine function:	1
signal name:	'AB'

(c) Edit parameters for Detection_200586989

refinementName:	Detection_200586989
isInitialState:	✓
isFinalState:	☐
originalName:	Detection (S0)
elementURIFragment:	//@innerStates.0/@regions.0/@innerStates.0

Figure 4.24: Mapping of (a) input signals, (b) output signals and (c) states

The component's task is to highlight currently active states. These states are computed by the simulation engine as seen in the previous section. More precisely, the simulation engine computes the FragmentURI, a unique identifier of a `State` EMF object. By using this identifier, the object to highlight (i. e., a GEF *EditPart* representing this `State` instance) can be retrieved by polling the specific GMF editor. The highlighting of this visualization follows a more general pattern:

1. A *trigger* indicates that the active state(s) have changed and the visualization should update its highlighting. This is usually done in a synchronous manner, as explained in Chap. 5. The component's input is a list of active state FragmentURIs. In every execution step it computes the difference of the current and the last active states. If there exists a difference, the trigger becomes `true`. Otherwise it is `false`.

2. A visualization *effect* is the resulting action of a visualization. In this case it is the highlighting of a state by coloring its bounds in red.

3. A *combination* of trigger and effect combines both and makes sure that the following equivalence always holds:

   $$\textit{trigger} \Leftrightarrow \textit{effect}$$

   This means that a specific state is visualized using the effect, if and only if the trigger for this specific state holds. Otherwise the visualization effect has to be removed, if it was set before.

The implementation of triggers, effects and combinations is detailed elsewhere [8].

5 Execution Framework

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. It does not do any simulation computation itself, but it combines simulation components, visualization components, and a user interface to control execution within the KIELER application, as indicated in Fig. 5.1.

These components can simply be constructed using the Java language by implementing some commonly defined interfaces. A generic approach on how to prepare the implementation of such simulation engines themselves for a specific DSL has just been presented in the previous chapter. In this chapter, a concrete implementation strategy used in the KIELER project will be given. Additionally, the Execution Manager infrastructure and implementation details are analyzed.

5.1 Motivation

As outlined in Chap. 1 and presented in Chap. 3 with EMF, there exists a well established framework for Eclipse that helps building domain specific languages in a model driven approach. Additionally, there exists an IEEE standard for a modeling and simulation high-level architecture [33]. No approved Eclipse project covers the latter yet. Because in the KIELER project there is a need for being able to simulate created models, the KIELER Execution Manager bridges this gap.

It is desirable to have an architecture that satisfies the following requirements:

Modularity: Several interacting but independent components should be able to participate in a simulation run.

Performance: It should be possible by design that these components are able to perform their tasks as concurrently as possible.

Extensibility: Simulation components need to have a very flexible and clear interface. On the one hand the interaction should conform to a commonly agreed standard, but on the other also be very efficient.

Flexibility: It must be ensured that several kinds of tasks can be fulfilled by such components, e. g., user interaction, visualization, recordings, online-debugging, various communication, validation or other analysis functionality.

Usability: The user interface must also be simple and clear allowing the user to do as much as possible with a minimum number of interaction steps.

Interactivity: The user should be able to interact with the model during simulation. This requires a discrete, stepwise execution.

5.2 Framework Overview

The KIEM framework provides the main simulation infrastructure for the KIELER project. Fig. 5.1 shows it as a gray box where so-called *DataComponents* can be attached. These components are able to communicate with each other via interfaces provided by KIEM. Hence, KIEM can be seen as an implementation of a communication bus between independent and concurrent DataComponents. As also depicted in Fig. 5.1, these DataComponents can cope with various tasks ranging from simple input/output facilities over communication or recording processors to simulation engines that can be less or more complex and generic.

In the following sections these DataComponents, the interaction and scheduling mechanisms, and other concepts that lead to the current design of KIEM are discussed.

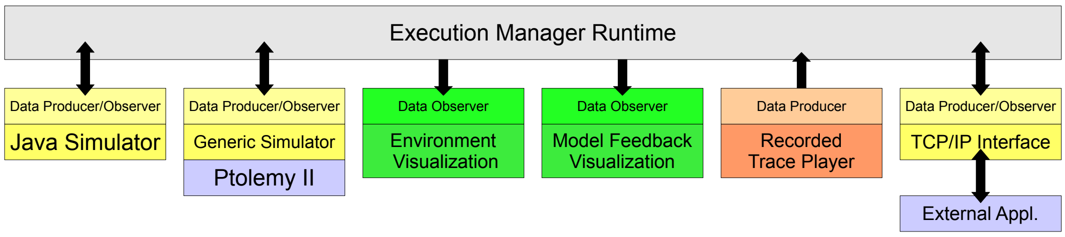


Figure 5.1: Schematic overview of the Execution Manager infrastructure

5.2.1 DataComponents

DataComponents are the building blocks of executions in the KIEM framework. These components are pure Java code that is restricted to meet a special interface so that the Execution Manager is able to address all components and interact with them in the same way. Therefore the Eclipse plug-in concept is used (see Sec. 3.1). In particular, the Execution Manager provides an extension point that DataComponents can implement. 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 the same time. Fig. 5.1 shows an example setup.

Therefore DataComponents can be classified according to their type of interaction into four categories:

1. *Initializer* DataComponents usually neither observe nor produce any data. Examples are the running of a web server (see Sec. 5.3.8) during the execution or the synchronization of a data base before and after a simulation run. Other

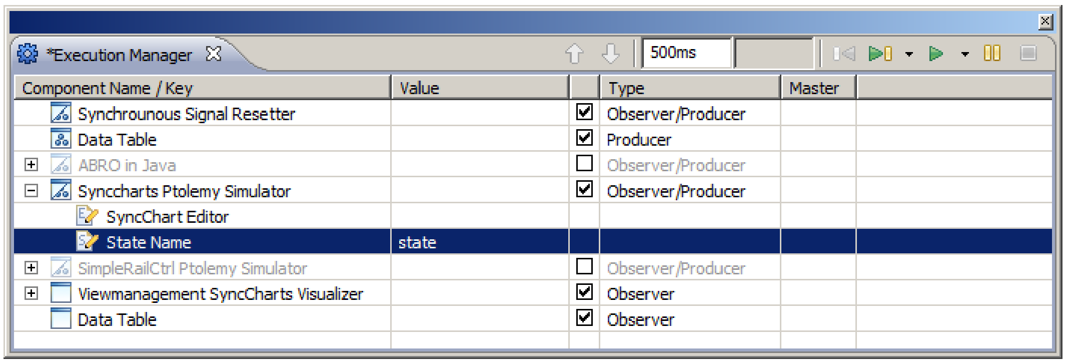
tasks could be a M2M transformation or code generation and compilation as presented in the previous chapter.

2. *Pure observer* DataComponents do not produce any data, e. g., 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, such as simulation engines that react to input with some output.

In Fig. 5.1 also the type of a DataComponent is denoted at the top of each box representing one.



*Execution Manager

500ms

Component Name / Key	Value		Type	Master
Synchrounous Signal Resetter		✓	Observer/Producer	
Data Table		✓	Producer	
⊞ ABRO in Java		☐	Observer/Producer	
⊟ Synccharts Ptolemy Simulator		✓	Observer/Producer	
SyncChart Editor				
State Name	state			
⊞ SimpleRailCtrl Ptolemy Simulator		☐	Observer/Producer	
⊞ Viewmanagement SyncCharts Visualizer		✓	Observer	
Data Table		✓	Observer	

Figure 5.2: GUI of the Execution Manager

5.2.2 User Interface

The GUI of KIEM is implemented as an Eclipse View and associated with the KIEM plug-in[1]. If this plug-in is loaded, the Eclipse View of Fig. 5.2 can be made visible and positioned around the editor pane.

Mainly in focus is a list of all loaded DataComponents that extend a special extension point (see Sec. 5.3). These components carry an icon that symbolizes the type of DataComponent as explained above. All components can be scheduled linearly. This is done by changing their position in the list using the up and down arrow buttons. The scheduling affects the interaction with other DataComponents and is explained in Sec. 5.2.4. DataComponents may provide properties that are configurable by the user directly inside the KIEM UI.

[1] `de.cau.cs.kieler.sim.kiem`

The scheduling (i. e., the ordered list of DataComponents) is saved together with their property settings in an `*.execution` file. Because KIEM implements a *saveable View*, this can be achieved by using the normal Eclipse *save* or *save as* functionality while the current focus is inside KIEM.

The GUI provides intuitive buttons to control the stepwise execution of all scheduled and active DataComponents:

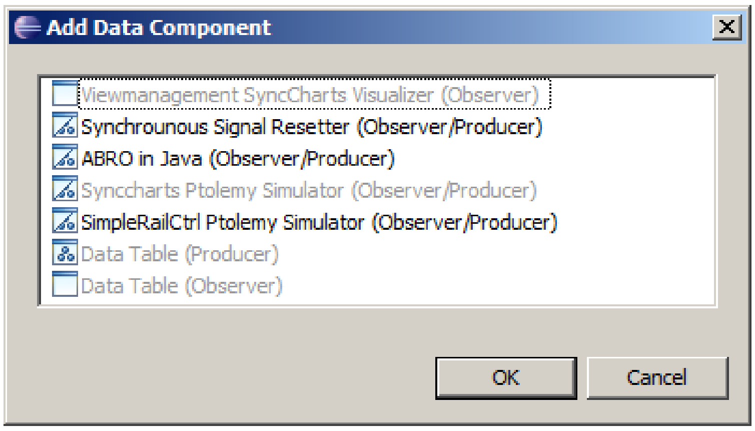
- Make an execution step backwards into the history
- Make a forward execution step
- Run the execution, i. e., make several execution steps with the user defined aimed step duration
- Pause a running execution
- Stop and terminate an execution

These buttons can be found in the upper right area of the KIEM View. Next to the up and down button is the text field that lets the user set an aimed step duration. This is the time an execution step should take. It depends strongly on the time spent by all DataComponents to execute a step. If they are faster than the set desired step duration, the execution (in run mode) stalls up to reaching the aimed duration for the step before executing the next one. The step and the run button come with an optional menu, which lets the user jump to a specific step (lying in the past) or run to a specific step (lying in the future).

