Christian-Albrechts-Universität zu Kiel

Diploma Thesis

The KIELER Textual Editing Framework

Özgün Bayramoğlu

January 22, 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

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

The main focus in the KIELER project is the efficient modeling of complex systems. To this end, we would like to be able to make use of the advantages of both textual and graphical editors that work together in a synchronized manner. As an application of this approach a textual editor for *SyncCharts* has been developed in this work using TMF Xtext. To accomplish this objective we have defined a new textual language for *SyncCharts*, namely KITS, developed its grammar and a textual editor for it. As a second step the textual editor has been synchronized with *ThinKCharts*, a graphical *SyncCharts* editor, which was previously developed for KIELER. Both editors are Eclipse plug-ins and thus highly extensible. Furthermore we have followed a model-based development approach and made use of different code generation frameworks. In particular, we utilize TMF Xtext for the generation of the textual development environment that provides its users with convenient editing tools like an outline view, code completion, templates, pretty printing and the like along with an ANTLR like parser. The synchronization has been realized by taking advantage of the command and observer pattern implementations within the GMF.

Dedicated to *Sevinç Apaydın*, my Germany-mother, and *Salih Talha Aksoy*, my life-time hero and granddad—for they have always made me happy, lighted up my way and warmed my soul. "Can the sunset harm the sun?"
Huzur içinde yatın.

It was a long shot, but it was worth the effort. I have learnt so much, got to know to people that have become unsubstituable parts of my life by now, and last but definitely not least, I have had the most fun! I would like to thank ...

Ünver Aksoy for always motivating me and reminding me of what is really important.

Fevziye Aksoy and Faik Bayramoğlu, for not giving up believing in what they have made years ago: Me.

Hauke Fuhrman for leading by example and being a model to learn from—not only about the topic of my theses but also about how to work efficiently and stay cheerful and fun at the same time.

Rita Marnau and *Amelie Kurland* for helping me with the outline of this work.

Burcu Ögüt for being Burcu.

Wolfgang Röttger for actually understanding me.

Matthias Schmeling for being my all-in-one friend: We have laughed about the most mindless things and talked about the most personal things; I have never been able to work with someone as easily as with you, Matthias.

Falk Starke for always having time for my endless questions, being my best friend in the worst times and for being probably the best listener I have seen.

Erk Tatari for his loving support during this work.

Professor Reinhard von Hanxleden for being an open, reachable professor who genuinely cares about what his students do and how they do it, also for providing us with a supportive atmosphere at the institute.

Contents

List of Figures

List of Acronyms

AST	Abstract Syntax Tree
DOM	Document Object Model
DSL	Domain Specific Language
EBNF	Extended Backus-Naur-Form
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
FSM	Finite-State Machine
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
IDE	Integrated Development Environment
I/O	Input/Output
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIT	KIel statechart extension of doT
KITS	KIELER Textual Language for SyncCharts
M2M	Model-To-Model
MDA	Model Driven Architecture
MDSD	Model-Driven Software Development
MOF	Meta-Object Facility
MWE	Modeling Workflow Engine
oAW	Open ArchitectureWare
OMG	Object Management Group
PIM	Platform Independent Model

PSM	Platform Specific Model
SDK	Software Development Kit
SUD	System Under Development
TCS	Textual Concrete Syntax
TMF	Textual Modeling Framework
UI	User Interface
UML	Unified Modeling Language
XMI	XML Meta-data Interchange
XML	Extensible Markup Language

1 Introduction

What you see in Fig. 1.1 are two different representations of the same: The left one is a graphical visualization of the *Mandelbrot set*[1] whereas the one on the right hand side is a textual description of its formula[2].

The graphical view of the set enables us to quickly gain an overview, is less formal and allows an easy orientation while the textual view has its strong point in simplicity and a finer granularity. It depicts how the parameters of the building blocks of the set were configured that were not visible in the graphical view, hence details the internals of the system. Apparent is also that it is easier to load and save, exchange, compare, print, share and edit [20].

Generally speaking, visual descriptions have a topological character contrary to flat textual descriptions that lack spacial organization. It is this topological construction that our minds understand apace, thus we also use graphical representations for debugging simulations for instance. Nonetheless the rapid observations we make with the aid of diagrams also result equally rapidly in the need of editing, configuring, and hence restructuring our models. This is the point where we run up against the drawbacks of a graphical visualization. Consider the amount of time you would need merely to create a node in the middle of a highly complex diagram as an example. How much is the time needed to really add the node and the time needed to finish the subsequent tasks like rearranging related nodes? As a result, one of the most perspicuous criticisms of graphical views is the amount of resources lost in task-irrelevant occupations. In their presentation [38], von Hanxleden and Prochnow present a comparison of editing speeds and traceability of textual versus graphical models coming to the same conclusion: Although they model the same system, different views are cut for different tasks.

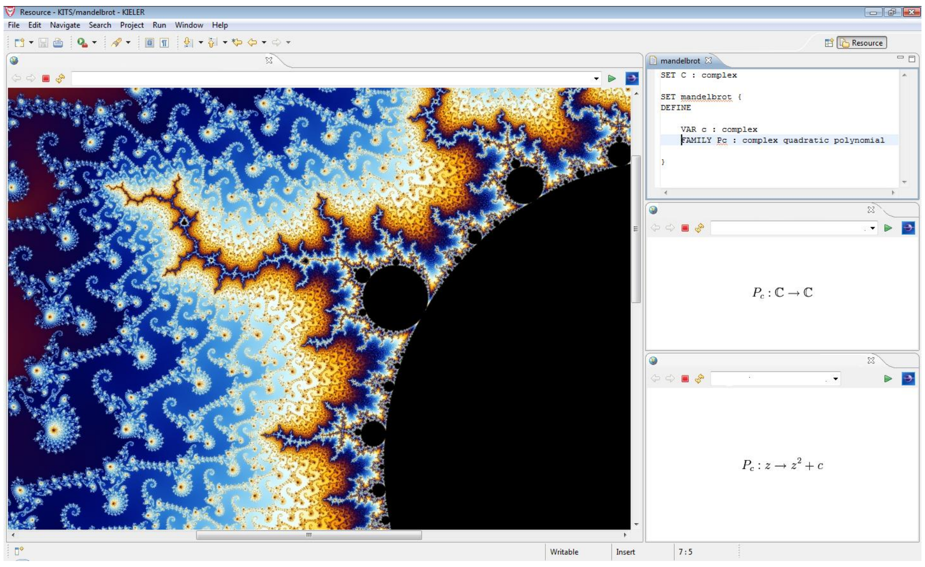
The user has a more observant role when using a graphical editor. Thus graphical editors should be more “active” meaning that they should manage tasks like auto-layout, zooming, simulation and automate editing in a structure-based manner.

In textual editors, on the other hand, the user takes over a more active role, hence textual editors should provide help with appropriate orientation aids like highlighting, overview, textual navigation, pretty printing and validation help such as code completion.

Eventually we think of textual and graphical representations as the two halves of our brain: Just like using both sides of our brain increases our efficiency, using both textual and graphical views will enhance the editing quality. Thus text-based mod-

[1] `http://en.wikipedia.org/wiki/Mandelbrot_set`

[2] Here merely two informal representations of a Mandelbrot set out of many possible are given. We are not claiming for their correctness or completeness.


Resource - KITS/mandelbrot - KIELER
File Edit Navigate Search Project Run Window Help
Resource
mandelbrot
Writable
Insert
7:5

```
SET C : complex

SET mandelbrot {
DEFINE

    VAR c : complex
    FAMILY Pc : complex quadratic polynomial

}
```

$P_c : \mathbb{C} \rightarrow \mathbb{C}$

$P_c : z \rightarrow z^2 + c$

Figure 1.1: Mandelbrot visualization

eling is not merely an alternative to its graphical counterpart, it also complements graphical models perfectly.

A direct implication of these considerations is the synchronization of textual and graphical views. Their integration can be desired in different levels [16]: Text can serve as the initialization block of the graphical model without allowing a textual editing of the system. An alternative and more fine-grained integration is done by views that update their contents according to changes in other views. The highest level of integration is achieved by supplying a hybrid view, e.g. textual components appearing in graphical components, where details can be edited.

The strategies to combine both views can be summed up as follows: Either both editors work directly on the semantic domain model and the respective views are updated with respect to the changes, as illustrated in Fig. 1.2 Or the separation of concrete and abstract syntax is not fully accomplished and we need an intermediate layer between the concrete representation and the abstract model to transform one into the other. This approach is visualized in Fig. 1.3. The first approach is used by the so-called *structure editors*. A textual structure editor projects the abstract syntax tree directly in the text. As a result the user directly edits the tree while entering text [43]. Although there are frameworks that successfully follow this approach, such as *JetBrains MPS*[3], in Eclipse [9] this infrastructure is not yet implemented [36]. To be more concrete, in the graphical modeling framework that we will get acquainted with in the following chapters, diagram-related data are clearly separated from the underlying semantic model. Unfortunately this is not the case for textual editors in Eclipse [36]. As a consequence, our synchronization has to follow the second strategy.

[3]http://www.jetbrains.com/mps/index.html

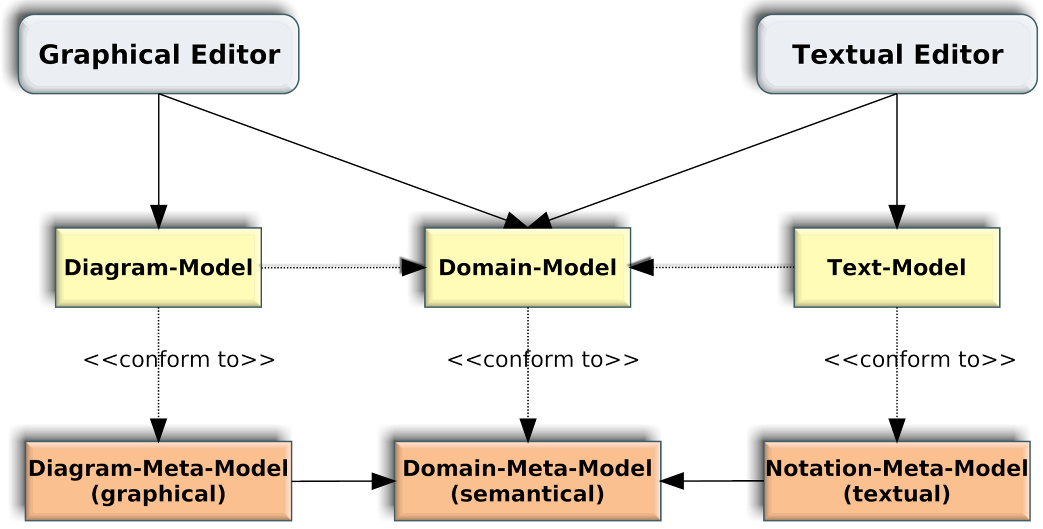


Figure 1.2: Synchronization of structure editors

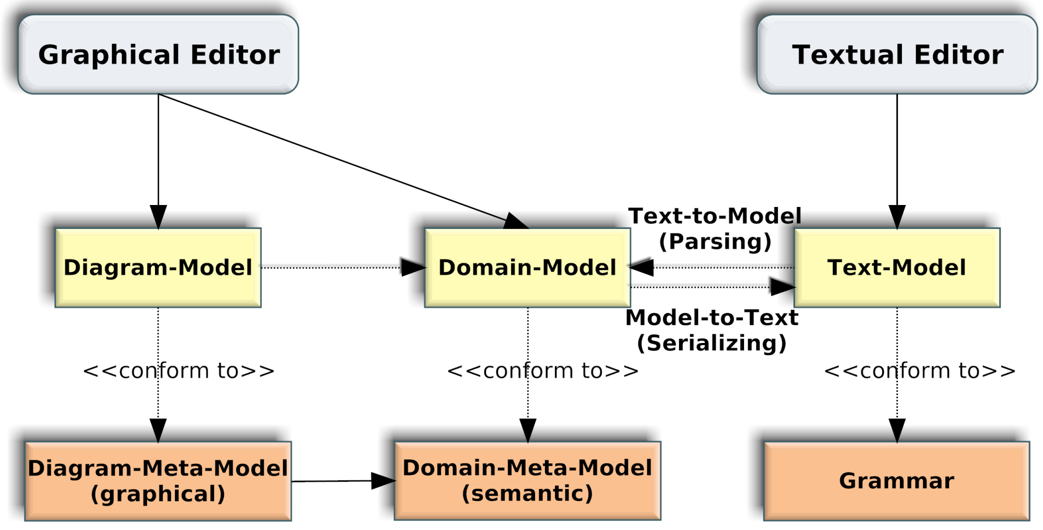


Figure 1.3: Synchronization of a textual editor that separates the text model from the underlying domain model

To conclude this introductory motivation, there is a debate in the literature, led off by a 22 years old paper, *No Silver Bullet* [3], we want to get briefly into. In this paper Brooks analyzes the nature of the difficulties in software development and determines that they are of two different natures: *Essential* difficulties have an abstract characterization and address the nature of software. Hence they remain the same under different representations, which make up the second category "*accidents*". Brooks passionately argues that efforts in improving software representation or tool support are not at all able to result in any serious solutions for essential problems:

> *How much more gain can be expected from the exploding researches into better programming environments? One's instinctive reaction is that the big-payoff problems were the first attacked, and have been solved. Language-specific smart editors are developments not yet widely used in practice, but the most they promise is freedom from syntactic errors and simple semantic errors. ... Surely this work is worthwhile, and surely it will bear some fruit in both productivity and reliability. But by its very nature, the return from now on must be marginal.*
>
> Brooks, *No Silver Bullet*, 1987

He concludes his paper with the result that we should concentrate on how to employ and grow *great designers*. A detailed answer to his widely cited article can be found in [22]. We take position in this discussion as follows: Minds of great designers do not work somehow or in some mystical ways. Research shows that they make more use of both sides of their brains which can be supported and even learnt as some claim. Arguing that investigating in better programming environments is peripheral and the real solution is to hire great designers is like appreciating the mind of a great software designer however not appreciating the efforts that make use of the way these minds work in order to bring out their full potential. Every great designer will most probably increase his or her efficiency and ability to cope with essential difficulties of software, when supported by better programming environments.

Let us conclude this section by commenting a second citation from the same paper:

> ***Graphical programming.*** *A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming. ... Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will. ... Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram. The so-called "desktop metaphor" of today's workstation is instead an "airplane-seat" metaphor. Anyone who has shuffled a lap full of papers while seated between two portly passengers will recognize the difference —– one can see only a very few things at once.*

Brooks, *No Silver Bullet*, 1987

At the end, the pilot of the same airplane is making the plane fly hopefully very well by the aid of a number of even smaller screens, which are also not attacking the very essential nature of the problem of flying an airplane. What we need is not to see the whole picture with the best resolution in every single part of it. Instead we rather need environments that provide us a "control panel" with different views displaying task-relevant parts while supporting task-relevant operations.

1.1 Contribution of this work

This work addresses the problems that arise in the graphical modeling of complex systems by providing an alternative textual editing environment.

Hence the first contribution of this thesis is a concrete syntax for SyncCharts that aims a compact description of the underlying system along with an editing framework for this language that supports its users with content assist, textual navigation, a tree-based outline, semantic validation, syntactical highlighting, templates and a pretty printer. With the introductory motivations in mind, this textual editor has been synchronized with an existing graphical editor in a further step.

In addition, a full synchronization of these editors leads to fundamental conflicts due to their different editing policies. This work solves them by synchronizing contents on file save.

1.2 Outline

This chapter will introduce the frameworks, concepts and languages that are fundamental for our concerns and conclude by defining the concrete problems that build the scope of this study.

In Chap. 4 and Chap. 3 we present our solution approaches to these problems before unfolding details of their implementation in Chap. 6 and in Chap. 5. Finally, an evaluation of our results along with suggestions for future work are given in the concluding Chap. 8.

2 Application Field: SyncCharts in KIELER

Wir backen kleine Brötchen…
dafür aber knackig!

Hauke Fuhrmann

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is the successor research project of the Kiel Integrated Environment for Layout (KIEL)[1] at the Christian-Albrechts-Universität zu Kiel aiming a convenient modeling of complex systems. What we mean by "convenient" has hopefully been sufficiently motivated in the introduction. Let us therefore turn our attention to the systems we want to model. At this point, let us also make a convention and call these systems the "System Under Development (SUD)" throughout this work.

If you would take a moment right now and have a look around you, it is quite possible that you would see some *reactive systems* around. *Reactive* means that the system is able to react to internal and external stimuli, and reactive systems are everywhere from telephones to avionics systems. In fact, the latter ones even played a historical role in the making of *Statecharts* [23].

In 1982 Harel consulted a team developing an avionics system. Surprisingly, he recalls, did he realize that the engineers had fundamental knowledge in the behavior of system parts in particular yet could not compute the next step of their SUD in a deterministic way, when given a set of circumstances. This had its roots in the fully inadequate system specification. Thus he developed *Statecharts* at the end of this project by enhancing Finite-State Machine (FSM)s with visual constructs that were able to express topological relationships. From this point of view, *Statecharts* are to FSMs, what hyper-graphs are to graphs [21].

The first of his visual constructs are dashed lines inside a state that express *orthogonality* meaning that we are in both sides of the dashed lines at the same time (logical AND). The second visual construct is the grouping of states into one superstate, thus abbreviating a subset of the FSM. This construct introduces hierarchy, this time meaning that we have a logical XOR.

As can be surmised from the difficulties the aforementioned avionics team experienced, it has proved hard to convey the behavior of reactive systems [21]. Especially plain automata do not cater for hierarchy, synchronization and concurrency support. *Statecharts* can be seen as a (maybe revolutionary) meta-approach to devise this

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

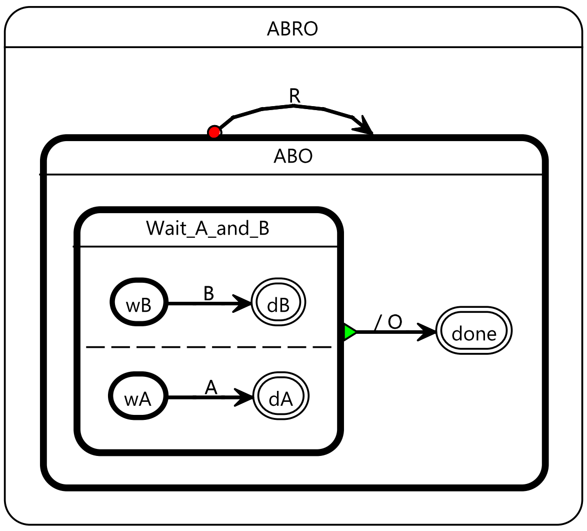


Figure 2.1: ABRO á la KIELER

complex behavior yet it is also fundamental to have a sound semantics in order to be able to compute the next reaction of our SUD in a deterministic manner.

To this end, Esterel[2], a synchronous imperative programming language, has been developed and is powerful enough to support the necessities of *synchronous* reactive systems (synchronicity is explained below). Esterel has the advantages of a textual language, along with its drawbacks. A visual alternative is given by *SyncCharts*. They inherit *Statecharts*' looks and Esterel's semantics. Their behavior can vary from that of *Statecharts*—in particular they do not support inter-level transitions due to their strict containment policy [1]. Throughout this work, *SyncCharts* is the graphical language itself while *syncChart* refers to a particular model visualized in SyncCharts.

KIELER has a graphical SyncCharts editor, namely *ThinKCharts*, that was developed by Schmeling [39]. In Fig. 2.1 a syncChart is represented that is visualized with this editor. Let us refer to this syncChart as ABRO and take a closer look at it to see the main SyncCharts building blocks:

ABRO has three cascaded states, ABRO, ABO and Wait_A_and_B. This is the visualization of the aforementioned hierarchy concept. States containing other states are called *macro-state*s. If we are in a macro-state, we are instantaneously in its initial state. Hence every macro-state has to have exactly one initial state. Consequently, once ABRO is activated, we are instantaneously in Wait_A_and_B.

In Wait_A_and_B we see the aforementioned dashed lines, expressing orthogonality, which means that we split our position at that level in two different branches and are at the same time in both wA and wB because they are the initial states of

[2] http://www-sop.inria.fr/meije/esterel/esterel-eng.html

respective *regions* in Wait_A_and_B.

As can be seen, syncCharts have an almost autonomous behavior due to their reactive nature. Remarkable is also that our syncCharts has changed its *active state(s)* —the state(s) that are reacting— a number of times but we referred to these changes as being *instantly* as if these steps did not take any time.

This is indeed true and has its roots in the time notion in SyncCharts: Synchronous systems are considered to be infinitely fast therefore react instantaneously. Time is simulated as continuously reoccurring discrete events, which are realized by an environment signal called tick. In each tick, our reactive system reacts to stimuli, e.g. by enabling transitions if their premises were satisfied. Thus the configuration of a reactive system is changed in a number of *micro-steps* inside of one tick. However because time only "flows" from tick to tick, these intermediate computations are considered to happen all at the same time, with other words, none of the computations take any time —not very realistic, however deterministic.

A reactive system that conforms to this time notion and fulfills the "synchrony hypothesis" is called a *synchronous* reactive system. The synchronous hypothesis adds the restriction that the value of each signal and variable has to be known before computing the next step from the input. In fact this restriction ensures the deterministic behavior of a reactive system.

The tick signal can be seen as a synchronizer for all computations. The number of micro-steps computations consist of can vary greatly from each other. Therefore the length of a tick is set to the duration of the longest computation. This way, all computations are brought to an end by tick, like a conductor pinching his finger and thumb to cut-off the music.

In our example, if A and B are present while we are in (wA, wB), both their transitions will be taken. We will move to a final state in each region leaving nothing to do in Wait_A_and_B. This is the premise of a *normal termination*, indicated by a green triangle, therefore it will be enabled and let O be emitted immediately.

Hereby O is a *signal*, the means of communication in SyncCharts. A syncChart reacts by sending *pure* or *valued* signals. Pure signals are either *present* or not; valued signals also have a type and hold a value of that type. So far our syncChart represents a system that wants to *write* something (O) yet awaits two "things" to do so (A,B). These could be, e.g. an address and a datum to be written into this address or sharpening a pencil and finding a paper, if we want. In Wait_A_and_B we wait until both have appeared (not necessarily in the same tick). When they have, we perform the writing immediately.

The transition on ABO adds an additional reset functionality to this system. We can send the signal R to our system that enables this particular transition. The red dot at its origin indicates it is a *strong abortion.* Strong abortions are so strong that they preempt any further activity in their source state. This effectively resets our system. Think about the situation that our syncChart reads both R and B while in (dA,wB) : R will *trigger* the strong abortion, the strong abortion will put the system back to (wA,wB), and as a side-effect set the system status to IDLE, meaning that R will not have any effects in this tick anymore.

Although this introduction on SyncCharts is not sufficient for a profound understanding of the language, it should be sufficient for our purposes. So far, we have seen what we want to model. Let us now turn our attention to *how* we will model SyncCharts. In particular, we will follow a model-based development approach, hence fundamental modeling concepts build the base for understanding the considerations in this work —as such they will be introduced in the next section. Thereafter we will proceed with the frameworks we make use of, particularly for the graphical editor in KIELER and for the KIELER textual editing framework.

2.1 Model Based Software Development

It makes a fundamental difference whether we think in terms of *function*s or *object*s when we develop software. Analogously, *model*s taking centre stage in software design requires a completely new way of thinking and is referred to as Model-Driven Software Development (MDSD).

A model describes a system in a modeling language, e.g. in Unified Modeling Language (UML) [33] while its meta-model describes the modeling language. A meta-meta-model adds another abstraction layer on top of this to define the description of modeling languages. Hence every model needs to conform to its meta-model as it is defined and described by it.

Roughly, when we follow a model-based development approach, we start from an abstract model and transform it into more specific models through to executable code.

Model Driven Architecture (MDA) [31] is a specification of this approach, initiated by the Object Management Group (OMG) [32]. First of all, MDA emphasizes the need for a thorough modeling of the software architecture prior to entering the implementation phase [26].

