Christian-Albrechts-Universität zu Kiel

Bachelor Thesis

Interactive Transformations for Visual Models

Ulf Rüegg

March 11, 2011

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl.-Inf. Christian Motika

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

Model transformations are an integral part of Model Driven Engineering (MDE). But these transformations are mostly executed silently and at once in the background. The user does not gain any insight into the process of the transformation, and the concrete coherences between the input model and its transformed version do not become clear. In this thesis, an approach to visualize a transformation is presented. The overall transformation is broken down into *steps* of a certain granularity. As a result the transformation can be executed *step-wise* by successively performing single steps. Each intermediate step is illustrated visually in order to improve the overall comprehensibility. Erroneous transformations can be examined easier as it is possible to locate the part of the transformation where the error is introduced more precisely.

SyncCharts and Esterel are graphical and textual programming languages, respectively, used for the design of reactive systems. A transformation between the two languages is desirable in order to utilize each representation's advantages. A graphical representation improves comprehension while it is faster to edit a textual representation. An implementation of a transformation that transforms an Esterel program into a SyncChart and which is executable in steps is presented in the context of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project.

Contents

List of Figures

Listings

Abbreviations

ATL	Atlas Transformation Language
BNF	Backus–Naur Form
CEC	Columbia Esterel Compiler
CPN	Colored Petri Net
DSL	Domain Specific Language
EBNF	Extended Backus–Naur Form
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
IBM	International Business Machines
IDE	Integrated Development Environment
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KiVi	KIELER Viewmanagement
KlePto	KIELER leveraging Ptolemy Semantics
M2M	model-to-model
MDE	Model Driven Engineering
MOF	Meta Object Facility
MWE	Modeling Workflow Engine
MWE2	Modeling Workflow Engine 2
oAW	openArchitectureWare
OCL	Object Constraint Language

OMG	Object Management Group
SC	Synchronous C
SSM	Safe State Machines
TGG	Triple Graph Grammars
ThinKCharts	Thin KIELER SyncCharts Editor
UML	Unified Modeling Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XSD	XML Schema Definition
XML	Extensible Markup Language

1 Introduction

These days, *embedded* [LS11], *real time*, and *reactive* systems are omnipresent. Reactive systems are in continuous interaction with the environment. They receive information about the environment's current state via sensors and react with a certain functionality. Such systems are, for instance, used in real time systems. Real time systems focus on predictability and correctness rather than performance, and have to consider the physical environment. Embedded systems perform computations in the context of other technical systems without the user noticing.

To illustrate consider the following systems: A mobile phone, the latest TV set, a modern kitchen, or many parts of an automobile. Each of these devices contains an electronic component controlling its function.

Furthermore, all of these devices have high requirements concerning safety, security, reliability, and usability. They are getting more complex and their development gets even harder, with relation to the previously mentioned requirements. Maintainability, readability, and adaptability are just some of the demands for code, diagrams, and everything else that is produced during the process that leads to a finished product. However, as the systems get more complex, their code and diagrams get more complex, too. Several unstructured and hardly readable diagrams combined with poor tooling are presented by von Hanxleden and Fuhrmann [vHF10]. Diagrams like these make it difficult to consider all interdependencies and to gain a good overview. Thus, the need for appropriate programming languages and proper tools to support the developer in his work is quite obvious.

Examples of such programming languages, designed for the development of reactive systems, are *Esterel* [Ber00] or the graphical programming language *SyncCharts* [And96]. Further noteworthy languages are *Lustre* [HR01] and *Signal* [GGBM91]. Some tools for the development of reactive systems are *LabView* [Nat08], *SCADE* [Est06], and *Ptolemy II* [EJL+03].

Comparing graphical and textual editing yields the following result. On the one hand, the advantage of a graphical programming language is a faster understanding of the essential meaning of a diagram and its context, especially for an outside person. On the other hand, in most cases the practiced developer prefers to edit his code textually. This is usually faster and seems more natural [PTvH06]. Also, high quality revision management exists for textual code while the comparison of graphical models still faces some obstacles [Sch08]. For this reason a transformation between a graphical and a textual representation is desirable. However, it is mandatory that semantics can be preserved and the expressiveness of either representation can be covered by using the other representation's syntax.

1.1 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] is an academic research project that aims at improving the pragmatics of model based design. In this context pragmatics relate to all practical aspects of handling a model. This includes all facets of editing as well as browsing [FvH10b]. Main aspects are, for example, *automatic layout*, *structure-based editing*, and proper *view-management* [FvH10a, FvH10b], all of which are supposed to support the developer in his work.

KIELER is created by using Eclipse and its modeling facilities[2] and is designed as a testbed for new concepts of model-based design. Figure 1.1 shows an overview of all components of KIELER. The projects are structured by their contribution either to pragmatics, or to semantics, or to syntax.

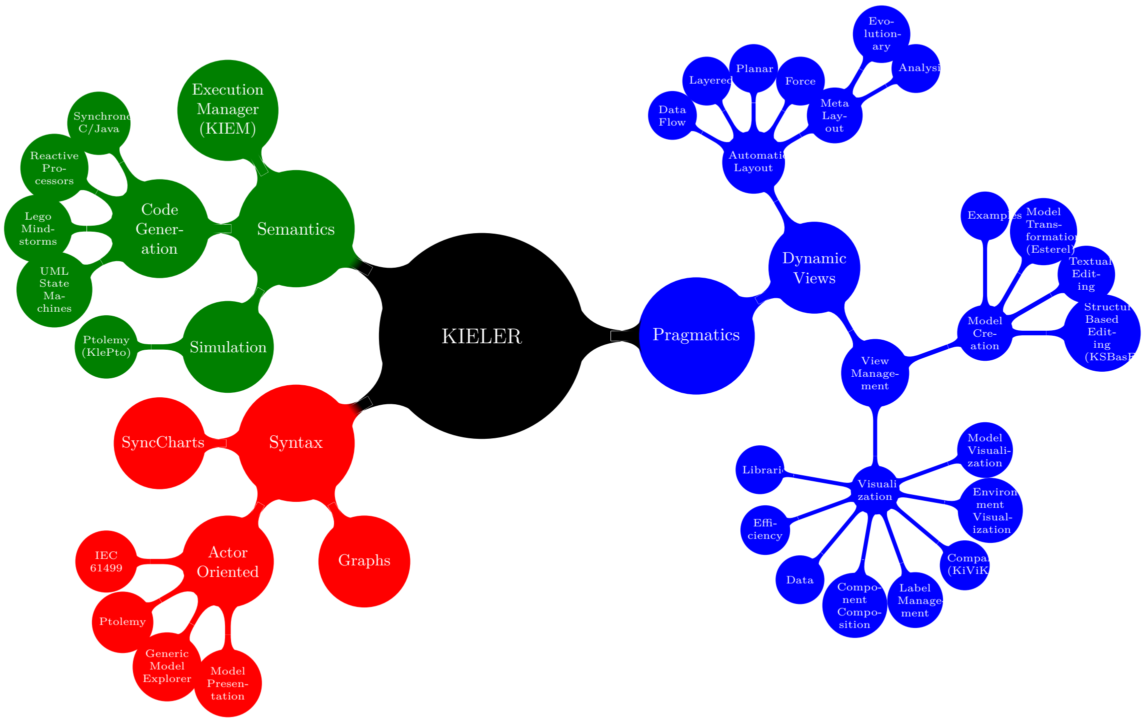


Figure 1.1: Overview of KIELER [Fuh11]

[1] http://www.informatik.uni-kiel.de/rtsys/kieler/
[2] http://www.eclipse.org/modeling/

1.2 Problem Statement

The main goal of this thesis is to examine a solution and to give a possible implementation for transforming one representation into the other one. It is not of importance whether the transformation is a text-to-text or a text-to-graphic representation. The exemplary implementation presented in this thesis sets its focus on the text-to-graphic transformation of Esterel code to SyncCharts.

With regard to the tooling, the application of any transformation should appear intuitive and has to be self-explaining as it aims at supporting the developer in his daily work. Therefore, it has to be integrated seamlessly into Eclipse and has to follow well-known editing paradigms.

To provide visual feedback during a transformation it is necessary to split the overall process into *steps*, which are basically a specified extract of the overall transformation. Thus, it is possible to perform single steps and visualize each intermediate stage on the way from the source representation to the final result. For convenience different modes of executing such a transformation should be offered. *Step-wise* execution with visual feedback aims at supporting the understanding of the transformation as it is performed slowly and the transformation's line of action becomes visible. It can also be used for *debugging*, the locating of mistakes made by the programmer. *Headless* execution, the execution without any visual feedback, serves as the natural choice to transform models quickly. Also, it is mandatory to provide a mechanism to roll back single transformation steps in case the developer wants to retrace an explicit transformation step.

First, a foundation for the convenient use of Esterel code is essential as motivated in the introduction. This includes both, integrating it into the Eclipse context, and providing commonly known programming paradigms like code completion, navigation, and formatting, on-the-fly syntax, and error checking.

Second, a base for the step-wise execution of arbitrary by using Eclipse technology has to be developed.

Third, the Esterel to SyncCharts transformation, which serves as the primary example, has to be implemented in a form that is easily executable. The execution modes mentioned above have to be provided. As much existing work as possible, provided by KIELER and possibly other sources, should be reused.

Fourth, because the sole application of generic transformation rules yields a rather canonical and verbose SyncChart, a further post-processing is necessary to bring it into a humanly better readable form. Fortunately, this can be done by some optimization rules that simplify the SyncChart into a semantically equivalent, yet more intuitive and comprehensible one.

Finally, adequate testing facilities have to be provided for everything that is developed. To sum up the following tasks can be determined:

1. Provide facilities to handle Esterel code in the context of KIELER.

2. Develop an approach to apply and to handle transformations visually.

3. Implement both, the SyncCharts to Esterel transformation and the SyncCharts optimization.

4. Define proper testing criteria and provide rudimentary tests.

1.3 Structure of this Document

The last part of Chapter 1 introduces Esterel and SyncCharts as it is essential for the reader to know the basic concepts of these languages to understand the approaches presented in this thesis.

Chapter 2 presents work related to this thesis, e. g., the first thoughts on the Esterel to SyncCharts transformation by Prochnow et al. [PTvH06]. Additionally, handling of model transformations and solutions to introduce visual debugging are considered.

Chapter 3 introduces all used technologies that are used for the implementation. Among these are basic Eclipse mechanisms, Xtext[3], Xtend[4], and several used features of KIELER.

Afterwards, the integration of Esterel into the KIELER context is presented in Chapter 4, including the adaption of an existing Esterel grammar into Xtext.

Chapter 5 starts with introducing a generic approach to deal with the Xtend-based model-to-model (M2M) and in-place transformations. Also, the theoretical foundations of the Esterel to SyncCharts transformation rules, as well as the SyncCharts optimization rules, are presented in further detail rule by rule. The concrete example implementations are discussed in Chapter 6.

Chapter 7 depicts the used validation approaches and presents several results produced by the implementation.

Finally, Chapter 8 summarizes the results of this thesis and gives a short outlook on topics worth being discussed in the future.

1.4 Esterel v5

As stated in the introduction the Esterel language is used for the approaches and implementations presented in this thesis. Therefore, the basic constructs of Esterel have to be introduced.

Esterel v5 is described by Claus Traulsen [Tra10] in the following way.

> "Esterel [BC84, PBEB07] is an imperative synchronous language. Esterel programs communicate with the environment and internally via signals, which are either present or absent during one instant. Signals are set present by the `emit` statement and tested with the `present` test. Local signals can be declared by using the `signal` statement. Signals

[3]http://www.eclipse.org/Xtext/
[4]http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html

> are absent per default: A signal is only present in a tick if it is emitted in this tick. Esterel statements can be either combined in sequence (`;`) or in parallel (||). The `loop` statement simply restarts its body when it terminates. All Esterel statements are considered instantaneous, except for the `pause` statement, which pauses for one instant. The `suspend` statement suspends its body when a trigger signal is present. Exception handling is done via named exceptions, called traps. The `trap` statement declares the scope of an exception. When the exception is raised with an `exit` statement, the control is transferred to the end of this trap scope. If multiple, different exceptions are raised in the same tick, the trap with the outermost scope is taken.
>
> From this small set of *kernel statements* derived statements are declared. This includes simple statements like `halt=loop pause`, which stops forever, but also the `abort` and `weak abort` statements, which terminate their bodies when the trigger signal is present. Weak abortion permits the execution of its body in the instant the trigger signal becomes active, strong abortion does not. Both kinds of abortions can be either immediate or delayed. The immediate version already senses for the trigger signal in the instant its body is entered, while the delayed version ignores it during the first instant in which the abort body is started.
>
> Beside the pure status, a signal can also contain an additional value. This value is persistent over ticks, if the signal is not emitted. If a valued signal is emitted multiple times within a tick, a commutative and associative function must be given to combine the signals. This ensures that the signal value is unique within a tick. Esterel also has a notion for variables, which can have different values within a tick. However, they cannot be read and written in parallel, hence all race conditions are syntactically excluded."

Listing 1.2a shows the `ABRO` program, which is the *Hello World* of Esterel. In line 1 a `module` called `ABRO` is defined. Signals are defined in line 3 and 4; `A`, `B` and `R` as input, `O` as output signal. In line 6 the program starts a `loop` which is reset upon the presence of `R`. Line 7 specifies two parallel `await` statements that wait for the occurrence of `A`, respectively `B`. The `emit` statement in line 8 emits `O` after the termination of the previous parallel block. Finally, in line 11 the module is completed by the `end module` keyword. For further information concerning the behavior of each statement see the Esterel Primer [Ber00].

For the sake of completeness, it should be mentioned that the current version of Esterel is v7. It comes with new constructs that are especially useful in hardware design, for instance, arrays and bit vectors. The approaches presented here focus on v5, hence v7 will not be discussed in further detail. For an overview of the difference see Traulsen [Tra10].

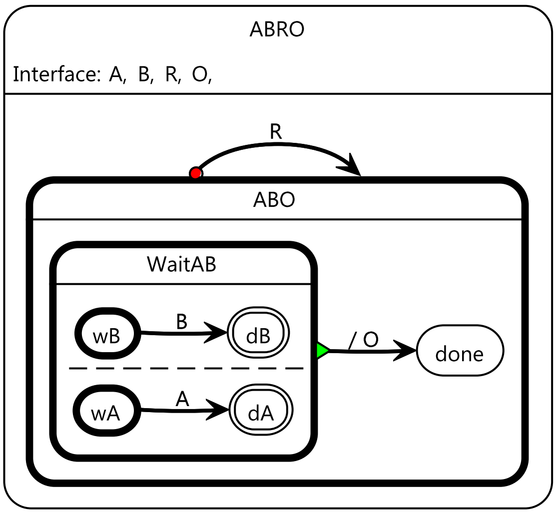
```
module ABRO:

input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O;
each R

end module
```

(a) Esterel



(b) SyncCharts

Figure 1.2: The `ABRO` program

1.5 SyncCharts

Chapter 5 explains the transformation of Esterel to SyncCharts. Chapter 6 presents the actual implementation. Therefore, it is essential to know the syntax and basic semantics of SyncCharts.