The current execution step number is displayed in the text field on the left side of the buttons. If the current tick is a new one, the number is just displayed as is. If it is a so-called *history step* (i. e., a step lying in the past) then its number is surrounded with brackets. For example "`[3]`" is the execution step number three that already was computed before.

DataComponents that are scheduled (i. e., appear in the list) can be marked as active or passive. All active ones are visualized with a black text color having a checked selection box. Passive DataComponents (e. g., the ones grayed out in Fig. 5.2) are not considered when KIEM invokes an execution step for every (active) DataComponent in the list.

DataComponents do not need to appear in the list. They can be deleted from the schedule by pressing the del key. Additionally, there is a dialog (see Fig. 5.3) that allows the user to add DataComponents, whose plug-ins have been loaded. Some special DataComponents may be *multi-instantiable* and can be added multiple times to the schedule list, each *instance* having its own property values.



Viewmanagement SyncCharts Visualizer (Observer)
Synchrounous Signal Resetter (Observer/Producer)
ABRO in Java (Observer/Producer)
Synccharts Ptolemy Simulator (Observer/Producer)
SimpleRailCtrl Ptolemy Simulator (Observer/Producer)
Data Table (Producer)
Data Table (Observer)

Figure 5.3: Dialog for adding DataComponents to Execution Manager schedule

Master Component

A DataComponent may be a *master*. In this case it takes over the role of the user and is able to trigger play, step, pause, and stop functionality or may change the step duration time. A *Master Component* should not change the scheduling, add or remove DataComponents. If a Master Component could delete itself, the expected behavior is unclear. The same is true for adding other Master DataComponents. Behavior of such a Master would depend on the presence and behavior of other DataComponents what should be avoided, apart from normal data interaction during an execution. In a schedule, there should be at most one enabled master present. If this is not the case, KIEM will automatically disable all but one master.

A Master Component may be interesting for a remote cooperation of several KIEM DataComponents. This enables for example special DataComponents to communicate via a remote connection and duplicate user GUI commands. This way a simulation could take place spatially separated from a visualization. In Sec. 5.3.8 an implementation is described that employs this pattern, by visualizing simulation results on a mobile phone that originally are computed in an Eclipse instance running on a computer. For the remote connection, in this example, the Internet and the TCP protocol is utilized.

Proxy Editor

The KIEM GUI is implemented as an Eclipse View, because it is intended to be used together with an open editor (see. Sec 3.1). The disadvantage of this is that saveing and opening of `*.execution` files actually is not seamlessly integrated into the Eclipse Workspace.

To deal with this problem, the KIEM GUI implements the interface `ISaveable-Part2`. That makes KIEM having a *saveable View*. Whenever the KIEM GUI has the focus, using the Eclipse *save* or *save as* actions, triggers KIEM to save the execution scheduling file.

The standard opening of files can only be done by Eclipse Editors. These usually define the extension point `org.eclipse.ui.editors` to define the standard extension, an icon and the class providing the editor. Because KIEM cannot be an Eclipse View and an Eclipse Editor at the same time (see Sec. 3.1), an additional *proxy editor* is used for opening `*.execution` files. In its `init()` method it passes the `IEditorInput` parameter, containing the file to be opened, on to the single KIEM View instance. This will handle all further opening procedure in its `openFile()` method.

5.2.3 Data Pool

The stepwise execution takes the scheduling into account that is given by the order of the DataComponent list (see Sec. 5.2.4). Within one step, all DataComponents marked as active are called to execute a step in this order.

General Interaction

Components may interact by exchanging data in order to communicate with each other. The Execution Manager collects and distributes sets of data from and to each active DataComponent. Therefore it needs some kind of memory as an 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. The data in the pool have no predefined meaning, hence all meaning is given to them from the interacting components.

Dropping and Requesting Data

Fig. 5.4 shows an example setup with (A) one producer-only DataComponent, (B and C) two observer and producer DataComponents, and (D) one observer-only DataComponent (see. Sec. 5.2.1). These are all assumed to be active so they are scheduled by the Execution Manager in the same order in which they appear in the scheduling list, in Fig. 5.4 this is from left to right.

The data pool is organized using revision numbers. DataComponents can drop new data into the pool, in this case they are producer DataComponents that return a non-`null` value. These data get a new revision number in the pool. For example this is the case for the green data package of component A that gets a revision #101.

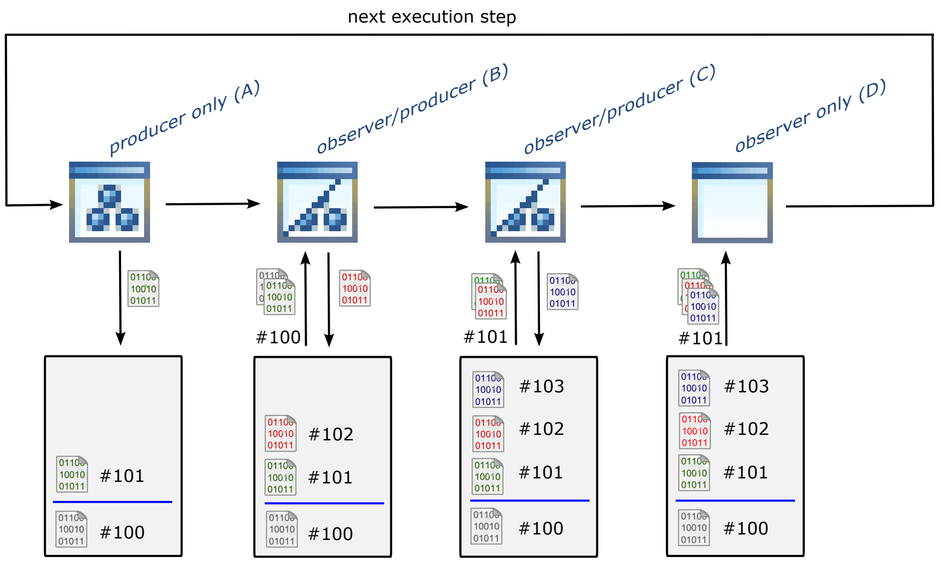


Figure 5.4: Data pool evolvement example for one execution step (schematic)

DeltaObserver

If DataComponents are observers, there are two options:

1. They only want to observe the delta of data values

2. They want to observe the whole data of the pool

In the first case the DataComponent is only interested in data that were added to the data pool since it was last (successfully) called. Therefore the Execution Manager denotes the revision number of the last successful call. These components are called *delta observer*.

In the second case the DataComponent is interested in the whole data of the pool.

Fig. 5.5 shows an execution of 5 steps with three participating DataComponents. One producer generates (different) data at steps 1, 3, 4, and 5. It does not produce any data in step 2. Both observers get skipped in steps 2 and 3 (e. g., because they might be a little slow).

In step 1 both observers get the blue data. Because there is no data in the pool, both inputs (their parameter values) are the same. In steps 2 and 3 both get skipped. Their `step()` method is not called in these steps. In step 4 the DeltaObserver gets only the red and the yellow one, produced in steps 3 and 4 (these are the delta values to the last time it was not skipped, i. e., step 1). The other observer gets the whole data of the pool, namely the blue, the red, and the yellow data. In step 5 the recently produced green data are the only ones that arrive at the DeltaObserver while the other observer gets all ever produced data.

Note that JSON data consist of key/value pairs. Data with equal keys may override each other giving precedence to the most recent produced value. However, the strategy illustrated in Fig. 5.5 remains the same.

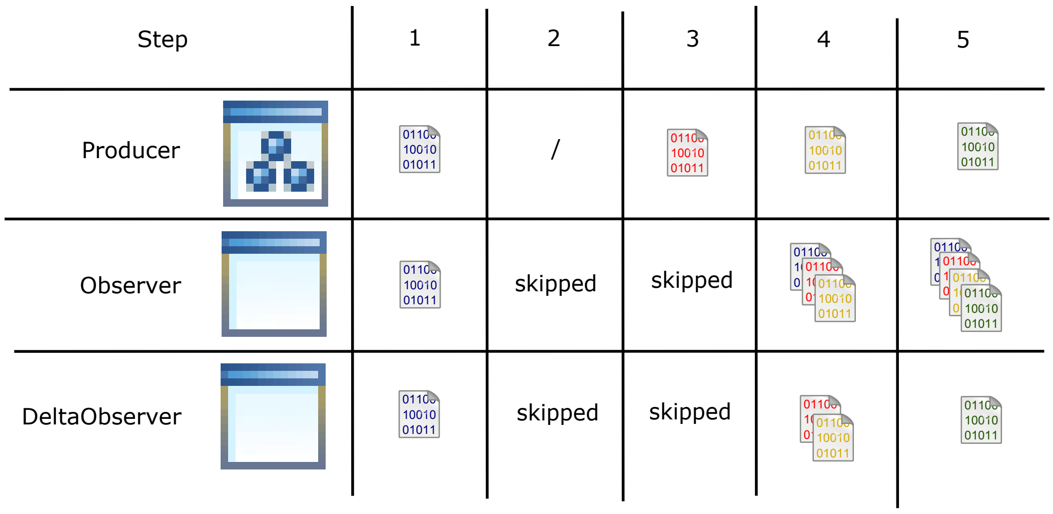


Figure 5.5: DeltaObserver vs. observer DataComponent

Data History

The dash within the data pool in Fig. 5.4 symbolizes the limit up to that (individual) revisions are held in memory. Older revisions get merged together under the newest and then smallest number as it is the case for data of revision #100 (and below) in the example. This limit is also the boundary for the (history) steps backwards during the execution. In the current implementation it is a constant set to a value of `100000`. Hence, the maximum number of steps to go backwards into the history depends on the number of producer DataComponent that insert non-`null` values into the data pool. In case DataComponents want to be executed in past (history) steps, they will always get the full pool data as inputs in such history steps.