Hence software is first designed and structured in a Platform Independent Model (PIM) that is an abstraction of the given problem from any platform specification. MDA intends to use the PIM to generate executable code. Being an intermediate step between the PIM and the generated code, a so-called Platform Specific Model (PSM) contains platform-specific data yet is not executable.

Since we need to be able to express more abstract artifacts as we increase our abstraction level, we also need different languages for different modeling layers. UML is a modeling language that supports the MDA to describe models and the meta-models they need to conform to. The highest abstraction level can be defined by another OMG language, namely Meta-Object Facility (MOF) [35].

When developing software in a model-based manner we automate the code generation and concentrate on the transformation definitions for our models as well as their semantic validation.

To this end various different MDA-enabled development tools such as code generation frameworks, model transformation or model validation languages have been developed. Let us introduce three such languages that are extensively used by dif-

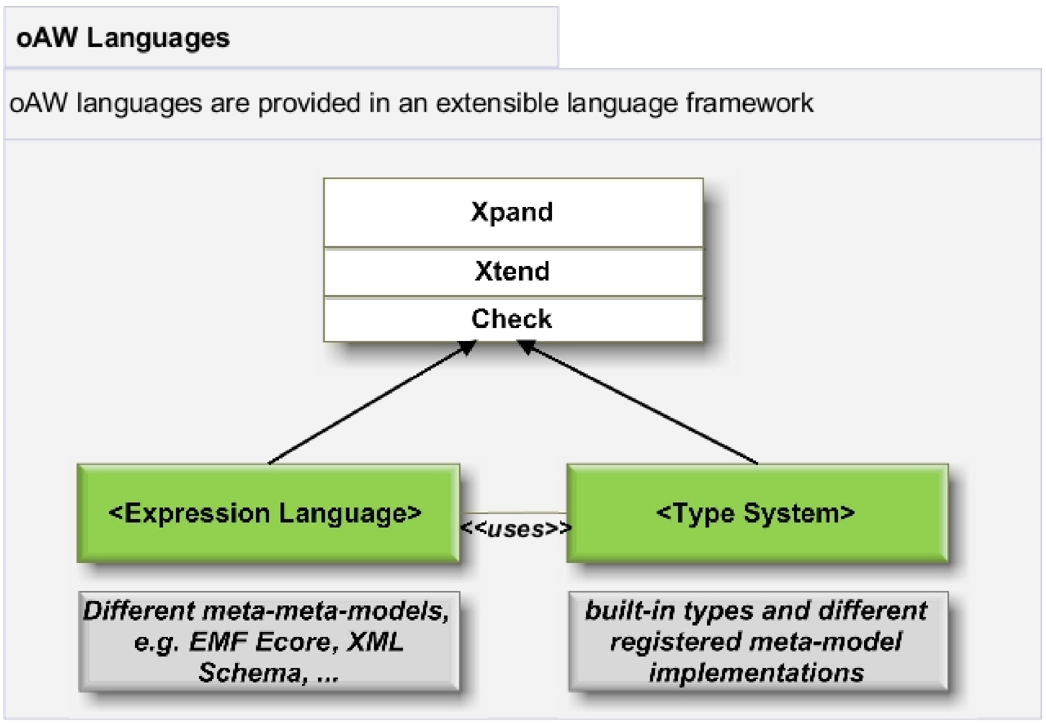

oAW Languages
oAW languages are provided in an extensible language framework
Xpand
Xtend
Check
<Expression Language>
<<uses>>
<Type System>
Different meta-meta-models, e.g. EMF Ecore, XML Schema, ...
built-in types and different registered meta-model implementations

Figure 2.2: Overview of the oAW language framework

ferent Eclipse modeling frameworks.

2.1.1 oAW Languages

Open ArchitectureWare (oAW)[3] has developed three languages that support MDSD, namely *Xtend*, *Xpand* and *Check*, that have been included into the umbrella Eclipse project for modeling[4], which we will introduce in the next section.

Xtend is used to define additional functionality on top of our models. *Xpand* is a template language that is utilized for code generation from our models, while *Check* is designed to validate models.

These three languages share a common base allowing them to operate on the same (meta-)meta-models to have a very similar syntax. [11] This common base consists of a *type system* and an *expression sublanguage*. The type system provides us with some built-in types and implementations of registered meta-models. The expression sublanguage abstracts and uniforms different meta-meta-models like Eclipse Modeling Framework (EMF) ECore, and defines a statically typed expressions language by referencing the type-system. Their hierarchy is visualized in Fig. 2.2.

[3] http://www.openarchitectureware.org/

[4] http://www.openarchitectureware.org/staticpages/index.php/oaw_eclipse_letter_of_intent

Listing 2.1: Validation rules in Check

```
import synccharts;

//ERROR example
context State if (type == StateType::CONDITIONAL) ERROR "Conditional
  states must not contain actions!" :
  (   (entryActions == null) || (entryActions.size == 0))
  && ((innerActions == null)||  (innerActions.size == 0))
  && (( exitActions == null) || ( exitActions.size == 0));

//WARNING example
context Signal if (type != ValueType::PURE) WARNING "KLEPTO:
  Only pure signals are supported in the simulation!" :
 false;
```

Check A meta-model can be seen as a grammar of a language in the sense that it defines the syntactical structure that its models have to conform to. However we also need to ensure that our models are not just syntactically but also semantically correct. *Check* is a language used to this end.

We define our semantic restrictions in the form of *Check* rules. In Listing 2.1, examples of validation rules used in KIELER are presented.

A Check rule can define an ERROR or a WARNING. Each Check rules starts with a *context*. The context is the scope of the particular rule. The context can be limited further via an additional query, as was done in the first rule in Listing 2.1: This Check rule is effective for conditional states only. The context definition and the rule type,i. e. ERROR or WARNING, are followed by a string. This string is the message that the user will see if the subsequent condition does *not* hold. The condition of a Check rule defines what *should* be hence the warning, defined by the second rule in Listing 2.1, is shown for every pure signal.

As can be noticed, the *Check* syntax is intuitive and allows using self-defined types such as *State* because of the aforementioned EMF ECore integration to the type system.

Xtend Xtend is used to define extensions on our meta-model in a non-invasive way. The additionally defined functions can then be used as libraries from other languages are based on the oAW expression framework. We also have the possibility to escape to Java inside our *Xtend* extensions.

Xpand If we have developed a Domain Specific Language (DSL), like we will do in this work, in almost every case we will also want to make something executable from it. The transformation of a DSL to executable code can be done via templates, Listing 2.2: Sample extension with Xtend

```
import syncharts;

List  sortTransitionPrios (State  transitions ):
   transitions . outgoingTransitions .sortBy(e|e. priority );
```

Listing 2.3: Referencing Xtend extensions from an Xpand template

```
import syncharts ;
<<EXTENSION template::helper>>

<<DEFINE state FOR State−>>
  <<this.id>>:
  <<EXPAND callJSONState>>
   <<FOREACH sortTransitionPrios(this) AS outTrans−>>
       <<IF ((Transition)outTrans).isImmediate−>>
           <<EXPAND transition FOR (Transition)outTrans−>>
       <<ENDIF−>>
   <<ENDFOREACH−>>
 <<ENDDEFINE>>
```

written in *Xpand*. Listing 2.2 and Listing 2.3 show how functions, declared in *Xtend*, can be referenced from an *Xpand* template.

2.2 Modeling in Eclipse — Tool Introduction

Eclipse supports MDSD with various frameworks. In this section we introduce the Eclipse modeling frameworks, that are relevant to our work.

2.2.1 Eclipse

Thinking of Eclipse [9] as merely a Software Development Kit (SDK) or a Java Development Environment is like seeing a hair dryer in a power outlet and thinking that this power outlet is a hair dryer provider. Eclipse is rather like a power outlet provider and manager than the outlet itself or the machine plugged to it. Thus the main purpose of the kernel of Eclipse is providing a mechanism that allows tool developers to develop tools that are perfectly integrable with existing ones.

Eclipse is based on units that provide certain functionality, called *plug-ins*. Each plug-in can define *extension points* which are comparable with a power outlet. Furthermore each plug-in can also define *extensions*, meaning that they extend the functionality of an extension point, defined by another plug-in. Again, this is comparable with inserting the hair dryer into the power outlet.

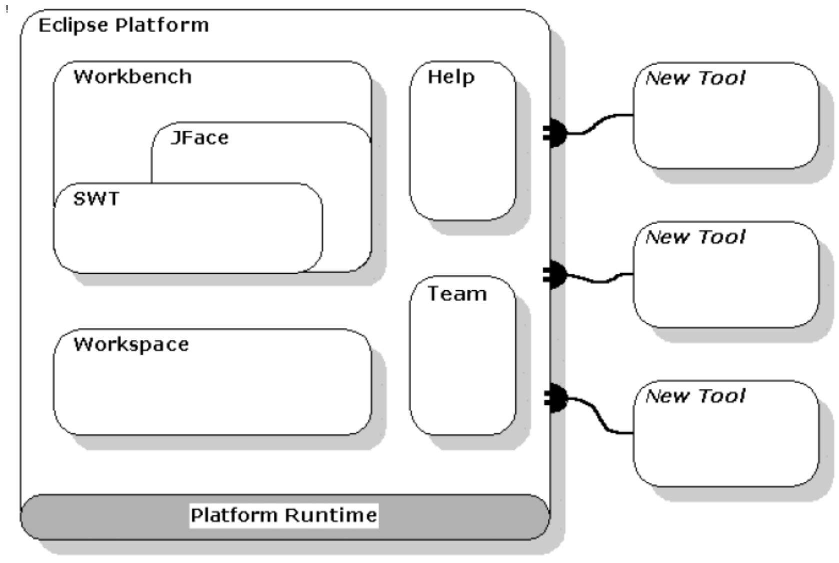


Figure 2.3: Eclipse platform overview

Extension points and extensions are the connection points of plug-ins and the *Eclipse runtime* manages their cooperation by using the *manifest* file of a plug-in that declares the dependencies of a plug-in. This allows the Eclipse runtime to gather all needed information to manage the plug-in interconnection without having to activate them.

Almost everything we see when working with Eclipse is an Eclipse plug-in —the Java development tools, the workbench, even the workspace. An overview of the Eclipse Platform can be seen in Fig. 2.3. Eclipse makes extensive use of the Adapter Pattern [17]. The adaptable workspace *resources* are an example of this. A workspace in Eclipse is simply a container for the files the Eclipse tools are operating on. An Eclipse workspace comes with an adjustable *history* and an adjustable *marker* mechanism. It is also possible to track the changes in workspace resources via *resource deltas* [8] efficiently.

An Eclipse workbench has one or more workbench windows, consisting of *views* and *editors*. To read more about the various terms, and differences between, e.g., views, viewers and editors, please refer to `http://www.eclipse.org/eclipse/faq/eclipse-faq.html`.

As for the deployment with Eclipse, the developer is supported with the *Eclipse Rich Client* platform that merely loads the chosen plug-ins, thus hiding everything else the user is not interested in. This allows us to provide our users with a rich client application that is extensible yet not filled with unneeded features, allowing an easier start-up and optimized use.

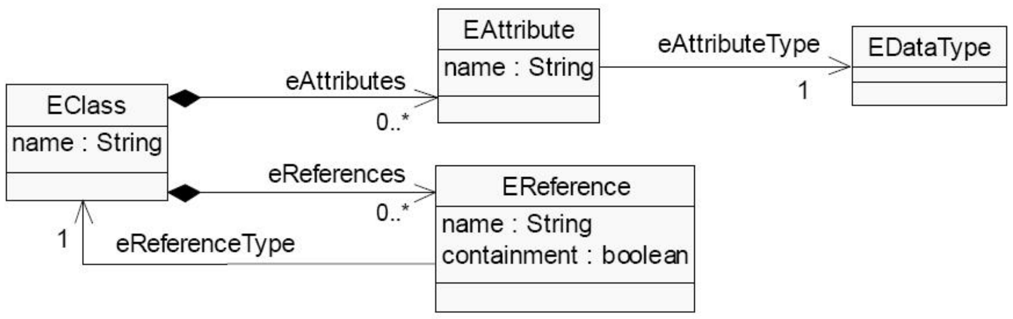


Figure 2.4: A minimal subset of the EMF Ecore meta-model

2.2.2 The Eclipse Modeling Framework

EMF [30] models depict well-structured formalizations of our domain, in our case SyncCharts. Later we are concerned with the concrete graphical or textual notation elements which we "build our sentences" with, however first we have to structure what we want to model. Hence EMF models build the base of our editors, and EMF is a framework that provides us with a model-based approach to construct this base.

However a model that cannot be processed is pointless. The strong point of EMF is its ability to generate Java code from this model. The generated code is organized in two Eclipse plug-ins respectively to instantiate and edit our model as well as an additional plug-in that holds further code for a fully functional editor of our model. The structure of the code, generated by EMF, is designed following patterns [17] that have proved useful for object oriented design.

From the MDSD perspective, an EMF model can be seen as the PIM because it does not contain any platform-specific data and serves as a base for all further development. The so-called *genmodel* that is generated from the EMF model adds platform-specific configuration data to setup the code generation process on top of the PIM [42] thus corresponding to our PSM. From this point of view the generation of a genmodel from an EMF model or the code generation from the genmodel can be seen as a Model-To-Model (M2M) transformation. An overview of how EMF is related to MDA can be found in [10].

The meta-modeling language in EMF is called *Ecore*. It covers a subset of the widely known UML meta-model however concentrates only on its essential parts. In this manner *Ecore* sometimes refers to *essential core* meaning that it builds the core of EMF and contains only essential information making it light-weight in contrast to the UML meta-model. The Ecore meta-model itself is given in form of an EMF (meta−)model to which our EMF models has to conform. However our models do not necessarily have to be written in Ecore at the first place for EMF to understand them. EMF can also synthesize them from annotated Java interfaces, Extensible Markup Language (XML) documents or Rational Rose models. An abridged version of the Ecore meta-model is given in Fig. 2.4. The SyncCharts meta-model, the base of our

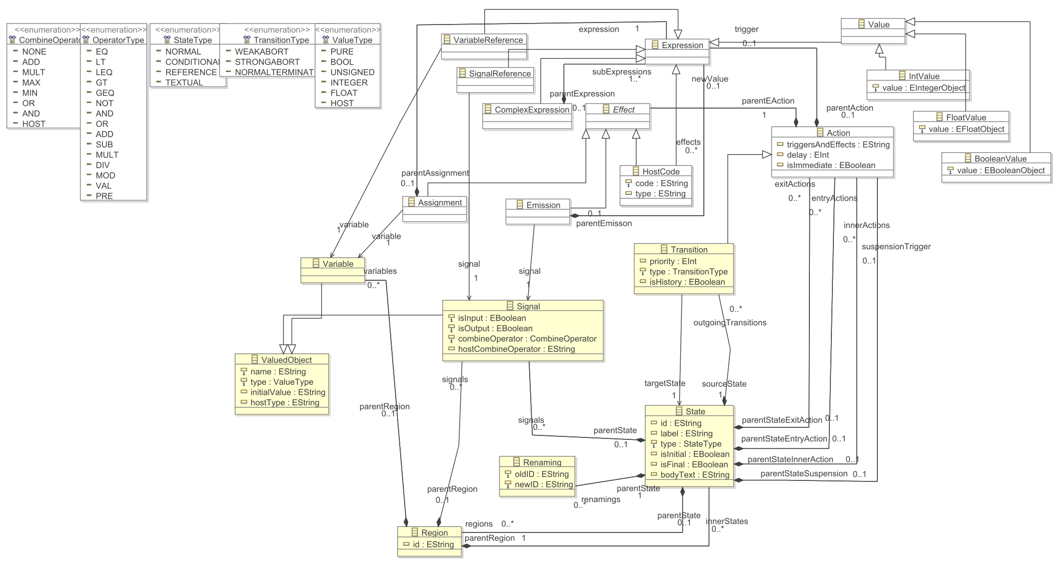


Figure 2.5: KIELER SyncCharts meta-model is conform to Ecore

editors in KIELER, that conforms to this meta-model is presented in Fig. 2.5.

A detailed understanding of EMF implementation details or aspects and patterns used in EMF is beyond the scope of this work. The reader is invited to investigate the official answers of Eclipse to frequently asked questions [2] and referred to the EMF book that was written by its developers, Merks et al. [40]. We will limit us to the concepts which are relevant to our work and which are used for our plug-ins:

The Patterns in Generated Code

When code is generated from an EMF model, for each class in the model, an interface and an implementation of this interface is generated. This pattern, separating interfaces and implementations, has proved helpful for model-based development, e.g., in the Document Object Model (DOM) [40].

Model instantiations using the model code are strongly encouraged to be made via generated singleton factories which reflects another implemented pattern, namely the *Factory* pattern.

Additionally each EClass extends a *Notifier* enabling it to notify its listeners about changes in its state. Thus, EMF also uses the *Observer* pattern [17]. A change observer (or *listener* as often referred to in literature) is called an *Adapter* in EMF for an Adapter not only listens to changes of an object, it also extends, i.e. adapts, the behavior of the object it observes.

Finally all editing tasks, e.g. menu actions, are implemented through the *Command* pattern as can be seen in Fig. 2.6.

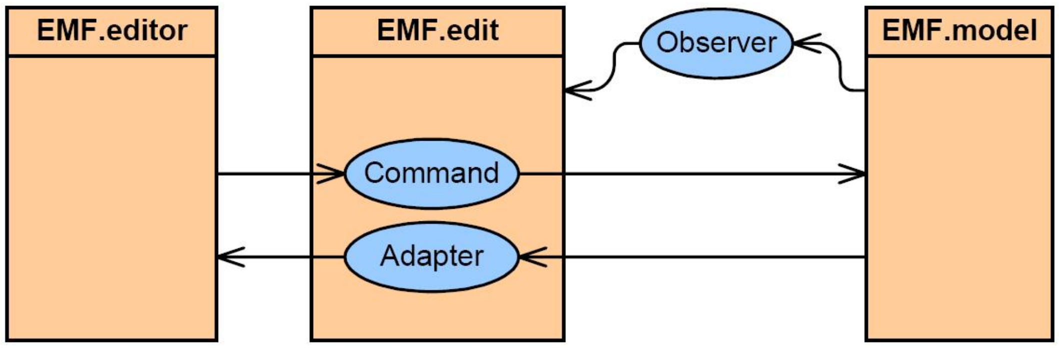


Figure 2.6: Patterns in EMF

The EMF Persistence API and EMF Resources

EMF enables data integration with an efficient and configurable persistence API. EMF models are per default serialized with a generic XML Meta-data Interchange (XMI) serializer yet they can be changed by other parser and serializer implementations as we will do for our Xtext resources later.

In EMF a *Resource* represents a physical storage that holds our models, e.g. a file in the file system. The fundamental difference between the two main associations in the Ecore meta-model, *references* and *aggregations* in EMF, visualizes itself during serialization. An aggregation is a compositional reference, meaning that a class is composite of its aggregated elements, e.g. a book and its pages. Creating a resource and adding a root object to its contents will result in having its aggregated children also persistent in the same resource. However other objects, which are only referenced from the root object, are serialized in separated resources, which is a fundamental difference. Hence aggregation is like being in something and dependent to it. A pregnant woman "aggregates her child" while for instance this thesis only references its author, i.e. its author will not die or vanish just because someone deleted this work. For our concerns the persistence environment of EMF plays a fundamental role, hence they are further detailed in Chap. 5.

To sum up, EMF, given a model as an input, can generate Java code to instantiate our model, persist these instances and edit them with a fully functional editor by just clicking a few buttons. That brings us to the widely discussed question, whether modeling and programming are rapidly becoming two interchangeable terms.

2.2.3 The Eclipse Modeling Project

With especially the latest contributions like the Textual Modeling Framework (TMF), Eclipse is being increasingly used as a DSL toolkit [19]. To this end, different modeling and code-generation frameworks are utilized that are gathered together in the TMF. Hence Eclipse Modeling Project (EMP) does not define a separate framework but serves as an umbrella project for related technologies.

The EMP frameworks that are relevant for our objective are EMF, TMF and Graphical Modeling Framework (GMF), and they are detailed in different sections of this work.

2.3 Technologies and Frameworks behind KIELER Editors

This section details the frameworks that are used to develop the SyncCharts editors in KIELER

2.3.1 The ThinKCharts' Way of Thinking

ThinKCharts was developed using GMF [18]. Thus an introduction in how to develop graphical editors with GMF can be found in [39]. As our aim is to build a textual editor that is to be synchronized with ThinKCharts, we will provide an overview how GMF editors are organized internally to see the big picture. This will allow us later to explain how the cooperation of both editors is organized.

The editing workflow in GMF editors is represented in Fig. 2.7. GMF editors are like orchestras (and we will write the lyrics to their song). On the one hand side we have the abstract model, developed with EMF, that represents SyncCharts in an abstract way. What EMF and EMF models are is detailed in Sec. 2.2.2. Let us think of this model as the notes of the song. On the other hand side we have Draw2D figures like nodes and connections that are what we see later in the canvas of our graphical editor. Hence we will think of them as the performers on the stage. The Graphical Editing Framework (GEF) [24] supplies *EditPart*s that link EMF models (Model) to Draw2D figures (View) and controls the editing mechanism in GMF editors (Controller). This kind of organization in editors, where the abstract model, the view and the editing task are separates in well-defined layers, is thus called the Model-View-Controller design and is a useful pattern. From this point of view the EditParts are the conductors in our orchestra. Yes, we said "conductors" because this is a very strange orchestra where there is exactly one conductor (EditPart) for each note (model element) and each performer (figure). Modifying this bijective relationship can be desired in case we want to assign two different shapes to a model element. Although this is not an easy task it was implemented in [39] for instance: In ThinKCharts the states are able to change their shapes depending on their attributes, e.g., they have thick borders when they are initial.

Let us for a moment imagine that we are sitting among the audience while a GMF orchestra is performing Tchaikovsky's Piano Concerto No.1. All of a sudden we shout out loud: *"Remove the pianist!"*. What then would happen is that the conductor would check if this is a valid *request*. If it is (which I doubt in this case), he would *command* the performer and its note to vanish. In other words, if a user clicks on a button in a GMF editor to remove a node, he or she invokes a request that is first validated by the corresponding EditPart. The EditPart then translates this request into a command and delegates it to the model element and its figure.

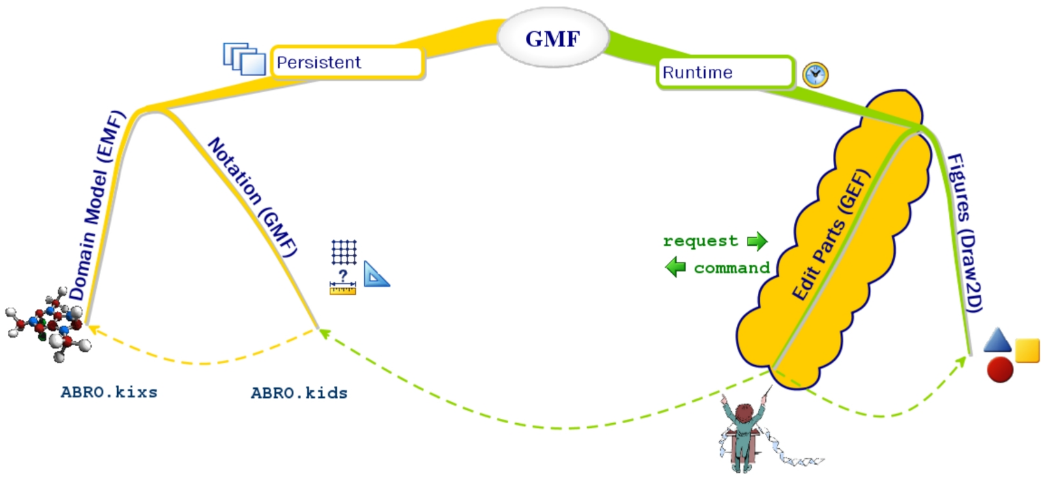


Figure 2.7: Editing workflow in GMF editors

This is where GMF comes into play. In fact, the EditParts do not directly interact with the model elements. The interaction with model elements takes place via the *notation* model. The notation is a persistent model that holds diagram information like the size and position of figures. As a result the graphical information is in a separate model than the semantic information. Due to this separation of graphical and semantic data, we can use the same notation model for different models. This modularity is one important added value of using GMF.