Claus Traulsen [Tra10] describes SyncCharts as follows:

> "SyncCharts (also called *Safe State Machines*) are a Statecharts dialect with a synchronous semantics that strictly conforms to the Esterel semantics. [...]
> A procedural definition of the semantics of SyncCharts is given by [And03]. The basic object in SyncCharts is a *reactive cell*, which is a state with its outgoing transitions. Reactive cells are combined to *state-transition graphs*, called *state regions* in other Statecharts dialects. These are flat automata with exactly one *initial state*, which is indicated by a bold border. A *macro-state*, like the `control` state on the example, consists of one or more state-transition graphs. Additionally, SyncCharts can contain *textual macro-states*, which consist of plain Esterel code. States can also have *internal actions*: `on entry`, `on exit` and `on inside`. An `on exit` action is executed whenever the state is left, whether this is done via an outgoing transition or a parent state of this state is left itself. SyncCharts inherit the concept of signals and valued signals from Esterel. Hence a transition trigger can consist of an event, which tests for presence and absence of values, and a conditional, which may compare numerical val-

> ues. Characteristic for SyncCharts are the different forms of preemption, expressed by different state transition types. *Weak and strong abortion* transitions as well as *suspension* can be applied to macro-states. Strong abortions are indicated by a red dot on the arrow tail, like the transitions that restart the controller for each valid input in the example. Weak abortions are drawn as plain arrows. A variant of weak abortions are *weak-delayed abortions*, which only activate the target state in the next instant. They make sure that states are not transient, what can both simplify the compilation and the understanding of a SyncCharts. A macro-state can either be left by an abortion, which has an explicit trigger, or by a *normal termination*, which is taken if the macro-state enters a terminal state. Normal terminations are indicated by a green triangle at the arrow tail. Analogously to Esterel, all transitions can either be immediate or delayed, where a delayed transition is only taken if the source state was already active at the start of an instant. In contrast, immediate transitions may be taken as soon as the state becomes active; this enables the activation and deactivation of a state multiple times within one instant. Delayed transitions can also be *count delayed*, i. e., the trigger must have been evaluated to true for a specific number of times, before the transition is enabled. When a state has more than one outgoing transition, a unique *priority* is assigned to each of them, where lower numbers have higher priority. Weak abortions must have lower priority than strong abortions, and if a normal termination exists, it always has the lowest priority."

The `ABRO` program should serve as an illustration, see Figure 1.2b. In the upper part of the SyncChart the four signals `A`, `B`, `R` and `O` are defined. The inner state `ABO` has a self-transition with `R` as the trigger, hence the transition is taken upon the presence of `R`. The `WaitAB` state models the waiting for the signals `A` and `B` as parallel regions. Within those regions the final state is reached by a transition with `A`, respectively `B` as trigger. Finally, upon termination of the `WaitAB` state `O` is emitted as an effect of a normally terminating transition.

2 Related Work

There are two essential aspects in this thesis:

1. Esterel is considered with relation to proper tooling, and the synthesizing of SyncCharts from Esterel is presented.

2. Model transformations are inspected, especially a possibility to execute a transformation step by step. The proper visualizing of intermediate steps is addressed as well as the possibility to debug transformations.

2.1 Esterel

The Columbia Esterel Compiler

The Columbia Esterel Compiler (CEC)[1] [EZ07] is an open-source Esterel compiler. It supports Esterel v5 and can translate Esterel source code into a C program. To recognize the Esterel constructs an Esterel grammar is used. The grammar is represented in Extended Backus–Naur Form (EBNF) and serves as one of the references for the construction of a grammar in this thesis.

During the translation several passes are performed, e. g., one pass is the creation of an *abstract syntax tree*. Each pass' result can be saved individually and is represented by an XML file. Furthermore, the CEC allows to expand an Esterel program containing several `module`s into a single `module`. This is done by replacing each `run` statement by the corresponding `module`. This expansion provides a straightforward solution to handle multiple modules but reduces modularity and readability of the Esterel code.

2.1.1 Statecharts to Esterel

Seshia et al. propose a translation of Statecharts to Esterel [SSBD99]. It aims at the formal verification of Statecharts by using tools created for the verification of Esterel programs.

Esterel Studio

Esterel Studio[2] is an environment for the design of control-flow models, see Figure 2.1 for a screenshot. The tool supports the simulation of these models' functionality.

[1] http://www.cs.columbia.edu/~sedwards/cec/
[2] http://www.esterel-technologies.com/

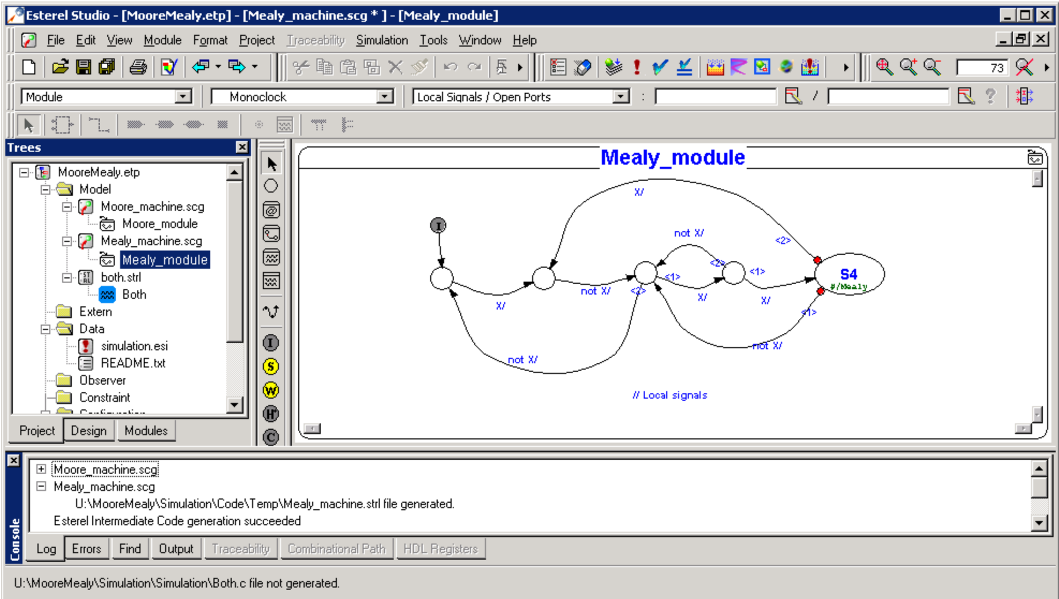


Figure 2.1: Esterel Studio GUI [Mot09]

It also supports formal verification and automated generation of Very High Speed Integrated Circuit Hardware Description Language (VHDL) and *Verilog* code.

Esterel Studio also provides a translation from Safe State Machines to Esterel. The translation was proposed by André [And03].

2.1.2 Synthesizing SyncCharts from Esterel

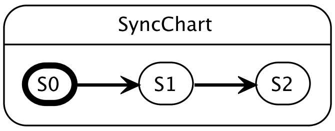
First ideas concerning the topic of the synthesization of Safe State Machines (SSM) from Esterel were presented by Prochnow et al. [PTvH06]. An exemplary implementation was provided by Kühl [Küh06] who also verified the correctness of the presented approaches with formal proofs. The implementation was done in the context of Kiel Integrated Environment for Layout (KIEL). However, no automatic testing of the implementation was performed.

KIEL[3] [PvH07] is the predecessor of KIELER and is a standalone Java application possessing already many of the features provided by KIELER. KIEL lacks modularity, reusability, and maintainability due to the implementation by using only native Java. These problems are solved in KIELER with the help of the generic concepts provided by Eclipse, e. g., the plug-in mechanism.

A first implementation of utilizing the modularity of KIELER was done by Lukaschewitz [Luk10].

Both implementations mentioned do not provide any visualization of the actual transformation. In both cases the Esterel file is taken as an input, and the completely transformed SyncChart is presented afterwards. This work tries to enhance the

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



(a) SyncCharts

```
S0:
    PAUSE;
    GOTO(S1);
S1:
    PAUSE;
    GOTO(S2);
S2:
    HALT;
```

(b) SC–Code

Figure 2.2: Small example of the conversion of a SyncChart to Synchronous C (SC) code [Ame10]

comprehensibility of the transformation by providing a mechanism to gain insight into intermediate steps.

2.2 Model Transformations

2.2.1 Transformation Languages

Matzen [Mat10] gives a small overview concerning the topic of model transformations and evaluates several transformation languages to choose the correct language for a certain use case. The selection of Xtend as the transformation language used in this thesis bases on Matzen's explanations. A disadvantage of Xtend is the lack of debugging facilities. The Atlas Transformation Language, also discussed by Matzen, comes with built-in, well-known debugging facilities [JABK08], a simple *break point* system to stop the control flow at a certain line of code.

In his work Matzen introduces a framework for *structure-based editing*. It allows the developer to apply ready-made operations to a certain model. The changes to the model are applied immediately so that the user receives direct feedback.

2.2.2 KIELER Transformations

There exist several M2M transformations in KIELER.

Amende [TAvH11, Ame10] presents a way to synthesize SC [vH09] code out of SyncCharts, see Figure 2.2 for an example. SC is an extension of the C language enriched by *deterministic concurrency* and *preemption*. Former approaches transformed SyncCharts to Esterel and the Esterel code to C code. This approach has several drawbacks, which can be prevented by using SC, e. g., the fact that it is hard to see the coherences between the generated C code and the SyncChart.

Motika developed KIELER leveraging Ptolemy Semantics (KlePto)[4] [Mot10, MSF+11],

[4] http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KlePto

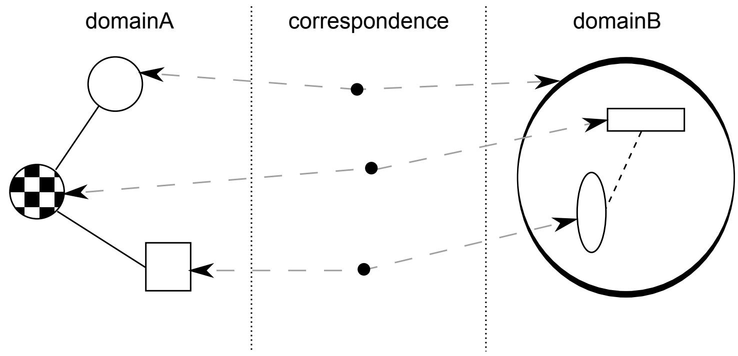


Figure 2.3: Schematic Triple Graph Grammars (TGG)

a framework that is capable of transforming a SyncChart to Ptolemy to simulate the functionality of a SyncChart by using Ptolemy semantics.

Those two transformations are executed without the user having the possibility to see any intermediate steps. A step-wise execution might be useful, especially in terms of debugging.

To reach this target it has to be evaluated if it is possible to split the transformation process into pieces, and which step granularity seems to be reasonable. For instance, the SC generation might be visualized by transforming a SyncChart's states one after another and building up the SC code piece by piece.

2.2.3 Triple Graph Grammars

TGGs [KW07] are a formalism used to define the correspondences between two distinct models. A triple graph consists of two graphs representing the models and a third correspondence graph associating objects of the two models with each other. In contrast to Xtend, TGGs are not restricted to one direction of the transformation. A possible way to use TGGs for synchronizing models is presented by Giese and Wagner [GW06].

Such a synchronization might be interesting to connect an Esterel file to a SyncChart and to allow editing any of the two representations while synchronizing the other one on-the-fly.

2.2.4 Visual Debugging

Jacobs and Musial [JM03] introduce an approach to perform visual debugging by using the Unified Modeling Language (UML). To do so they link the program execution to an UML object diagram and enhance its presentation by graph layout, color encoding, and focus and context. Such a representation provides, additionally to the program state, further structural information.

Schoenboeck et al. [JS09] note the problems of missing facilities to support the debugging and the understanding of transformations. They try to solve the problem by using *Transformation Nets* which are a Domain Specific Language (DSL) on top of Colored Petri Nets (CPNs).

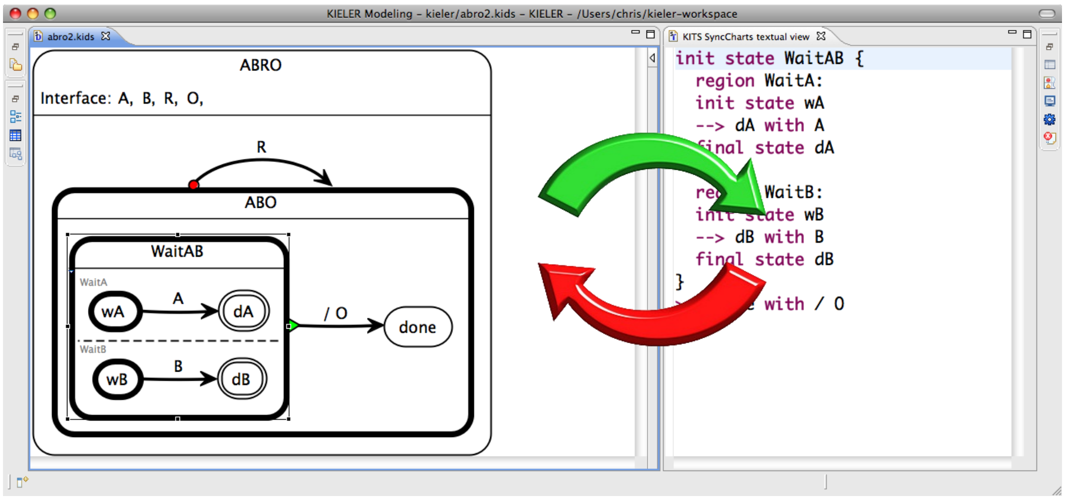


Figure 2.4: Integrating a graphical and a textual SyncCharts representation [Sch11]

2.2.5 Integrating Textual and Graphical Modeling

The advantages of providing a textual and also a graphical representation of a model were depicted in Chapter 1. An approach that goes beyond a plain transformation from one model to another one is presented by Schneider [Sch11]. Schneider discusses the seamless integration of textual and graphical modeling. This includes the immediate *synchronization* of either model upon any changes that were made in the other model. He also provides an exemplary implementation by using KIELER's SyncCharts editor and a textual representation for SyncCharts, which he created, see Figure 2.4 for a screenshot.

Such an approach is more complex than the transformation presented in this thesis and demands high similarity of the two representations used. Esterel and SyncCharts are convertible but each direction yields strongly deviating results, even if the semantics are the same.