5.2.4 Linear Scheduling

All components have in common that they are called by the Execution Manager in a linear order. This can be defined by the user in an *execution setting* and exactly reflects the order of the DataComponent list in the KIEM View shown in Fig. 5.2. 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

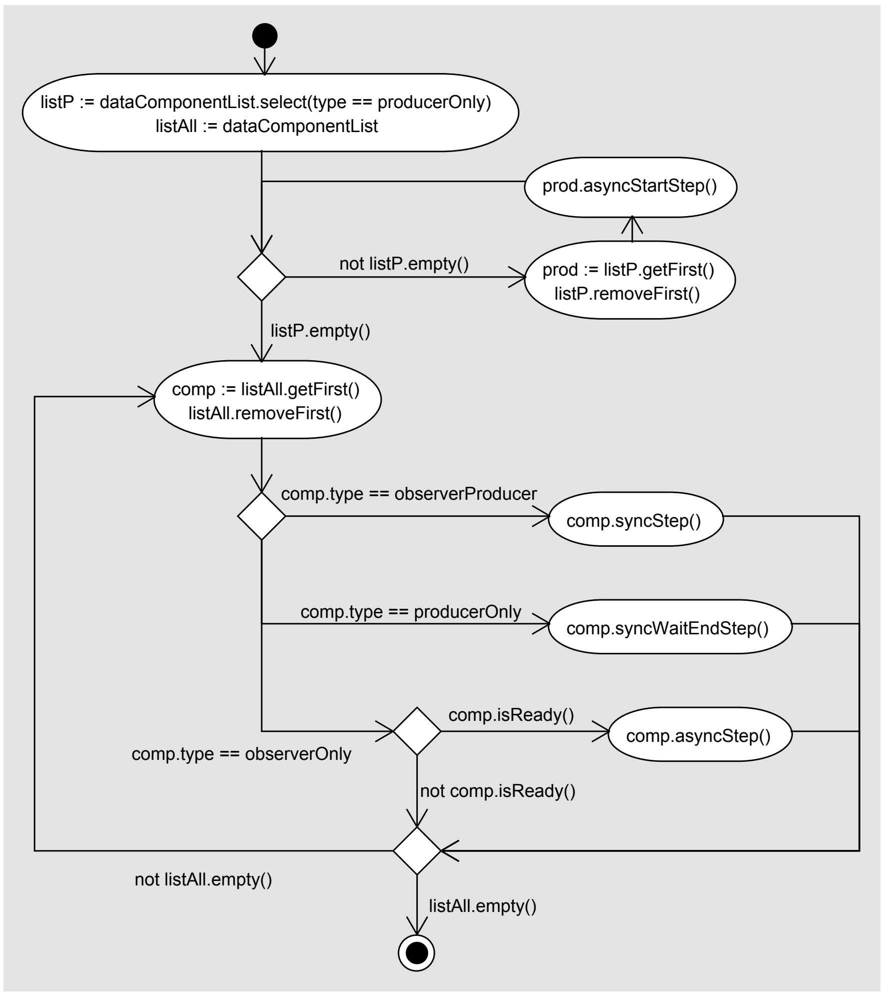


Figure 5.6: Linear scheduled execution of DataComponents

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. In particular, components that cannot run concurrently due to implementation and scheduling restrictions share a common thread. Apart from that, components 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 enforce thread safety. There are also additional scheduling differences between the types of DataComponents listed in Sec. 5.2.1. 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 DataComponent depends on their (nonexistent) output.

The latter leads to the more detailed scheduling scheme for one execution step shown in the activity diagram of Fig. 5.6. This is divided into two main parts:

1. Call all producer-only DataComponents and

2. Execute all DataComponents according to their scheduling order.

The first processes the list `listP` containing DataComponents that only produce data. These do not depend on the (output) data of other components, so their calculation can be started asynchronously at the beginning of an execution step. When this list is empty, all such DataComponents have been processed. Note that all DataComponents that only produce or only observe data run in their own threads. The `asyncStartStep()` method for example will resume a blocked computation of a waiting producer-only DataComponent thread.

In the second part the list of all DataComponents is walked in the order scheduled by the user:

- For a observer and producer DataComponent there does not exist a different thread. The method `syncStep()` runs in the same thread as the execution scheduling itself because it anyways blocks until the DataComponent has finished its computation.

- The computation of a producer-only DataComponent has already been started (s.a.). At this point it must be considered that later scheduled components may depend on the outputs of the current producer DataComponent. This implies that the process needs to block with the call of `syncWaitEndStep()` until the computation has finished.

- Observer-only DataComponents do not produce any data. This implies that other DataComponents cannot depend on them. Hence observer-only DataComponents can be called asynchronously (`asyncStep()`). Because the computation may need some time, an observer-only DataComponent's thread may still process an older invocation of the same method. In this case `isReady()` will return `false` and the component is skipped in this step. Note that skipping of DataComponents can be dangerous and may lead to unexpected and unpredictable behavior in some situations. Hence, the designer of a component should trade of a blocking scheduling scheme against the possibilities and consequences of the described time lacks.

  An example could be an environment visualization component that may spend much time in rendering the graphics. Additionally, this time could vary. Here skipping makes sense, because otherwise all DataComponents are blocked by the rendering process.

If the list afterwards is not walked through completely the process will continue with the next component. After all components have eventually been processed once, the execution step completes.

5.2.5 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 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 in order to notify the user.

Extensibility: The data format chosen in the implementation relies on 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, a mobile phone Java ME[2] application has been developed that can fully interact with the Execution Manager, see also Sec. 5.3.8.

Flexibility: For example 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, DataComponents need to make sure to reset them to be absent for a next synchronous tick. This can be done by introducing a separate DataComponent, which resets all present signals, or in special cases this may be done by the communicating DataComponents itself. Such a specification can be used to built several DataComponents that are able to interact following the synchronous semantics of their external signals within an execution of KIEM. An example of this is given in Sec 5.3.5.

Co-Simulation: Co-operative simulation allows the execution of interacting components run by different simulation tools. For each different simulation tool a

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

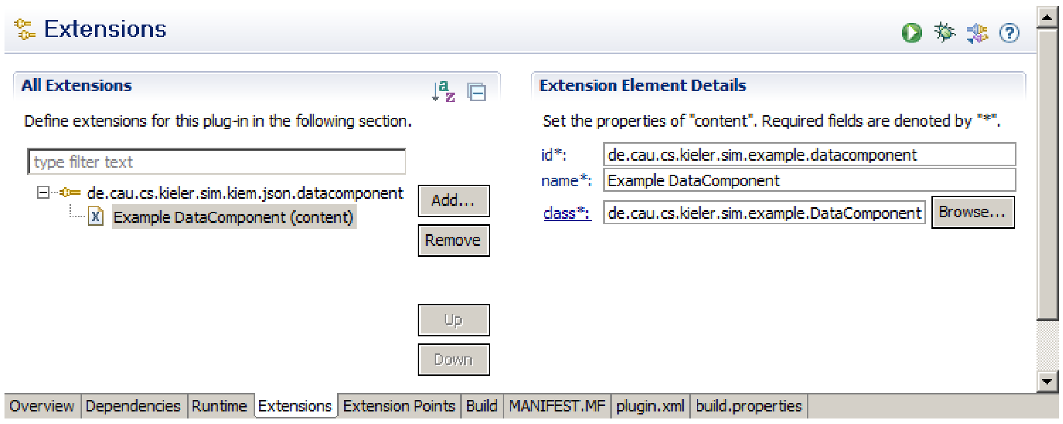
specific interface DataComponent just needs to be defined. This way Matlab/Simulink for example could co-simulate with a state machine model and an online-target debugging interface to get a model- and hardware-in-the-loop setup, which is useful for designing embedded/cyber-physical systems.

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 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 very well during interactive debugging sessions.

5.3 Implementation Details

After a framework overview of KIEM has been given in the previous section, this section presents some implementation details.

In particular, the extension point used to implement a DataComponent is discussed together with the abstract superclass of such components. This is followed by some details about the handling of concurrent threads of KIEM used during the execution. Additionally, optional properties of DataComponents are discussed as well as the data model and a representation for synchronous signals. Finally, three example implementations for DataComponents are given.


Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

de.cau.cs.kieler.sim.kiem.json.datacomponent
Example DataComponent (content)

Extension Element Details

Set the properties of "content". Required fields are denoted by "*".

id*: de.cau.cs.kieler.sim.example.datacomponent
name*: Example DataComponent
class*: de.cau.cs.kieler.sim.example.DataComponent

Figure 5.7: Extension definition for a DataComponent

5.3.1 Extension Point

As introduced in Chap. 3, JSON is used as the concrete data exchange format. To implement a DataComponent, one has to choose from two Eclipse extension points that are offered by KIEM:

1. `de.cau.cs.kieler.sim.kiem.json.datacomponent`
2. `de.cau.cs.kieler.sim.kiem.string.datacomponent`

The first one is based on a special JSON implementation[3] for Java. It uses `JSON-Objects` for the communication with KIEM. The second one uses string representations, conforming to the JSON format, in order to communicate. This extension point for example could be used for an embedded target where the implementation must consider memory limitations. In Fig. 5.7 an example DataComponent's `plugin.xml` that extends KIEM using the first extension point is shown. The name defined here is used by KIEM as a display name for the DataComponent (instance) representation in the scheduling list of the GUI (see Sec. 5.2.2).