The notation definition does not *contain* semantic data however it references our EMF model. This reference is exactly the point where GMF bridges the two technologies GEF and EMF, and it is visualized by a dashed yellow arrow in Fig. 2.7.

In fact, the motivation behind GMF was that EMF and GEF were widely used frameworks, however it was time-consuming to develop GEF editors "manually". Hence GMF comes with a development environment to enable a model-based development of GEF editors. To do so, GMF introduces three intermediate models that depend on the EMF model: The *graphical definition* model to hold the figures, the *tool definition* model to hold the palette elements in the editor, and finally the *mapping model* to map these two and the EMF model. The latter is again an artifact to increase modularity. Tooling and graphical definitions are likely to be similar in different applications, thus the mapping model is able to map existing tool or graphical definition to different domains hence enabling their re-usability. This is illustrated in Fig. 2.8. A deeper understanding of GMF can be gained from the detailed tutorial in [6].

2.3.2 Choosing a Textual Modeling Framework

The number of frameworks to develop domain-specific textual editors is increasing along with the quality of generated editors. As we are using Eclipse as our development framework we will confine us to the alternatives within Eclipse.

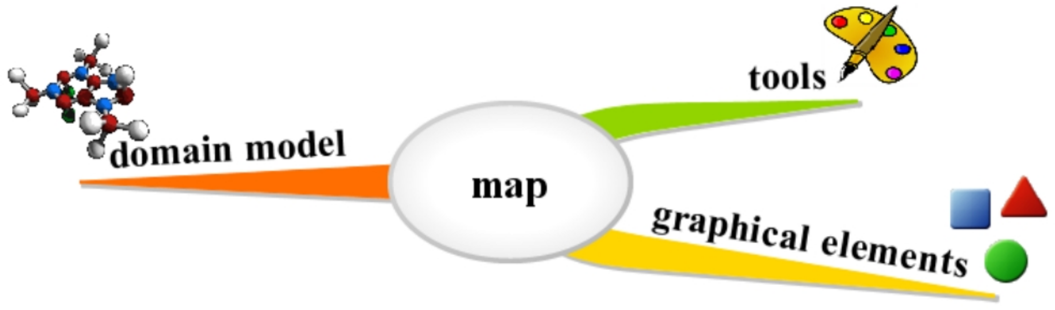


Figure 2.8: GMF map-models increase the modularity in the development process

In Chap. 2 we mentioned the difficulties an avionics team experienced while developing an avionics systems. One of the origins of this problem was the difference between the backgrounds of the experts that had to work on the same system. They lacked a simple, common language that concentrates on the aspects of their SUD. This justifies the introduction of DSLs which are languages that are cut off to a specific domain hence are effective to express domain-specific problems however cannot be used for other aims in contrary to their general purpose counterparts like Java. In this manner SyncCharts can be seen as a graphical DSL. DSLs can be categorized in different ways: Graphical DSLs use graphical notation elements (figures) while textual DSLs have a textual "alphabet". We can use an existing general purpose language (called host language) and narrow it down to a DSL in which case we have an *internal DSL*. An example is *GoogleGuice*[5] with Java as its host language. We will get acquainted with this internal DSL in Sec. 2.3.2. In contrary, external DSLs are defined with a custom syntax and need their own parsers, of which our textual SyncCharts language will be an example of.

At this point we need to distinguish between the so-called *abstract syntax* and the *concrete syntax* of a DSL. The concrete syntax of a language determines the structure of the user input. Models, entered as text or diagrams by the user have to conform to the concrete syntax. The abstract syntax, however, depicts the internal structure of how models are persisted. Models that are persisted by the parser have to conform to the abstract syntax.

The concrete syntax of a language is not necessarily a textual syntax —it can also be graphical. SyncCharts, for instance, has a graphical concrete syntax consisting of circles and arrows. The abstract syntax of SyncCharts in KIELER is modeled in EMF and can be seen in Fig. 2.5.

When we develop textual editing frameworks we can either derive the abstract syntax from the concrete syntax or we can start with an existing abstract syntax and derive our concrete syntax from it. These two different strategies are implemented respectively by the two frameworks Textual Concrete Syntax (TCS) and Xtext within the umbrella project TMF in Eclipse.

[5] `http://code.google.com/p/google-guice/`

Textual Concrete Syntax

TCS annotates meta-models with syntactic elements and synthesizes a grammar file from the annotated meta-model. This approach allows the quick transformation from model to text as well as text to model.

Since however TCS lacks the extensibility and modularity of Xtext we will utilize Xtext in this work. Therefore we will introduce it in greater detail.

TMF Xtext

Xtext takes the concrete syntax as its starting point that is specified in a grammar. This grammar language is a DSL itself and as a proof-of-concept it was developed in itself [7]. The probably most fundamental characteristic of an Xtext grammar is that it combines both abstract and concrete syntax definitions in the same file.

An Xtext grammar roughly consists of a list of *parser rules* and *terminal rules* (also referred to as *lexer rules*). These rules are given in an Extended Backus-Naur-Form (EBNF) [45]. An example of parser rules is seen in 2.9(a).

In an Xtext grammar, the first parser rule is set as the *entry rule*, and depicts the root element of the abstract syntax. For instance, the entry rule in 2.9(a) is the *Region* rule and corresponds to a Region element in KIELER SyncCharts meta-model. This is an example of how the aforementioned combination of domain and concrete model is defined. Let us have a closer look at the derivation of an abstract models from an Xtext grammar.

Parsing and EMF model inference Parsing in Xtext can be divided in two main steps: *lexing* and *parsing*. In the lexing phase a sequence of *token*s are derived from the input text using the terminal rules. Tokens are typed atomic parts of the input, defined by the terminal rules. Keywords, ID, STRING, INT or WS (white-space) are examples. The latter one is called *hidden terminal* in Xtext. They may occur between tokens in any number; the parser skips them when seen. It is possible to define which terminals to hide per parser rule. When the token stream has been derived, the parser goes through the parser rules in the grammar and searches for matching patterns in the token stream. It does so by looking at as many number of tokens as it needs to decide which rule it should try. Once it has chosen a rule, it reads from the token stream and tries to consume tokens until it reaches the end of a rule. If it does not succeed —because for instance the grammar rule prescribed a mandatory keyword that cannot be found in the text at that point —it detects that it has reached a *dead end* and rolls back to its last point to try another rule.

When a sequence of parser rules has been found that consumes the whole token stream, the Xtext parser creates the so-called parse tree consisting of terminals and non-terminals. In Xtext this tree is also called the *node model* because it has been implemented with a tree structure consisting of *composite* and *leaf* nodes. The root of this node model is a so-called NodeAdapter and references its according EObject[6].

[6]An EObject is the EMF class for an Object

Hence our semantic model is connected to the parse tree with the NodeAdapter.

We have said that the nodes in the parse tree reference EObjects. So when have objects been created? This is the characteristic of Xtext: While the parser is creating a parse tree, it also creates objects at some points and connects them to the parse nodes. What we mean by "some points" is explained below. The so created objects are then added to a so-called XtextResource by the parser before they are linked in a further step. According to Efftinge and Völter [12] the separation of linking from the parsing phase enables the implementation of more complicated linking semantics independent from the concrete textual syntax.

Let us now turn to how and when Xtext creates our objects: We mentioned that the parser rules might contain some additional constructs like actions or assignments. These are used to influence the instantiation of our abstract EMF model. A parser rule in Xtext has a name and a return type followed by a colon and semicolon. Between the latter two a number of keywords, assignments or actions can be declared.

Keywords represent non-semantic concrete syntax elements that are not matched by our abstract syntax, namely our EMF meta-model. In 2.9(a) region, init, final and state are keywords.

The semantically relevant parts are the assignments, actions and the return type of a parser rule. Every parser rule has a return type even if not explicitly declared. In that case the parser assumes that the name of the parser rule is to be taken as the return type. Roughly for each parser rule that has at least one assignment in its body an EClass with the corresponding return type will be created. In case we already have a meta-model and we do not want Xtext to create new classes but to instantiate existing ones instead, we have to import this meta-model at the beginning of our grammar. Thereafter we will be able to reference existing classes as return types of our parser rules.

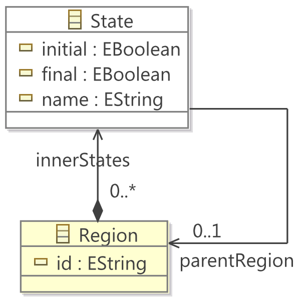
When the parser enters a parser rule it does not immediately create a new object. As a matter of fact it does so only after it has entered the first assignment in a parser rule. This is why parser rules that do not contain any assignments are marked as *abstract parser rule*s and serve as super-classes in our models.

An assignment, as the name predicts, assigns a value, read by the parser, to the corresponding feature of our object. The two different features types in EMF, namely *single-valued* or *multi-valued*, are matched by two different assignment rules in Xtext: The assignment operator "$=$" holds single values whereas the add operator "$+=$" allows multiple values to be written in the same feature. If there is a terminal on the right hand side of an assignment, then we have an EAttribute else the right hand side is the name of another rule in which case we have an EReference to its class. In case we want to cross-reference existing objects we need to put the EReference type on the right hand side of our assignment between brackets. In2.9(a) name is an attribute of State whereas innerStates declares a containment reference from Region to State that can hold an arbitrary number of states but must contain at least one. The non-containment reference parentRegion cross-links an existing Region. The so-called "Region action" makes the parser create a Region object as soon as it enters the Region rule instead of waiting until the first assignment and is given in curly

```

Region :
    {Region}
    ('region')? (id=ID)?
    (innerStates+=State)+
;
State :
     parentRegion=[Region]
    ( initial ?='init')?( final ?='final')? 'state' name=ID
;
```

(a) An example of Xtext parser rules



(b) The model inferred from the parser rules

Figure 2.9: Model inference in Xtext

brackets.

In 2.9(a) we see the EMF model that was generated from this small grammar. We have set the lower bound of the innerStates reference to 1 in our grammar however Xtext has generated an EReference with the lower bound 0. This is a convention to relax the generated meta-model from restrictions as much as possible[7]. Another change the model inference algorithm of Xtext makes is lifting common attributes to superstates. If A and B both extend S and have an attribute name, for instance, then in our generated meta-model, the name attribute would be added as to S.

The structure of the parse model in Xtext is presented in Fig. 2.10.

So far we have seen how the generated parser processes text and instantiates our models. How do we generate this artifact in Xtext? Let us overview the main Xtext infrastructure before concluding this section.

Generation and Configuration in Xtext The Xtext generator is a so-called Modeling Workflow Engine (MWE) file that roughly is a list of *fragments* and is written in a declarative XML-like language.

After the Xtext generator is configured, e.g. by declaring some paths, where code is to be generated in, an arbitrary number of language configurations follow. These again consist of a number of fragments for tasks like loading an ePackage, model

[7]This discussion is still open and can be followed from `https://bugs.eclipse.org/bugs/show_bug.cgi?id=266830`

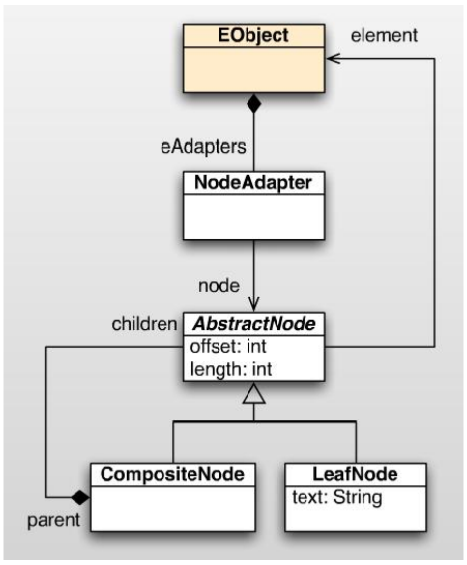


Figure 2.10: Xtext runtime: NodeAdapters bridge parse tree to the AST (from [4])

parsing, validation, code generation and the like. After triggering the workflow, Xtext goes through these steps and provides us with the first (untainted) version of a fully functional editor with content assist, an outline view, pretty printer, templates, syntactical highlighting and source navigation. If there are two language configurations in our workflow then two parsers will be generated and so on. Nevertheless there is still need to modify and change some of the generated parts of the resulting Integrated Development Environment (IDE) in most cases.

Xtext utilizes a dependency injection framework called *Google Guice* to allow an convenient modification of generated code. This framework introduces *binding*s and *module*s to manage the dependency injection in Java.

Google Guice bindings map implementations to interfaces. For instance if we have an interface called IClass and an implementing class ClassImpl, the binding bind(IClass.class).to(ClassImpl) will mark the particular implementation class in order to refer to it whenever an IClass implementation is needed. Everywhere where we need an instance of type IClass, instead of creating an IClass object with new calls or factories, we can tell Google Guice to supply us one. This is done by inserting annotation @inject. GoogleGuice then instantiates the bound implementation class and “injects” it into our code, e.g. in a class field Fig. 2.11. Such implementation classes are called *service*s. Furthermore special Java classes that have a list of bindings are called *module*s. In Xtext almost everything is a service hence replaceable and modifiable. Hence Google Guice makes an very fine-granulated configuration within Xtext manageable and even easy. In Listing 2.4 we see how the Xtext service, responsible for accessing the grammar is bound and injected. IGrammarAccess is the Xtext interface for classes that provide a grammar access. When we generate code an implementation of this interface, called KitsGrammarAccess, is generated along with

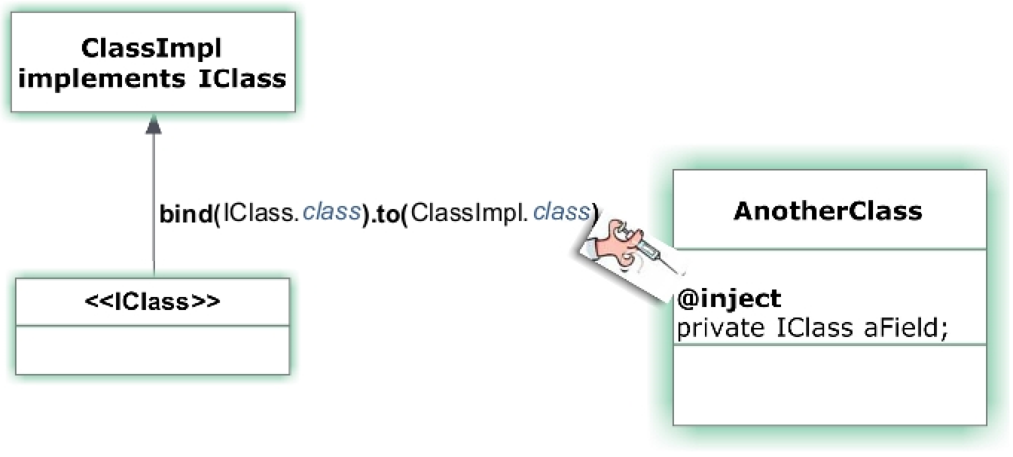

ClassImpl implements IClass
bind(IClass.class).to(ClassImpl.class)
<<IClass>>
AnotherClass
@inject
private IClass aField;

Figure 2.11: A very simplified look at dependency injection with Google Guice

a module that holds a list of bindings. One of them, the bindGrammarAccess, maps *IGrammarAccess* to *KitsGrammarAccess*. Furthermore KitsParser has a private field, annotated with @inject. When a KitsParser is created, Google Guice will instantiate a KitsGrammarAccess and inject it into this field. If we want to change the behavior of grammar accessing in Xtext, we write our own grammar access service, i.e. extend KitsGrammar and change the corresponding binding in our module. We can be sure that at all points where an *IGrammarAccess* was injected our implementation will be used without having to change anything else.

In this work we will utilize TMF Xtext. This has several reasons:

In [15] Fowler suggests that there are two fundamental trade-offs of using external DSLs. The time needed to implement a *translator*, that will parse and make something executable from DSL files is the first one. A possible solution to this is using a parser generator, of which Xtext is one.

The second main drawback is that external DSLs lacks the (semantic) integration into the language environment, utilised. Thus many features that have become widely common — because most of the IDEs support them— are missing and have to be manually implemented if we work with an external DSL. This is the second main advantage of Xtext. By generating not only a parser or serializer but also an IDE it closes this very important gap that, again according to Fowler, was why language oriented programming has not been caught on so much.

Finally just like EMF balances its model-based generative nature with the possibility to modify the generated code on very different levels, with GoogleGuice Xtext also has become a highly configurable framework. The separation of generated code from manually written parts is successfully managed by clearly defining the modification hooks in separate modules for runtime or User Interface (UI) configuration.

Hence the concepts, patterns and technologies used to build the infrastructure of Xtext are likely to increase the use of external DSLs and effectively become the new

Listing 2.4: A service binding in Xtext

```
public abstract class AbstractKitsRuntimeModule
                          extends DefaultRuntimeModule {

    public Class<? extends org.eclipse.xtext.IGrammarAccess>
        bindIGrammarAccess() {
            return de.cau.cs.kieler.synccharts.dsl.
                            services.KitsGrammarAccess.class;
    }
}

public class KitsParser extends AbstractAntlrParser {

    @Inject
    private KitsGrammarAccess grammarAccess;
    ...
}
```

state-of-the-art for employing external DSLs, cut well to the needs of our domains.

2.4 Framing the Scope of this Thesis

In this section we will state the concrete problems that frame the scope of this thesis: What are our expectations from a textual description language for SyncCharts, and what are the problems that arise when we try to synchronize an Xtext editor with a GMF editor?

2.4.1 On the Need for a New SyncCharts Description Language

KIEL, the predecessor of KIELER, supports a textual language for describing Statecharts, namely the KIel statechart extension of doT (KIT) [46]. In this section, this language is introduced and its design is analyzed. The result justifies the need for a new descriptive textual language.

KIT is designed to support Statecharts in general, however this thesis concentrates on SyncCharts which is a dialect of Statecharts. Hence this particular discussion covers describing Statecharts in general while we limit us to SyncCharts in the following chapters.

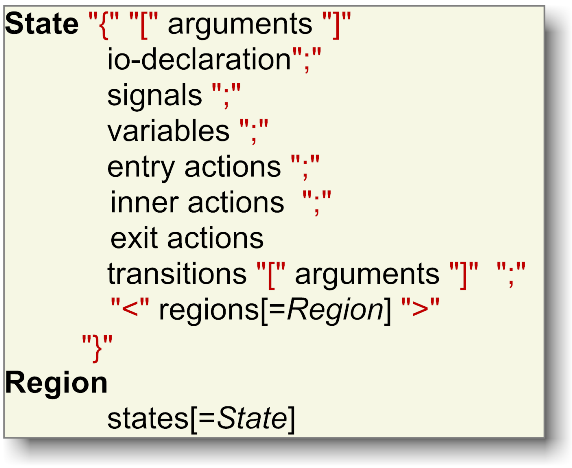
KIT is the first descriptive language used for Statecharts synthesis along with a language called RSML [37, page 25]. The advantage of KIT in relation to RSML is a clear separation of the textual syntax from graphical information. Listing 2.5 represents the ABRO example in KIT.

As can be seen, a statechart in KIT is introduced by the keyword statechart. Next, optional statechart arguments provide some meta-information. Then, the *Input/Output (I/O)-declaration* as well as the state and transition declarations take place between a pair of braces. Hereby, an I/O-declaration is an optional set of input signals, output signals and variables.

States in KIT are declared by simply typing their names, which is convenient. All state arguments follow in a pair of brackets, with preceding tags such as type or label. Marking states as initial can alternatively be done by an initiating "$\rightarrow$".

Listing 2.5: ABRO in KIT

```
statechart abro[model="Esterel Studio";version="5.0"]{
 input A;
 input B;
 input R;
 output O;
 {
  ->ABO;
  ABO{
   Wait_A_and_B{
     ->wB;
     wB->dB[type=sa;label="wB"];
     dB[type=final];
    ||
     ->wA;
     wA->dA[type=sa;label="wA"];
     dA[type=final];
   };
   ->Wait_A_and_B;
   Wait_A_and_B->done[type=nt;label="/ O"];
   done[type=final];
  };
  ABO->ABO[type=sa;label="R"];
 };
};
```



```
State "{" "[" arguments "]"
          io-declaration";"
          signals ";"
          variables ";"
          entry actions ";"
          inner actions  ";"
          exit actions
          transitions "[" arguments "]"  ";"
          "<" regions[=Region] ">"
        "}"
Region
          states[=State]
```

Figure 2.12: The simplified structure of Statecharts in KIT

Analogously transition triggers and effects are surrounded by brackets. The priority of a transition and transition arguments are also given in brackets. The simplified overall structure of a statechart in KIT is given in Fig. 2.12.

Assessment of KITs Language Design

The state done in Listing 2.5 is only referenced once as a target state of a transition. However we need a second assignment to mark it as final. The need for such additional assignments hampers the quick description of a statechart—especially as it grows more complex. Remarkable is also that every assignment ends with a delimiter.

In KIT, arguments are embraced by brackets; state contents by curly braces and regions by square brackets. Having to change between various parenthesis is confusing.

Again, in Listing 2.5, the type and label arguments in wB->dB[type=sa;label="wB"]; follow a state although they describe a transition. In the next line, however, the same state is succeeded by another very similar argument ([type=final]) that this time describes a state. This is not intuitive.

Additionally, all state and transition types could be abbreviated by initiating keyword such as initial, final, conditional or symbols such as "o->".

Finally, transition declarations require a source state thus the user has to reference the same source state multiple times in case there is more than one transition, going out from the state.

Discussion of Existing Textual SyncCharts Languages

These shortcomings motivate us to investigate how the problem of representing Statecharts is addressed by other textual languages. Generally speaking, existing Statecharts textual languages are grouped regarding to their application fields, which are *data persistence*, *data exchange*, *description* and finally the *execution of Statecharts*:

Data Exchange Languages These languages are used to integrate Statecharts in different programs. XMI [34], UXF [41] and SCXML [47] are examples of languages that are used to exchange statecharts.

Data Persistence Languages Different commercial tools offer different serialization formats for persisting statecharts. Most of these formats are proprietary. StateFlow [28] and the scg format of Esterel Studio [13] are examples of persistence languages.

Description Languages Languages that concentrate on describing statecharts aim to ease the development process. We have already seen an instance: KIT is a Statecharts description language that inherits some concepts from other languages such as Dot[8] and ARGOS [27] [37, page 25]. Further examples of description languages are SVM [14] and UMC [29]. Both employ *explicit declarations*, i.e. the declaration of statechart elements may occur at different positions in the text. As a result, the textual structure differs from the topology of the statechart. Furthermore the latter is used for validation of statecharts. Descriptive languages are generally used as an intermediate format that is synthesized from a statechart. According to Prochnow [37, page 25] a direct synthesis of Statecharts from a descriptive language has not been done often. In this sense, KIT and the approach followed in this work to synthesize a syncChart from a descriptive language are novel.

Programming Languages Programming languages such as Esterel aim to execute Statecharts. An Esterel program can be transformed to a syncChart [25] and vice versa hence they are equivalent in expressiveness. Also, a transformation between SyncCharts and C has been developed by von Hanxleden [44].

The evaluation of different aspects of these languages yield to different criteria and questions:

Usability Ideally, an additional IDE support should not be enforced or fundamental when using a language. The syntax should be intuitive enough to enable the employment of plain text editors. However recent developments in language-sensitive editors or structure-editors [43] may change the importance of this criterion.

Text length Various criteria such as the number of words can quantify the amount of needed resources for textually describing a statechart, that can be a relevant factor in different cases.

[8]http://www.graphviz.org/doc/info/lang.html

User Base The user base that is addressed by a language is another fundamental factor that effects the language design. Also whether or not the language is widely spread is relevant for the learning curve, which is the next criterion:

Learning Curve For the evaluation or design of a language, the time needed by an inexperienced user to get acquainted with the language should be considered. This time, referred to as the learning curve, is a fundamental factor.