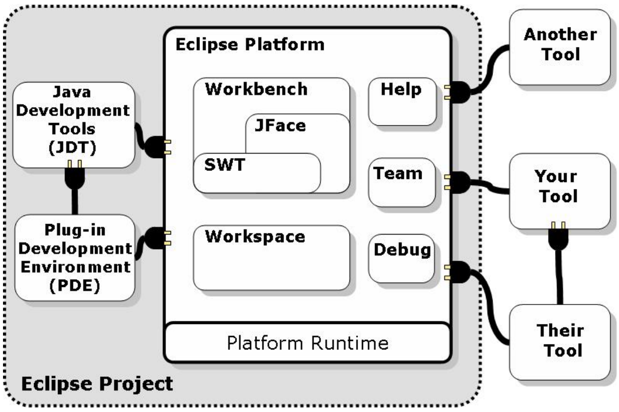


Figure 3.1: Eclipse's plug-in architecture as presented in its documentation[3]

3 Used Technologies

3.1 Eclipse

Eclipse[1] was originally developed by IBM as a *Java* Integrated Development Environment (IDE) and is written in the Java language itself. Nowadays, it is maintained as an open-source project and is the host for further languages, such as *C*, *C++* and *PHP*. Eclipse is also the host for a broad variety of modeling utilities[2]. The Eclipse plug-in mechanism allows incremental development of complex projects and will be discussed in the next sections.

3.1.1 Plug-in Mechanism

An Eclipse project can be developed by composing small pieces of functionality, so-called *plug-ins*. Each of these plug-ins depends on others and provides some new functionality. Figure 3.1 presents an overview of the plug-in architecture as presented in the Eclipse documentation[3]. This can also be consulted to gain further information on this topic.

Plug-ins enable a project to be more modular, expandable, and adaptable than plain Java applications. To be used by other plug-ins, a plug-in has to specify an interface or, in Eclipse terminology, *extension points*. These are XML schema definitions containing the extension point's specifications.

[1] http://www.eclipse.org
[2] http://www.eclipse.org/modeling/
[3] http://help.eclipse.org/galileo/nav/2

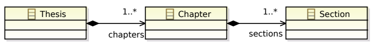
M3	meta-metamodel	MOF, Kermeta, KM3, Ecore
M2	metamodel	UML, Petri nets, Xtext, DSLs
M1	model	
M0	real world object	

Figure 3.2: The Meta Object Facility (MOF) architecture [Sch11]

3.1.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)[4] is used to define the abstract syntax of a certain domain in the form of metamodels. It adheres to the MOF[5] standard defined by the Object Management Group (OMG). The MOF standard defines a four layer architecture describing models.

M0: Physical objects of the reality, e. g., a screwdriver.

M1: A model of an M0 object, e. g., the description of the screwdriver.

M2: A metamodel of an M1 model, e. g., the specification of the model of a screwdriver.

M3: A meta-metamodel of an M2 metamodel, which specifies the structure of a metamodel.

These four layers are illustrated with some simple examples in Figure 3.2.

In EMF so-called *ecore* metamodels (M2) are defined by using class diagrams, annotated Java code, or XML Schema Definition (XSD). With the help of such a metamodel EMF can generate Java code to work with concrete models.

3.1.3 Xtext

The Xtext[6] project was integrated into Eclipse in 2009 and previously developed in the context of the openArchitectureWare (oAW) framework[7]. It aims at the creation of textual Domain Specific Languages (DSLs), which is done by providing an EBNF-like grammar. DSLs are small programming languages dedicated to a specific domain. In contrast to *General Purpose Languages*, such as Java, they are only intended to solve problems within their particular domain in a clear and compact way [Fow05].

[4] http://www.eclipse.org/emf/
[5] http://www.omg.org/mof/
[6] http://www.eclipse.org/Xtext/
[7] http://www.openarchitectureware.org/

Therefore, they are not capable to provide solutions for problems arising from a totally different domain.

Xtext is used in Chapter 4 to define a grammar for the Esterel language, and as a result it allows to use Esterel in the context of Eclipse and KIELER.

An additional advantage of Xtext is the automatic generation of a textual Eclipse editor with functionalities like *syntax highlighting* or *code completion*. These can be customized in nearly every way the developer wants it to be. Xtext also supports grammar inheritance enabling the developer to extend existing grammars or his own one to achieve good *modularization* and *reusability*.

Some of the basic elements of Xtext's syntax are shown below. To illustrate the use, Listing 3.1 shows a small example grammar which models a telephone book with persons. In line 5 and 6 the corresponding rule `PhoneBook` is specified, which starts with the string `"phonebook"` and expects at least a name and one `Entry`. An `Entry` is defined in line 8 and 9 and demands a `Person` as well as a `PhoneNumber`. For a `Person` fields for the name, optionally for the age and for the telephone number are specified. A `PhoneNumber` consists of a pair of integers indicating the region code and the actual number.

```
1  grammar de.cau.cs.kieler.pb.PhoneBook with org.eclipse.xtext.common.Terminals
2
3  generate phoneBook "http://www.cau.de/cs/kieler/pb/PhoneBook"
4
5  PhoneBook:
6     "phonebook" name=ID ":" entries+=Entry+;
7
8  Entry:
9     "person" "(" person=Person ")" number=PhoneNumber ";";
10
11 Person:
12    forename=ID surname=ID ("," "age" age=INT)?;
13
14 PhoneNumber:
15    code=INT "/" number=INT;
```

Listing 3.1: Xtext specification of a phone book

Listing 3.2 shows a possible instance of such a telephone book. Two `persons`, `Donald Duck` and `Scrooge McDuck`, are recorded with forename, surname, and age. One `person` is just recorded with forename and surname, obviously a female for whom it is decent not to mention her age. Furthermore, for each `person` a telephone number is stated.

```
phonebook Duckburg :
   person (Donald Duck, age 30) 121/1354;
   person (Scrooge McDuck, age 98) 212/9843;
   person (Daisy Duck) 121/2343;
```

Listing 3.2: A possible instance of the previously specified phone book

Cardinal Operators

- `?` optional element
- `*` arbitrary number of elements
- `+` at least one element

Assignment Operators

- `=` assignment of one element
- `+=` assignment of an element to a list of elements
- `?=` boolean assignment, assigns `true` if element exists, `false` otherwise

Further Rules

- `|` marks an alternative
- `[ ]` cross-reference to an existing element
- `{ }` simple action, enforces the creation of a specific type

3.1.4 Xtend

Besides Xtext, Xtend is a former project of the oAW framework, which has been integrated into Eclipse. Xtend is part of the Xpand[8] project. Originally, Xtend was used to define *extensions* to a metamodel that was used with Xtend. Nowadays it has evolved to a fully functional programming language. It is used particularly to define transformations based on metamodels[9]. Therefore, the term of an extension is equivalent to a method or a function.

Model transformations can be classified [MG06] with relation to the layers introduced in Section 3.1.2. A transformation is called *exogenous* if the M1 source model and the M1 target model are derived from different M2 metamodels, *endogenous* transformations base on the same M2 metamodel and are called *in-place* if the source and target M1 model is the same instance.

Xtend is used to implement the Esterel to SyncCharts transformation rules as well as the SyncCharts optimization rules presented in Chapter 5. The first-mentioned transformations are exogenous, they transform Esterel elements to SyncCharts elements. The latter are endogenous in-place transformations as they optimize the structure of a single M1 model.

[8] http://wiki.eclipse.org/Xpand
[9] http://blog.efftinge.de/2006/04/model2model-transformation-with-xtend_15.html

Expression Language

Xpand/Xtend comes with a simple expression language[10], which is a mixture of the Object Constraint Language (OCL) and Java. The most important functionality is described as follows:

1. Arithmetic and boolean operators or operations, e. g., `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`.
2. Collections providing known operations from functional and declarative programming languages, e. g., `{1,2,3}.select(e|e > 2)`.
3. Simple data types (Integer, String, Boolean, Real).
4. Conditional expressions (`if`, `switch`).
5. Chained expressions (`fstExpr -> sndExpr -> lastExpr`).
6. The `let` expression to instantiate local variables.

Example

There are three features of Xtend that are exemplified in this section.

First, the so-called *member syntax* allows the invocation of methods in two different ways. Either the method's name is written followed by the arguments in parentheses or the first argument of the method is written followed by a dot, the method's name and possibly further arguments.

Second, in case more than one method is specified with the same name but with arguments of distinct type, *multiple dispatch* decides which method to call. This decision is made at runtime and depends on the type of the passed arguments.

Third, Xtend's built-in collection types provide special operations, for instance, to select a subset of elements or to iterate over the whole collection. To iterate over a list it is sufficient to apply a method demanding an argument with the type of the collection's elements. In that case the method is called individually for each element.

Listing 3.3 shows a small transformation example based on the previous phone book DSL.

The phone book metamodel is imported by using the `import` statement in line 1. Afterwards, a method called `transformPhoneBook` is defined in line 3, which calls two further methods, `reduceAge` and `addGyroGearloose`.

Multiple dispatch can be observed in line 4, 8, and 12. The `reduceAge` method call in line 4 yields the selection of the method defined in line 8 as the type of `book` is `PhoneBook`.

Line 9 reveals a facet of Xtend's collection types. The method `reduceAge` is applied to each element of the `entries` collection.

[10] http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html

```
1 import phoneBook;
2
3 Void transformPhoneBook(PhoneBook book):
4    reduceAge(book) ->
5    book.addGyroGearloose()
6 ;
7
8 Void reduceAge(PhoneBook book):
9    book.entries.reduceAge()
10 ;
11
12 Void reduceAge(Entry e):
13    let p = e.person:
14    if p.age != null && p.age > 90 then
15       p.setAge(p.age - 20)
16 ;
17
18 Void addGyroGearloose(PhoneBook book):
19    let entry = new Entry:
20    let phoneNumber = new PhoneNumber:
21    let gyro = new Person:
22    gyro.setForename("Gyro") -> gyro.setSurname("Gearloose") ->
23    phoneNumber.setCode(231) -> phoneNumber.setNumber(2221) ->
24    entry.setPerson(gyro) -> entry.setNumber(phoneNumber)
25 ;
```

Listing 3.3: Xtend example of transforming a phone book

In lines 18–25 the method `addGyroGearloose` yields the creation of a new phone book entry using local variables and chained expressions.

The execution of this transformation in the context of the previously introduced `Ducksburg` phone book yields the result shown in Listing 3.4. In line 3 the age of `Scrooge McDuck` dropped from 98 to 78 and in line 5 the new `Entry Gyro Gearloose` can be seen.

```
1 phonebook Duckburg :
2    person (Donald Duck, age 30) 121/1354;
3    person (Scrooge McDuck, age 78) 212/9843;
4    person (Daisy Duck) 121/2343;
5    person (Gyro Gearloose) 231/2221;
```

Listing 3.4: Transformed version of the previous Ducksburg.pb

Execution of Extensions

Xtend's metamodel Extensions can be executed by using a so-called `XtendFacade`. All of the required metamodels have to be registered. The file containing the extensions has to be specified. The method that should be called has to be passed as well as the demanded arguments.

Another way to execute extensions is to use the Modeling Workflow Engine (MWE) and the Modeling Workflow Engine 2 (MWE2), respectively, which need the same information as before. MWE[11] is independent of any Java code but lacks flexibility. This is because always an entry extension has to be called with the root element of a model. Just the `XtendFacade` is used in the context of this thesis, therefore, MWE will not be described in further detail.

Listing 3.5 presents a small example of how to use the `XtendFacade`. In line 1 the facade is instantiated by calling the static `create` method. As parameter the location of the file containing the extensions is passed. In line 3 the actual method `someTransformation` is invoked with the string `AnyParameter` as the parameter.

```
1 XtendFacade facade = XtendFacade.create("path/myExtensions");
2
3 facade.call("someTransformation", new Object[]{"AnyParameter"});
```

Listing 3.5: Calling an Xtend method via the `XtendFacade`

Java Extensions

Xtend provides the possibility to escape to Java in case its expressiveness is not sufficient, for instance, in case some information should be printed to Java's standard output stream. In this case, any static Java method can be called. The Java method is then evaluated as usual while Xtend awaits the return of the called method.

Listing 3.6 shows an Xtend method `debug` that takes one argument of the type `EObject`. The `debug` method of the `Utils` class in the `de.cau.cs.kieler.pb` package is called with `obj` as the argument. The Java method, which is presented in Listing 3.7, just prints the result of the argument's `toString()` method to the standard output stream.

```
import ecore;

Void debug(EObject obj):
   JAVA  de.cau.cs.kieler.pb.Utils.debug(org.eclipse.emf.ecore.EObject);
```

Listing 3.6: Xtend calling a Java method

[11] http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)

```java
package de.cau.cs.kieler.pb.Utils;

public final class Utils{
   public static void debug(final EObject obj) {
      System.out.println(obj.toString());
   }
}
```

Listing 3.7: Static Java method in a utility class

Global Variables

Global variables have to be passed either to the MWE2 workflow or to the calling `XtendFacade` prior to runtime. They can be used to pass information to Xtend, which does not change during runtime, or in some cases it is not convenient to pass all information by parameters.

Inside Xtend they can be retrieved by using the `GLOBALVAR` keyword.

3.2 JUnit

JUnit[12] is a testing framework written in Java and originally created by Kent Beck and Erich Gamma. It is used to create automated and repeatable tests of software components, also referred to as *white-box testing*.

The JUnit framework serves as an automatic testing facility for the implementations presented in Chapter 6. The testing strategies and the definition of proper JUnit tests are discussed in Chapter 7.

The usage of JUnit is simple. As shown in Listing 3.8 a mere Java class serves as a frame. The `@Before` annotation depicts a method executed before the actual test methods. It can be used to initialize everything needed. Each test case is bundled into a method annotated with the keyword `@Test`. To identify the correctness of the tested functionality either *assertions* or *exceptions* can be used. This does not represent the whole functionality of the JUnit framework but is enough to understand its usage within this thesis. For further information see the JUnit documentation.