Both extension points are based on the same (abstract) super-class `Abstract-DataComponent` and itself implement an interface called `IDataComponent`. To conform to the interface, every DataComponent needs to supply code for the following methods:

`initialize()`: This method is called during the initialization phase. That is, when the execution is about to begin, but has not yet begun. The initialization phase begins for example when the user clicks on the play, step, or pause button in the GUI or such function is triggered from somewhere else (e. g., a Master Component, see. Sec 5.2.2).

In this method a simulation DataComponent may trigger the memory allocation and the loading of the model to simulate.

`wrapup()`: When the execution has ended (normally) this method is called. For instance, the stop button in the GUI was hit or a Master Component triggered the execution stop. The `wrapup()` method is not called, when the execution was abnormally aborted by an error. This is because if an error occurs in this method, this could result in a deadlock.

`step()`: All DataComponents are periodically invoked by the Execution Manager during an execution by calling their `step()` methods. This will be done once for an execution step, as explained in Sec. 5.2.4. In this method a component may react to inputs (parameter) with computed outputs (return value). The parameter is of type JSON as well as the return value, both respecting the implemented extension point.

[3]`http://www.json.org/java`

Listing 5.1: AbstractDataComponent implementation example

```java
public class DataComponent extends JSONObjectDataComponent
                           implements IJSONObjectDataComponent {

   boolean waitI;
   boolean doneI;


   public void initialize() {
      waitI = true;
      doneI = false;
   }

   public boolean isObserver() {return true;}

   public boolean isProducer() {return true;}

   public JSONObject step(JSONObject jSONObject)
                  throws KiemExecutionException {

      JSONObject returnObj = new JSONObject();
      if (waitI && jSONObject.has("I")
         && (JSONSignalValues.isPresent(jSONObject.get("I")))) {
         //take transition from waitI to doneI
         //  when signal I is present
               transition_waitI_doneI();
         //output signal O
         returnObj.accumulate("O", JSONSignalValues.newValue(true));
      }

      return returnObj;
   }

   private void transition_waitI_doneI() {
      waitI = false;
      doneI = true;
   }
}
```

`isObserver()`: If a DataComponent needs inputs, i. e., needs data from another DataComponent (e. g., a user input facility DataComponent as presented in Sec. 5.3.7), it must return `true` in this method. Otherwise it can safely return `false`. In the latter case the parameter will always be `null` but the DataComponent may have extra computation time (see Sec. 5.2.4).

`isProducer()`: If a DataComponent wants to produce data, e. g., in order to communicate with other DataComponents, this method must return `true`. If a component does not produce any data, it may return `false`. Because the latter will have other scheduling consequences this is not always preferred, even if no data are produced. If `false` is returned here, the Execution Manager will ignore any data returned. In this case, `null` should be the first choice for a reasonable return value.

Feature	**Method**	**Description**
communication	`provideFilterKeys()`	Returns a `String[]` array that leads to a more selective data communication.
communication	`isDeltaObserver()`	In case of new steps, only difference data are transmitted to the component if it returns `true` here.
execution	`checkProperties()`	Is used in order to check the property settings.
execution	`isHistoryStep()`	Is used by a component to decide between old and new execution steps.
execution/ scheduling	`notifyEvent()`	This method is called when GUI or execution events occur (e. g., pressed buttons, (de)activating, adding, deleting, schedule changes).
scheduling	`isMultiInstantiable()`	Returns a `boolean` value that indicates whether the component is thread safe.
scheduling	`isHistoryObserver()`	Can be used to signal KIEM that the component can handle (already computed) old data.
parametrization	`getDataComponentId()`	Implements a basic unique identification value computation.
parametrization	`provideProperties()`	Extra properties of type `KiemProperty[]` can be provided as a return value.
parametrization	`provideEvent-OfInterest()`	Returns events of interest (type `KiemEvent`). For these events KIEM will call the `notifyEvent()`.
coordination	`isMaster()`	A component may flag that it implements master functionality.
coordination	`isMasterImplemen-tingGUI()`	A master may also implement the functionality originally provided by the KIEM GUI. This way a button hit results in the master to react.
coordination	`masterCommand()`	These methods can only be called by master components in order to imitate user inputs.

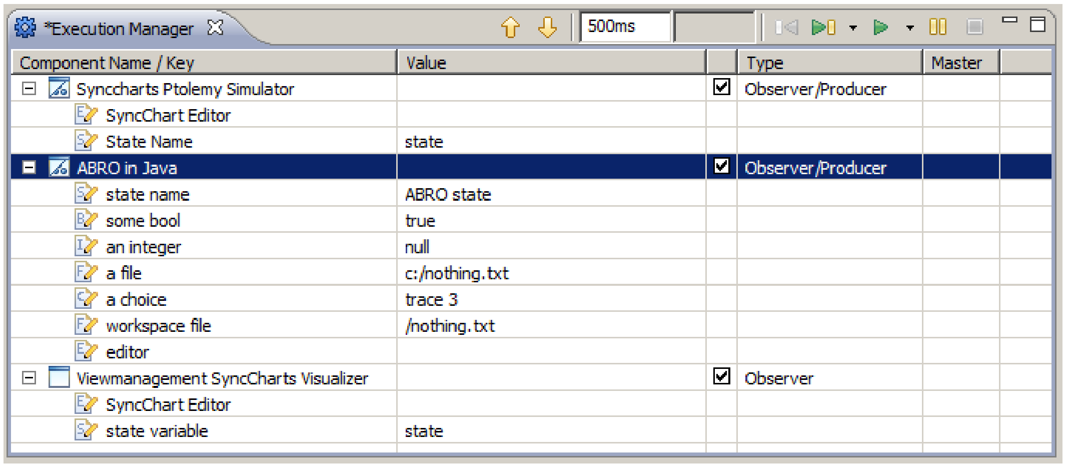
Table 5.1: Main methods of the central abstract class `AbstractDataComponent`

Listing 5.1 shows an example implementation of a DataComponent with a minimal set of methods needed to conform to the interface. In its `initialize()` method of line 8, the initial state is set to be `waitI`. This component claims in line 13 that it wants to observe values. Consequently, it usually can expect a non-`null` parameter in its `step()` method of line 17.

It further claims in line 15 to produce values. Consequently, values returned in the `step()` method will be considered by KIEM. This DataComponent utilizes signal representations as to be discussed in Sec. 5.3.5. In lines 22 and 23 it is tested if a signal `I` is within the observed data and additionally is present. If this is the case and the component is in state `waitI`, the `transition_waitI_doneI()` method executes the transition to state `doneI`. Additionally an output value `O` is produced when taking this transition.

5.3.2 The Abstract Class `AbstractDataComponent`

The abstract class `AbstractDataComponent` plays a central role in the KIEM execution framework. As seen earlier, all DataComponent implementations are implicitly based on it, because the respective extension points are. This class provides basic common features needed for communication, execution, scheduling, parametrization, and coordination. In Table 5.1 most of the methods are listed with a brief description that can be used or overwritten in order to specialize a DataComponent.



Component Name / Key	Value		Type	Master
Synccharts Ptolemy Simulator		☑	Observer/Producer	
SyncChart Editor				
State Name	state			
ABRO in Java		☑	Observer/Producer	
state name	ABRO state			
some bool	true			
an integer	null			
a file	c:/nothing.txt			
a choice	trace 3			
workspace file	/nothing.txt			
editor				
Viewmanagement SyncCharts Visualizer		☑	Observer	
SyncChart Editor				
state variable	state			

Figure 5.8: KIEM DataComponent properties

5.3.3 DataComponent Properties

For the sake of usability, KIEM provides an extra functionality for defining user configurable properties for a DataComponent instance. For every DataComponent added to the scheduling list of KIEM a new property setting is defined.

Fig. 5.8 shows the KIEM GUI with three DataComponents, each having their own properties. In the GUI these can be folded `[-]` and unfolded `[+]`. Values can be edited directly in the table. Properties may be of different types. For example the *ABRO in Java* DataComponent in Fig. 5.8 has a `String` *state name* property. The user may add/edit arbitrary text in the value column. The same is true for the other property example types listed below (boolean, integer, file, choice, workspace file, and editor).

This way the user is able to configure component instances. All property settings are saved together with the scheduling in the `*.execution` files (see Sec. 5.2.2) and restored when such schedulings are loaded into KIEM.

In the default case DataComponents do not have any properties to set. But provision of properties can easily be done by overriding the method `provide-Properties()` of the super class `AbstractDataComponent`. In this method an instance of type `KiemProperty[]` must be constructed. This array contains the properties in the same order as KIEM will show them for each instance of the respected DataComponent.

A `KiemProperty` is again an abstract class and may be implemented to provide flexible property types. For convenience reasons, some default implementations of standard property types come along with KIEM:

- `KiemPropertyTypeString`
- `KiemPropertyTypeBool`
- `KiemPropertyTypeInt`
- `KiemPropertyTypeChoice`
- `KiemPropertyTypeEditor`
- `KiemPropertyTypeFile`
- `KiemPropertyTypeWorkspaceFile`

For each property type instance a name and a default value must be set. The value may later be changed by the user in order to configure the referred DataComponent instance respecting the property type. This is done in the same user interface (UI) as the scheduling of the components itself. An example can be seen in Fig. 5.2 of the KIEM GUI. The editing of values is done within the table using special `CellEditors` appropriate to the type of property.