Type of Information A textual Statecharts description can allow the specification of additional data like graphical notations, layout parameters, modeling tool definitions and version numbers. Whether or not such meta-information is enforced and clearly separated from the base language is another important point.

Structural Similarity We have seen that declarations of statechart elements can occur in different positions in the text. The alternative approach is using *implicit declarations* which results in a textual structure that is reminiscent of the topology of a statechart. Explicit declarations can prove useful for complex statecharts while implicit declarations are more intuitive. Generally, description Statechart languages should have a clear structure allowing a quick perception of the overall organization. However not every language is designed to be read by human:

Naturally the criteria are not equally weighted for every application field. For instance structural similarity and usability are not the most fundamental criteria for persistence languages.

An overview of existing textual Statecharts languages, categorized after the application fields that were listed above, is given in Fig. 2.13.

The presented considerations yield to the need of a textual SyncCharts description language that avoids the drawbacks of KIT. However the attempt to design a new concrete syntax invokes various difficulties such as the introduction of abbreviating symbols without overloading them or supporting states without explicitly declared identifiers.

Establishing conclusive decisions regarding these questions is not easy. Although it seems like there is a lot maneuvering room and freedom, the problem of designing a language according to the aforementioned criteria brings its own restrictions along. The discussions during the design process[9] were made by reference to several (informal) syntax proposals, some of which can be found in App. A.

In Chap. 3 we will present our solutions to the questions above while the implementation of the parser and the IDE is detailed in Chap. 5.

We will go on with the main problems in the synchronization of GMF and Xtext editors in the following section.

[9]http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Meetings

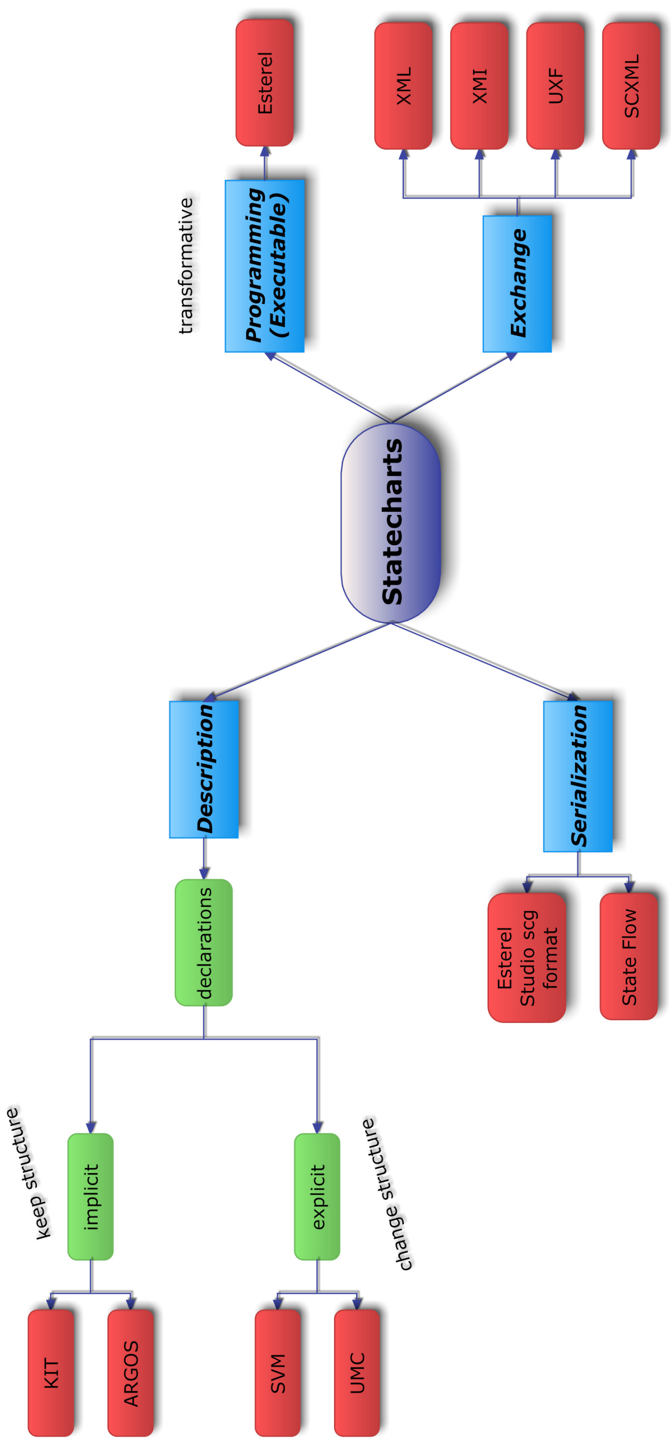


Figure 2.13: An overview of the existing textual SyncCharts languages

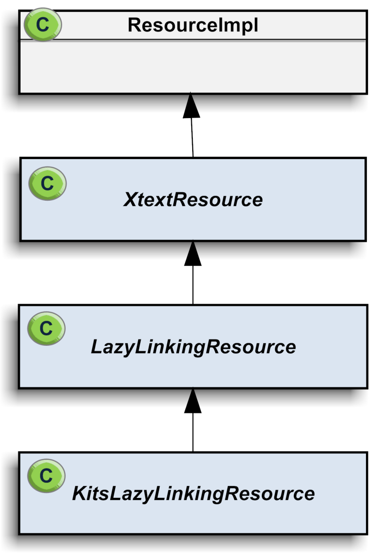


Figure 2.14: Type hierarchy of an Xtext Resource

2.4.2 On the Synchronization of Xtext and GMF Editors

We can recall from Sec. 2.3.1 and Fig. 2.7 that GMF editors need two persistent models, namely a GMF *notation model* and an EMF *semantic model*. The model representation in a GMF editor is done at runtime using merely these two persistent models. From the point of view of GMF, how the EMF model was serialized or synthesized is a black-box. Hence if an Xtext resource can be stored as an EMF resource, then it can serve as a serialization back-end for a GMF model.

The good news is that every Xtext resource indeed is an EMF resource. This is visible in the type hierarchy of an XtextResource, and can be seen in Fig. 2.14. Here, ResourceImpl is the superclass of EMF resources.

Effectively, not only GMF, but also any EMF-based framework can use Xtext to serialize their models. The EMF integration of Xtext is pictured in Fig. 2.15. Xtext components such as the linker, parser or the serializer use the *XtextResource* to create an EMF model from its textual content. For their concerns, the upper half of the figure is a black-box. The *XtextResource* is then serialized in a domain-specific-language of choice.

Furthermore the XtextResource can be processed by other frameworks, e.g. to generate code—or synthesize a diagram.

As an implication of these considerations, the GMF editor should be able to use the domain model instance that has been parsed by the Xtext parser as its "input." According to this strategy, which is visualized in Fig. 2.16, the Xtext parser will instantiate the EMF model when the user enters text. Since they share the same domain model instance, GMF is aware of these changes and adapts its contents accordingly. Adapting of the textual content to changes in diagram follows the same

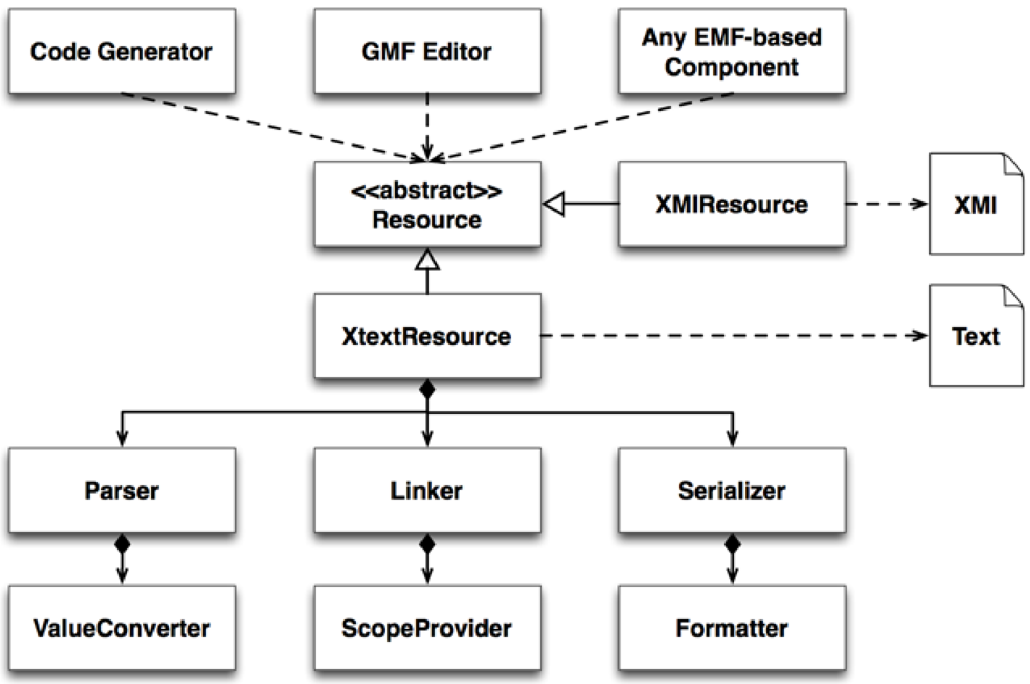


Figure 2.15: EMF integration of Xtext (from [7])

path. This approach requires one model instance at runtime that is then read and written by both editors. It is a common way to combine pure EMF editors with GMF editors for a full synchronization, as detailed in [5]. Unfortunately, it quickly yields to conflicts when applied to an Xtext and GMF editor.

The origin of conflicts in this approach is that GMF editors have a *canonical edit policy*. The canonical edit policy means that every figure that does not have a semantic element is deleted from the diagram. Recall that figures are saved in notation model, and semantic elements are saved in the domain model. Effectively, this policy means that the notation model is synchronized with the underlying domain model at all times.

On the other hand side, Xtext has a *partial parsing strategy*. The partial parsing policy means that, after a change in the text, Xtext calculates the affected subtree of the parse model, removes the nodes in this field, and parses the subtree again. Effectively, new model elements are created.

So sharing a model instance between Xtext and GMF results in many invalid intermediate models that are visible in the GMF editor due to its canonical edit policy. The arising conflict is presented in Fig. 2.17:

For instance if we make changes in our GMF editor, the corresponding part of the EMF tree adapts to changes and notifies Xtext about the modification 2.17(a). Xtext, due to its partial parsing policy, calculates the “effected” area in the model,

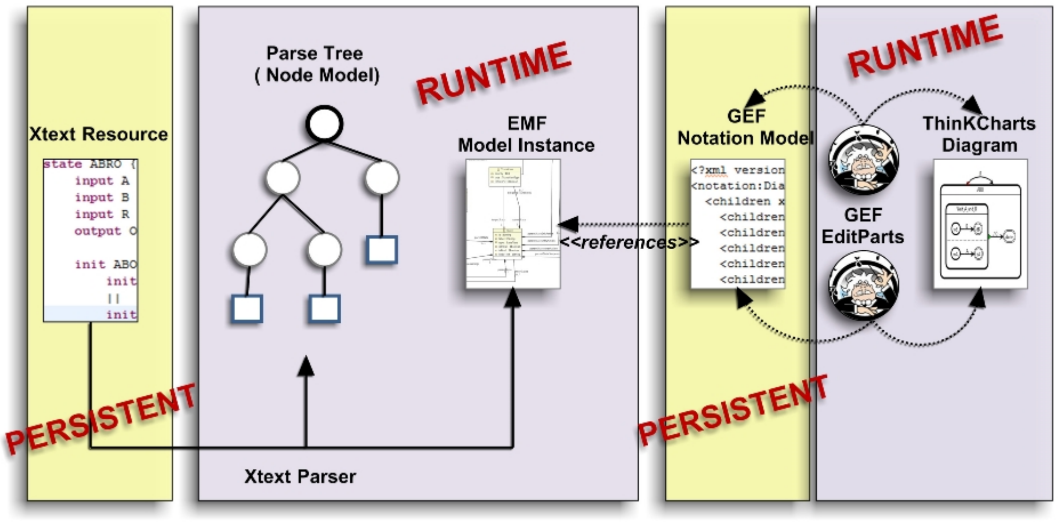
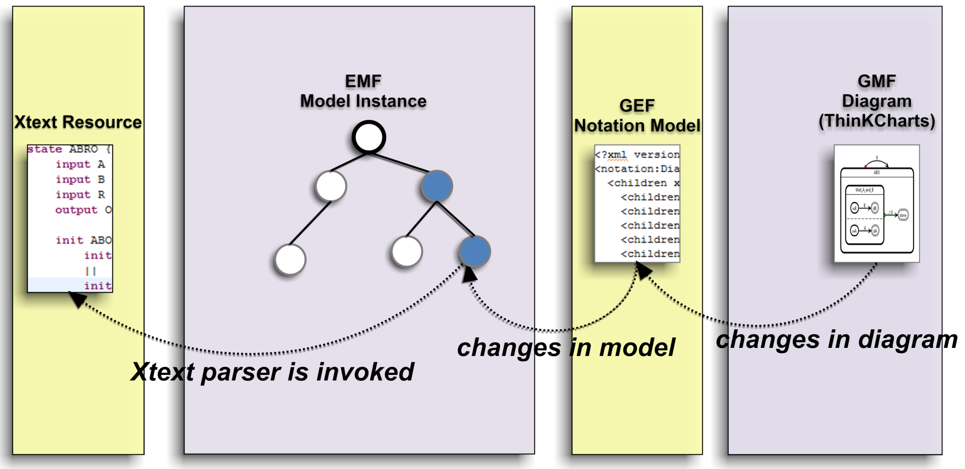


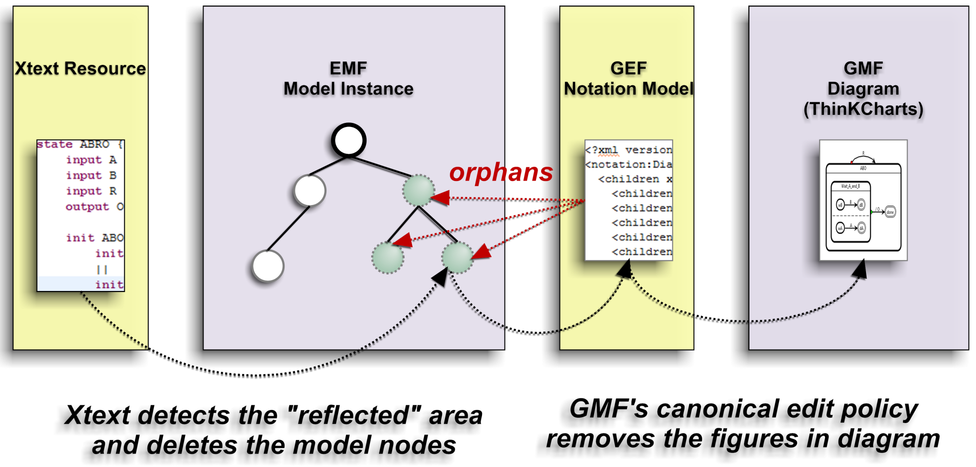
Figure 2.16: Architecture of a full synchronization between Xtext and GMF

removes the elements to re-parse the corresponding field again. As a result, the reference between the notation elements and the deleted domain model is broken. The corresponding notation elements do not have any semantic model elements anymore. As such they are also removed from the GMF editor due to its canonical edit policy 2.17(b). In the next step, the Xtext parser parses the text and instantiates the model again. However our domain file now has new model elements in the changed area. Effectively GMF does recognize that the new nodes are identical to the old nodes. Hence all manually changed diagram information, such as position or size, gets lost with the deleted figures 2.17(c). This is the first problem we have to solve.

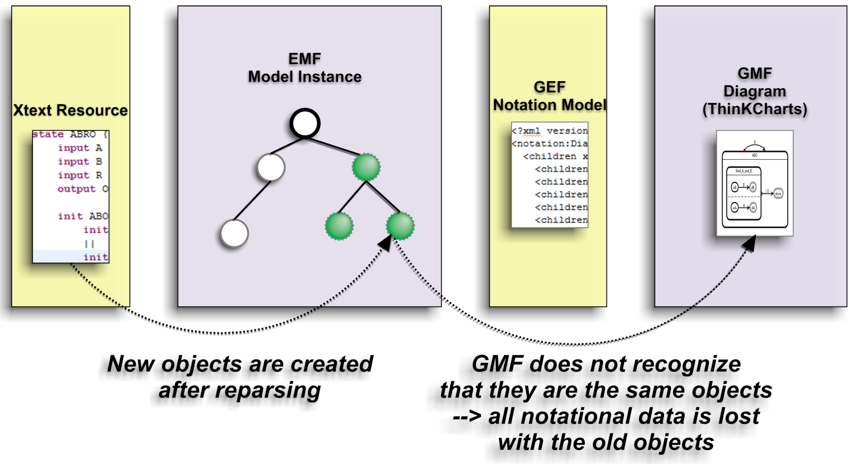
There are further considerations when synchronizing Xtext and GMF editors: At different points in this work we have mentioned that Xtext generates an EMF model merely from a grammar file. This is not entirely correct. Xtext eventually creates an EMF model but it needs more than the grammar to do so. The main concentration of the grammar is to define an Abstract Syntax Tree (AST) along with the transformation from text to AST (parsing) and backwards (serialization). The AST-to-model transformation is later made by the linker in the linking phase. Also other semantic aspects like validation of constraints or model-to-model transformations are either added on top of the AST by other fragments of the Xtext framework or have to be manually implemented at appropriate docking points. Only after these subsequent steps do we have a model that is not only grammatically correct but also correct according to our EMF meta-model. Hence the Xtext grammar has its strong point more in the lexical aspects of a language than the semantic ones. Effectively it is not powerful enough to cover all features of an EMF model. As a result a grammar that parses as much as possible and leaving semantics to the subsequent steps should be preferred as this approach will at least ease the serialization phase. This effectively will produce a set of grammatically correct models that is greater than the set of



(a) Semantic model adapts to changes in GMF editor


Xtext Resource
EMF
Model Instance
GEF
Notation Model
GMF
Diagram
(ThinKCharts)
orphans
Xtext detects the "reflected" area
and deletes the model nodes
GMF's canonical edit policy
removes the figures in diagram

(b) Xtext reacts to changes in semantic model by reparsing the entire affected area



(c) Loss of notational information after reparsing of the model by Xtext

Figure 2.17: Conflicts arising from different edit policies of Xtext and GMF

semantically valid models but the elimination can be done in a separate validation step so that we still will not allow the users to enter invalid models in the editor even if they would conform to our grammar.

Another issue that arises from the differences in the expressiveness of Xtext and GMF are fields that are mandatory in our model but not integrable in our grammar. For instance, we have bi-directional references in our grammar like the relationship between a state and its region. When a state contains a region, the opposite reference in a region, called parentState, is set automatically to the container state by EMF. In the Xtext grammar however we cannot introduce a rule to set the opposite of this bi-directional relationship without further ado as this would be part of a left-recursive call: We would have to call a state rule from our region rule and a region rule from our state. Furthermore, the value of the parent region is derived from the state, and "a particular region" cannot be expressed merely with an Xtext grammar rule. As a result we only have a rule for setting the inner state of a region but not for setting the parent region of a state. When the Xtext editor is synchronized with the GMF editor, the parent region of each state will automatically be set. However the Xtext serializer will fail to serialize a feature for which there is no grammar rule.

The last conflict arises from the index-based URIfragments that are used within EMF to reference cross-document objects. We introduce fragments in Chap. 5. Mainly, the problem is that we need a way to reference our model elements, that are serialized in KIELER Textual Language for SyncCharts (KITS) from GMF's notation file. By default each EMF model is assigned a fragment that defines it in its resource file. This fragment is created by numbering the elements in a containment reference according to the order of their appearance. When an element is removed from a containment list, indices of remaining elements change accordingly, and the mapping from the notation file to the domain elements changes. Hence we have to define a new policy to provide fragments.

A further need for the synchronization is implementing a *formatter* for our serializer. To enable a good user experience, actions to navigate between both editors, e.g. selecting a node in a diagram and opening it in the text, should also be implemented.

So far we have stated main problems that arise in both defining a textual syntax and synchronizing ThinKCharts with a textual editor. Let us now turn to our solution approaches to these problems.

3 The KIELER Textual SyncCharts Editor

In Sec. 2.4.1 we have presented our considerations regarding textual SyncCharts languages and concluded with the need of a new textual language, namely KITS. In this section, we will have a closer look at our conclusions behind the design of this new language.

3.1 Language Design Concepts

KITS is a descriptive language, and hence it will not merely be read and written by machines. Therefore its syntax is similar to the topology of a graphical syncChart. Also KITS' syntax does not contain any meta-data information such as version numbers or graphical data.

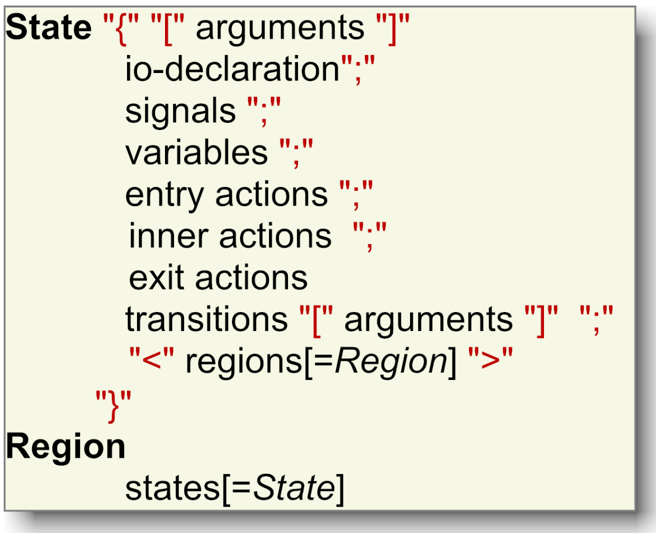
KITS does not employ *white-space semantics*, which would make a rapid development without specific tool support virtually impossible.

One of the difficulties we have seen when analyzing KIT was the deployment of different parentheses. KITS deploys only curly braces for inner states of a macro-state. The only delimiters used other than this are semicolons to end a transition.

SyncCharts in KITS conform to the following overall structure: Every syncChart starts with a *region* that contains *variable*s and *state*s. Furthermore each state can declare *signals*, *entry-*, *inner-* or *exit-action*s as well as an arbitrary number of regions to hold its inner states. Suspension triggers have been included to states to obtain a compact structure. The structure of SyncCharts in KITS is presented in 3.1(a).

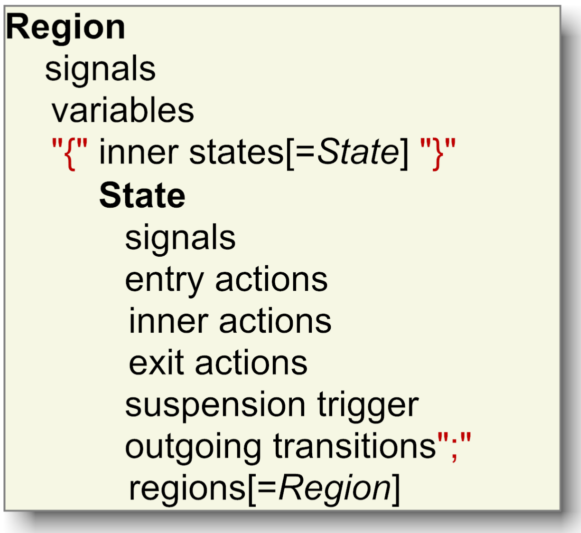
As the SyncCharts meta-model used in KIELER defines that every state has to have a parent region, KITS has to enforce a region declaration before each state declaration. However, our editor implicitly creates needed regions, hence they do not have to be explicitly introduced. Users can optionally declare and name regions with the keyword `region`. This is illustrated in 3.2(a) and 3.2(b).