Furthermore, JUnit is well integrated into Eclipse which provides all functionality to execute a test and to present its results clearly. See Figure 3.3 for a screenshot. The left half shows a list with all specified tests if they were successful and the time their execution took. The right half presents further information about the failure of the selected test. In the given case this is an `Exception` with the information message *This is obviously wrong.*

[12] http://junit.sourceforge.net/

```java
import org.junit.*;
import static org.junit.Assert.*;

public class JUnit {

  int three = 3, nine;

  @Before
  public void setup() {
    nine = 9;
  }
  @Test
  public void testSquareRoot() {
    assertEquals((int) Math.sqrt(nine), three);
  }
  @Test
  public void testEquality() {
    assertTrue(nine == three);
  }
  @Test(expected = ArithmeticException.class)
  public void testDivisionByZero() {
    float f = 5 / 0;
  }
  @Test
  public void throwException() throws Exception {
  if (1 != 0)
    throw new Exception("This is obviously wrong!");
  }
}
```

Listing 3.8: JUnit class file

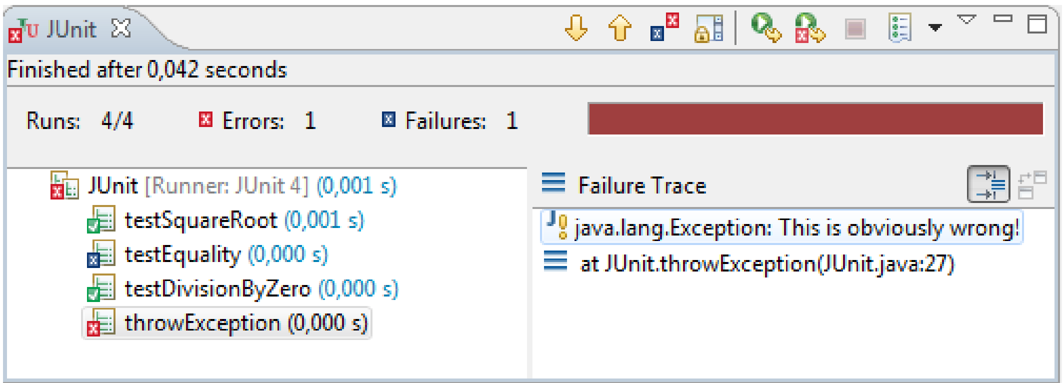


Figure 3.3: JUnit test result

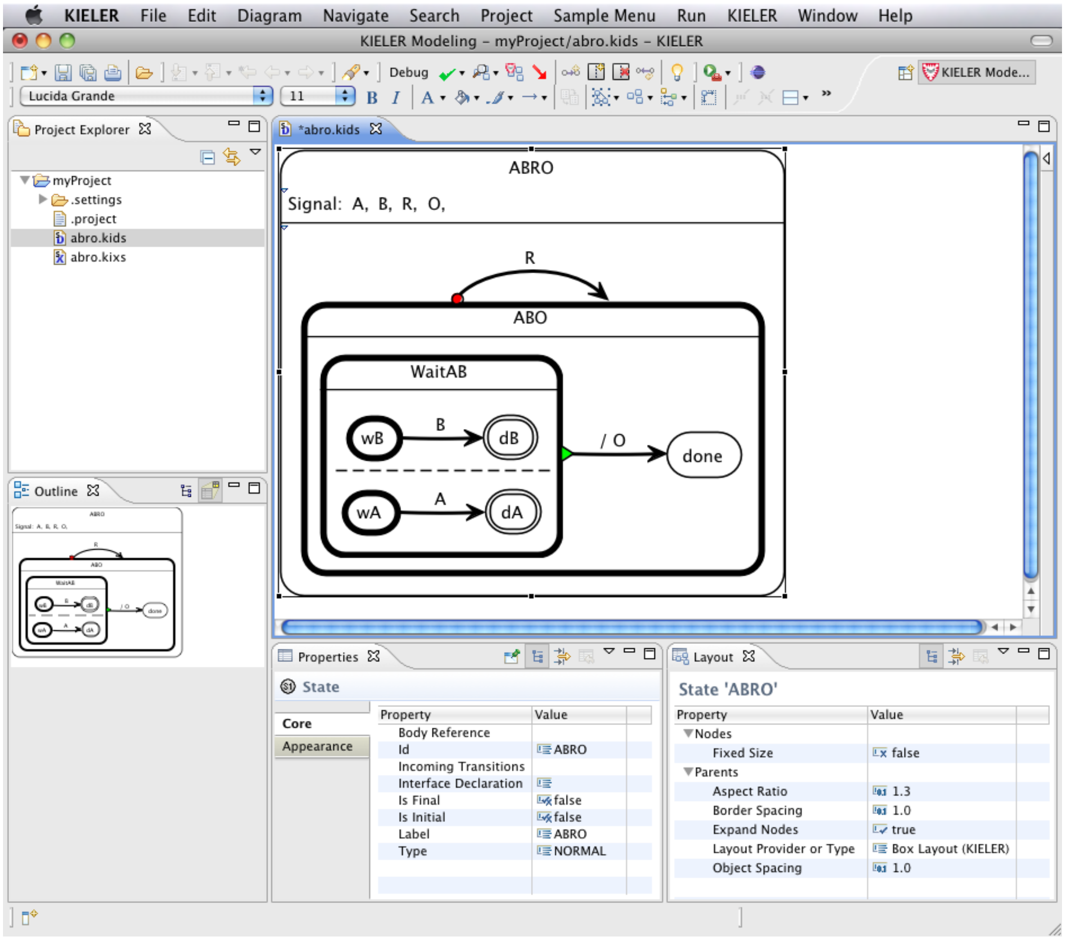


Figure 3.4: The KIELER GUI with its ThinKCharts editor [Fuh11]

3.3 KIELER

In the next sections some features of KIELER are introduced in further detail. These features are used by the exemplary Esterel to SyncCharts transformation implementation presented in this thesis. They provide the layer allowing interaction of the user with the transformation and allow the step-wise execution of model transformations. Figure 3.4 shows a first screenshot of the KIELER workbench.

3.3.1 KIELER Execution Manager

The KIELER Execution Manager (KIEM)[13] [MFvH09, Mot09] is a generic execution infrastructure based on Eclipse that can be used to execute arbitrary domain-specific models. KIEM itself just provides an interface and represents a frame for such executions, e. g., scheduling several so called `DataComponents` representing different units

[13]http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIEM

of an execution. A possible setup can be one `DataComponent` producing data of a simulation and another one visualizing this data.

Graphical User Interface

KIEM provides a GUI with several buttons that control the execution. The user is able to *step* the execution or *run* it with a certain delay between two consecutive steps. The execution can be *paused* or completely *stopped*. Additionally, there is the possibility to do *backward steps*.

One can also assemble some `DataComponents` in a *schedule* and enable or disable them for the current execution. These buttons, together with a predefined schedule to simulate SyncCharts, can be seen in Figure 3.5.

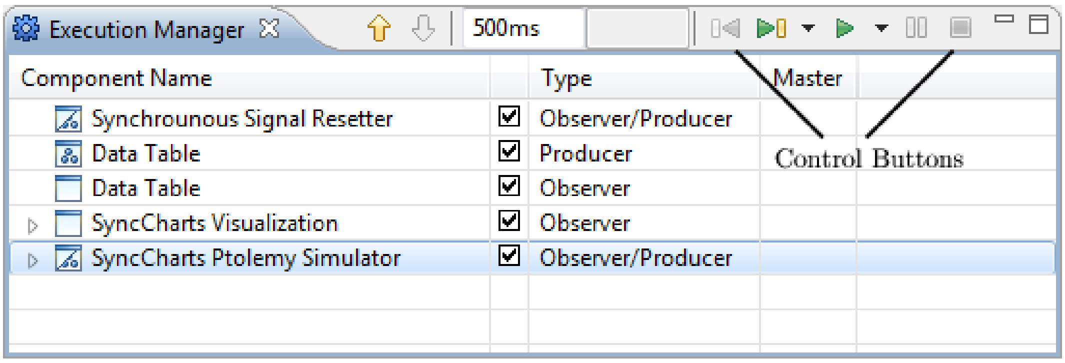


Figure 3.5: KIEM User Interface

DataComponent

A *DataComponent* provides several methods which the developer is supposed to overwrite. They are automatically called by KIEM during an execution. The most important ones are:

1. **`initialize():`** This method is called prior to the begin of the execution.
2. **`step():`** For each step that is initiated by either KIEM or the user this method is called.
3. **`wrapup():`** Once the execution is finished or the user aborts it this method is called.
4. **`isProducer():`** This predicate should return true if the DataComponent produces any data that might be used by other DataComponents.
5. **`isObserver():`** This predicate should return true if the DataComponent wants to observe data produced by another one. The specification has, like 4., implications on the scheduling [Mot09] of the different DataComponents.

3.3.2 KIELER Viewmanagement

KIELER Viewmanagement (KiVi)[14] [Mül10] is intended to manage arbitrary visual effects occurring during model based design. KiVi bases on the ideas presented by Fuhrmann and von Hanxleden [FvH10b].

Basically it consists of three pieces:

`Trigger`: A `Trigger` is supposed to listen to certain events and inform KiVi upon occurrence. Then, KiVi executes all interested `Combination`.

`Effect`: An effect is the actual action that is executed. For example, it might manipulate the view to meet certain criteria.

`Combination`: The `Combination` is the logic that decides how to react when it is informed by a trigger about the occurrence of a certain event.

KiVi tries to keep the `Combinations` as simple as possible. They are the piece of code each developer has to write himself, for instance, to apply a view management effect. `Triggers` and `Effects` can often be reused as some of them are used in different contexts. An example of such a use could be applying *automatic layout* to a diagram either upon a *button click* or a *model change*.

The generic approach to execute implementations presented in Chapter 6 uses a KiVi `Effect`. An advantage is the possibility to apply further `Effects` in combination with a transformation without much effort.

[14]https://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KiVi

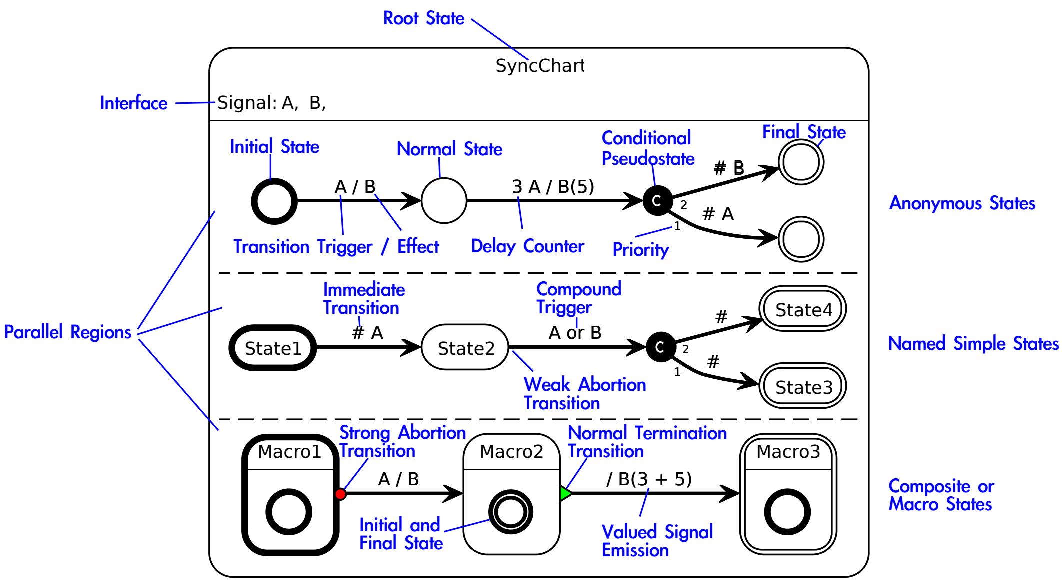


Figure 3.6: Syntax of ThinKCharts [Fuh11]

Furthermore, the interface connecting the user inputs with the accordant program behavior is implemented by using a `Combination`, which allows the communication with other viewmanagement elements and is well expandable.

3.3.3 Thin KIELER SyncCharts Editor

The Thin KIELER SyncCharts Editor (ThinKCharts) has been developed as a demonstration tool for new approaches of graphical modeling. See Figure 3.4 for a screenshot. Its first implementation was provided by Matthias Schmeling [Sch09] and uses EMF and Graphical Modeling Framework (GMF). For further information on the GMF part refer to the work of Schmeling.

The editor bases on the metamodel shown in Figure 3.7. All elements of SyncCharts, as introduced in Section 1.5, are represented by the metamodel. The concrete syntax is similar to the one used by Esterel Studio and is presented in Figure 3.6. A macro state containing all expressiveness of SyncCharts can be seen. All black elements are syntax, while the description of each element is presented in blue.

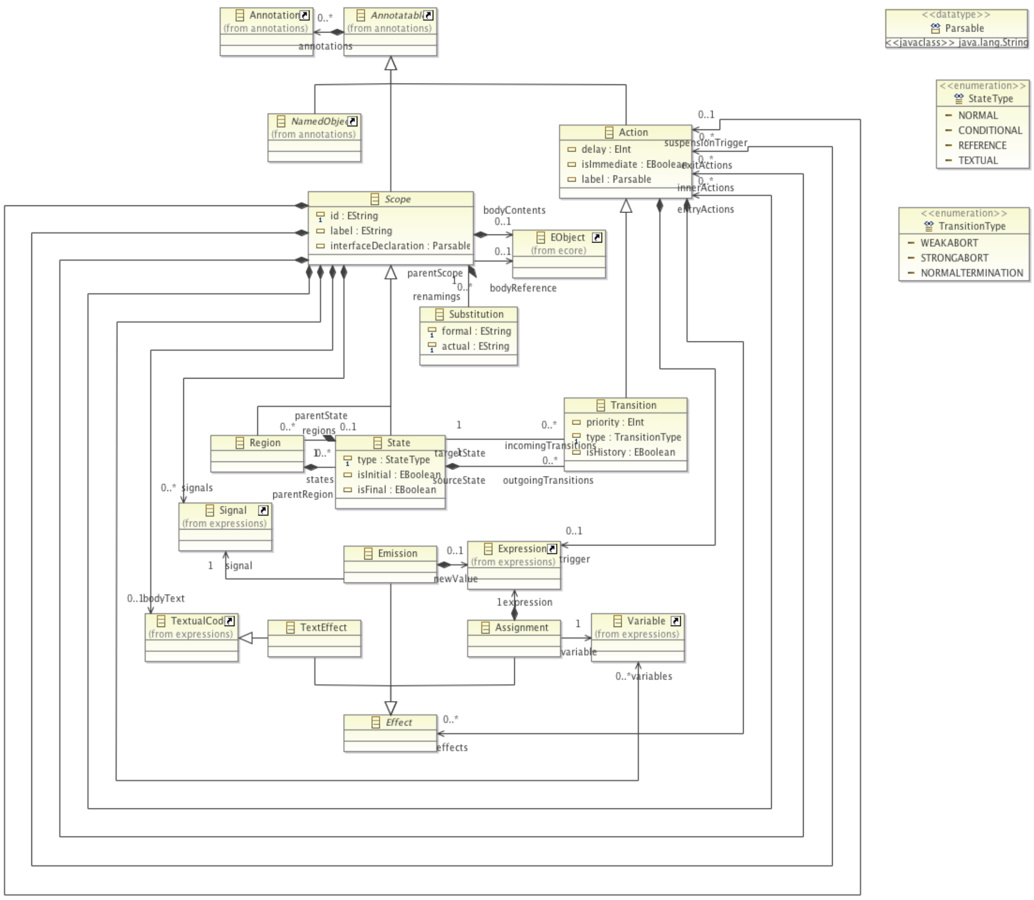


Figure 3.7: ThinKCharts metamodel

4 Adaption of the Esterel Grammar in KIELER

There are several Esterel grammars [Ber00, PBEB07] and tools to work with Esterel, such as Esterel studio[1]. But to satisfy the requirements stated in Chapter 1, e. g., usability in the form of sophisticated tooling and the integration into KIELER, it is necessary to adapt such a grammar in the Eclipse context.

In this chapter the adaption of the Esterel language grammar is presented. Furthermore, some small extracts of the actual grammar and solutions of the main obstacles concerning the chosen approach are given.

4.1 Concept

Due to the intention of this thesis to use Esterel in the context of KIELER a good integration into the current infrastructure is mandatory.