Examples for the implementation or provision of a `KiemProperty[]` array is given in Listing 5.2. In line 3 the most simple (implicit) string constructor is used in order to generate a property that allows to set and edit a `String` value. In lines 5 and 7 properties for values of the types `boolean` and `int` are created. For the boolean type a drop-down list lets the user choose from the values `true` and `false`. Lines 9 and 16 show a examples about how to specify file types. For the GUI, these

Listing 5.2: Implementing (optional) properties for a DataComponent

```
 1  public KiemProperty[] provideProperties() {
 2     KiemProperty[] properties = new KiemProperty[7];
 3     properties[0] = new KiemProperty("state name",
 4                                     "state");
 5     properties[1] = new KiemProperty("some bool",
 6                                     true);
 7     properties[2] = new KiemProperty("an integer",
 8                                     2);
 9     properties[3] = new KiemProperty("a file",
10                                      new KiemPropertyTypeFile(),
11                                      "c:/nothing.txt");
12     String[] items = {"trace 1","trace 2", "trace 3", "trace 4"};
13     properties[4] = new KiemProperty("a choice",
14                                     new KiemPropertyTypeChoice(items),
15                                     items[2]);
16     properties[5] = new KiemProperty("workspace file",
17                                     new KiemPropertyTypeWorkspaceFile(),
18                                     "/nothing.txt");
19     properties[6] = new KiemProperty("editor",
20                                     new KiemPropertyTypeEditor(),
21                                     "");
22     return properties;
23  }
```

types come with enhanced text fields linked to special dialogs in order to let the user choose a file. How to provide a more generic list of items to choose from is presented in lines 12 and 13. The editor type of line 19, lists all opened editors together with the opened files.

In order to provide new types, the given example implementations listed above can be considered a construction manual. It is basically done by deriving from the abstract class `KiemProperty`. A custom `KiemProperty` needs to implement the `IKiemProperty` interface and hence to provide the following two methods:

- `getValue()`
- `setValue()`

It must be ensured that all property values are the canonical string representatives and hence all property values need to be serializable. The `Object` values of the first two methods depend on the cell editor used by this property type. By default this is the `TextCellEditor` that handles strings. One can override the `provideCellEditor()` method and provide another cell editor here. For example the `ComboBoxCellEditor` operates on integer values. It must be considered that only the string representation is the one that will be stored (in a scheduling file) and should be unique to be distinguishable. Finally, the method `provideIcon()` can be overridden to supply a different than the default icon for the GUI.

5.3.4 Scheduling and Concurrency Handling

The scheduling concept followed by the ideas described in Sec. 5.2.4 is not trivial when it comes to producer-only or observer-only DataComponents. In both cases the flow of control is forked and concurrency handling is needed:

- Producer-only DataComponents can start computation right at the beginning of an execution step.
- Observer-only DataComponents may compute concurrently to other components until they are scheduled in the next execution step.

For both kinds of DataComponents the `Execution` class instance, which can be seen as the central scheduler of KIEM, creates a worker thread in its constructor. Benefits for such worker threads are that there will not be much overhead for creating and disposing threads. This additional time is only needed once at the beginning of an execution.

The class `ObserverExecution` implements the behavior of a worker thread operating on a DataComponent that is a pure observer. The class `ProducerExecution` implements the behavior of a worker thread operating on a DataComponent that is a pure producer. In the following both types of threads and their interaction with the single `Execution` scheduling instance are analyzed.

`ObserverExecution`

In Listing 5.3 this interaction scheme for observer-only DataComponent threads is shown. A central role plays the private `done` variable. For this the following invariant `IV` holds outside the synchronized blocks:

$$\texttt{IV} : \texttt{done} == \text{true} \Leftrightarrow \text{thread is sleeping}$$

It ensures that a synchronized call of `step()` that should awake the worker thread is never missed. Only if the thread is already sleeping (i. e.,`done` is `true`) it is woken up. But because `done` is set to `true` (line 31) in another synchronized block just before the thread goes to sleep in line 34, the invariant is always preserved. More over, when `step()` returns `false`, `done` was not `true` just before the invocation. This implies that the call to `dataComponent.step()` in line 39 had not returned yet, and/or lines 29 to 34 had not been executed as well.

The latter case may occur when the component needs more computation time than one execution step lasts.

Not shown in the simplified version of Listing 5.3 is the fact that the (input) data must be set before the invocation of line 39 and there is a possibility to set the `stop` flag variable. Also errors of the DataComponent-call are being caught and handled as well as a distinction of the correct extension point and respective type castings.

Listing 5.3: Simplified observer-only DataComponent thread

```java
public class ObserverExecution implements Runnable {

    private boolean done;
    private boolean stop;
    private DataComponent dataComponent;
    private JSONObject data;

    public ObserverExecution() {
        this.stop = false;
        this.done = true;
    }

    public synchronized boolean step() {
        // check if we already done
        if (!done) {
            // deadline missed
            return false;
        } else {
            // deadline met
            this.done = false;
            // awake this thread
            this.notify();
            return true;
        }
    }

    public void run() {
        while (!this.stop) {
            synchronized (this) {
                // now we got the result and are done
                this.done = true;
                // go to sleep
                if (done) {
                    this.wait();
                }
            }

            // do asynchronous call - do not use any return value
            dataComponent.step(this.data);

        } // next while not stop
    }

}
```

ProducerExecution

In Listing 5.4 a similar interaction scheme for producer-only DataComponent threads is shown. The done variable also plays a central role for the synchronization. Note that a producer-only DataComponent must be called twice by the scheduler, i. e., the Execution instance.

The computation is started in the beginning of an execution step. Therefore step() is called. It is assumed that done is true and any results of previous execution steps have been reaped. Calling of step() sets the done variable to

Listing 5.4: Simplified producer-only DataComponent thread

```java
public class ProducerExecution implements Runnable {

    private boolean done;
    private boolean stop;
    private DataComponent dataComponent;
    private JSONObject data;

    public ProducerExecution() {
        this.stop = false;
        this.done = false;
    }

    public synchronized void step() {
         done = false;
         // this will awake the execution (blockinWaitUntilDone()) AND the waiting
         // producer thread but only the producer thread will proceed, because
         // done is guaranteed to be false!
         // the execution will fall asleep until the producer has finished doing
         // its step
         this.notifyAll();
    }

    public synchronized void blockingWaitUntilDone() {
         while (!this.done) {
              // we pass the lock to someone else because we are still waiting for
              // done to become true
              this.notifyAll();
              this.wait();
         } // end while
        // at this point done is true and we may reap the results now
    }

    public void run() {
        while (!this.stop) {
            // caller step() must ensure that done == false (before)
            if (!done) {
                 synchronized (this) {
                     // now we got the result and are done so we set done to true
                     // to indicate that the results can be reaped
                     // while done is true we sleep and awake the execution
                     // (blockingWaitUntilDone()) that is possibly waiting
                     this.done = true;
                     // only proceed if done == false
                     // (hence blockingStep() was called)
                     while (this.done) {
                            this.notifyAll();
                            // at this point blockingWaitUntilDone() can return
                            this.wait();
                     }
                     // at this point we know that someone wants us to make a step
                     // and done is false
                 }//end synchronized
                 // do asynchronous call
                 this.data = compJSON.step(null);
            } // end if not done
        } // next while not stop
    }
}
```

`false` and promptly returns.

In the `run()` method, the worker thread waits for this condition (line 48) and is woken up by the `notifyAll` statement of line 20. The worker thread then calls the `DataComponent.step()` method in line 54 and blocks until it returns a (produced) value. During this, the `done` variable is `false` (as set in line 14).

The scheduling protocol enforces the rule that before another invocation of `step()` is allowed to occur, the scheduler must reap the execution results (or wait for them first). Therefore `blockingWaitUntilDone()` is called by the `Execution` instance, whenever this DataComponent is scheduled. If `done` is `true`, the method just returns, and the produced data can be processed.

In the other, more interesting case, the execution thread must still wait for the results (line 28). Note that the constructs in lines 24-29 and lines 45-49 are symmetric. Whenever the call of line 54 returns, the worker may enter the synchronized block eventually. This is because a `blockingWaitUntilDone()`-call may eventually wait and a `step()`-call is not allowed by protocol. Moreover line 42 will eventually set `done` to `true` and before leaving the synchronized block, the worker thread is going to sleep. Only after this, the possibly waiting `Execution` instance is woken up and now sees the `done` == `true` condition.

5.3.5 Synchronous Signals

Using the JSON format as an implementation for the concrete data model, one has to deal with the restrictions that come along with JSON itself and with the ones that follow by the implementation of the communication mechanisms provided by KIEM:

- A `JSONObject` is a list of key/value pairs,
- The key is of type `String`,
- The value is an object that may itself be a `JSONObject` again, and
- All data are represented as a single `JSONObject`.

Synchronous signals normally have a name, a value, and a presence status. The presence status must be consistent for one tick instance. A signal is present within an execution step, if and only if its presence status is set to `true`.

If this is carried into the KIEM execution, a signal may be some object with a `String` name-key in the overall `JSONObject` data list. Analogous to the above, an observer DataComponent's input signal should only be present, iff any producer DataComponent's output signal having the same name, was set to be present in the same execution step, where the producer DataComponent was scheduled before the observer DataComponent. If no such producer DataComponent exists, the signal should be absent (i. e., not be present). But this implication is not reflected directly by the data pool.

In the data pool, if any DataComponent in any execution step ever has produced a value with that name, then this value persists. Hence the presence of a signal cannot be implied by the presence of a data value (representing a signal).

For this reason signal presence or absence has to be made explicit. And because data persist across execution steps, such signals must be set to absent in the beginning of each step. Because this turns out to be a common convention, in order to deal with synchronous signals in a reasonable way, a proposed solution includes a special representation format for signal values and a special DataComponent to reset these.

Representation format

The proposed representation format is implemented in the class `JSONSignal-Values`. Signals should contain not just a value but instead of that a `JSONSignal-Values`. That is, they contain a `JSONObject` as their value, where at least one parameter is called *present* and this parameter is of type `boolean`. It should indicate the presence or absence of the signal. Another parameter is called *value* and contains the normal value of this signal. But this parameter is optional, because pure signals do not have any value. Examples for this representation are given in Table 5.2.