KITS abbreviates state modifiers and transition types by keywords or symbols. As a concept, we have avoided a syntax that is mainly symbolic or uses merely keywords as this will overload symbols or keywords and hamper the readability. Furthermore we have enhanced the expressiveness of KIT by adding local signal declarations to states.



```
State "{" "[" arguments "]"
          io-declaration";"
          signals ";"
          variables ";"
          entry actions ";"
          inner actions  ";"
          exit actions
          transitions "[" arguments "]"  ";"
          "<" regions[=Region] ">"
       "}"
Region
          states[=State]
```

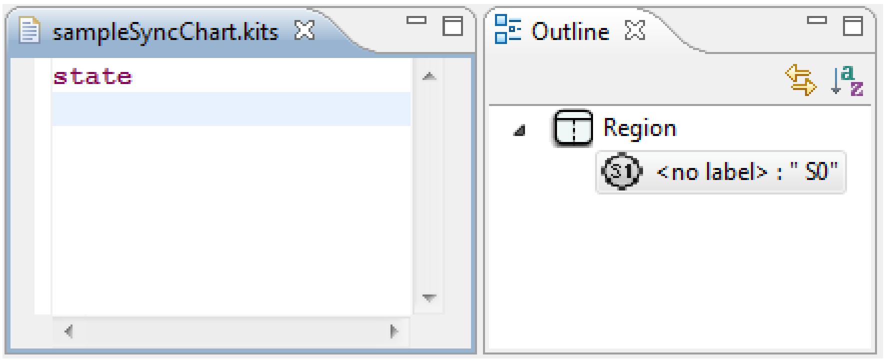
(a) SyncCharts building blocks in KITS



```
Region
   signals
    variables
    "{" inner states[=State] "}"
        State
          signals
          entry actions
          inner actions
          exit actions
          suspension trigger
          outgoing transitions";"
          regions[=Region]
```

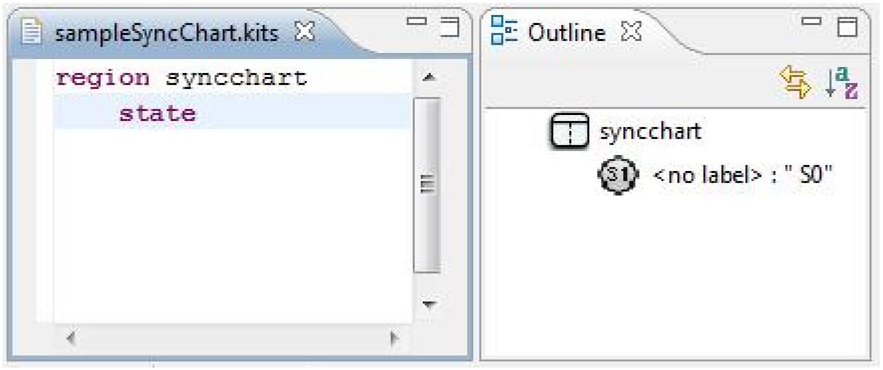
(b) SyncCharts building blocks in KIT

Figure 3.1: Changes in the overall structure and delimiter usage



```
state
```

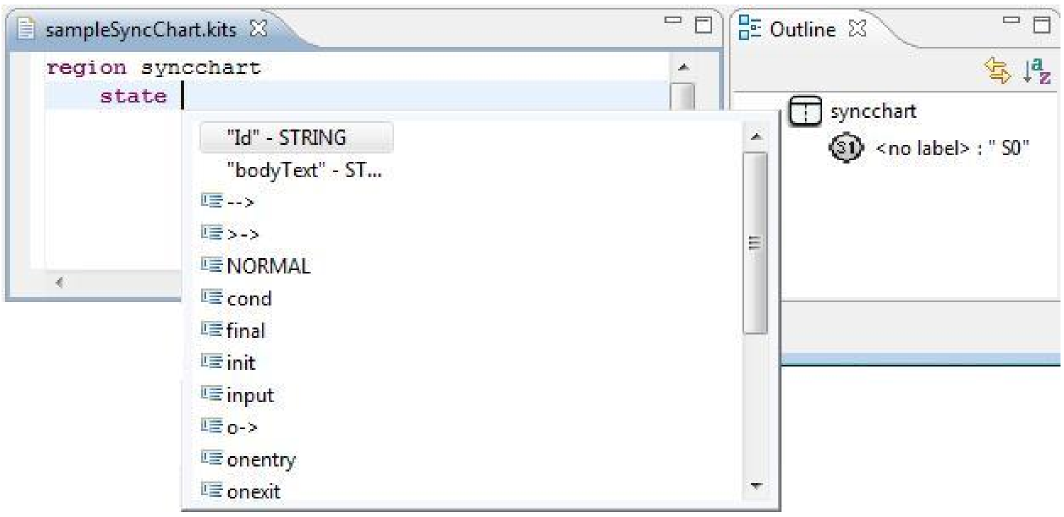
(a) Region initialization in KITS



```
region syncchart
     state
```

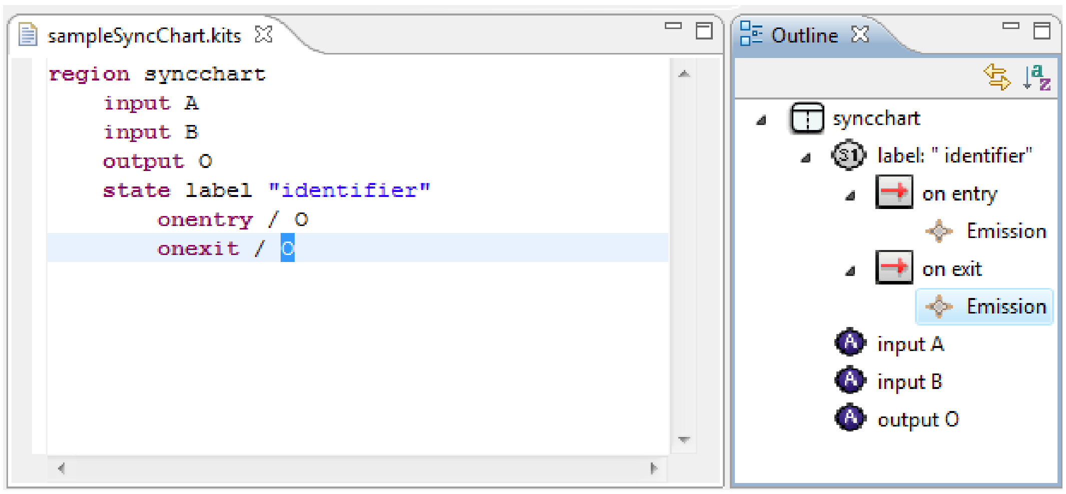
(b) Explicit region declaration in KITS

Figure 3.2: Simple syncCharts in KITS



```
region syncchart
    state
```

Figure 3.3: Content assist example in KITS editor



```
region syncchart
    input A
    input B
    output O
    state label "identifier"
        onentry / O
        onexit / O
```

syncchart
label: " identifier"
on entry
Emission
on exit
Emission
input A
input B
output O

Figure 3.4: Sample state with actions, label and identifier in KITS

3.2 SyncCharts in KITS

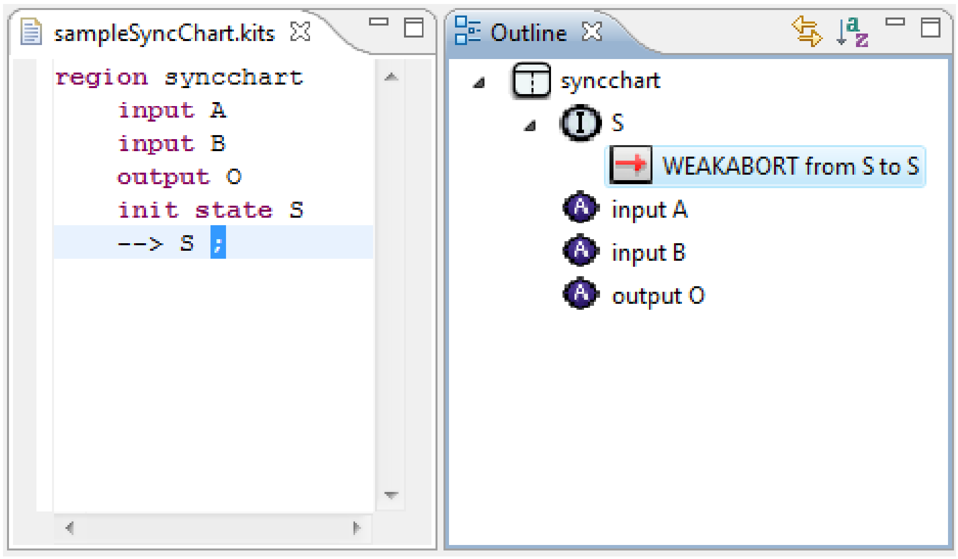
A concept in KITS is to make state declarations, being the common use-case, as comfortable as possible. To this end the order of the *state modifiers* init and final is not fixed. *State types* can also be omitted. In this case the state will have the default type normal. State types in KITS are conditional, reference or textual. The types reference and textual have been included into the language specification in order to support further enhancements such as defining states that are described in a specific program code. Alternatively, the keyword state will also be sufficient to declare a state.

A further concept is the support for *anonymous states*. Mandatory state names can hamper a convenient development—especially of larger syncCharts. Generally, naming states is redundant in case a state is needed only once, e.g. as the final or initial state of a region. Hence KITS supports anonymous state declarations.

An anonymous state would not be referable, e.g. by other tools that operate on the same model. Therefore they are automatically assigned an identifier of the form Sn where n is an index that is unique in the corresponding region. Consequently the scope of a state in KITS is framed by its region.

A state also has an optional label. Labels are what the user "normally" enters to name his or her state. As we will later see in the synchronized version of our editor, labels are also shown in the diagram nodes whereas identifiers can be seen in an extra properties view. In Fig. 3.4 a state with both a label and identifier is presented. In order to distinguish labels from identifiers, the latter are noted between quotes. The outline view is configured to show both.

KITS enables transition declarations without explicit source states. In this case the last declared source state is assigned as the source of the transition as illustrated in Fig. 3.5.



```
region syncchart
    input A
    input B
    output O
    init state S
    --> S ;
```

```
syncchart
    S
        WEAKABORT from S to S
    input A
    input B
    output O
```

Figure 3.5: Omitted source state in KITS

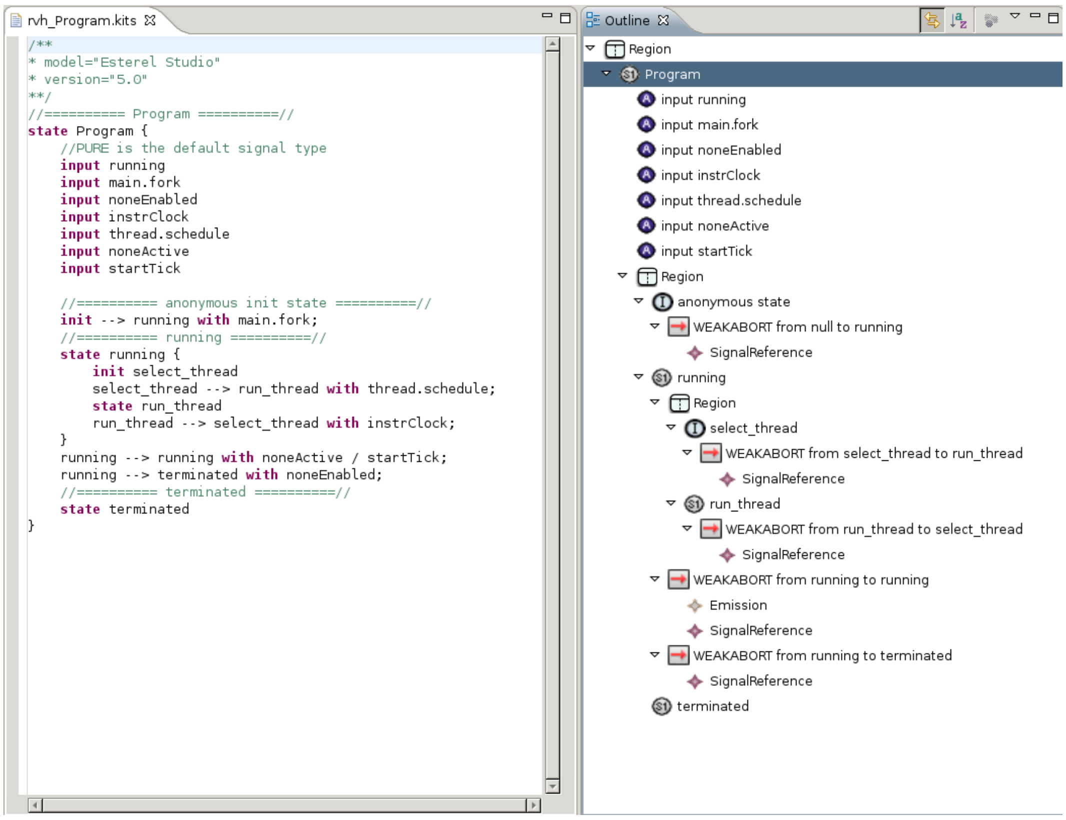
Transition types are denoted by symbols: -> means *weak abortion*, o-> indicates a *strong abortion* and finally >-> stands for a *normal termination*. A transition conforms to the form sourceState <n> targetState "with" # delay trigger effect "history" where

- sourceState is the optional source state reference,
- <n> is an optional priority where n is at least 1,
- targetState is the mandatory target state reference,
- with is a keyword that precedes the transition label,
- # is an optional tag for immediate transitions,
- delay is an integer that defined the transition delay,
- trigger, effect are the triggers and effects of the transition, and finally
- the keyword history marks a history transition.

A more complete sample can be found in Fig. 3.6.

Scopes

The visibility of SyncCharts elements is fundamental for a sound semantics. To this end, KITS assigns so-called *scopes* to every element that can be referenced. We have implemented a *ScopeProvider* that is used by our content assist and linker to find the objects that can be referenced from a given context. State, signal and variable scopes were defined to prevent the input of incorrect models.

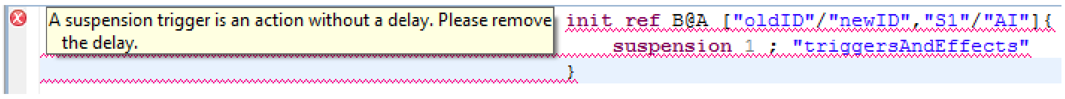


```
/**
* model="Esterel Studio"
* version="5.0"
**/
//========== Program ==========//
state Program {
    //PURE is the default signal type
    input running
    input main.fork
    input noneEnabled
    input instrClock
    input thread.schedule
    input noneActive
    input startTick

    //========== anonymous init state ==========//
    init --> running with main.fork;
    //========== running ==========//
    state running {
        init select_thread
        select_thread --> run_thread with thread.schedule;
        state run_thread
        run_thread --> select_thread with instrClock;
    }
    running --> running with noneActive / startTick;
    running --> terminated with noneEnabled;
    //========== terminated ==========//
    state terminated
}
```

```
Region
  Program
      input running
      input main.fork
      input noneEnabled
      input instrClock
      input thread.schedule
      input noneActive
      input startTick
    Region
      anonymous state
        WEAKABORT from null to running
          SignalReference
      running
        Region
          select_thread
            WEAKABORT from select_thread to run_thread
              SignalReference
          run_thread
            WEAKABORT from run_thread to select_thread
              SignalReference
        WEAKABORT from running to running
          Emission
          SignalReference
        WEAKABORT from running to terminated
          SignalReference
      terminated
```

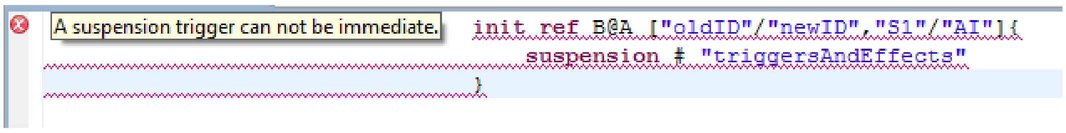
Figure 3.6: Sample syncChart in KITS



A suspension trigger is an action without a delay. Please remove the delay.

```
init ref B@A ["oldID"/"newID","S1"/"AI"]{
        suspension 1 ; "triggersAndEffects"
}
```

Figure 3.7: Validation in KIELER textual SyncCharts editor - 1



A suspension trigger can not be immediate.

```
init ref B@A ["oldID"/"newID","S1"/"AI"]{
        suspension # "triggersAndEffects"
}
```

Figure 3.8: Validation in KIELER textual SyncCharts editor - 2

Validation

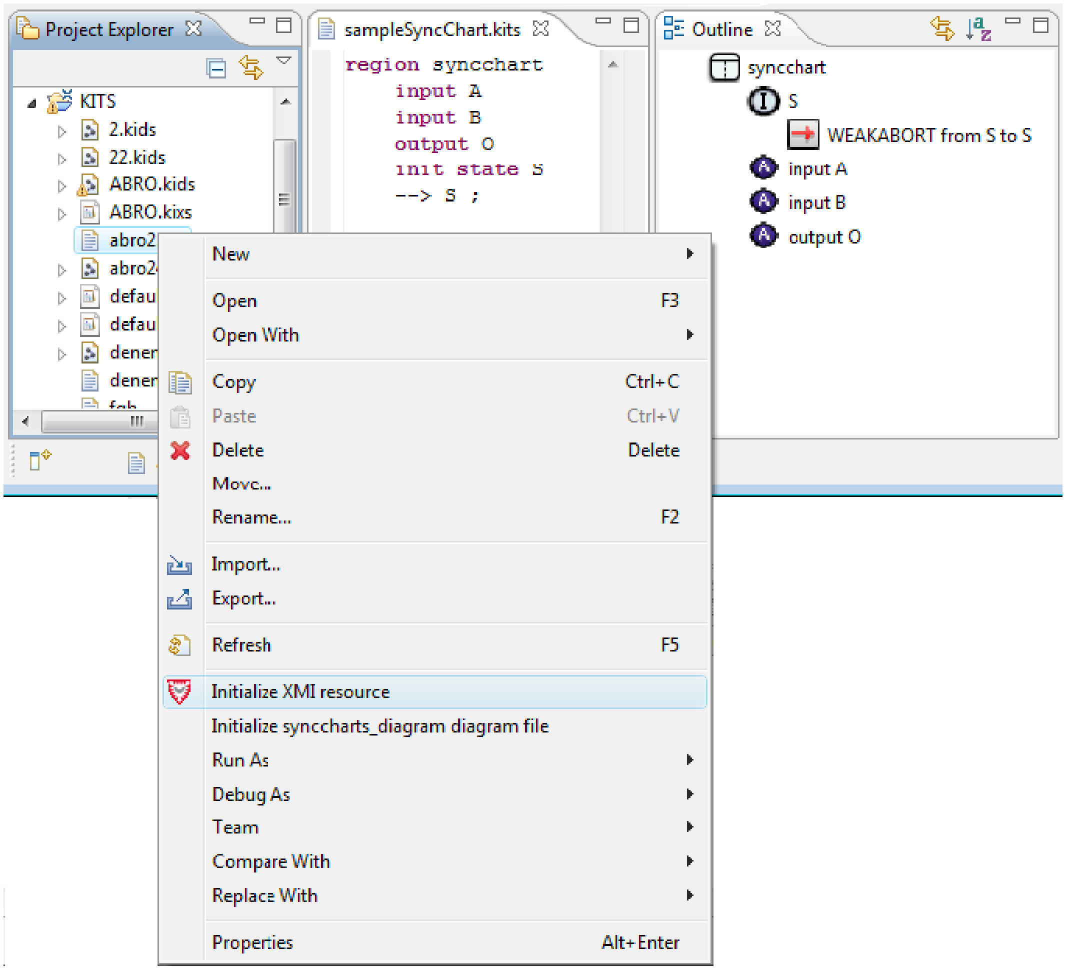
As we have discussed in Sec. 2.4.2, the grammar language deployed by Xtext is expressive with regards to the lexical aspects of a language, yet the models it accepts should further be restricted to those that are also semantically correct. As KIELER integrates a number of tools that operate on the same model, the semantic validation of the underlying system should be tool-independent. Therefore the semantic restriction rules are attached directly to the meta-model in KIELER. The validation language used is *Check*, which has been introduced in Sec. 2.1.1. As an XtextResource "listens" to the EMF meta-model, it is able to give visual warnings in case a rule has been violated. Examples are presented in Fig. 3.7 and in Fig. 3.8.

KITS2XMI

As we recall from Sec. 2.2.2, EMF provides a default XMI parser and serializer for its models. By deploying an Xtext parser and serializer we have "disabled" the default XMI serialization. However in some cases a serialization to XMI might be desirable. To this end, an action has been introduced that can be triggered by a right-click on a file that has the .kits extension as seen in Fig. 3.9. This action then serializes the model in the default XMI syntax and saves it in a resource with the same name but this time with the extension .kixs.

3.3 IDE Support for KITS

As can be seen in the figures, the editing of syncCharts is supported by an outline view. The outline view has been configured to show a detailed tree-based overview of the syncChart that is being developed. As such, it shows the source and target states of transitions, if set, triggers and effects of actions, both labels and identifiers of states among details of transition effects and triggers. A further assistance is given by the outline view by highlighting selected elements in the text as can be seen in Fig. 3.4.



```
region syncchart
    input A
    input B
    output O
    init state S
    --> S ;
```

Figure 3.9: The alternative XMI serialization of a model in KITS

Another editing help is the content assist. This is shown in Fig. 3.3. Further aid is provided by textual navigation that highlights the declaration of an element when the user hits the F3 key while hovering with the mouse over it.

Finally, pre-defined templates and a project wizard aim to ease and fasten the development process.

4 A Semi-Full Synchronization Approach for Xtext and GMF

In Sec. 2.4.2 we have seen the difficulties that arise when an Xtext editor shares the same model instance with a GMF editor. This short section summarizes our approaches that address these problems.

As visualized in 2.17(b), the main problem in the synchronization of Xtext and GMF editors is that the Xtext parser makes constant changes to the model that are also visible to the GMF editor when they share the same model instance. The GMF editor, due to its canonical edit policy, immediately reflects these changes in the diagram. This result is a diagram that re-iteratively changes its contents and displays wrong models.

The consequently arising question is: When can we be sure that the model is valid, and hence can be visualized? The answer is shown in Fig. 4.1: On file save.

The Xtext editor operates on its own copy of the model, and hence the changes it applies to the model are invisible to the GMF editor at this point. When the textual file has been changed, the model instance that is used by the GMF editor is also updated.

Again, due to its canonical policy, the GMF editor immediately reflects the changes and displays the model—this time, a correct one.

To sum up the idea in this "semi-full synchronization", Xtext and GMF should share the same model—not, however, the same model instance. As a result, the editors do not adapt their contents against a "volatile" runtime model, instead they refer to persistent, valid models.

An immediate result of this strategy is that there can be more than one "dirty"

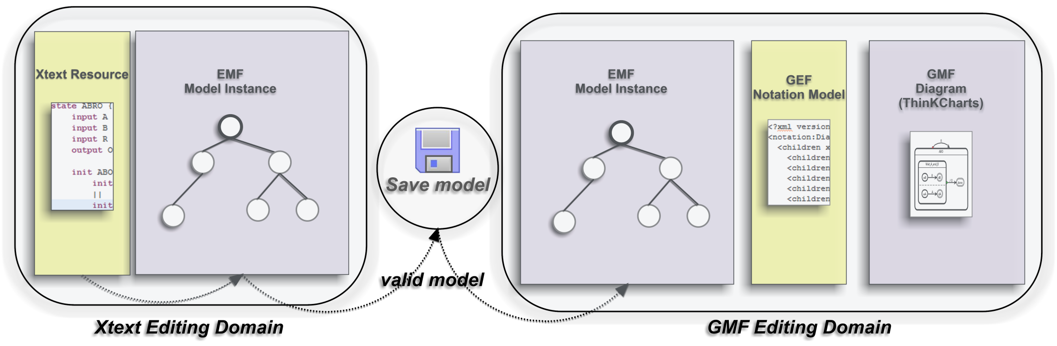


Figure 4.1: Separated editing domains for Xtext and GMF editors

editor that operates on SyncCharts at the same time. A dirty editor is an editor that has unsaved modifications. Since both Xtext and GMF have their own model instances, they can apply changes to their model independent from each other.