KIELER already provides an expression language based on Xtext. It is called `KExpressions` and is used by the ThinKCharts editor. Its expressions are similar to the expressions used in Esterel. Furthermore, Xtext generates automatically sophisticated tooling, which can be customized to a nearly arbitrary extent. Hence, Xtext seems to be the natural choice to realize a new implementation of an Esterel grammar.

Existing Esterel grammars are usually documented in Backus–Naur Form (BNF) or in EBNF and are therefore easily transferred into Xtext as the syntax is similar. The primarily used references for the presented adaption are Potop-Butucaru [PBEB07] and Berry [Ber00].

In the following, the reused infrastructure of KIELER is briefly introduced.

4.1.1 KExpressions

`KExpressions` is an expression language, which is developed by using EMF and Xtext. Its connections within KIELER can be seen in Figure 4.1. The `annotations` metamodel specifies how to annotate a plain object. The `KExpressions` metamodel uses it and serves as the expression language for the `synccharts` metamodel. The dotted box indicates that the expressions will be reused in the `esterel` metamodel as well.

[1] http://www.esterel-eda.com

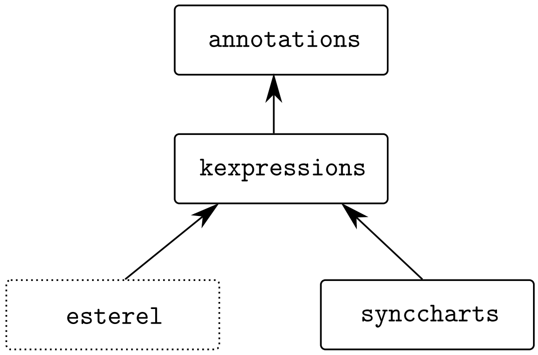


Figure 4.1: Dependencies of some of KIELER's metamodels

`KExpressions` provides basic arithmetic and boolean expressions and obeys common precedence rules. It also comes with the possibility to define TextExpressions, which is handy to define host code in Esterel.

Such expressions may be:

1. Valued Expressions: `(1 + 2 / 4 mod 5)`.
2. Boolean Expressions: `(A and B or not (5 > 4))`.
3. Text Expressions: ”`printf(...)`”.

Additionally, it specifies basic description of an interface declaration for signals and variables. Signals and variables can have a *type* and an *initial value*. Furthermore, signals can also be marked as *input*, *output*, *inputoutput*, or *return*.

Listing 4.1 shows the definition of two signals `A` and `B` with an initial value and a type in line 2. Afterwards, two variables `v1` of type `integer` are specified with an initial value.

```
1 // signals
2 input A := 4 : integer, output O := 1.4f : float
3 // variables
4 var v1 := 1, v2 := 2 : integer
```

Listing 4.1: Declaration of several signals and variables

To get an overview of `KExpressions`'s metamodel, see Figure 4.3. It can be seen that `Signals` and `Variables` are `ValuedObjects` which are additionally `Annotatables`. The `annotations` metamodel can be seen in Figure 4.2 and defines essential functionality to annotate `NamedObjects` with arbitrary `Annotations`. `Expressions` are either `ComplexExpressions` or plain `Values`. `ComplexExpressions` can be composed `OperatorExpressions` with an operator, e. g., `+` or $*$, `ValuedObjectReferences` that, for instance, refers to an existing `Signal`, or to a `TextExpression`. `Values` can be floats, integer, or booleans.

Another crucial reason for the re-use of `KExpressions` is the fact that the definition of the SyncCharts metamodel bases on it. Hence, the rules for an Esterel to SyncCharts transformation can be kept simple. There is no need to transform Esterel expressions into a form, valid in SyncCharts, as they use the same elements.

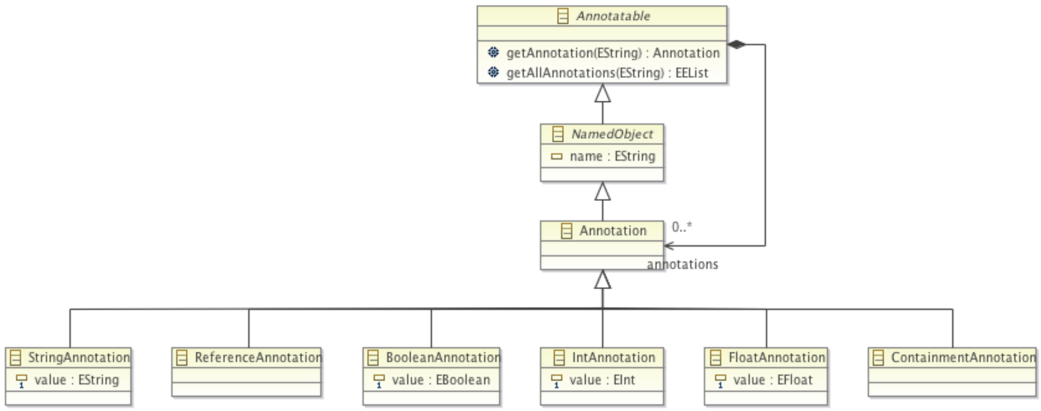


Figure 4.2: The `annotations` metamodel

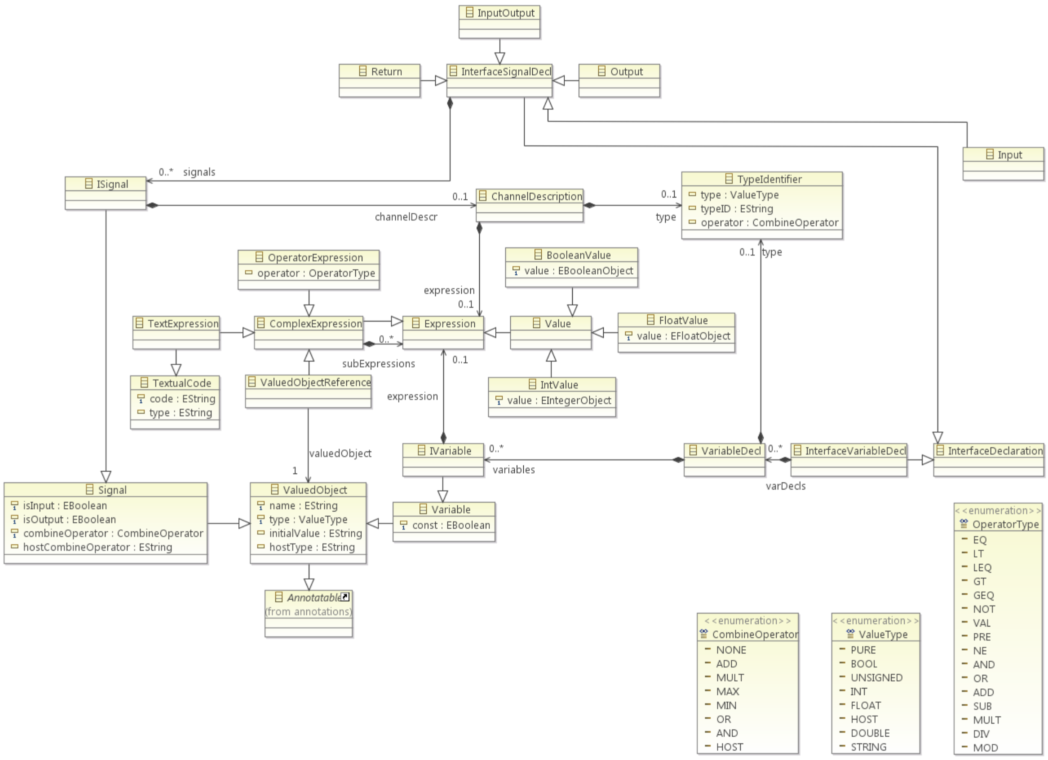


Figure 4.3: The `KExpressions` metamodel

4.2 Implementation

In this section some pieces of the created Xtext grammar are presented. The basic comprehension of the grammar is necessary to understand the implementation of the Esterel to SyncCharts transformation.

The root element of an Esterel program is the `Program` element, which is defined in Listing 4.2 in line 1 and 2. It can contain several modules and can be commented in the Esterel typical way using `%{` as starting, `}%` as ending delimiter. Lines 3 to 6 specify a `Module` which consists of a name, a possible interface declaration, and a body with several statements.

Listing 4.3 shows the way statements can be combined. This can be done either in sequence or in parallel. Each `AtomicStatement` is defined in its own separate rule (see line 9-12), which is shown in Section 5.2 together with its transformation into an equivalent SyncChart.

```
1 Program hidden(Esterel_SL_Comment, Esterel_ML_Comment, WS):
2    (modules+=Module)*;
3 Module:
4    "module" name=ID ":" (interface=ModuleInterface)? body=ModuleBody end=EndModule
        ;
5 ModuleBody:
6    statements+=Statement;
```

Listing 4.2: Esterel `program` and `module`

```
1  Statement:
2     Sequence ({Parallel.list+=current} "||" list+=Sequence)*;
3
4  Sequence returns Statement:
5     AtomicStatement ({Sequence.list+=current} ";" list+=AtomicStatement)* ";"?;
6
7  AtomicStatement returns Statement:
8     Abort | Assignment | Await | Block | ProcCall | Do | Emit | EveryDo |
9     Exit | Exec | Halt | IfTest | LocalSignalDecl | Loop | Nothing | Pause |
10    Present | Repeat | Run | Suspend | Sustain | Trap | LocalVariable |
11    VarStatement | WeakAbort;
```

Listing 4.3: Esterel `statements`

4.2.1 Obstacles

Interface Declarations

As stated above, `KExpressions` already provides basic support for the declaration of signals and variables. However, Esterel allows the definition of domain specific *types* which can be used as the type of a signal or a variable. Therefore, the `TypeIdentifier` rule is overwritten to allow the reference of newly specified types.

Also, Esterel's interface declaration allows the declaration of *Types*, *Sensors*, *Relations*, *Tasks*, *Functions*, and *Procedures*. Adding those is not much of a problem as all of them can be combined with alternatives in one rule, as shown in Listing 4.4.

```
ModuleInterface:
   (intSignalDecls+=InterfaceSignalDecl
   | intTypeDecls+=TypeDecl
   | intSensorDecls+=SensorDecl
   | intConstantDecls+=ConstantDecls
   | intRelationDecls+=RelationDecl
   | intTaskDecls+=TaskDecl
   | intFunctionDecls+=FunctionDecl
   | intProcedureDecls+=ProcedureDecl)+;
```

Listing 4.4: A `module`'s interface

Removing Left-Recursion

Xtext uses an LL-Parser to evaluate the input. For this reason it does not support *left recursion*, which is used a lot in EBNF grammars. An example of a left recursive rule, taken from the grammar used as the prototype, is shown in Listing 4.5. As one can see there, the `SignalDeclList` rule references itself as the left symbol of the rule in line 3.

To transfer such rules into a form that is valid in Xtext one can use the so-called left-factoring. This is shown in Listing 4.6. The left recursion is removed by using a new rule `SignalDecls`, which either specifies a `SignalDecl` or another `SignalDeclList`. The latter needs to be written in parentheses or any other symbols assuring unambiguity.

However, this is not what is wanted as it introduces additional syntax. A better solution is to use Xtext's list assignment mechanism which yields cleaner results. Additionally, considering the programmatic use lists can be processed easier compared to strongly nested classes.

An example of the latter conversion can be seen in Listing 4.7. Here, at least one `SignalDecl` has to be specified, followed by an arbitrary amount of `SignalDecls`, each separated from the other one by a comma.

```
1 SignalDeclList:
2    SignalDecl |
3    SignalDeclList ',' SignalDecl;
```

Listing 4.5: Left-recursive grammar rule taken from [Ber00]

```
SignalDeclList:
   left=SignalDecls (',' right=SignalDecls)?;

SignalDecls:
   SignalDecl |  '(' SignalDeclList ')';
```

Listing 4.6: *Left-factored* result

```
SignalDeclList:
   signals+=SignalDecl (',' signals+=SignalDecl)*;
```

Listing 4.7: Using Xtext's list assignment

Embedding further Expressions

Unfortunately, the reused `KExpressions` does not provide all of the expressions included in Esterel. The missing expressions are *function calls*, *traps*, and *constants*.

To solve this problem the expressions need to be hooked into the existing `KExpressions` implementation. For such cases Xtext's inheritance mechanism allows to override methods of the parent grammar, which is done for the mentioned expressions as shown in Listing 4.8.

In line 10-11 the `TrapExpression` is defined. The use of an `ISignal` to reference traps is a result of the internal implementation of traps. Afterwards, in line 13-14 a `FunctionExpression` is specified, which consists of a reference to a function name and a various number of `Expressions` as parameters. At last, a `ConstantExpression` can either be a referenced constant or any `ConstantAtom`, e. g., an integer or a float.

The insertion into the existing `AtomicExpression` rule of the original `KExpressions` grammar can be seen in line 1-8 where the new expressions are added to the `AtomicExpression` as alternatives.

4.2.2 Result

In Figure 4.4 the generated Esterel editor and a small sample input consisting of the `ABRO` module and `anotherModule` can be seen. On the left bottom a *program* outline is presented containing all main elements of the current program. Above this, small *error markers* indicate the missing letter *p* in `outut` and the not yet finished `emit` statement. At this point, also the *code completion* is worth to be mentioned. It suggests finishing the letters `emi` to the `emit` statement. The last point to observe is the *syntax highlighting*, which highlights each Esterel keyword in a bold, dark violet font.