Static helper methods for handling such signals are also included in the `JSON-SignalValues` class. This includes:

- `setPresent()` / `isPresent()`
- `isSignalValue()`
- `getSignalValue()`
- `newValue()`

JSON representation	Description
`a:{present:true, value:10}`	Valued signal a, present, with an integer value of 10
`a:{present:false, value:"hello signal"}`	Valued signal a, absent, with a string value of “hello signal”
`b:{present:true}`	Pure signal b, present
`b:{present:false}`	Pure signal b, absent

Table 5.2: JSON representations of synchronous signals

`setPresent()` is used to set a present status of a `JSONSignalValue` instance. `isPresent()` queries the presence status of a `JSONSignalValue` instance. To check whether a `JSONObject` instance has a `present` key, `isSignalValue()` can be used. `getSignalValue()` returns the *value* key of a `JSONSignalValue` instance. Finally, `newValue()` creates a new pure or valued signal's `JSONSignal-Value` instance.

Listing 5.5: Synchronous signal resetter

```
1  public JSONObject step(final JSONObject allDataIn)
2                                            throws KiemExecutionException {
3      JSONObject allDataOut = new JSONObject();
4       String[] fieldNames = JSONObject.getNames(allDataIn);
5       if (fieldNames != null) {
6           for (int c = 0; c < fieldNames.length; c++) {
7               // extract key, value from JSONObject
8               Object obj = allDataIn.get(fieldNames[c]);
9               String key = fieldNames[c];
10              if (obj instanceof JSONObject) {
11                  // if signal
12                  if (JSONSignalValues.isSignalValue((JSONObject) obj)) {
13                      // if present
14                      if (JSONSignalValues.isPresent((JSONObject) obj)) {
15                          // modify and set absent
16                          JSONSignalValues.setPresent((JSONObject) obj, false);
17                          // add to return JSON value only if
18                          // signal was changed
19                          allDataOut.accumulate(key, (JSONObject) obj);
20                      }
21                  }
22              }
23          }
24     }
25     return allDataOut;
26 }
```

SyncSignalResetter

The synchronous signal resetter is an example plug-in[4] that comes with KIEM. This component must always be scheduled first within an execution step, at least before all relevant DataComponents that take part in synchronous communications.

Its `step()` method can be seen in Listing 5.5. In line 3 a `JSONObject` for the return values is created. Now all input data are processed in an iterative manner (line 6). The name of a datum is extracted in line 8. If this is a signal value and it is present then it is set to absent and accumulated (line 19) in the object to be returned. Otherwise it is either not a signal or a signal that already is absent. Because in the latter two cases, no updates of the data pool are required, this DataComponent not needs to consider these. All updated signals, i.e, signals that were present and now are to be set to absent, are returned in line 25.

5.3.6 Example 1: ABRO in Java

The following most basic example plug-in[5] illustrates the implementation of a DataComponent while using the synchronous signal conventions of the previous section.

[4]`de.cau.cs.kieler.sim.syncsignalreset`
[5]`de.cau.cs.kieler.sim.abro`

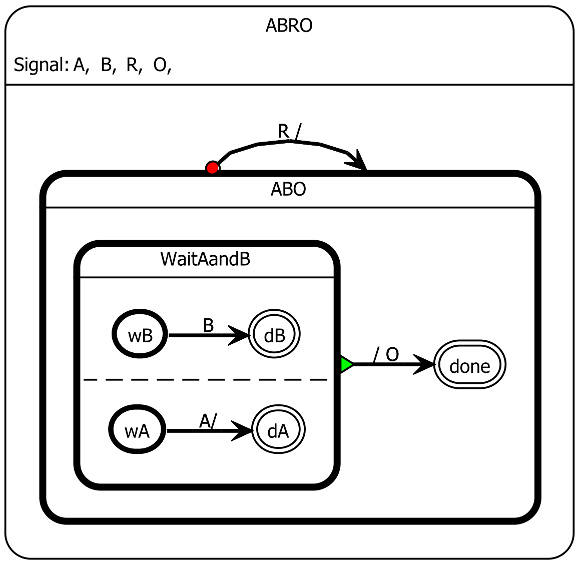


Figure 5.9: ABRO example as a SyncChart

SyncCharts were introduced in Sec. 4.2. Consider the SyncChart in Fig. 5.9. This specifies the famous ABRO example [3], the *hello world* of the synchronous world. It has three input signals `A`, `B`, and `R` and one output signal `O`.

The behavior is as follows:

- The system is concurrently waiting for signal `A` and signal `B` in states `wA` and `wB`.
  - If signal `A` is present and `wA` is the current state, state `dA` is entered.
  - If signal `B` is present and `wB` is the current state, state `dB` is entered.
- Whenever `dA` and `dB` are the current states, immediately `O` is emitted and the state `done` is entered.
- Whenever `R` is present, nothing of the above happens (preemption), but the system is in states `wA` and `wB` afterwards.

This behavior can also be expressed by a DataComponent. Consider Listing 5.6 that implements a simple ABRO simulator assuming all signals are reset using the synchronous signal resetter presented in the section before.

In lines 3 to 7, boolean state variables are declared for all possible states of the system. The `initialize()` method resets the states to the initial ones calling the `resetABO()` method in line 9. The method `wrapup()` must be implemented due to the interface requirements, but is left empty in this case. In lines 11 and 12 the

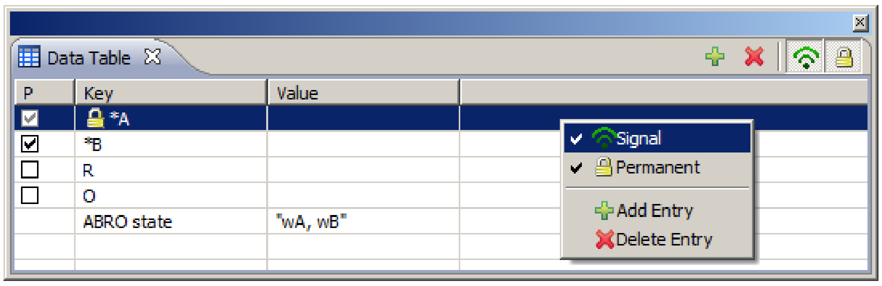
Listing 5.6: ABRO in Java example DataComponent

```
 1 public class DataComponent extends JSONObjectDataComponent
 2                                         implements IJSONObjectDataComponent {
 3     private boolean wA;
 4     private boolean wB;
 5     private boolean dA;
 6     private boolean dB;
 7     private boolean done;
 8 
 9     public void initialize() {resetABO();}
10     public void wrapup() {}
11     public boolean isObserver() {return true;}
12     public boolean isProducer() {return true;}
13 
14     public JSONObject step(final JSONObject jSONObject)
15                                         throws KiemExecutionException {
16         JSONObject returnObj = new JSONObject();
17         if (jSONObject.has("R")
18            && (JSONSignalValues.isPresent(jSONObject.get("R")))) {
19              resetABO();
20         } else {
21             if (wA
22                && jSONObject.has("A")
23                && (JSONSignalValues.isPresent(jSONObject.get("A")))) {
24                  wA = false; dA = true;
25             }
26             if (wB
27                && jSONObject.has("B")
28                && (JSONSignalValues.isPresent(jSONObject.get("B")))) {
29                  wB = false; dB = true;
30             }
31             if (dA && dB) {
32                 dB = false; dA = false;
33                 done = true;
34                 returnObj.accumulate("O", JSONSignalValues.newValue(true));
35             }
36         }
37         return returnObj;
38     }
39 
40     private void resetABO() {
41         wA = true;   wB = true;
42         dA = false;  dB = false;
43         done = false;
44     }
45 }
```

component declares that it is going to observe data (i. e., the input signals A, B, and R) and also produce data (i. e., the output signal O).

The most interesting method `step()` begins in line 14. It takes `jSONObject`, a `JSONObject` instance, as an argument that may contain all input signals. In line 18 it is tested whether signal R is present. Before the `JSONSignalValue` of signal R can be tested it must be checked if there exists such a signal name in the data at all. The latter is done in the same line calling `jSONObject.has("R")` in line 17. The same holds for testing the other input signals A in lines 21 to 23 and B in lines

26 to 28. If `R` is present, the system is reset calling the `resetABO()` once again. In the other case, it may react to `A` or/and `B`. It may do so by changing states from `wA` to `dA` and `wB` to `dB` accordingly. If both state `dA` and `dB` are present at the same time, state `done` is entered and signal `O` is emitted by accumulating it to the returned `JSONObject` instance.


Data Table

P	Key	Value
✓	*A	
✓	*B	
☐	R	
☐	O	
	ABRO state	"wA, wB"

Figure 5.10: DataTable plug-in visualizing ABRO simulation results

5.3.7 Example 2: Data Table

The data table plug-in[6] is a very useful example DataComponent. It is intended to be used to display data and also to manually inject new/changed data values. The GUI of the data table can be seen in Fig. 5.10.

As the global `JSONObject` is a list of key/value pairs, each of such pairs can be displayed in the data table that has a key and a value column. Both are editable cells, meaning that the user is able to edit the key and/or the value directly in the table. A new table entry can be created using the *Add Entry* command (e. g., from the context menu or the tool bar). A key/value pair can be deleted, using the *Delete Entry* command or by pressing the del key.

Because the data table displays and modifies data entries, it takes over the role of an observer and producer. Usually the user edits some values in the data table, then triggers an execution step, and afterwards wants to see the results in the data table. Hence, the data table does not behave like a simulator component (e. g., the ABRO DataComponent seen in Sec. 5.3.6) that computes new outputs using inputs. In this case it is the other way round. Moreover the data table modifications by the user refer to the inputs of DataComponents in the next execution step, while other (not modified) data are the output of DataComponents in the current execution step. To handle this, there are two specialties about the data table:

- The data table is split into two components, a producer part and an observer part. The producer part injects modified or new data and should be scheduled before any other DataComponent that expects to get inputs from the data

[6]`de.cau.cs.kieler.sim.table`

table. The observer part updates the data table with the current values and is reasonably scheduled after any other DataComponent that one wants to observe the outputs from.