When their contents are saved, the default Eclipse policy overwrites changes in the unsaved editors. Consequently, we need a concurrent modification policy.

We handle concurrent modification by registering listeners to Xtext and ThinKChart EditorParts at workbench start. At well-defined points,e. g. when the ThinKCharts editor has been opened or when the resource set has been changed, we can make queries and and react to changes in the KITS editor or ThinKCharts.

Among other things, we listen to the event, that is created before saving an editor. At this point, we detect a concurrent modification by simply iterating over the dirty editors and checking if there is more than one. If so, we prompt a user inter-action and let the user decide whether or not to save the file.

A further problem, we have pointed to in Sec. 2.4.2, was caused by index-based fragments. This conflict is solved by overwriting the so-called *fragment provider* to generate name-based fragments instead of index-based fragments in order to correctly cross-reference objects in the domain model from a GMF editor.

Finally, the action to open selected diagram elements in the KITS editor makes use of the fact that Xtext parser nodes reference their corresponding model elements. When the user selects a node in ThinKCharts, for instance, its parser node can be determined. This parser node contains, among other things, the offset and the length of our model. Hence we can easily use the selection provider of our textual editor to highlight the selection given by the particular offset and length.

5 The Development of the SyncCharts Editor with Xtext

We first have to write the Xtext grammar for our language. As we have seen in Sec. 2.3.2 our Xtext grammar will both define the syntactical structure of our language *and* specify how the Xtext parser will later instantiate EMF models. To this end, we either have to tell Xtext to generate a new EMF model or import an existing one.

A SyncCharts EMF meta-model already exists in KIELER and can be seen in Fig. 2.5. Hence we want to import this meta-model rather than generating a new one.

Importing meta-models can be done in two different ways in Xtext: Using their nsURI or their platformURI. *URI*s are used to uniquely identify resources —or objects within resources as we will see in Chap. 6— and extensively utilized in the persistence framework of EMF.

EMF has a pattern for building any kind of *URI*. Every *URI* consists of three main parts, called *scheme*, *scheme-specific part* and an optional *fragment*.

The *scheme* specifies the type of the *URI*, e.g. file or platform. The *scheme-specific part* is further detailed in different parts but they are not relevant for our concerns. Roughly, we can say that it gives a hierarchical path to our resource starting from a so-called *authority*, e.g. a host in a network system, or *device*, e.g. C as the preceding drive in a file path. The last part of a *URI*, the fragment, is used to identify a part of the resource hence has to be unique in this resource. We will need a deeper introduction to *URI fragment*s in Chap. 6.

Every EMF model resides in an EPackage that can be addressed by it nsURI or platformURI. The platformURI of our meta-model is, according to the pattern above, platform:/resource/de.cau.cs.kieler.synccharts/model/synccharts.ecore and specifies a relative path of the meta-model in our workspace. We utilize this *URI* to import our SyncCharts meta-model.

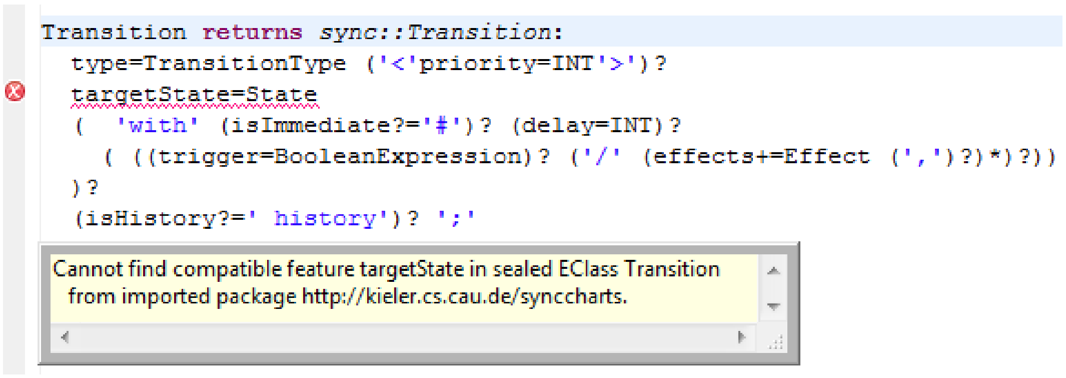
Imported meta-models can optionally be assigned an alias to ease their reference in the grammar. We set sync as our alias and are thereafter able to use existing types as return types of our parser rules as can be seen in the Region rule in Listing 5.1.

Our rule returns an instance of the existing model element Region, accepts an optional keyword region as well as an optional identifier. It also has three assignments. The first two are grouped together by the *alternative operator* $|$. The *add operator* $+=$ means that we have a multi-valued attribute on the left hand side of our assignment, i.e. lists. These lists are called variables and signals according to the features of a Region in our meta-model. The right hand side of the assignments are calls to other rules in our grammar. The *cardinality operator* of a rule determines how many

Listing 5.1: Importing existing meta-models in Xtext

```
import "platform:/resource/de.cau.cs.kieler.syncharts/model/syncharts.ecore"
       as sync

Region returns sync::Region:
{sync::Region}
  ('region')? (id=ID)?
  ( variables+=Variable | signals+=Signal)(innerStates+=State)+;
```



```
Transition returns sync::Transition:
  type=TransitionType ('<'priority=INT'>')?
  targetState=State
  (  'with' (isImmediate?='#')? (delay=INT)?
     ( ((trigger=BooleanExpression)? ('/' (effects+=Effect (',')?)*)?))
  )?
  (isHistory?=' history')? ';'
```

Cannot find compatible feature targetState in sealed EClass Transition from imported package http://kieler.cs.cau.de/syncharts.

Figure 5.1: Restrictions caused by importing meta-models in an Xtext gramar

times it can be or at least has to be called. "∗" means *zero-or-more* hence we can add an arbitrary number of elements to both variables and signals by calling these assignments as much as we need but we can also omit it. This is different for the last assignment: Its cardinality operator is "+" meaning that we will need at least one State in our list but then we can add as many as we need to.

Importing and referencing an existing meta-model is a great convenience in case the meta-model is given as in our case however it also brings fundamental restrictions along. We have to stay conform to the existing structure hence we cannot change the meta-model, e.g. by creating other elements or changing reference types.

For instance, in a transition, we want to be able reference existing states as the target state. However we also want to be able to create a new state if no identifier is specified in the target. The input A -> final; should create a new final state and set it as the target. To do so we would need a containment reference to a state from our transition which does not exist in our meta-model. The resulting conflict is shown in Fig. 5.1 We present our approach to solve this below.

After having completed our grammar we will configure our *MWE workflow*. We specify what we want to generate, such as an ANTLR parser or a project wizard, and remove pre-defined fragments, we do not need, such as the fragment for generating an EMF model. After the workflow is configured, we trigger it to generate our artifacts.

Xtext generates a main plug-in that holds the runtime code along with a second plug-in that holds UI relevant artifacts. A third plug-in can optionally be generated that holds helpers for source code generation from our DSL. It is important to clearly separate manually written code from generated code hence Xtext divides the code in each of these plug-ins in two folders, calls src and src-gen.

During the entire development process, the code in any src-gen folder must remain unchanged as it will be overwritten during every code generation. In order to enable a fine-grained manipulation Xtext utilizes *Google Guice* and provides us with a *module* for both runtime and UI plug-ins in their src folder. We know from Sec. 2.3.2 that a *module* declares a list of *binding*s hence every modification we need to make is done in two steps:

Service Implement a new *service* or extend and overwrite an existing one

Binding Bind it in the corresponding *module* to be used instead of the default implementation

Let us elucidate this with an example runtime problem that needs to be solved: The *scoping*.

5.1 Scoping with Xtext

Any cross-reference, we have in our model, needs to be resolved. In case we want to emit a signal in our transitions, we need to cross-reference existing signals. If we try to emit a signal that either has not been declared anywhere or that was declared beyond our *scope* we should catch this mistake and return a linking error.

Hence a *scope* in Xtext defines the set of elements that can be "reached" from the context of a cross-reference. When the linker tries to resolve cross-references, it iterates only the elements in the given *scope* and returns an error if nothing that matches the user input has been found.

Consequently, there are two fundamental questions according to cross-reference:

Candidates How do we collect the elements that are in our *scope*?

Matching How do we decide if the input *is* in the *scope*?

The default linking semantic, pre-defined by Xtext, builds scopes in a nested manner that is comparable with the scoping in, e.g. Java[1].

According to this scoping policy every signal, defined in a region, would be visible to every transition in all inner states of this particular region. However we would not be able to reference them from a second region in the same hierarchy level.

We might want to change this in order to differentiate between input and output signals: In the effects of a transition, we want to "see" only the input signals of our default scope while in the effect of a transition we merely consider the output signals.

[1]`http://www.eclipse.org/Xtext/documentation/0_7_2/xtext.html`

Xtext supplies a hook to alter the default scoping. The default scoping is given by the SimpleNameBasedScopeProvider and is bound by to the framework by our runtime module, namely DefaultRuntimeModule.

On top of a Google Guice module that manages the dependency injection “alone” Xtext additionally introduces a hierarchical structure of modules. Thus we have five modules that are nested and may override the binding in their superclasses.

The AbstractRuntimeModule extends our DefaultRuntimeModule and overrides the binding for scoping and returns another implementation that is generated and named after the name of our DSL: KitsScopeProvider.

The result is that we have a generated scope provider implementation that is left blank and is intended to be modified. In case, we do not carry out any modifications to KitsScopeProvider, the implementation will fall back to the default semantic in SimpleNameBasedScopeProvider.

Hence we merely need to specify our intended scope implementation for signals in transitions and Xtext will organize the dependency injection in the background.

Since we want to define the scope for merely signal emissions in triggers and signal references in effects we would need to go through a lot of elements in our meta-model and find the correct references. This would result in a code that needs to introduces many nests if . . . else queries. Xtext supports us here by introducing a declarative support.

5.1.1 Polymorphic Dispatcher

KitsScopeProvider extends a class, called AbstractDeclarativeScopeProvider, that provides a polymorphic dispatch mechanism to accordingly delegate functions depending on their names. We can implement a method with the name scope_Emission_signal. The declarative support, then, applies it only to signal references in emissions.

So far we have seen how to influence the building of scopes. How does Xtext decide whether or not the input elements is in the calculated scope? The default matching policy is made by comparing the names of elements. Thus we need to have a attribute name in our Signal class. Since signals in our meta-model indeed have a name attribute, this policy satisfies our needs.

The State elements in KIELER, however, lack a name attribute. Instead they have an id and a label. Therefore, the default scoping will not be able to “see” existing states, and hence this will result in linking errors. We need a way to “tell” Xtext to take our scope_Emission_signal implementation for emissions and compare names of elements. Yet *delegate* to another implementation in case of target states to compare the candidates in our scope with the input using their id or label attributes.

5.1.2 Delegation of dependencies

Xtext supplies an AbstractDelegatingScopeProvider that is annotated with a specific name and has methods to set and return a ScopeProvider. Our custom implementation KitsScopeProvider is a subclass of this class. Hence whenever Xtext has to

Listing 5.2: From input to linked model - 1

```
init A
   --> B;
final B
```

calculate a scope for a reference that was not explicitly defined by our implementation it will consult AbstractDelegatingScopeProvider to get a scope. AbstractDelegatingScopeProvider will return ScopeProvider that then calculates the needed scope. This is another point where we can influence the scoping. Google Guice allows us to bind implementations dependent to annotations therefore we can use the annotation of AbstractDelegatingScopeProvider to specify which ScopeProvider it returns. By default it returns a SimpleNameBasedScopeProvider that resolves cross-references by comparing name attributes. Hence we extend this class so that if Xtext comes across a State element, the matching will be done by the label[2] attribute. As a last step we bind it using named binding in our module. In Fig. 5.2 we illustrate the scoping and dependency injection workflow.

We have mentioned the linker and that it uses the calculated scopes to resolve cross-references. Let us now have a closer look at what happens behind the scenes when a user input is parsed and how the path from the input to resolved references look like.

5.2 Parsing and linking with Xtext

What happens when a KITS file is loaded? First the tokenizer builds a *token* stream, then our parser goes through the grammar rules and creates a *parse tree* (or *node model* in other words) and instantiates our meta-model at the same time. The resulting model instance is hereby connected to the parse tree. Thereafter the parse result is added to the contents of an XtextResource before the linking is triggered. The linker first clears all cross-references —if there are any— and installs proxies for objects that are cross-referenced. Hence only then when we know that we have created all objects and installed all proxies the cross-references will be solved.

Consider that our input is the (not very useful) snycChart in Listing 5.2.

The token stream that the tokenizer will create from this input is shown in Listing 5.3. Their corresponding node model and AST are given in Fig. 5.3 and in Fig. 5.4.

Hence the parser registers nodes that represent cross-reference by setting their grammar element to CrossReferenceImpl. These nodes are represented as proxy elements until they are resolved. By default they will only be resolved on demand as Xtext follows the *lazy linking* pattern. Whether or not to lazily resolve the cross-references is specified by the *LazyLinkingResource* that represents our file. Hence

[2]We will see that the id of a state is an internal attribute that is set automatically. The user has the option to explicitly specify it however it is not enforced.

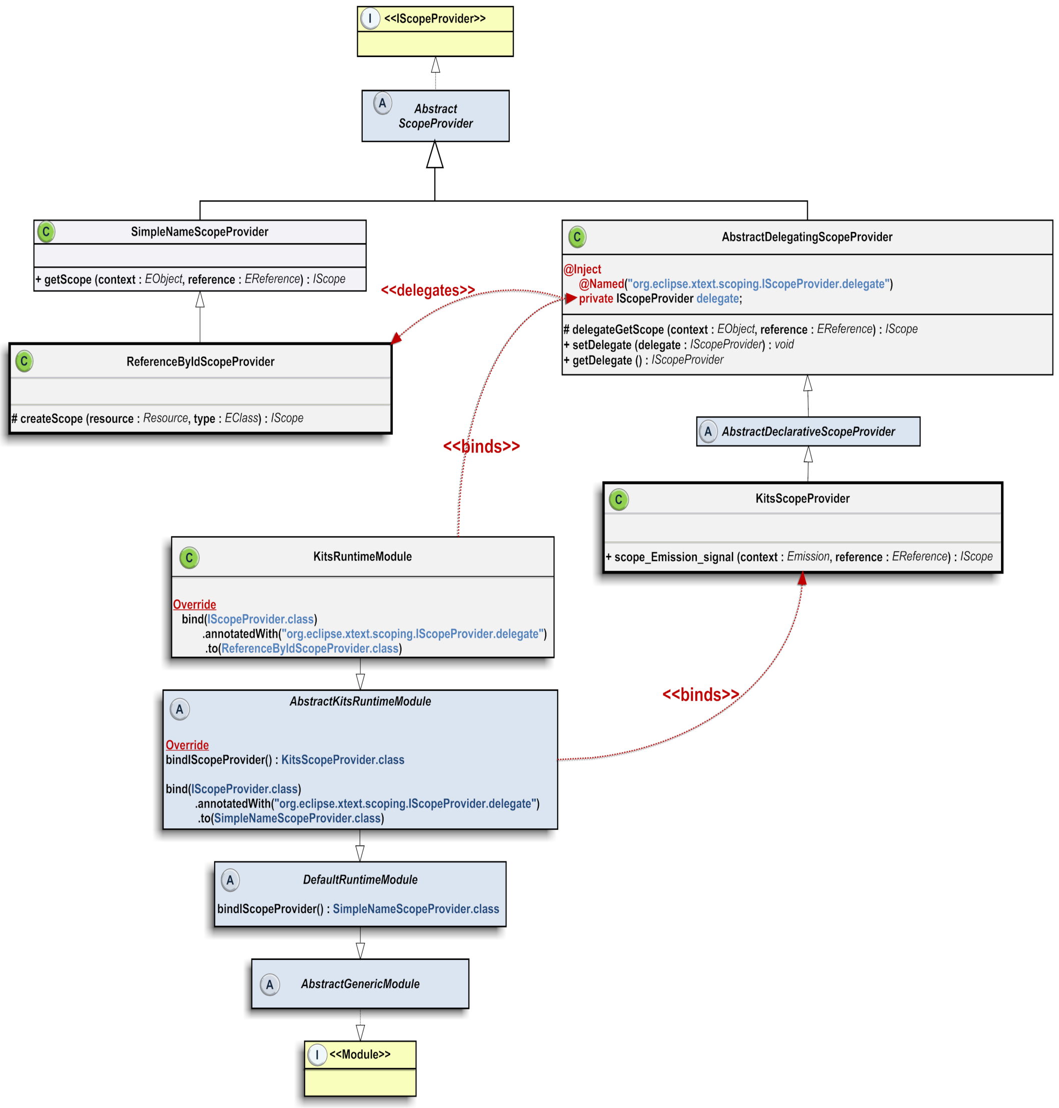

I <<IScopeProvider>>
A Abstract ScopeProvider
C SimpleNameScopeProvider
+ getScope (context : EObject, reference : EReference) : IScope
C AbstractDelegatingScopeProvider
@Inject
@Named("org.eclipse.xtext.scoping.IScopeProvider.delegate")
private IScopeProvider delegate;
# delegateGetScope (context : EObject, reference : EReference) : IScope
+ setDelegate (delegate : IScopeProvider) : void
+ getDelegate () : IScopeProvider
<<delegates>>
C ReferenceByIdScopeProvider
# createScope (resource : Resource, type : EClass) : IScope
<<binds>>
A AbstractDeclarativeScopeProvider
C KitsScopeProvider
+ scope_Emission_signal (context : Emission, reference : EReference) : IScope
C KitsRuntimeModule
Override
bind(IScopeProvider.class)
.annotatedWith("org.eclipse.xtext.scoping.IScopeProvider.delegate")
.to(ReferenceByIdScopeProvider.class)
<<binds>>
A AbstractKitsRuntimeModule
Override
bindIScopeProvider() : KitsScopeProvider.class
bind(IScopeProvider.class)
.annotatedWith("org.eclipse.xtext.scoping.IScopeProvider.delegate")
.to(SimpleNameScopeProvider.class)
A DefaultRuntimeModule
bindIScopeProvider() : SimpleNameScopeProvider.class
A AbstractGenericModule
I <<Module>>

Figure 5.2: Dependency injection workflow on the example of scoping

Listing 5.3: From input to linked model - 2

```
The token stream ...
[init,' ',A,\n,-->,B,;,\n,final,' ',B]

... is synthesized from the terminal rules:
[KEYWORD,WS,ID,WS,KEYWORD,ID,KEYWORD,WS,KEYWORD,WS,ID]
```

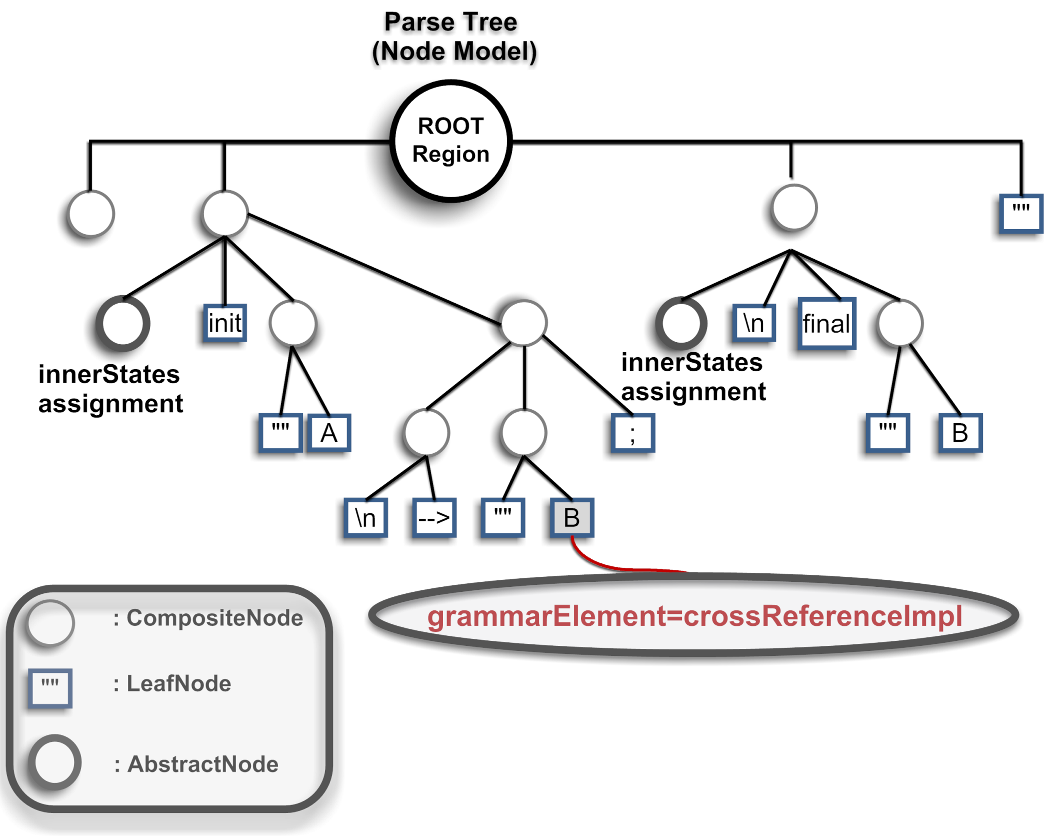


Figure 5.3: From input to linked model - 3

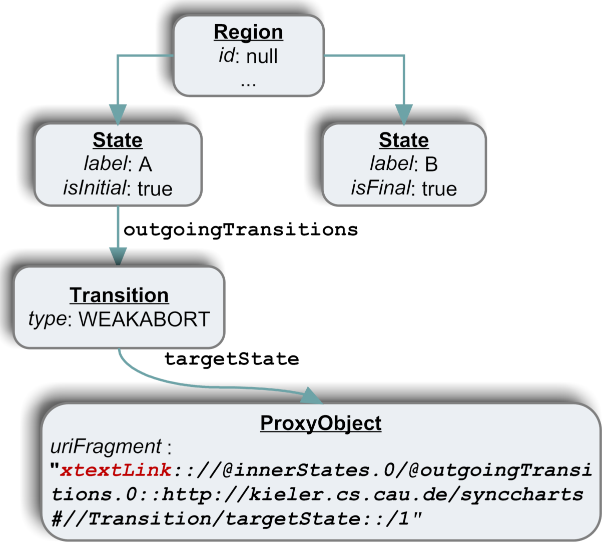


Figure 5.4: From input to linked model - 4

the resolving policy can also be changed to an *eager resolving* extending this class and binding it with the aforementioned Google Guice dependency injection.

When the object for our proxy is asked (for instance by our outline view to adapt its labels according to the object properties) its uriFragment will reveal that it is a cross-reference because the it starts with the string xtextLink_ and this denotes cross-references in Xtext. As the next step the linker will try to link the proxy to the correct object.

This is the point where our linker utilizes the scope provider: It will ask the scope provider to calculate the scope for target state elements and decide whether B points to an existing state.

Consequently we will be confronted with a linking error if our input is, e.g. init A –> final;. Can we prevent this and make our parser or linker create a state instead of creating errors when final is read?

5.3 Anonymous States in KITS

One difficulty, we have pointed out when analyzing KIT in Sec. 2.4.1, was that state modifiers and state type had to be set in an extra assignment even if we needed a particular state only once as the target of a transition. However, as we have seen our meta-model predefines the targetState reference to be a cross-reference hence we can only reference existing states at this point.

To change this behavior, we need to

1. find the point where our linker fails to resolve the targetState reference and provokes an error,
2. remove the linking error,
3. create a new state instead,
4. remove also the syntax error,
5. extract the specified state type and
6. set the type of the newly created state.

Merely removing the linking error and creating a new state will not be sufficient since another error will arise at that point —a syntax error— as our grammar does not allow for anything else than an ID after the transition type, i.e. after "$--\,>$" in our example. Hence we also need to remove the syntax error. If we leave it here, the users still will have to assign the types of states in an extra assignment thus we will read the token that causes the syntax —final in this case— error and set the state type or modifier of the state according to it.