For further information concerning the grammar see Listing A.1 in the appendix.

```
AtomicExpression returns kexpressions::Expression:
   FunctionExpression
   | TrapExpression
   | BooleanValue
   | ValuedObjectTestExpression
   | TextExpression
   | '(' BooleanExpression ')'
   | ConstantExpression;
TrapExpression returns kexpressions::Expression:
   {TrapExpression} "??" trap=[kexpressions::ISignal|ID];
FunctionExpression returns kexpressions::Expression:
   {FunctionExpression} function=[Function|ID] "(" (kexpressions+=Expression (","
       kexpressions+=Expression)*)? ")";
ConstantExpression returns kexpressions::Expression:
   {ConstantExpression} (constant=[Constant|ID] | value=ConstantAtom);
```

Listing 4.8: Embedding further expressions

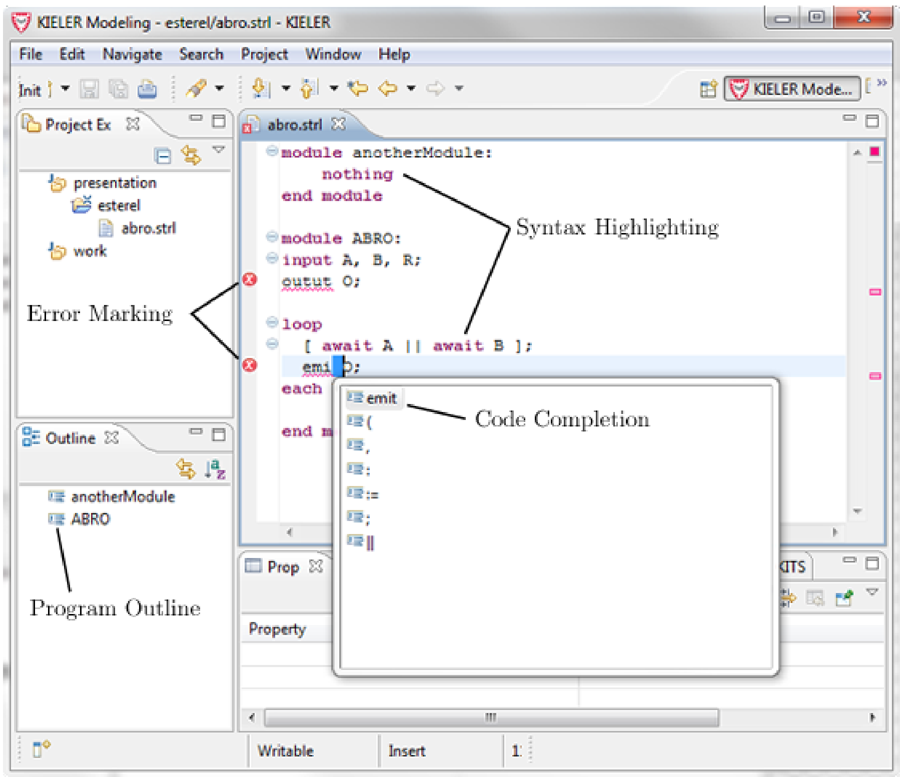


Figure 4.4: Xtext Esterel editor

5 Visual Transformation

The meaning of the term *transformation* is ambiguous. It refers to completely different issues depending on the area of application in which it is used. In the context of this thesis it is used in the four following ways.

First, it stands for the transformation of a certain domain's model into another model (M2M). Second, it denotes an in-place transformation operating on a single model within the same domain. Third, it can indicate an arbitrary extract of one of the two transformations mentioned. This, for instance, might be just a single step of a transformation that takes ten steps overall (the size of a step depends on its definition). Last, the term can be used representatively for the sum of all previously stated points.

This chapter is divided into three parts. In Section 5.1, criteria that are essential to any transformation are pointed out. This section includes specifying the abstract dimension of an arbitrary transformation. Based on these findings a possible implementation is presented.

Afterwards, Section 5.2 introduces the theoretical foundations of the Esterel to SyncCharts transformation. The foundations were presented and proved by Kühl [Küh06] and implemented in KIEL. In this thesis they are just revisited and placed into the new context of KIELER. The actual implementation is documented in Section 6.1.

Section 5.3 presents optimization potential of SyncCharts as presented by Kühl.

5.1 A Generic Approach

The following requirements have to be considered to provide a generic approach for arbitrary transformations.

Reusability The approach has to be usable within different domains and with diverse technologies.

Expandability An existing transformation has to be changeable and to be expandable easily. It should be possible to add further transformations without difficulty.

Comprehensibility The functionality of the transformation must be quickly understandable for developers and users. Developers should be able to write new transformations without the need of working themselves through the whole implementation first. Users should have the possibility to retrieve visual feedback to gain a good understanding of the transformation process.

Usability A pleasant and logical interaction with the transformation process has to be guaranteed by any user interface that is provided.

With the help of these requirements an abstract description of a transformation can be specified including all common information for arbitrary transformations and domains.

Two facets can be distinguished. First, there is the description of the actual transformation which should be processed. This is basically a matter of what should be done. Second, information about the context, in which a transformation takes place, is required. This is a matter of where, how, and when.

The first point will be referred to as *Transformation Description*, the second one as *Transformation Context*. In the following, both facets are described in further detail.

Transformation Description

Elements Which elements of a model should be transformed?

Name The name of the transformation that should be executed.

Transformation Context

Transformation Description Which transformation should be executed?

Domain In which domain is the current transformation executed? Is it *in-place* or *M2M*?

Definitions Where are the transformation rules defined?

Execution Environment How can the transformation be executed?

Modalities When and in which way should it be executed (e. g., step-wise or batch)?

5.1.1 Graphical User Interface

The implementation of the user interface depends on the type of the performed transformation.

For instance, the synthesis of SyncCharts from Esterel requires step-wise execution with back steps. In contrast to this, a single button is enough for a transformation that replaces all signals `s` by a signal `s2` within a given SyncChart.

Therefore, the presented user interface is already adjusted to the needs of the synthesis of SyncCharts from Esterel. However, it is suitable for other transformations that have a successive character.

Performs a step.

Performs one step backwards if possible.

Performs the transformation from Esterel to SyncCharts until all Esterel elements are transformed.

In case there are Esterel elements left, these are transformed first. Afterwards, a complete SyncCharts optimization is applied until no more states can be optimized.

The user is able to determine the context that should be transformed by selecting an element within the editor.

5.2 Esterel to SyncCharts Transformation

In this section the concept of each Esterel to SyncCharts transformation rule is presented. Esterel statements are nested hierarchically. This fact offers the opportunity to construct atomic rules that handle just one statement a time. Each of these rules can be applied individually and is assured to yield a correct result as the rule was proven formally.

The following sections serve as a reference for each Esterel statement's transformation into an equivalent SyncChart. They are structured in the following way:

1. A grammar snippet is shown describing the complete expressiveness of the current Esterel statement.
2. A very brief description of the Esterel statement is given.
3. The equivalent SyncCharts macro state is characterized by a text and a representative SyncChart diagram. A reference is given to the page of Kühl's work where the proof of this equivalence is presented.
4. The Xtend transformation of the particular statement is listed.

In the presented SyncCharts diagrams the following notations are used to keep the diagrams as general as possible.

e1, en, ex: an effect (e. g., `/ O`).

t1, tn, tx: a trigger (e. g., `1 < 5`).

se1, sen: a signal expression (e. g., `A and B`).

v1, vn: a variable.

sig1, sign: a signal.

`s1`, `sn`: a statement.

`proc1`: a procedure.

`trap1`, `trapn`: a trap.

`trapex1`, `trapexn`: a trap expression.

. . . : indicates a sequence of elements (e. g., several states or transitions).

Xtend transformations

The transformation rules are implemented by using Xtend. For further details of the motivation concerning the use of Xtend see Matzen [Mat10].

Each Esterel *statement* is transformed in the context of an explicit SyncCharts *state*. For this reason, it is possible to name all Xtend methods in the same way, pass the state and the statement as arguments, and let Xtend's *multiple dispatch* choose the fitting method. Also, the first and the last part of each rule is the same, e. g., in each rule the current state's name is changed according to the current Esterel element. This functionality is moved to two methods called `initializeRule()` and `finalizeRule()`. Listing 5.1 shows the common structure of all implemented transformation rules in pseudo code. Figure 5.1 depicts the execution of the transformation of the `abort` rule in a schematic way.

```
Void initializeRule(State, EsterelObject):
   // ...
;

Void finalizeRule(State, EsterelObject):
   // ...
;

Void rule(State, EsterelObject):
   // create new elements with let, e.g., let r = new Region
   initializeRule()

   // do statement specific transformation

   finalizeRulel()
;
```

Listing 5.1: Pseudo code describing the basic structure of a transformation rule

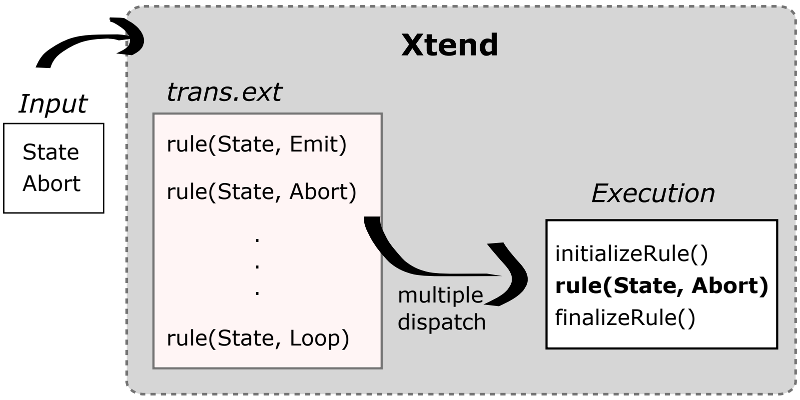


Figure 5.1: Schematic rule execution

5.2.1 nothing

Listing 5.2: `nothing`'s grammar snippet

```
Nothing:
   {Nothing} "nothing";
```

Statement Description

The `nothing` statement terminates instantaneously.

Equivalent macro-state

Immediate termination is achieved by a state marked initial and final [Küh06, p. 48].

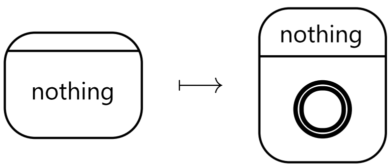


Figure 5.2: `nothing`'s transformation

Transformation

```
Void rule(State s, Nothing n):
   let nState = new State:
   let r = new Region:
   initializeRule(s, n) ->
   s.regions.add(r) ->
   r.states.add(nState) ->
   nState.setIsFinal(true) ->
   nState.setIsInitial(true)
;
```

Listing 5.3: `nothing`'s transformation snippet

5.2.2 pause

Listing 5.4: pause's grammar snippet

```
Pause:
   {Pause} "pause";
```

Statement Description

The pause statement pauses for one instant.

Equivalent macro-state

A pause for one instant is modeled by an initial state connected to a final state by a weakly aborting transition [Küh06, p. 50].

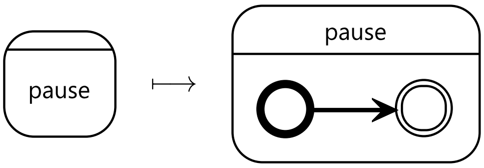


Figure 5.3: pause's transformation

Transformation

```
Void rule(State s, Pause p):
   let r = new Region:
   let initS = new State:
   let finalS = new State:
   let t = new Transition:
   initializeRule(s, p) ->
   s.regions.add(r) ->
   r.states.add(initS) ->
   r.states.add(finalS) ->
   initS.setIsInitial(true) ->
   finalS.setIsFinal(true) ->
   // add transition
   t.setType(TransitionType::WEAKABORT) ->
   t.connectTransition(initS, finalS)
;
```

Listing 5.5: pause's transformation snippet

5.2.3 `halt`

Listing 5.6: `halt`'s grammar snippet

```
Halt:
   {Halt} "halt";
```

Statement Description

The `halt` statement pauses without terminating.

Equivalent macro-state

An ever lasting pause is modeled by a single initial state [Küh06, p. 49].

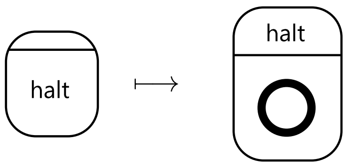


Figure 5.4: `halt`'s transformation

Transformation

```
Void rule(State s, Halt h):
   let r = new Region:
   let ns = new State:
   initializeRule(s, h) ->
   s.regions.add(r) ->
   r.states.add(ns) ->
   ns.setIsInitial(true)
;
```

Listing 5.7: `halt`'s transformation snippet

5.2.4 abort

Listing 5.8: abort's grammar snippet

```
Abort:
   "abort" statement=Statement "when" body=AbortBody;

AbortBody:
   AbortInstance | AbortCase;
AbortInstance:
   delay=DelayExpr ("do" statement=Statement "end" (optEnd="abort")?)?;
AbortCase:
   cases+=AbortCaseSingle (cases+=AbortCaseSingle)* "end" (optEnd="abort")?;
AbortCaseSingle:
   "case" delay=DelayExpr ("do" statement=Statement)?;
```

Statement Description

The abort statement terminates its body upon the occurrence of a certain delay expression.

Equivalent macro-state

The abort statement is transformed into an initial state containing the body statement. All cases are modeled by weakly aborting transitions that possess the delay expression as a trigger. They are connected to a final state. Each of these states contains the do statement if it exists.

The priority of each transition depends on the textual order of the Esterel code.

To cover the case that abort terminates without occurrence of any delay expression a final state connected by a normally terminating transition is added [Küh06, p. 51].

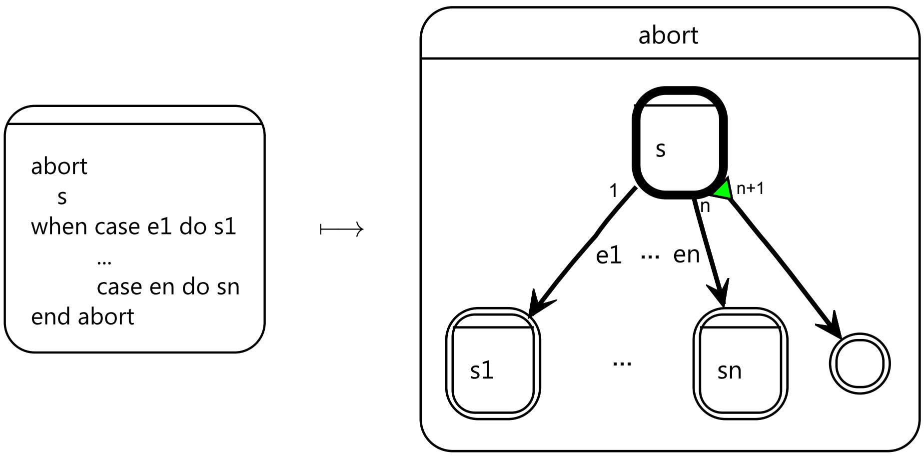


Figure 5.5: `abort`'s transformation

Transformation

```
Void rule(State s, Abort a):
   ruleAbort(s, a)
;

// helping rule, to allow usual aborts as well as weak aborts to be handled
Void ruleAbort(State s, Abort a):
   let r = new Region:
   let initState = new State:
   let caseStates = {}:
   let finalState = new State:
   let toFinalTrans = new Transition:
   initializeRule(s, a) ->
   // add new state
   s.regions.add(r) ->
   r.states.add(initState) ->
   r.states.add(finalState) ->
   // connect initial and final state with of a normal termination
   toFinalTrans.setType(TransitionType::NORMALTERMINATION) ->
   toFinalTrans.setPriority(1) ->
   initState.setIsInitial(true) ->
   finalState.setIsFinal(true) ->

   // just an instance or cases?
   (AbortInstance.isInstance(a.body) ?
      // INSTANCE
      handleAbortCaseSingle(r, initState, 1,
         ((AbortInstance)a.body).delay,
         ((AbortInstance)a.body).statement,
         WeakAbort.isInstance(a))
   :
      // CASES
      (toFinalTrans.setPriority(((AbortCase)a.body).cases.size + 1) ->
      handleAbortCases(r, initState, 1, ((AbortCase)a.body).cases, WeakAbort.
          isInstance(a)))
   ) ->
   toFinalTrans.connectTransition(initState, finalState) ->
   // handle abort's body statement
   finalizeRule(initState, a.statement)
;

Void handleAbortCases(Region parent, State source, Integer prio, List[
    AbortCaseSingle] cases, boolean weak):
   (cases.size > 1) ?
      (handleAbortCaseSingle(parent, source, prio, cases.first().delay, cases.
          first().statement, weak) ->
      handleAbortCases(parent, source, prio + 1, cases.withoutFirst(), weak))
      :
```

```
    handleAbortCaseSingle(parent, source, prio, cases.first().delay, cases.first
        ().statement, weak)
;

Void handleAbortCaseSingle(Region parent, State source, Integer prio, DelayExpr
    expr, Statement body, boolean weak):
  let caseState = new State:
  let trans = new Transition:
  caseState.setIsFinal(true) ->
  parent.states.add(caseState) ->

  // create and add transition (weak abort needs strong abortion!)
  if !weak then trans.setType(TransitionType::STRONGABORT) ->
  trans.setPriority(prio) ->

  // handle delayexpr
  trans.addDelayToTrigger(expr) ->

  trans.connectTransition(source, caseState) ->
  // care! the "do" body of abort might be null
  if (body != null) then
    // set body text
    finalizeRule(caseState, body)
;
```

Listing 5.9: `abort`'s transformation snippet

5.2.5 assign

Listing 5.10: `assign`'s grammar snippet

```
Assignment:
   var=[kexpressions::IVariable|ID] ":=" expr=Expression;
```

Statement Description

The `assign` statement assigns an expression to a variable.