- In order to display both in the same table, instead of having two tables, modified or new data are marked in the table (with a "*") while observed data are not marked.

The data table additionally is aware of the synchronous signal conventions introduced in Sec. 5.3.5. If a key/value pair conforms to this convention (i. e., has a boolean *present* parameter) then it is interpreted to be a signal and to have any further (optional) values in the *value* parameter. If the signal is present, the respective check box is checked. If a signal is absent, it is not checked. The signal presence can also be altered by the user and will result in a modification marker as explained above. A key/value pair can be modified to be a signal (and vice versa) using the command *Signal* (context menu).

In addition to this, an entry can be marked to be *permanent*. In this case a lock-icon is displayed next to the entry. The producer part of the data table outputs this entry in every following execution step, as if the entry has been modified each time. For example this can be used to permanently emit signals. Note that a locked entry cannot be edited, except for its presence status for convenience, and that outputs of other DataComponents with the same key may not be visible in the data table.

Synchronization

Because the data table consists of two DataComponents operating on the same table, some additional synchronization is needed. For this, the singleton Eclipse View of the table is used, more specifically, its one and only instance of the table data (`TableData`). The Eclipse View records when the user is currently editing an entry. If this is the case, no updates of the observer part and also the producer part are reflected in the table. If there is no editing ongoing, the entries can safely be updated. The producer part always returns the entries in its `step()` method that are tagged as modified. This tag is then removed.

5.3.8 Example 3: Mobile Data Table

As mentioned in Sec. 5.2.5 and as an example of the extensibility of the KIEM framework, a mobile phone application was developed that serves as a *mobile* data table. This application is split into four different parts:

Mobile application: This is the client program running on the mobile Java ME compatible phone. An Internet TCP connection is needed to connect to the server part. It can be used to modify or delete data, insert new data and to remotely control the execution.

Server DataComponent: This is a DataComponent that is neither a producer nor an observer but takes over the role of a master. It opens a port where the client software is able to connect itself to. It further takes commands from the client (e. g., to make an execution step), and relays them by calling the adequate master methods (see. Sec. 5.2.2). Additionally, it offers its TCP stream for reading and writing to the observer and producer component.

If the master component is enabled, the mobile data table may rule the execution. If the master component is disabled, the execution is controlled by the KIEM GUI and not by the mobile application.

Observer DataComponent: This operates mostly like the data table observer. But instead of using the Eclipse View to display updated data, the TCP stream is utilized to relay this data to the mobile application.

Producer DataComponent: This component takes input from the mobile application using the TCP stream provided by the master (server) component. This is done asynchronously while the user edits the entry. If a step is made, all updated data in the buffer of the producer DataComponent is returned by its `step()` method.

Some screen shots of the running mobile application part are depicted in Fig. 5.11. In the first one the connection settings dialog is displayed. The second image shows the modification of the signal `A` that has no value. Because the mobile data table is visualizing the ABRO simulation example from above, the last two screens show the steps `1` and `2`. In the first step the user sets the signal `A` and `B` to present. This is visualized with a modification tag ("$\star$") before the signal name. The user then triggers an execution step pressing the step button. Once this has completed, the resulting (present) signals `A`, `B` and `O` are displayed.

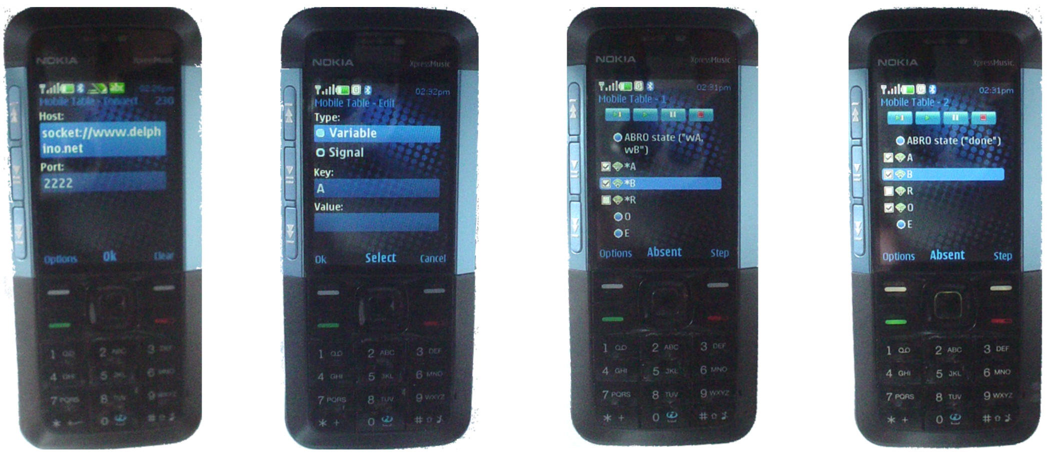


Figure 5.11: Mobile data table application visualizing ABRO simulation results

6 Conclusions

As stated in Sec. 1.2, the problem covered by this diploma thesis is twofold:

1. Develop a generic mechanism, to construct simulation engines for arbitrary DSLs.

2. Offer a generic execution framework to use these simulation engines, embedded into an existing, and MDSD based DSL toolchain that is complete w.r.t. the abstract syntax.

With this segmentation in mind, the following summarizes the results and gives ideas for future work.

6.1 Results

6.1.1 Semantics

As stated in Sec. 1.2, there exists good support for defining the abstract syntax of a DSL. In the context of MDSD, the Eclipse EMF/GMF toolchain allows to generate feasible and customizable editors from this specification.

However semantic aspects that are needed for simulation, have not bean addressed previously in the same manner. This thesis presented an approach for a denotational transformation. Sec. 2.2 gave an overview about how this gap can be filled and reasons about the approach of this work.

In Chap. 4, two case studies to illustrate this approach were presented:

Model Railway Controllers: This first case study showed how to construct a tailored (simple) DSL out of a domain description. It further illustrated how a simulation engine for this DSL can be derived from the abstract syntax and how this compares to code generation.

SyncCharts: The second case study demonstrated how the transformation can be accomplished for an already existing language having lots of non-trivial properties.

Both case studies showed the integration in current MDSD technologies. In particular, common Eclipse technologies, such as EMF, GMF, Xtend, and Xpand, were employed. Additionally, the Ptolemy II tool and the C language were utilized and integrated to execute models in order to simulate them.

As claimed, no new language or notation was introduced. The approach is not bound to any concrete language, because it is conceptually open to other transformation languages. Common technologies were used for the framework and the model transformations. Concerning the target group of domain experts, the operationally driven and not too formal approach seemed to be an appropriate trade-off. To accomplish this, it was shown that the abstraction can be maintained on a high-level using M2M transformations, e. g., concurrency problems can be left for the concurrency-experts by utilizing Ptolemy.

Hence, all aims, namely simplicity, flexibility, usability, extensibility, comprehensibility, and abstraction, have been reached.

Model Railway Controllers

Various problems for the Model Railway Controller simulation implementation showed up when trying to generate C code. Especially concurrency in combination with the state machine pattern and a synchronous pattern with enforced thread synchronization were the main goals reached in this part of the work.

When defining the M2M transformation in Xtend, merging of control data and the Model Railway interface actor were the most difficult tasks to accomplish. Concurrency, synchronization, and a state based representation came for free by employing Ptolemy's strengths.

SyncCharts

Defining the M2M transformation for SyncCharts turned out to be non-trivial in some special parts. Still it was shown how their basic features can be represented:

1. **Hierarchy** is basically supported by Ptolemy's hierarchical structure.

2. **Compound events** are also basically supported, employing Ptolemy's expression language requiring only little adaption.

3. **Parallelism** is one of the strengths of Ptolemy. Little adaption was necessary, such as merging local signals in a loop.

4. **Signal broadcast** was the hardest task, because Ptolemy's modularity contradicts with finding a globally consistent signal assignment. The introduced SyncChart director handles this issue by the presented must-cannot analysis crossing hierarchy levels.

5. **Other** features like the normal termination are solved separately. Preemption is natively supported by Ptolemy's ModalModels.

Additionally it was shown how to bridge some general interfacing problems with Ptolemy (e. g., using the special IO Actor).

6.1.2 Execution Framework

The presented execution framework serves as a light-weight implementation of the IEEE standard (see Sec. 2.2.4).

Arbitrary interacting components are supported and various existing example components were presented. The interface between the framework and such components is kept simple but flexible. Optional, extensible property features were introduced to enhance the usability. Additionally, the user interface and handling is similar to several tools presented in Chap. 2 in order to lower practical barriers.

As further claimed, flexible representations in the data model are possible, the common Eclipse plug-in concept and extension points are used, which are both common standards. The framework behavior like the scheduling or the data management was kept as simple as possible.

Hence, all aims, namely simplicity, flexibility, extensibility, comprehensibility, and uniformity, are fulfilled.

6.2 Future Work

Although all initially set aims have been met as summarized above, there are several ideas for future work that may arise from this diploma thesis.

6.2.1 Specific Model Transformations

Representing time: In the M2M transformation for the Model Railway Controller language, currently "time" advances with each tick. This is not the preferable option. One would favor to represent time as an optional input signal from the environment in a *multiform notion of time* manner.

Optimize code: In the M2T transformation (code generation), one may possibly be able to further optimize the C code by for example carefully reducing synchronization barriers.

Transient states: For the SyncChart transformation, only the `NORMAL` state type is supported yet. It is preferable to advance the transformation and additionally allow immediate transitions or conditional states. In order to do so, one would have to consider the following things:

1. Add an auxiliary parameter to denote such states in Ptolemy.
2. Adapt the SyncCharts director:
    - Fire such ModalModels more than once during a fixed point iteration.
    - Extend the computation of possibly emitted signals to signals of outgoing transitions of *all currently reachable* transient states.