To this end we extend the default resource implementation in Xtext, namely the LazyLinkingResource, and override its getEObject method. This method is responsible for managing the scoping-linking workflow and invokes errors in case the returned linkedObjects list for the object to be resolved is empty. Fig. 5.5 is a simplified

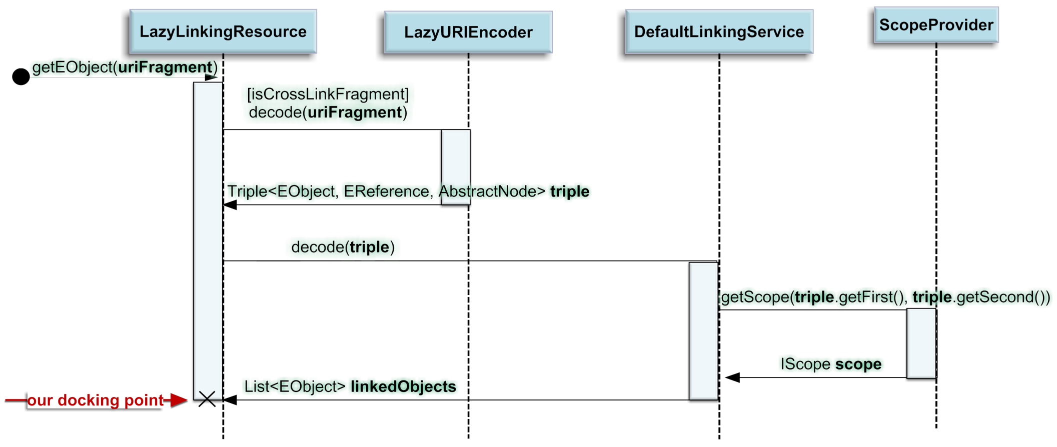


Figure 5.5: Simplified overview of scoping-linking workflow

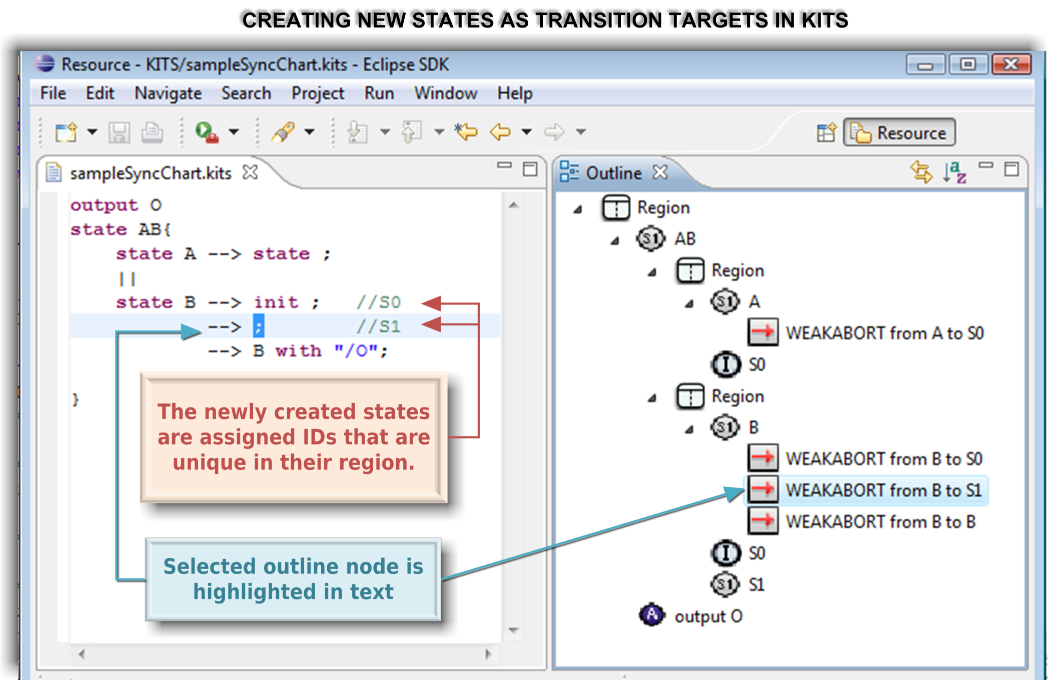
overview of the getEObject method and intends to summarize this workflow. We extract the transition from a “triple” that holds the context of our cross-reference, the reference itself and the parse node for it. After creating a new state and adding it as the target of our transition, we remove the linking error for this reference and read the text of the leaf node that causes the syntax error. We then remove the syntax error as well and set the state type as desired by the user. The result is presented in Fig. 5.6 and Fig. 5.7

As we can see, our example model in Fig. 5.6 and Fig. 5.7 are only grammatically correct yet they contain semantic errors. In a correct syncChart every region should have exactly one initial state and there must be an outgoing transition from an initial state. Furthermore we cannot have “isolated” states meaning that every state needs at least one incoming transitions. In Sec. 2.1.1 we have got acquainted with the validation language *Check*. Let us now see how we utilize this language to manage runtime validation.

5.4 Validation of SyncCharts

It is possible to check our constraints at different levels: We can register our *Check* rules directly at the meta-model level independent from our textual editor. This has the advantage that any diagram that operates on our meta-model will reflect the results of our validation. We have followed this approach in KIELER and attached our validation rules to our meta-model. Listing 5.4 and Fig. 5.8 show how this validation is done.

A second approach would be to define our constraints in Xtext. Xtext provides us with convenient helpers that allow to validate our models at runtime. We can either use the generated *Check* files to hold our *Check* rules or extend AbstractKitsJavaValidator to implement our validation in Java while making use of the declarative support that is provided by this class. In this case we have to ensure that our implementation

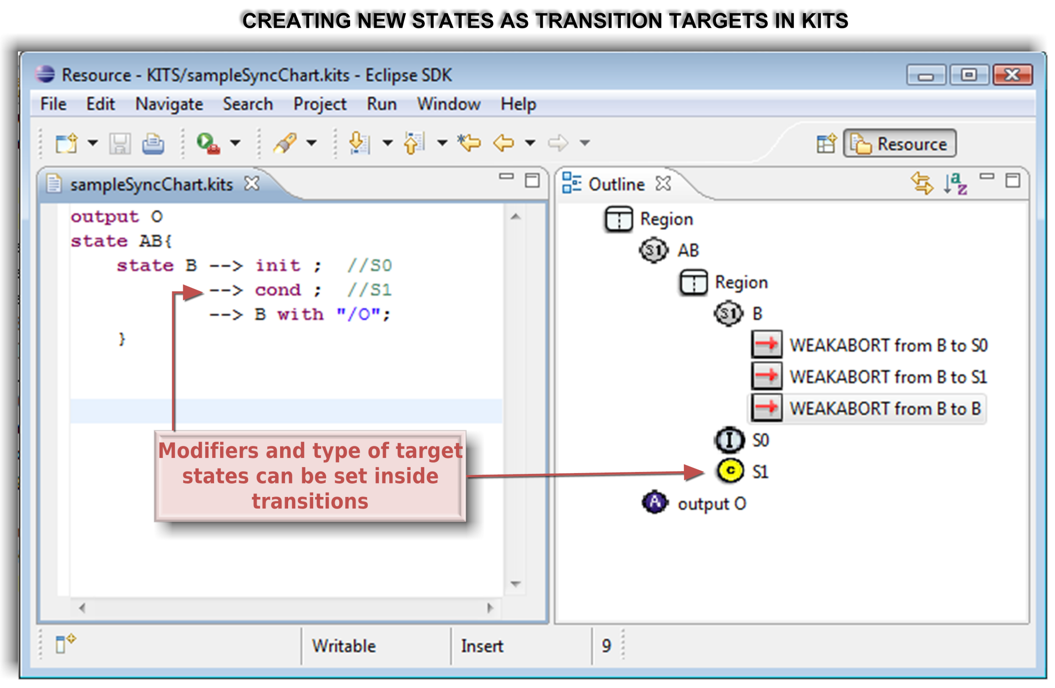

CREATING NEW STATES AS TRANSITION TARGETS IN KITS
Resource - KITS/sampleSyncChart.kits - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Resource
sampleSyncChart.kits
Outline
The newly created states are assigned IDs that are unique in their region.
Selected outline node is highlighted in text

```
output O
state AB{
    state A --> state ;
    ||
    state B --> init ;    //S0
            --> ;         //S1
            --> B with "/O";

}
```

```
Region
  AB
    Region
      A
        WEAKABORT from A to S0
      S0
    Region
      B
        WEAKABORT from B to S0
        WEAKABORT from B to S1
        WEAKABORT from B to B
      S0
      S1
  output O
```

Figure 5.6: Creating new states in transition targets in KITS


CREATING NEW STATES AS TRANSITION TARGETS IN KITS
Resource - KITS/sampleSyncChart.kits - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Resource
sampleSyncChart.kits
Outline
Modifiers and type of target states can be set inside transitions
Writable
Insert
9

```
output O
state AB{
    state B --> init ;   //S0
            --> cond ;   //S1
            --> B with "/O";
    }
```

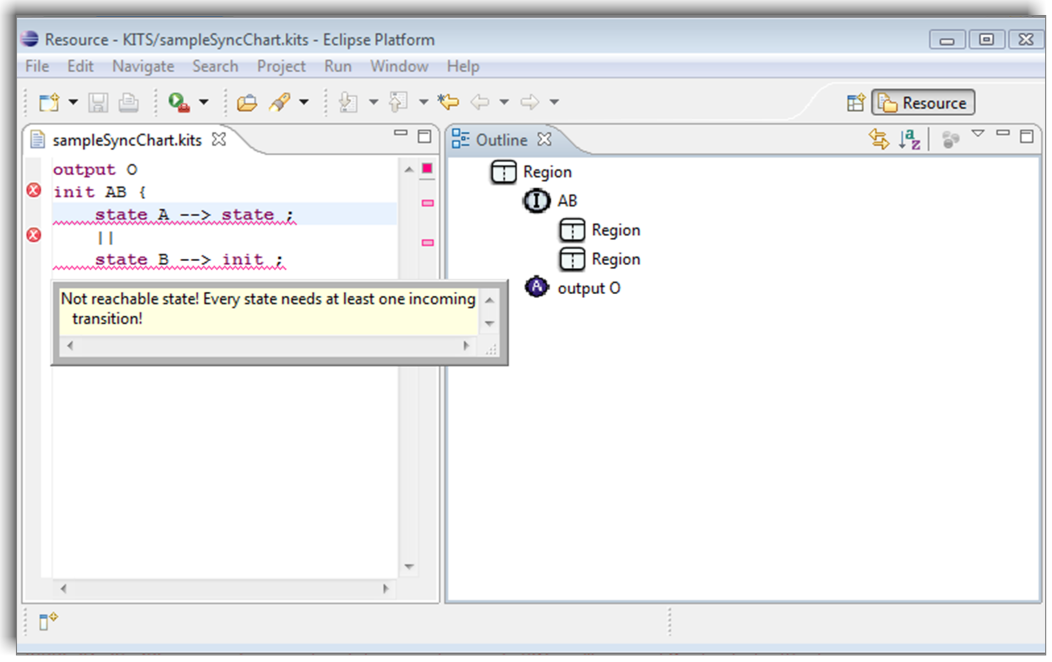
```
Region
  AB
    Region
      B
        WEAKABORT from B to S0
        WEAKABORT from B to S1
        WEAKABORT from B to B
      S0
      S1
  output O
```

Figure 5.7: Setting modifiers and the type of target states in KITS

Listing 5.4: Validation rules for SyncCharts in KIELER

```
context State if
  (parentRegion != null
  && parentRegion.parentState != null
  && isInitial == false)
  ERROR "Not reachable state!
      Every state needs at least one incoming transition!" :
  (parentRegion.innerStates.exists
   (e|e.outgoingTransitions.exists(t|t.targetState == this)));

context Region if (parentState != null)
   ERROR "Every region should have exactly one initial state!" :
   (innerStates.select(s|s.isInitial).size == 1);
```



```
output O
init AB {
   state A --> state ;
     ||
   state B --> init ;
```

Not reachable state! Every state needs at least one incoming transition!

Figure 5.8: Runtime validation in KITS

Listing 5.5: Declarative label provider in Xtext

```
public class KitsLabelProvider extends DefaultLabelProvider {
   String image(State s) {
      String myStateLabelImage = "State.gif";
      if (s. isIsInitial ()) {
         myStateLabelImage = "InitialState . gif ";
      }
      if (s. isIsFinal ()) {
         myStateLabelImage = "FinalState.gif";
      }
      return myStateLabelImage;
   }
```

is injected as we have seen in the aforementioned examples.

We have seen in the figures that the IDE supports us with various editing aids such as a content assist or outline view. Let us see how these refinements were done.

5.5 UI Refinement and Editing Support for KITS

We can refine the generated IDE in various ways and Xtext offers the same convenience mechanisms, we have seen, like the declarative support or a module for bindings regarding the UI.

An example for declarative support is the Xtext LabelProvider that holds modifications for labels in our views. We specify our icons in a declarative manner and Xtext utilizes them in view nodes such as the outline or the content assist. The results can be seen in the figures while an example label declaration can be found in Listing 5.5.

Furthermore we can modify the outline view using a so-called SemanticModelTransformer. Again we are provided with declarative support and can transform the hierarchical outline structure as well as the labels that are shown. An example declaration can be found in Listing 5.6.

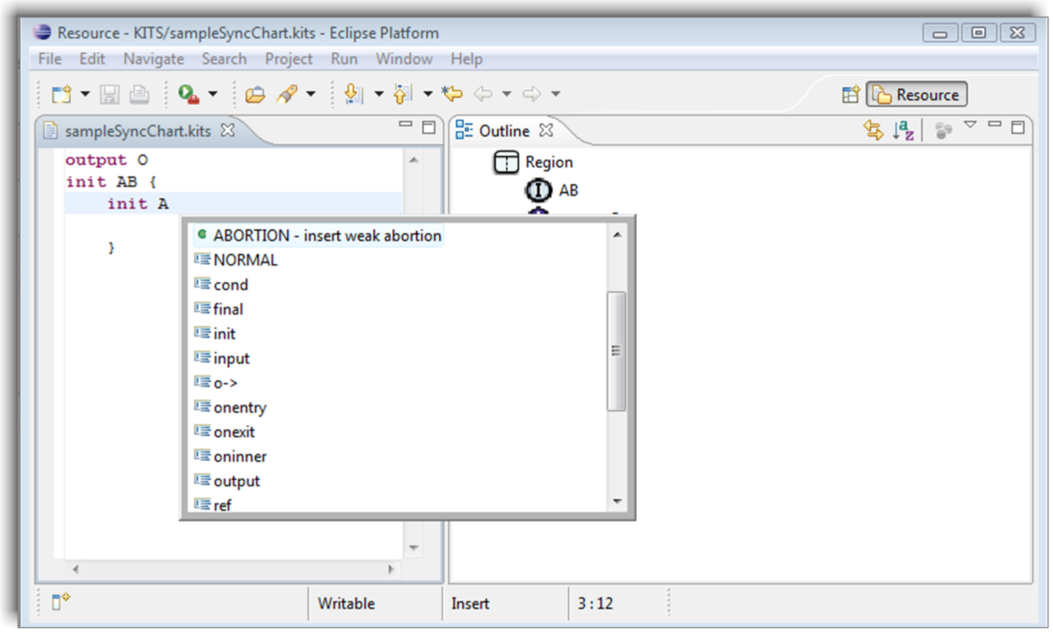
We conclude this section by giving another example of an IDE help, *templates*. We can define templates for our language that are shown in the content assist as can be seen in Fig. 5.9

Listing 5.6: Declarative semantic model transformer in Xtext

```
public ContentOutlineNode createNode
        ( Transition  semanticTransition ,
        ContentOutlineNode parentNode) {
    ContentOutlineNode node = super.
                newOutlineNode(semanticTransition,
                        parentNode);
    // Example: A --> B; note: each transition has a type
    String  transitionLabel  = semanticTransition .getType(). toString ();

    // from A
    if  (semanticTransition .getSourceState()  != null)  {
        State semanticSource = semanticTransition . getSourceState ();
        if  (semanticSource.getLabel()  != null
                && !(semanticSource.getLabel().trim (). equals ("")))

        {
            transitionLabel  = transitionLabel  + " from "
                + semanticTransition .getSourceState (). getLabel ();
        }
    }
    ...
    node.setLabel( transitionLabel );
    return node;
}
```


Resource - KITS/sampleSyncChart.kits - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
sampleSyncChart.kits
Outline
Region
AB
ABORTION - insert weak abortion
NORMAL
cond
final
init
input
o->
onentry
onexit
oninner
output
ref
Writable
Insert
3 : 12

```
output O
init AB {
    init A

    }
```

Figure 5.9: Template support in KITS

6 Sharing the Same Model with GMF

Sec. 2.4.2 described our approaches to address the problems, caused by synchronizing a graphical GMF editor with a textual Xtext editor. This section presents the details of the implementation of this solutions.

6.1 The Synchronization Infrastructure

As discussed in Sec. 2.4.2, the first difficulty in synchronizing ThinKCharts with a textual Xtext editor is caused by the conflicting edit policies of Xtext and GMF, and eliminates a full-synchronization of both editors. In Fig. 4.1 our solution to this approach is presented. Briefly, both, Xtext and GMF, editors work on their own copy of the model and "send" their changes on saving.

To be more concrete, all EMF based editors, along with Xtext and GMF, have an EditingDomain. The EditingDomain is the "workplace" of an Xtext or GMF editor.

Among other things[1], the EditingDomain holds the domain file.

Hence we do not need to change anything to create separate model instances for Xtext and GMF, they work on their own copies by default. Hence, our editors operate on the same model, but are in different EditingDomains.

We want to be able to "listen" to model changes in other editing domains, synthesize syncCharts from KITS files and serialize a syncChart in KITS. Hence adding merely an action to ThinKCharts that allows the initialization of a syncChart from a KITS file will not be sufficient as this will only cover SyncCharts synthesis and not the serialization of a syncChart in KITS.

Tho this end, we have to set the domain file extension of ThinKCharts to .kits. This modification results in the needed initialization and serialization.

6.2 Referencing Xtext Models from a GMF Editor

In Sec. 2.4.2 we have already seen how objects are reference in EMF: Every EMF resource is referenced via a unique URI, and every EMF model in a resource is identified by a *fragment* that is unique in its resource. Listing 6.1 shows a sample EMF model reference. The presented reference is taken from the notation model of the ABRO syncChart.

[1] An EditingDomain also holds a CommandStack to manage editing commands. Hence sharing the same editing domain effectively means sharing also the same CommandStack. Although it has some important implications [5] it is not relevant to our concerns.

Listing 6.1: Sample EMF object URI

```
<element xmi:type="syncharts:Signal"
          href="ABRO.kixs#//@innerStates.0/
                         @regions.0/
                         @innerStates.0/
                         @signals.0"/>
```

The notation model holds the figures in our diagrams as well as a reference to the elements that are visualized by the particular figure. In this case our figure represents a model element that is the "first region of the first inner state in a file called ABRO.kixs". In EMF, href precedes inter-documentary references. In this case, we have an inter-documentary reference because the notation and domain files are saved separately in KIELER. The extension .kixs depicts that our model was serialized using the default XMI serializer provided by EMF.

As can be seen, the default URIfragments in EMF are index-based thus not very permanent: If an element in a containment list is removed, all following elements are "moved up", and hence update their indices.

When we save our model in KITS, we need to ensure permanent URIfragments as they are the connection between the notation and the semantic model.

Generally speaking this is a fundamental difference between a serialization in the abstract syntax, e.g. in XMI and a serialization in the form of a concrete syntax, e.g. in KITS. Concrete syntax references are made *by name* whereas abstract syntax references are directly pointing to an *object*.

Xtext internally handles object references in a different way that is not relevant for our concernes. However, a FragmentProvider can be implemented to return fragments that are used by non-Xtext documents to reference objects, created by Xtext. The default implementation of the FragmentProvider does nothing, and hence returns index-based fragments of EMF.

Our approach to providing permanent object references is to overwrite the FragmentProvider that returns name-based fragments that conform to the pattern <eContainer.object>.

6.3 Interaction Code for Xtext and GMF

As our editors are only synchronized on saving file contents, there can be concurrent modifications on the same model, made in different editors. Furthermore, after initializing a syncChart from KITS an auto-layout should be triggered.

These adaptations are made by registering a KitsConcurrentModificationAdapter to both Xtext and ThinKChart editors as can be seen in Listing 6.2. The KitsConcurrentModificationAdapter is added to the active page of the Eclipse workbench, after the workbench has been initialized. Then the KitsConcurrentModificationAdapter iterates over the open editors and registers itself as a listener by ThinKCharts and

Listing 6.2: Sample EMF object URI

```
public KitsConcurrentModificationAdapter( final IWorkbenchPage activePage) {
    this.page = activePage;
    IEditorReference [] editorReferences = activePage.getEditorReferences ();

    for ( IEditorReference editorReference : editorReferences ) {
        IEditorPart editor = editorReference.getEditor( false );
        if ( editor instanceof SyncchartsDiagramEditor) {
            ((DiagramEditor) editor ).getEditingDomain()
                    .addResourceSetListener( this );
        }
        if ( editor instanceof XtextEditor) {

            ISourceViewer sourceViewer = ((XtextEditor) editor )
                    .getInternalSourceViewer ();
            if (sourceViewer instanceof ITextViewerExtension) {
                ((ITextViewerExtension) sourceViewer)
                        .appendVerifyKeyListener( this );
            }
        }
    }

}
```

the KITS editor. Consequently, it is able to react to changes in both editors at some well-defined points .

In particular, KitsConcurrentModificationAdapter implements three different listeners, namely a PartListener, a ResourceSetChangedListener and a VerifyKeyListener, as can be seen in Fig. 6.1.

Registering the KitsConcurrentModificationAdapter as a listener to resource set changes enables us to trigger an auto-layout at this point. The resourceSetChanged is fired *after* changes to the resource set have been committed, and hence this is a well-defined point for layouting our diagram.

Furthermore, KitsConcurrentModificationAdapter listens to workbench *parts*. Workbench parts are visible components of a workbench, i. e. editors or views. The KitsConcurrentModificationAdapter is thus able to trigger an initial auto-layout when a ThinKCharts editor is opened, as this is also a workbench part. This is seen in Listing 6.3.

When more than one SyncCharts editor is "dirty" at the same time, saving a (KITSor ThinKCharts) file overwrites changes in the other editor(s). To prevent this, KitsConcurrentModificationAdapter lets the user decide and asks whether the changes should be saved or not. To this end, it registers itself as a VerifyKeyListener to the Xtext editor, as can be also seen in Listing 6.3.