Equivalent macro-state

The assignment is realized as an effect of a transition connecting an initial state with a final state [Küh06, p. 52].

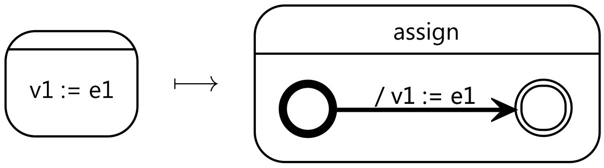


Figure 5.6: `assign`'s transformation

Transformation

```
Void rule(State s, esterel::Assignment assign):
   let r = new Region:
   let initS = new State:
   let finalS = new State:
   let t = new Transition:
   let sa = new synccharts::Assignment:
   initializeRule(s, assign) ->
   // setup states
   s.regions.add(r) ->
   initS.setIsInitial(true) ->
   finalS.setIsFinal(true) ->
   r.states.add(initS) ->
   r.states.add(finalS) ->
   // init transition
   t.connectTransition(initS, finalS) ->
   sa.setVariable(assign.var) ->
   // care to convert the expression prior to adding it
   if (assign.expr != null) then
      (sa.setExpression(((Expression) clone(assign.expr)).convertEsterelExpression
          ())) ->
   //add assignment to the transition
   t.effects.add(sa)
;
```

Listing 5.11: `assign`'s transformation snippet

5.2.6 `await`

Listing 5.12: `await`'s grammar snippet

```
Await:
   "await" body=AwaitBody;

AwaitBody:
   AwaitInstance | AwaitCase;

AwaitInstance:
   delay=DelayExpr ("do" statement=Statement end=AwaitEnd)?;

AwaitCase:
   cases+=AbortCaseSingle (cases+=AbortCaseSingle)* end=AwaitEnd;

AwaitEnd:
   "end" "await"?;
```

Statement Description

The `await` statement awaits the occurrence of certain delay expressions and executes the corresponding statement.

Equivalent macro-state

An initial state with an outgoing transition for each delay expression is created. All transitions are weakly aborting and have the respective delay expression as a trigger. Their target state is a final state with the corresponding do statement as their body. Each transition's priority depends on the textual order of the cases [Küh06, p. 53].

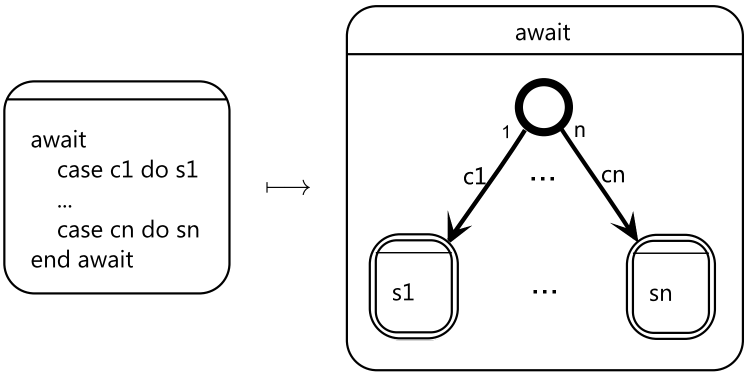


Figure 5.7: `await`'s transformation

Transformation

```
Void rule(State s, Await a):
   let r = new Region :
   let initState = new State:
   let caseStates = {}:
   initializeRule(s, a) ->
   // add state
   s.regions.add(r) ->
   initState.setIsInitial(true) ->
   r.states.add(initState) ->

   // just one instance or cases?
   AwaitInstance.isInstance(a.body)?
      handleAwaitCaseSingle(r, initState, 1,
         ((AwaitInstance)a.body).delay,
         ((AwaitInstance)a.body).statement)
   :
      handleAwaitCases(r, initState, 1, ((AwaitCase)a.body).cases)
;

// as abort and await cases are equal, we use AbortCaseSingles
Void handleAwaitCases(Region r, State previous, Integer prio, List[AbortCaseSingle
    ] cases):
   handleAbortCases(r, previous, prio, cases, true)
;

Void handleAwaitCaseSingle(Region r, State previous, Integer prio, DelayExpr expr,
     Statement body):
   handleAbortCaseSingle(r, previous, prio, expr, body, true)
;
```

Listing 5.13: await's transformation snippet

5.2.7 `do-upto`

Listing 5.14: `do-upto`'s grammar snippet

```
Do:
   "do" statement=Statement (end=DoUpto | end=DoWatching);

DoUpto:
   "upto" expr=DelayExpr;
```

Statement Description

The `do-upto` statement executes its body until a specified expression evaluates successfully. If the body terminates first, the execution is stopped until the expression evaluates.

Equivalent macro-state

An initial state contains the body statement and a strongly aborting transition with the signal expression as a trigger and a plain final state as the target. As there are no further transitions it is guaranteed that the macro state is not left prior to the occurrence of the signal expression [Küh06, p. 54].

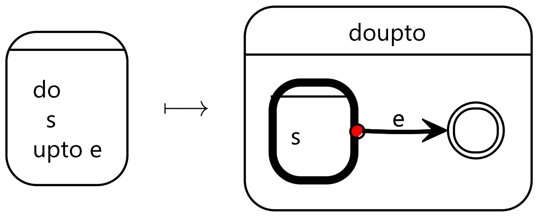


Figure 5.8: `do-upto`'s transformation

Transformation

```
Void rule(State s, Do d):
   // end is either DoUpto or DoWatching and just effects the outgoing context.
   // hence we transform the "do" statement here and then decide
   let r = new Region :
   let initState = new State:
   initializeRule(s, d) ->
   s.regions.add(r) ->
   r.states.add(initState) ->
   initState.setIsInitial(true) ->
   ((DoUpto.isInstance(d.end)) ?
      handleDoUpto(s, initState, r, (DoUpto) d.end)
      :
      handleDoWatching(s, initState, r, (DoWatching) d.end)
   ) ->
   finalizeRule(initState, d.statement)
;

Void handleDoUpto(State parent, State previous, Region parentR, DoUpto du):
   let finalS = new State:
   let t = new Transition:
   // set a more explicit state name
   parent.setLabelIfEmpty("Doupto State") ->
   parentR.states.add(finalS) ->
   finalS.setIsFinal(true) ->

   t.setType(TransitionType::STRONGABORT) ->
   t.connectTransition(previous, finalS) ->
   t.addDelayToTrigger(du.expr)
;
```

Listing 5.15: `do-upto`'s transformation snippet

5.2.8 `do-watching`

Listing 5.16: `do-watching`'s grammar snippet

```
Do:
   "do" statement=Statement (end=DoUpto | end=DoWatching);

DoWatching:
   "watching" delay=DelayExpr (end=DoWatchingEnd)?;

DoWatchingEnd:
   "timeout" statement=Statement "end" (optEnd="timeout")?;
```

Statement Description

The `do-watching` statement aborts the execution of its body as soon as a specified expression occurs. A possible timeout statement is executed. In case the body terminates first the `do-watching` statement terminates as well.

Equivalent macro-state

The body statement is added to a new initial macro state. A normally terminating transition leads to a simple, final state. The specified expression is added to a strongly aborting transition, which is connected to a final macro state. The latter contains either the timeout statement or a `nothing` statement [Küh06, p. 56].

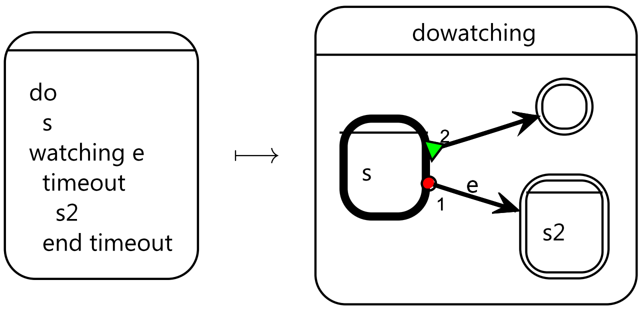


Figure 5.9: `do-watching`'s transformation

Transformation

```
// see doupto for the entry rule

Void handleDoWatching(State parent, State previous, Region parentR, DoWatching dw) :
   let abortF = new State:
   let normalF = new State:
   let abortT = new Transition:
   let normalT = new Transition:
   // set a more explicit state name
   parent.setLabelIfEmpty("Dowatching State") ->
   parentR.states.add(abortF) ->
   parentR.states.add(normalF) ->
   normalF.setIsFinal(true) ->
   abortF.setIsFinal(true) ->
   // transitions
   abortT.setType(TransitionType::STRONGABORT) ->
   normalT.setType(TransitionType::NORMALTERMINATION) ->
   abortT.connectTransition(previous, abortF) ->
   normalT.connectTransition(previous, normalF) ->
   // add delay
   abortT.addDelayToTrigger(dw.delay) ->
   // if timeout
   if (dw.end != null) then
      finalizeRule(abortF, dw.end.statement)
;
```

Listing 5.17: `do-watching`'s transformation snippet

5.2.9 emit

Listing 5.18: `emit`'s grammar snippet

```
Emit:
   "emit" ((signal=[kexpressions::ISignal|ID]) | tick=Tick) ("(" expr=Expression "
      )")?;
```

Statement Description

The `emit` statement emits a specified signal instantaneously.

Equivalent macro-state

The specified signal is emitted as an effect of a transition connecting an initial with a final state [Küh06, p. 57].

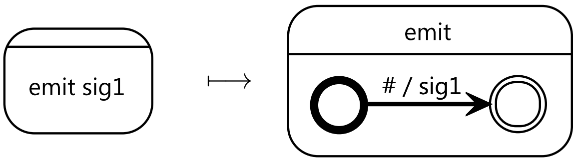


Figure 5.10: `emit`'s transformation

Transformation

```
Void rule(State s, Emit e):
   let initS = new State:
   let finalS = new State:
   let r = new Region:
   let emitTrans = new Transition:
   let emission = new Emission:
   initializeRule(s, e) ->
   s.regions.add(r) ->
   initS.setIsInitial(true) ->
   finalS.setIsFinal(true) ->

   // add new states to region
   r.states.add(initS) ->
   r.states.add(finalS) ->

   // add the effect
   emitTrans.setIsImmediate(true) ->
   emission.setSignal(e.signal) ->
   emission.setNewValue(convertEsterelExpression((Expression)clone(e.expr))) ->
   emitTrans.effects.add(emission) ->
   // add transition to state
   emitTrans.connectTransition(initS, finalS)
;
```

Listing 5.19: `emit`'s transformation snippet

5.2.10 every

Listing 5.20: every's grammar snippet

```
EveryDo:
   "every" delay=DelayExpr "do" statement=Statement "end" (optEnd="every")?;
```

Statement Description

The `every` statement starts the execution of its body upon the first occurrence of the specified delay expression. Afterwards, the body is started again each time the delay expression evaluates successfully. In case the statement is already running it is aborted first.

Equivalent macro-state

The macro state containing the body statement is entered by a transition, which has the delay expression as a trigger, connected to an initial, plain state. Furthermore, the macro state possesses a self-loop with the delay expression as a trigger [Küh06, p. 58].

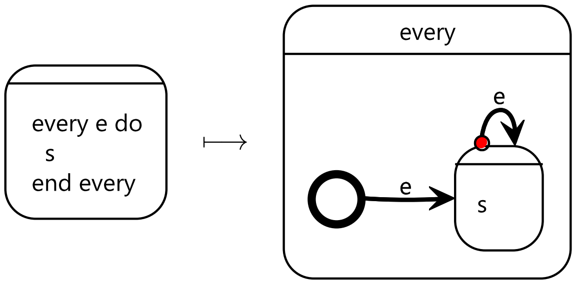


Figure 5.11: every's transformation

Transformation

```
Void rule(State s, EveryDo e):
   let r = new Region:
   let initS = new State:
   let everyS = new State:
   let initT = new Transition:
   let everyT = new Transition:
   initializeRule(s, e) ->
   // setup states
   s.regions.add(r) ->
   r.states.add(initS) ->
   r.states.add(everyS) ->
   initS.setIsInitial(true) ->
   // init transitions
   initT.setType(TransitionType::WEAKABORT) ->
   everyT.setType(TransitionType::STRONGABORT) ->
   initT.connectTransition(initS, everyS) ->
   everyT.connectTransition(everyS, everyS) ->
   // add delays
   initT.addDelayToTrigger(e.delay) ->
   everyT.addDelayToTrigger(e.delay) ->
   // recursive
   finalizeRule(everyS, e.statement)
;
```

Listing 5.21: `every`'s transformation snippet

5.2.11 `if`

Listing 5.22: `if`'s grammar snippet

```
IfTest:
   "if" expr=Expression (thenPart=ThenPart)? (elsif+=ElsIf)∗ (elsePart=ElsePart)?
       "end" (optEnd="if")?;

ElsIf:
   "elsif" expr=Expression (thenPart=ThenPart)?;

ThenPart:
   "then" statement=Statement;

ElsePart:
   "else" statement=Statement;
```

Statement Description

The `if` statement branches according to the evaluation of an arbitrary number of expressions.

Equivalent macro-state

Each branch is represented by a transition connecting a common initial state with a distinct final state. Each final state contains the corresponding statement. The trigger of each transition is the respective expression. Priorities are set according to the textual order [Küh06, p. 60].