Valued signals: Currently the simulation with KlePto only supports `PURE` signals. Signal values of combined signals are computed in order to prepare an extension that can handle valued signals. For such, one would need to:

1. Make use of (already computed) combined signal values.
2. Save these in Ptolemy parameters, as seen for counting the ticks in a transformed Railway Controller model.
3. Replace `?S` with the parameter name for a signal `S` in all transition labels.

`PRE`-operator: Although one can refer to the current presence status of signals in transition guards, this cannot be done across execution steps. Hence, the `PRE`-operator defined in SyncCharts is currently not supported. It could be interesting to enabled the usage of this by:

1. Using Ptolemy parameters like for valued signals (s.a.).
2. Extending the SyncChart director to save the current value and present status for the next tick.
3. Replace `PRE(S)` and `PRE(?S)` with the parameter name for a signal `S` in all transition labels.
4. Take special care of an important pre-requirement, namely modularization of signal broadcast (s.b.).

Emit input signals, read output signals: The presented must-cannot analysis performed by the SyncChart director considers only local signals. These are signals that are neither input nor output signals. As a consequence, input signals can only appear in transition triggers, and output signals can only appear in transition emissions. There are two different ideas to allow the reversal usage of output signals in transition triggers and input signals in transition emissions:

1. **Extended transition labels:** One could imagine to extend transition labels in a way that for every output signal which is read, an auxiliary local signal is used. This is emitted in all transitions that would also emit the original output label.

   Similarly input signals that are emitted, are replaced by auxiliary local signals that are added to transition triggers where the original input signals are combined in a disjunction.

2. **Parallel regions:** One could also imagine to add parallel regions to the topmost state. One would add an auxiliary output signal with the same name and modify the original output signal to be a local one. In the added parallel region containing just one state, one would emit the original output signal whenever the according retyped local one is present.

   Similarly auxiliary input signals could imply the emission of the according retyped local signals in such additional regions.

With this injective transformation one needs to make sure that the mapping remains correct. This is because the additional regions and states have no representation/meaning in the original SyncChart.

Modularization of broadcast: A major drawback of rising local signals as explained in Sec. 4.2.7 is that this limits the computation of a signal assignment to the current tick (see PRE-operator).

In order to enhance this, the possibly emitted signals that are currently computed in a BFS style call chain, could be extended by a convergecast as explained by Lynch [41, p.60]. For this, hierarchy and parallelism of Ptolemy models must be mapped to a graph structure. This can easily be done by assuming hierarchy as a parent-child relation and parallelism as a sibling relation of *entity nodes*.

The convergecast must be implemented in such a way that it propagates from every node that is responsible for deciding whether a signal must or cannot be emitted. This is basically already prepared, as the SyncChart director carries such a list of signals. Overridden signals must be considered in the transformation, e. g., rename them in a first approach, as it is currently done.

Extending the propagation in the right manner is the most crucial aspect here while also reconsidering this enhancement and formally reason about its correctness seams interesting.

6.2.2 General Model Transformation

Xtend Ptolemy API: Currently the presented model transformations are large collections of Xtend functions, which fulfill various tasks. Often these functions are specific to a given problem.

To enhance this, one could consider to modularize and generalize the Xtend functions and to advance their structure in order to acquire a simpler API for creating Ptolemy models.

Advance abstraction: Because in the current approach one has to build transformations for a specific DSL either from scratch, by reusing functions, or by making use of an API, one could imagine to follow a pattern that better fits to MDSD. This could be a generative approach for (basic) Xtend transformations. For example one could start with one-to-one relations in a first approach and advance this later.

Advance mapping: The mapping is currently implemented only rudimentarily. With a generative approach (s.a) this could also be enhanced by deriving the mapping from a more abstract transformation specification somehow.

Generic simulation components: The simulation components developed in this work do their job for each DSL. Because this seems to have a very high potential of similarities, one could try to combine these in a generic or abstract simulation component that can optionally be refined to retain flexibility.

6.2.3 Execution Framework

Macrosteps: There is just one notion of an *execution step*, which results in all active DataComponents to be called once by KIEM.

One could possibly consider to advance this to a notion of an optional *macro step* where a component may be called several times to compute for example a fixed point.

This may be further generalized according to Ptolemy's *Actor-Oriented-Design* where each DataComponent may be represented by a Ptolemy actor. At least two things might have to be investigated:

1. How to maintain the simplicity of the user interface and the DataComponent interface?
2. What about the efficiency of this approach?

Enhance example components: The presented example DataComponents could potentially be enhanced. For example the data table could be extended to display and allow editing of hierarchical JSON data.

Automate KIEM: Currently the user has to schedule DataComponents (or even load ready-to-use saved schedules) and manually control the execution. Master components are intended to be used also in combination with user interaction.

Possibly one wants to automate the process of executing DataComponents in a script-based or configurable way. Alternatively one may want to control the execution from the command line in order to use operating system scripts. Both ideas could be implemented as additional plug-ins that utilize the application programming interface (API) of KIEM in order to accomplish their tasks.

Advanced configurations: KIEM currently does not have a preference page to save additional settings like DataComponent timeouts. Also execution schedulings might be similar for a common diagram type.

It may improve the usability further to allow the user to customize execution schedulings for specific diagram types. An interface for these kind of settings could be realized as an Eclipse preference page.

6.3 Summary

In this diploma thesis a two-level approach on how to simulate models of a DSL was presented. This approach was integrated into the Eclipse EMF/GMF open source toolchain. Additionally it was presented how Ptolemy II can be exploited as a simulation back-end. The followed denotational transformation based approach leaves much space for flexibility, extension, and enhancements as stated in the section before. With KIEM, an execution framework exists that allows arbitrary components to interact in an execution. This provides an easy to use interface for various simulation, visualization, validation, and communication components.

Altogether, this thesis may serve as a good foundation for future research and development in the context of KIELER and the execution of models.

Bibliography

[1] Jauhar Ali and Jiro Tanaka. Converting Statecharts into Java code. In *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*, Dallas, Texas, June 2000. Society for Design and Process Science (SDPS).

[2] 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`.

[3] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France, April 2003. `http://www.esterel-technologies.com`.

[4] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[5] Charles André. Computing SyncCharts reactions. In *SLAP 2003: Synchronous Languages, Applications and Programming, A Satellite Workshop of ECRST 2003*, volume 88, pages 3 – 19, 2004.

[6] Charles André. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science*, 88:3–19, October 2004.

[7] Greg Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.

[8] Nils Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf`.

[9] Gérard Berry. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999. `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`.

[10] Gérard Berry. *The Esterel v5 Language Primer*, 1999. `ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps`.

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

[12] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the atl model transformation language: Transforming xslt into xquery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[13] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[14] 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.

[15] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison Wesley, 2009.

[16] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey City, NJ, USA, September 2005.

[17] 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.

[18] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[19] 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.

[20] Esterel Technologies. *Esterel Studio User Guide and Reference Manual*, 5.0 edition, May 2003.

[21] Esterel Technologies. *SCADE Technical Manual*, 5.1 edition, February 2006.

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

[23] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Proceedings of the 15th International Monterey Workshop on*

Foundations of Computer Software, Future Trends and Techniques for Development, LNCS, Budapest, September To appear. Also available as Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[24] 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.

[25] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison Wesley, 2009.

[26] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

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

[28] 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.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[30] Stephan Höhrmann. Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2006. `http://rtsys.informatik.uni-kiel.de/%7Ebiblio/downloads/theses/sho-dt.pdf`.

[31] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[32] Cornelis Huizing and Rob Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314. Springer-Verlag, 1992.

[33] 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`.

[34] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[35] Bilung Lee and Edward A. Lee. Hierarchical concurrent finite state machines in ptolemy. In *CSD '98: Proceedings of the 1998 International Conference on Application of Concurrency to System Design*, page 34, Washington, DC, USA, 1998. IEEE Computer Society.

[36] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[37] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[38] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12:231–260, 2003.

[39] Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. In Florence Maraninchi, Marc Pouzet, and Valérie Roy, editors, *International Workshop on Synchronous Languages, Applications and Programming (SLAP'05)*, Edinburgh, April 2005.

[40] Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. *EURASIP Journal on Embedded Systems, Special Issue on Synchronous Paradigms in Embedded Systems*, 2007. `http://www.hindawi.com/getarticle.aspx?doi=10.1155/2007/48979`.

[41] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[42] Mathworks Inc. *MATLAB reference guide*. Natick MA., 1993.

[43] Mathworks Inc. *Stateflow and Stateflow Coder for use with Simulink — User's Guide*. Mathworks Inc., 6 edition, 2004.

[44] Christian Motika. Modellbasierte Umgebungssimulation für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-st.pdf`.

[45] Object Management Group. MOF 2.0 Query/Views/Transformation RFP, April 2004. `http://www.omg.org/docs/ad/02-04-10.pdf`.

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

[47] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003.

[48] André Ohlhoff. Simulating the Behavior of SyncCharts. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aoh-st.pdf`.

[49] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005.

[50] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.

[51] 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.

[52] Matthias Schmeling. An Eclipse-Editor for Safe State Machines. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2009. `http://rtsys.informatik.uni-kiel.de/%7Ebiblio/downloads/theses/schm-st.pdf`.

[53] Klaus Schneider, Jens Brandt, Tobias Schüle, and Thomas Türk. Improving constructiveness in code generators. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, UK, 2005.

[54] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF - Eclipse Modeling Framework*. Addison Wesley, 2009.

[55] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[56] Sven Efftinge. Model2Model transformation with Xtend, 2006. `http://blog.efftinge.de/2006/04/model2model-transformation-with-xtend_15.html`.

[57] Paul Whitaker. The simulation of synchronous reactive systems in Ptolemy II. Master's thesis, EECS Department, University of California, Berkeley, 2001.

Index