When the editors are closed, the registered adapters are removed as is shown

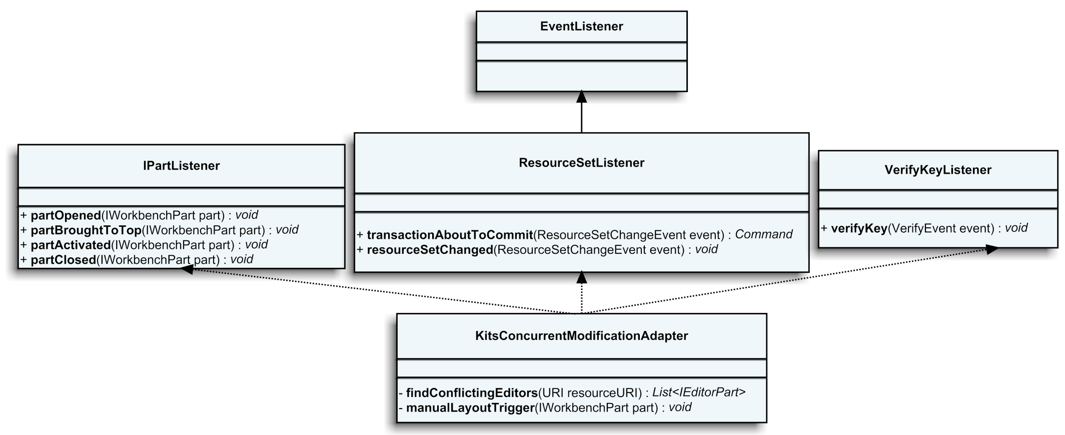

EventListener
IPartListener
+ partOpened(IWorkbenchPart part) : void
+ partBroughtToTop(IWorkbenchPart part) : void
+ partActivated(IWorkbenchPart part) : void
+ partClosed(IWorkbenchPart part) : void
ResourceSetListener
+ transactionAboutToCommit(ResourceSetChangeEvent event) : Command
+ resourceSetChanged(ResourceSetChangeEvent event) : void
VerifyKeyListener
+ verifyKey(VerifyEvent event) : void
KitsConcurrentModificationAdapter
- findConflictingEditors(URI resourceURI) : List<IEditorPart>
- manualLayoutTrigger(IWorkbenchPart part) : void

Figure 6.1: Concurrent Modification Listener in KITS

Listing 6.3: Reactions of the KitsConcurrentModificationAdapter to partOpened

```java
public void partOpened(final IWorkbenchPart part) {
    if (part instanceof DiagramEditor) {
        TransactionalEditingDomain editingDomain = ((DiagramEditor) part)
                .getEditingDomain();
        editingDomain.addResourceSetListener(this);
        manualLayoutTrigger(part);
    }
    if (part instanceof XtextEditor) {
        ISourceViewer sourceViewer = ((XtextEditor) part)
                .getInternalSourceViewer();
        if (sourceViewer instanceof ITextViewerExtension) {
            ((ITextViewerExtension) sourceViewer)
                    .appendVerifyKeyListener(this);
        }
    }
}
```

Listing 6.4: Removing the KitsConcurrentModificationAdapter

```
public void partClosed( final IWorkbenchPart part) {
    if (part instanceof DiagramEditor) {
        TransactionalEditingDomain editingDomain = ((DiagramEditor) part)
                .getEditingDomain();
        editingDomain.removeResourceSetListener(this);
    }
    if (part instanceof XtextEditor) {
        ISourceViewer sourceViewer = ((XtextEditor) part)
                . getInternalSourceViewer ();
        if (sourceViewer instanceof ITextViewerExtension) {
            ((ITextViewerExtension) sourceViewer)
                    . removeVerifyKeyListener( this );
        }
    }
}
```

in Listing 6.4.

6.4 Inter-documentary Navigation

The support of navigation between editors is partly supported. The navigation from ThinKCharts to KITS is implemented by making use of the NodeAdapter in Xtext while a navigation from KITS to ThinKCharts would need a query language to find the objects in the GMF editor. Sec. 9.1 refers to this as a future work.

Listing 6.5 shows the path from a diagram element in ThinKCharts to its selection in the KITS editor. The openInKitsEditor takes two arguments: The semanticElement is the object that has been selected in ThinKCharts. The targetPart is the KITS editor.

As we know from previous chapters, a parser node in Xtext references the object it represents. Furthermore, a parser node also holds the offset and length that its object needs in file. This data is used to find the scope that is to be highlighted in the KITS editor.

Since Xtext still is in development, the source code is under the sway of constant modification. To this end we give an overview of the Eclipse configuration, used for the latest version of this work, in App. B.

Listing 6.5: Opening ThinKCharts' diagram elements in KITS

```java
public static void openInKitsEditor(EObject semanticElement,
        IWorkbenchPart targetPart) {
    try {
        String uri = semanticElement.eResource().getURI().toString();
        if (uri.startsWith(PLATFORM_RESOURCE)) {
            String fileString = uri.substring(PLATFORM_RESOURCE.length());
            IFile file = ResourcesPlugin.getWorkspace().getRoot().getFile(
                    new Path(fileString));

            if (file != null && file.exists()) {
                NodeAdapter nodeAdapter = NodeUtil
                        .getNodeAdapter(semanticElement);
                if (nodeAdapter != null) {
                    CompositeNode parserNode = nodeAdapter.getParserNode();
                    if (parserNode != null) {
                        ITextEditor editor = (ITextEditor) targetPart
                                .getSite().getPage().openEditor(
                                        new FileEditorInput(file),
                                        EDITOR_ID);
                        editor.getSelectionProvider().setSelection(
                                new TextSelection(parserNode.getOffset(),
                                        parserNode.getLength()));
                    }
                }
            }
        }
    } catch (PartInitException e) {
        e.printStackTrace();
    }
}
```

7 Evaluation

This work has been motivated by the need to combine textual and graphical representations of complex systems for a convenient modeling. An example of such systems, SyncCharts in KIELER has served as the application field. The graphical concrete syntax of SyncCharts has been complemented with a textual concrete syntax. The resulting need of synchronizing both views has been addressed by developing a synchronization at a well-defined point, i.e. file save.

By default, EMF models are serialized in XMI. The XMI format of a persistent syncCharts is saved as plain-text. In contrary to the plain-text XMI editor, the language-specific Xtext editor allows a validation at runtime that prevents the user from "breaking" the model. Hence the default serialization language XMI can be defined as a set of strings whereas the domain-specific-language KITS is a set of valid syncCharts models.

Further added values of serializing the syncChart model in a domain-specific language are as follows: The editing of a syncChart in a language that is more "human-friendly" structured than XMI increases the editing speed and the readability of the model. Additional IDE support, such as content assist, a tree-based outline view, textual navigation and the like, enables a quick orientation in the model file. Furthermore, version controlling is done in a "familiar" format.

Xtext is a very powerful framework. The Xtext grammar holds not only the information to generate a parser for the textual input; it also defines the text-to-AST transformation. The resulting AST is transformed (linked) to the EMF model by a linker.

KIELER already has a SyncCharts EMF model, and hence our resulting model is pre-defined. This restricts us when designing the grammar. Also, as the expressiveness of the Xtext grammar language and the EMF Ecore language is different, we can only cover a subset of our model definition in the grammar.

The results are the need for validation—to meet the expressiveness of the pre-defined meta-model—and the need to alter the default linking semantic—to meet the pre-defined meta-model structure.

Without the modular infrastructure for dependency injection in Xtext, especially the latter task would be difficult to solve. Possibly, an intermediate meta-model would be needed, which had to be transformed to the SyncCharts meta-model. Hence Xtext is a generative framework with a lot of implicit conventions, however it also supports a fine-granulated modification of this default semantics by utilizing a powerful dependency injection framework.

The definite shortcoming of using Xtext is that it is still in development and not thoroughly documented. As a result, a considerable amount of time is needed to get acquainted with the internals of Xtext.

The synchronization of editors requires a deeper understanding of both Xtext and GMF. Especially GMF is a mature framework with a complex underlying structure. The conflicts between the foundational strategies of Xtext and GMF can be solved at the end of a longer research by possibly adding a semi-canonical edit policy to GMF.

Hence for the textual editing framework we had to alter the default semantics in Xtext to meet our requirements—whereas to implement the synchronization, we had to alter our requirements.

8 Conclusion

This work has followed a model-based approach to enhance the graphical modeling of SyncCharts in KIELER with a textual editing framework. To this end, we have first designed a new concrete syntax for SyncCharts and developed a textual editing environment using TMF Xtext.

The attempt to fully synchronize the resulting Xtext editor with the existing GMF editor, namely ThinKCharts, has resulted in several problems.

In particular, GMF editors follow a canonical edit policy. This means that their graphical content is synchronized with the underlying domain model at all times. Xtext, on the other hand, follows a partial parsing strategy, and hence constantly restructures the parse tree.

A solution to this conflict has to either change the behavior of GMF or Xtext or relax the requirements. A modification of the architecture of Xtext or GMF, e. g. by introducing a semi-canonical policy for GMF editors, is beyond the scope of this work. Hence the synchronization is triggered on file saving where we can be sure that the model is “final” and valid.

Both Xtext and GMF are modular, powerful frameworks. However internally they make various conventions and have a complex underlying structure. While using these promising frameworks generally yields in ever so quick results that are presentable, the more specialized the requirements are, such as their full synchronization, the closer we are likely to run up against their limits.

9 Future Work

9.1 Textual Editor

Serializing Meta-data As a future work, the syntax can be enhanced to hold meta-information such as diagram layout specification. This enhancement is useful as it would enable the persistence of layout information, specific to a syncChart. However a clean separation of the base language and the meta-data should be maintained and the meta-information should not be enforced by the grammar. Furthermore, the meta-data could contain model and version specifications, as was done in KIT. KIT is designed to support different Statechart dialects, and contains header-like state arguments that specify the dialect of the Statechart.

Inter-level Transitions The current naming convention specifies that states have identifiers that are unique in their region. This effectively sets the scope of a state to its region, hence inter-level transitions are not supported by acKITS. Hence inter-level transitions are already supported by ThinKCharts, altering the current naming conventions would be sufficient to add this feature.

Structuring via External Components The structuring of a snycChart via references to external states—states that were declared in external resources—is a conceivable enhancement to the current grammar. In the current version of Xtext, an external object can be referenced by using a String value that has been assigned to its importURI attribute. The predefined name, importURI, of the attribute is an Xtext convention. In this work, we have seen another such convention: The cross-referencing of objects can be done via a name attribute that has to be a String. Although such conventions provide a convenient way to add new language features, they also require an adaptation of the underlying meta-model. An alternative approach can be to alter the linking semantic to support inter-document references.

Further Syntax Enhancements Additional to the features that have been detailed in this work, the grammar of KITS supports renamings of typed values, declaration of textual states and integration of host code in actions. As a future work, it can be enhanced to support various language features such as subsequent variable declarations in a single assignment—as done in Esterel.

Outline Menu Actions A late enhancement of the outline view implementation in Xtext supports adding menu actions to the outline. Enhancing the outline view to provide model editing actions, such as removing or renaming SyncCharts elements, will result in an additional editor that enables a structure-based editing of the underlying model. As this new feature is not investigated in the scope of this work, the scope and amount of time needed to accomplish this objective cannot be estimated.

9.2 Synchronization

Semi-canonical Edit Policy for GMF The synchronization infrastructure depends on the limits of both Xtext and GMF. However, especially GMF has grown highly complex hence its core implementation patterns are not extensible in the scope of this work. The canonical edit policy is such a pattern. This policy interferes with a full-synchronization of Xtext and GMF. There is research to add a semi-canonical editing behavior to GMF but at this point this is still an open question.

Xtext Parser Wrapper The parser that is generated by Xtext does not have to be used in an Xtext environment. It is easily possible to utilize it in different contexts, such as a GMF editor. However to utilize other Hence a possible enhancement could be to provide an adapter for the Xtext parser that enables its integration into other editors.

EMF Index Roughly, EMF Index is a new project that aims to provide means to query EMF objects that are defined “anywhere”, e.g. also in a GMF editor. Depending on this project, the Xtext editor in this work can be enhanced to reference objects in ThinKCharts, e.g. in an action that opens a diagram element from its textual description in the KITS file.

10 Bibliography

[1] 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.

[2] John Arthorne and Chris Laffra. *Official Eclipse 3.0 FAQ*. Addison-Wesley, July 2004.

[3] Frederick P. Brooks Jr. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[4] Eclipse Europe Summit 2008. *Combining Graphics and Text in Modeling Tools*. `http://www.slideshare.net/schwurbel/combining-graphical-and-textual`.

[5] Eclipse Foundation. *Eclipse Corner Article: Integrating EMF and GMF Generated Editors*. `http://www.eclipse.org/articles/article.php?file=Article-Integrating-EMF-GMF-Editors/index.html`.

[6] Eclipse Foundation, Ottawa. *Graphical Modeling Framework Tutorial*. `http://wiki.eclipse.org/index.php/GMFTutorial`.

[7] Eclipse Foundation. *Xtext User Guide*. `http://www.eclipse.org/Xtext/documentation/0_7_2`.

[8] Eclipse Foundation, Ottawa. *Eclipse Platform Technical Overview*, version 3.1 edition, 19. April 2006. `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html`.

[9] Eclipse Software Foundation. Eclipse homepage, 2008. `http://www.eclipse.org/`.

[10] Dave Steinberg Ed Merks. Models to code with the eclipse modeling framework. In *EclipseCon 2005*, 2005. `http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial11final.pdf`.

[11] Sven Efftinge. *openArchitectureWare User Guide*. Eclipse Foundation, 2008. `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html`.

[12] Sven Efftinge and Markus Voelter. oAW xText: A framework for textual DSLs. In *Eclipse Summit Europe*, Esslingen, Germany, October 2006. http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf.

[13] Esterel Technologies. Esterel studio, 2008. http://www.esterel-eda.com.

[14] Thomas Huining Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference (SCSC 2003), Student Workshop*, pages 147–166. The Society for Computer Modelling and Simulation, July 2003. Montréal, Canada.

[15] Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2007. http://www.martinfowler.com/articles/languageWorkbench.html.

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

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[18] Richard Gronback. Graphical Modeling Framework, 2008. http://www.eclipse.org/gmf/.

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

[20] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007), Nashville, TN, USA, October 2007*, 2007.

[21] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988. http://doi.acm.org/10.1145/42411.42414.

[22] David Harel. Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25(1):8–20, 1992.

[23] David Harel. Statecharts in the making: A personal account. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, New York, NY, USA, 2007. ACM.

[24] Randy Hudson. Graphical Editing Framework, 2003. http://www.eclipse.org/gef/.

[25] Lars Kühl. Transformation von Esterel nach SyncCharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lku-dt.pdf.

[26] Ivan Kurtev. *State of the Art of QVT: A Model Transformation Language Standard*. Springer-Verlag, Berlin, Heidelberg, 2008.

[27] F. Maraninchi and Y. Rémond. Argos: An automaton-based synchronous language. *Computer Languages*, 27(27):61–92, 2001.

[28] Mathworks, Inc. Stateflow—design and simulate state machines and control logic, 2008. http://www.mathworks.com/products/stateflow/.

[29] Franco Mazzanti. *UMC User Guide (Version 2.5)*. Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo" (ISTI), Pisa, Italy, 2003.

[30] Ed Merks. Eclipse Modeling Framework, 2009. http://www.eclipse.org/modeling/emf/.

[31] Object Management Group. http://www.omg.org/mda.

[32] Object Management Group. http://www.omg.org/.

[33] Object Management Group. UML Homepage. http://www.uml.org/.

[34] Object Management Group. XML metadata interchange (XMI) specification, version 2.0, May 2003. http://www.omg.org/docs/formal/03-05-02.pdf.

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

[36] Roland Petrasch, Wolfgang Neuhaus, and Florian Fieber, editors. *Modellbasierte Software-Entwicklung für eingebettete Systeme*. Logos Berlin, July 2007.

[37] Steffen Prochnow. *Efficient Development of Complex Statecharts*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, Germany, 2008. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/spr-diss.pdf.

[38] Steffen Prochnow and Reinhard von Hanxleden. Comfortable modeling of complex reactive systems. In *Proceedings of Design, Automation and Test in Europe (DATE'06)*, Munich, Germany, March 2006.

[39] Matthias Schmeling. Thinkcharts—the thin KIELER SyncCharts editor. Master's thesis, Christian-Albrechts University of Kiel, 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf.

[40] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse).* Addison-Wesley Longman, Amsterdam, January 2009.

[41] Junichi Suzuki and Yoshikazu Yamamoto. Making uml models interoperable with uxf. In *The Unified Modeling Language: Beyond the Notation*, pages 78–91. Springer-Verlag, 1998.

[42] Petter Graff Vladimir Bacvanski. Mastering the eclipse modeling framework. In *EclipseCon 2005*, 2005. `http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf`.

[43] Markus Völter. Werkzeuge für domänenspezifische sprachen. March 2009. `http://www.heise.de/developer/artikel/Werkzeuge-fuer-domaenenspezifische-Sprachen-227190.html`.

[44] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.

[45] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[46] Mirko Wischer. Textuelle Darstellung und strukturbasiertes Editieren von Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/miwi-dt.pdf`.

[47] World Wide Web Consortium (W3C). State Chart XML (SCXML): State Machine notation for control abstraction, February 2007. `http://www.w3.org/TR/scxml/`.

A Textual Syntax Proposals

Here we present few informal proposals of how textual SyncCharts representations could look like. First the representation in KIT is given. The same syncChart that is presented in Fig. A.1 is used in all instances in order to allow a quick comparison.

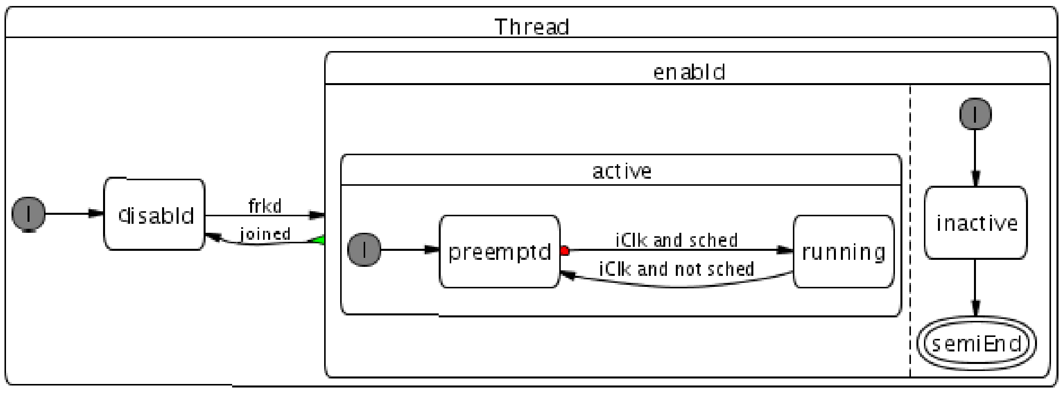


Figure A.1: Sample syncChart

Listing A.1: Sample syncChart in KIT

```
statechart Thread[model="Esterel Studio";version="5.0";]{
 input forked ;
 input joined ;
 input  instrClk ;
 input schedule ;
 input startTick ;
 input delay ;
 {
  disabled −>enabled[label="forked";];
  −>disabled;
  enabled{
    active {
     running−>preempted[label="instrClk and not schedule";];
     preempted−>running[type=wa;label="instrClk and schedule";];
     −>preempted;
    };
    active −>inactive[type=wa;label="startTick";];
    inactive −>active[type=wa;label="delay";];
    −>active;
  };
  enabled−>disabled[type=wa;label="joined";];
 };
};
```

Listing A.2: Proposal: JiTTY

```
Thread :
    sub: -> disabled
    sub: enabled
    -> (disabled)forked(enabled)
    >-> (enabled)joined(disabled)
;
enabled :
    sub: active
||
    sub: <new>
;
active :
    sub: -> preempted
    sub: running
    o-> (preempted)instrClk & NOT schedule(running)
    >-> (running)instrClk & schedule(preempted)
;
<new> :
    sub: -> inactive
    sub: ((semiEnd ))
    -> (inactive)(semiEnd)
;
```

Listing A.3: Proposal: KITTY

```
[Thread
 |->(disabled)-forked->[enabled
                     |[ active
                    |->(preempted)o-[instrClk & NOT schedule]->(running),
                    (running)-[instrClk & schedule]->preempted]
                    |->(inactive)->((semiEnd))],
 (enabled)>-joined->disabled]
```

Listing A.4: Proposal: DIRKIT

```
Thread:
    disabled . ini  −forked−> enabled;
   enabled >−joined−> disabled;
Thread\enabled:
    active ;
||
    inactive . ini  −> semiEnd.fin;
Thread\enabled\active:
   preepmtd. ini  o−instrClk & NOT schedule−> running;
   running  −instrClk & schedule−> preepmt
```

Listing A.5: Proposal: M@ILK->T

```
Thread|>
  −>o FROM:disabled
       [forked]  X [joined]
       TO: enabled
Thread\enabled\active|>
  −>o FROM: preempted
       [ instrClk  & NOT schedule] X [instrClk & schedule]
       TO: running
Thread\enabled\<newstate>|>
  −>o FROM: inactive
       []  X −
       TO: ((semiEnd)) <−
```

Listing A.6: Proposal: KITeX

```
\begin{state}{Thread}
 \state[ initial ]{ disbld }
 \begin{state}{enabled}
  \begin{state}{active}
   \state[ initial ]{preempted}
   \state[ final ]{running}
   \trans[s]{preempted}[instrClk & schedule]{running}
   \trans{running}[ instrClk & not schedule]{preempted}
  \end{state}
  \||
  \state[ initial ]{ inactive }
  \state[ final ]{semiEnd}
  \trans{ inactive }{semiEnd}
 \end{state}
\trans{ disbld }[ frk ]{enabled}
\trans[nt]{enabled}[joined ]{ disbld }
\end{state}
```

B Eclipse Configuration

Listing B.1: Eclipse configuration

```
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
EMF, GMF, GEF
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
"EMF − Eclipse Modeling Framework Runtime and Tools"
org. eclipse .emf (2.5.0. v200906151043)
"Graphical Editing Framework GEF"
org. eclipse . gef (3.5.0. v20090528−1511)
"Graphical Modeling Framework"
org. eclipse .gmf (1.2.0. v20090615−0700)
"GMF Models Bridging Plug−in"
org. eclipse .gmf.bridge (1.1.0. v20090528−1000)
"Trace Model"
org. eclipse .gmf.bridge. trace (1.0.200. v20090520−1343)
"GMF Tooling UI"
org. eclipse .gmf.bridge. ui (1.2.1. v20090812−1620)
"GMF Dashboard"
org. eclipse .gmf.bridge. ui .dashboard (2.0.0. v20090114−0940)
"GMF GenModel and Code Generation"
org. eclipse .gmf.codegen (2.2.1. v20090812−1620)
"GMF Tooling Commons Plug−in"
org. eclipse .gmf.common (1.1.100.v20090420−1925)
"GMF Common Core"
org. eclipse .gmf.runtime.common.core (1.2.0.v20090403−1720)
"GMF Diagram Core"
org. eclipse .gmf.runtime.diagram.core (1.2.1. v20090729−2029)
 "GMF Draw2d Additions"
org. eclipse .gmf.runtime.draw2d.ui (1.2.1. v20090825−1800)
"GMF Commands"
org. eclipse .gmf.runtime.emf.commands.core (1.2.0.v20090403−1720)
"GMF GEF Additions"
org. eclipse .gmf.runtime.gef . ui (1.2.0. v20090520−1343)
"GMF Notation Model Support"
org. eclipse .gmf.runtime. notation (1.2.0. v20090521−1925)
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
TMF, Google Guice, KIELER
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
Id: org. eclipse .xpand, Version: 0.8.0. v200911101353,
Id: org. eclipse .xpand.ui, Version: 0.8.0. v200911101353,
Id: org. eclipse .xtend, Version: 0.8.0. v200911101353,
Id: org. eclipse .xtend.check. ui, Version: 0.8.0. v200911101353,
Id: org. eclipse .xtend.typesystem.emf, Version: 0.8.0. v200911101353,
Id: org. eclipse .xtend.typesystem.emf. ui, Version: 0.8.0. v200911101353,
Id: org. eclipse . xtext, Version: 0.7.2, (Action grammar)
Id: org. eclipse . xtext, Version: 0.8.0. v200911101539, (KITS grammar)
Id: org. eclipse . xtext. activities , Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext .common.types, Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext .common.types.ui, Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext . generator, Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext . junit , Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext . logging, Version: 1.2.15,
Id: org. eclipse . xtext . ui .common, Version: 0.8.0.v200911101539,
Id: org. eclipse . xtext . ui . core, Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext . util , Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext .xtend, Version: 0.8.0. v200911101539,
Id: org. eclipse . xtext . xtext . ui , Version: 0.8.0. v200911101539,
Id: com.google.guice, Version: 1.0.1. v200911101451,
Id: de. itemis . xtext . antlr , Version: 0.7.2.200908121408, (Action parser)
Id: de. itemis . xtext . antlr , Version: 0.8.0. v200911110853, (KITS parser)
Id: de.cau.cs. kieler . syncccharts , Version: 0.1.0
```