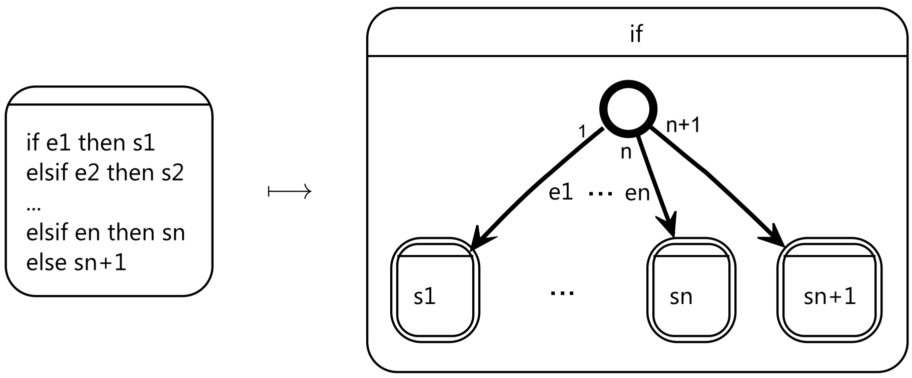


Figure 5.12: `if`'s transformation

Transformation

```
Void rule(State s, IfTest ift):
   let r = new Region:
   let initS = new State:
   let maxprio = 2 + ift.elsif.size: // priority for possible else case
   initializeRule(s, ift) ->
   s.setLabelIfEmpty("If State") ->
   s.regions.add(r) ->
   r.states.add(initS) ->
   initS.setIsInitial(true) ->
   // first if
   handleIfSingle(r, initS, 1, ift.expr, (ift.thenPart != null) ? ift.thenPart.
       statement : null) ->
   if (!ift.elsif.isEmpty) then // possible else ifs
      handleElseIfParts(r, initS, 2, ift.elsif) ->
   if (ift.elsePart != null) then // possible else
      handleIfSingle(r, initS, maxprio, null, ift.elsePart.statement)
;
Void handleElseIfParts(Region parent, State previous, Integer prio, List[ElsIf]
    elses):
   handleIfSingle(parent, previous, prio, elses.first().expr,
      (elses.first().thenPart != null) ? elses.first().thenPart.statement : null)
          ->
   if(elses.size > 1) then
      handleElseIfParts(parent, previous, prio+1, elses.withoutFirst())
;
Void handleIfSingle(Region parent, State previous, Integer prio, Expression e,
    Statement s):
   let ifS = new State:
   let ifT = new Transition:
   parent.states.add(ifS) ->
   ifS.setIsFinal(true) ->
   if (e != null) then // setup transition
      (ifT.setTrigger(convertEsterelExpression((Expression) clone(e)))) ->
   ifT.connectTransition(previous, ifS) ->
   ifT.setPriority(prio) ->
   // if thenpart, recursive
   if (s != null) then
      finalizeRule(ifS, s) ->
   // create artificial nothing
   if (s == null) then
      (let n = new Nothing:
      rule(ifS, n))
;
```

Listing 5.23: `if`'s transformation snippet

5.2.12 `local-signal`

Listing 5.24: `local-signal`'s grammar snippet

```
LocalSignalDecl:
   "signal" signalList=LocalSignalList "in" statement=Statement "end" (optEnd="signal")?;

LocalSignalList:
   {LocalSignal} signal+=ISignal
   ("," signal+=ISignal)*;
```

Statement Description

A local `signal` statement declares new local signals. Other declarations of the same signal on a higher hierarchy level are hidden.

Equivalent macro-state

The new signals are added to the interface declaration of the current macro state [Küh06, p. 61].

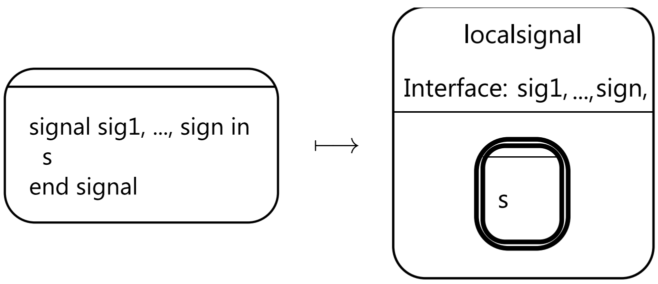


Figure 5.13: `local-signal`'s transformation

Transformation

```
Void rule(State s, LocalSignalDecl ls):
   let r = new Region:
   let sigS = new State:
   initializeRule(s, ls) ->
   // setup
   s.regions.add(r) ->
   r.states.add(sigS) ->
   sigS.setIsInitial(true) ->
   sigS.setIsFinal(true) ->
   // extract local signals
   ls.signalList.extractLocalSignals(s) ->
   // recursive
   finalizeRule(sigS, ls.statement)
;
```

Listing 5.25: `local-signal`'s transformation snippet

5.2.13 `local-variable`

Listing 5.26: `local-variable`'s grammar snippet

```
LocalVariable:
   var=InterfaceVariableDecl "in" statement=Statement "end" (optEnd="var")?;
```

Statement Description

The `var` statement indicates the declaration of new local variables.

Equivalent macro-state

The newly declared variables are added to the interface declaration of the current macro state [Küh06, p. 62].

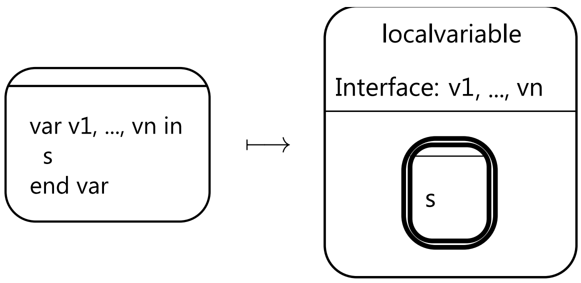


Figure 5.14: `local-variable`'s transformation

Transformation

```
Void rule(State s, LocalVariable v):
   let r = new Region:
   let varS = new State:
   initializeRule(s, v) ->
   // setup
   s.regions.add(r) ->
   r.states.add(varS) ->
   varS.setIsInitial(true) ->
   varS.setIsFinal(true) ->
   // add variables to state
   v.var.varDecls.extractLocalVariables(s) ->
   // recursive
   finalizeRule(varS, v.statement)
;
```

Listing 5.27: `local-variable`'s transformation snippet

5.2.14 `loop`

Listing 5.28: `loop`'s grammar snippet

```
Loop:
   "loop" body=LoopBody (end1=EndLoop | end=LoopEach);

EndLoop:
   "end" "loop"?;

LoopEach:
   "each" LoopDelay;

LoopDelay:
   delay=DelayExpr;

LoopBody:
   statement=Statement;
```

Statement Description

The `loop` statement executes its body and restarts it instantaneously upon termination.

Equivalent macro-state

The loop is realized as a macro state containing the body statement. The state holds a self-transition that terminates normally [Küh06, p. 63].

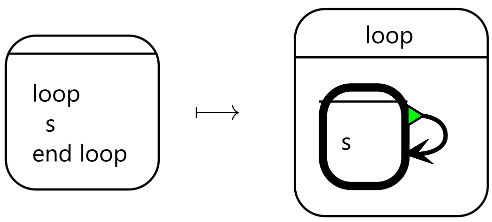


Figure 5.15: `loop`'s transformation

Transformation

```
Void rule(State s, Loop l):
   let r = new Region:
   let loopState = new State:
   let selfTrans = new Transition:
   initializeRule(s, l) ->
   // name has to be determined separately
   (LoopEach.isInstance(l.end) ? s.setLabelIfEmpty("LoopEach State")
      : s.setLabelIfEmpty("Loop State")) ->
   // setup state
   s.regions.add(r) ->
   r.states.add(loopState) ->
   loopState.setIsInitial(true) ->
   // setup transition
   selfTrans.setType(TransitionType::NORMALTERMINATION) ->
   selfTrans.connectTransition(loopState, loopState) ->

   // handle each case
   if LoopEach.isInstance(l.end) then
      (selfTrans.setType(TransitionType::STRONGABORT) ->
       s) -> selfTrans.addDelayToTrigger(((LoopDelay)l.end).delay) ->

   finalizeRule(loopState, l.body.statement)
;
```

Listing 5.29: `loop`'s transformation snippet

5.2.15 `loop-each`

Listing 5.30: `loop-each`'s grammar snippet

```
Loop:
   "loop" body=LoopBody (end1=EndLoop | end=LoopEach);

EndLoop:
   "end" "loop"?;

LoopEach:
   "each" LoopDelay;

LoopDelay:
   delay=DelayExpr;

LoopBody:
   statement=Statement;
```

Statement Description

The `loop-each` starts with the execution of its body immediately. Upon the occurrence of a delay expression the execution of the body is aborted and restarted. If the body terminates, the execution is paused until the delay expression occurs.

Equivalent macro-state

The realization corresponds to the simple `loop`. Additionally, the self transition is realized in the form of a strong abortion. The delay expression is added to the transition as a trigger [Küh06, p. 64].

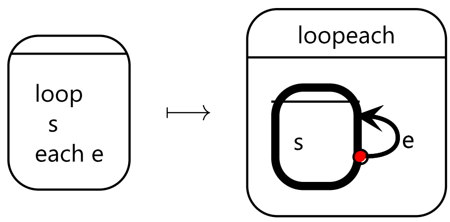


Figure 5.16: `loop-each`'s transformation

Transformation

```
Void rule(State s, Loop l):
   let r = new Region:
   let loopState = new State:
   let selfTrans = new Transition:
   initializeRule(s, l) ->
   // name has to be determined separately
   (LoopEach.isInstance(l.end) ? s.setLabelIfEmpty("LoopEach State")
      : s.setLabelIfEmpty("Loop State")) ->
   // setup state
   s.regions.add(r) ->
   r.states.add(loopState) ->
   loopState.setIsInitial(true) ->
   // setup transition
   selfTrans.setType(TransitionType::NORMALTERMINATION) ->
   selfTrans.connectTransition(loopState, loopState) ->

   // handle each case
   if LoopEach.isInstance(l.end) then
      (selfTrans.setType(TransitionType::STRONGABORT) ->
       s) -> selfTrans.addDelayToTrigger(((LoopDelay)l.end).delay) ->

   finalizeRule(loopState, l.body.statement)
;
```

Listing 5.31: `loop-each`'s transformation snippet

5.2.16 parallel

Listing 5.32: parallel's grammar snippet

```
Statement:
  Sequence ({Parallel.list+=current} "||" list+=Sequence)*;
```

Statement Description

All statements of a parallel statement are executed concurrently. parallel itself terminates as soon as all inner statements have been terminated.

Equivalent macro-state

To model an equivalent macro state SyncCharts' notation of regions is used. Each inner statement is placed within its own macro state and added to a shared macro state in parallel [Küh06, p. 65].

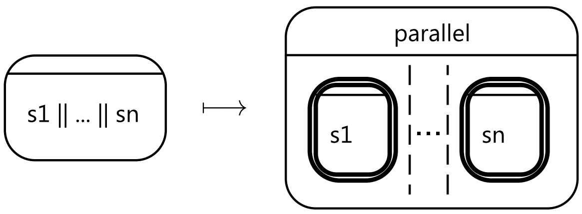


Figure 5.17: parallel's transformation

Transformation

```
Void rule(State s, Parallel p):
   initializeRule(s, p) ->
   ruleParallelRecursive(s, p.list.copyList())
;

Void ruleParallelRecursive(State parent, List[Statement] statements):
   // if inner parallel, extract the statements
   (Parallel.isInstance(statements.first()) ?
      (
      statements.addAll(((Parallel)statements.first()).list) ->
      statements.remove(statements.first()) ->
      ruleParallelRecursive(parent, statements)
      )
   :
      // else create this state and add it to the parallel region
      (
      let r = new Region:
      let s = new State:
      parent.regions.add(r) ->
      r.states.add(s) ->
      // add this to parallel
      s.setIsFinal(true) ->
      s.setIsInitial(true) ->

      if(statements.size > 1) then
         ruleParallelRecursive(parent, statements.withoutFirst()) ->

      finalizeRule(s, statements.first())
      )
   )
;
```

Listing 5.33: `parallel`'s transformation snippet

5.2.17 present

Listing 5.34: present's grammar snippet

```
Present:
  "present" body=PresentBody (elsePart=ElsePart)? "end" (optEnd="present")?;

PresentBody:
  PresentEventBody | PresentCaseList;

PresentEventBody:
  event=PresentEvent (thenPart=ThenPart)?;
PresentCaseList:
  cases+=PresentCase (cases+=PresentCase)*;
PresentCase:
  "case" event=PresentEvent ("do" statement=Statement)?;
PresentEvent:
  expression=SignalExpression | "[" expression=SignalExpression "]" | tick=Tick;
```

Statement Description

The present statement tests the instantaneous value of one or several signal expressions and branches accordingly.

Equivalent macro-state

The equivalent macro state correlates with the if statement presented in Figure 5.12. The only difference between them is the use of signal expressions in case of present [Küh06, p. 66].

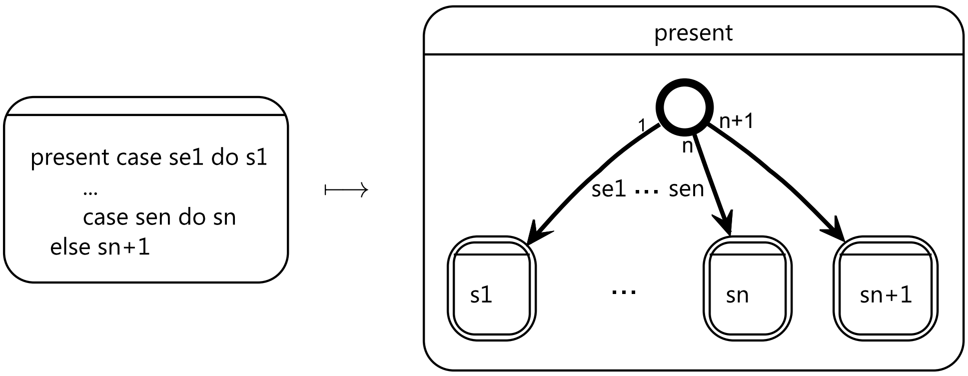


Figure 5.18: present's transformation

Transformation

```
Void rule(State s, Present p):
   let r = new Region:
   let pS = new State:
   initializeRule(s, p) ->
   s.regions.add(r) ->
   r.states.add(pS) ->
   pS.setIsInitial(true) ->
   (PresentEventBody.isInstance(p.body) ?
      // handle present ... then ... form
      (let event = ((PresentEventBody)p.body).event.expression:
       let thenPart = ((PresentEventBody)p.body).thenPart:
       handleIfSingle(r, pS, 1, event, thenPart != null ? thenPart.statement :
           null)
      )
      : // handle present cases
      handlePresentCases(r, pS, 1, ((PresentCaseList)p.body).cases)
   ) ->
   if p.elsePart != null then
      (let maxprio = (PresentCaseList.isInstance(p.body) ? ((PresentCaseList)p.
          body).cases.size + 1 : 2):
       handleIfSingle(r, pS, maxprio, null, p.elsePart.statement))
;
Void handlePresentCases(Region parent, State previous, Integer prio, List[
    PresentCase] cases):
   (cases.size > 1) ?
      (handlePresentCaseSingle(parent, previous, prio, cases.first().event.
          expression, cases.first().statement) ->
      handlePresentCases(parent, previous, prio + 1, cases.withoutFirst()))
      :
      (handlePresentCaseSingle(parent, previous, prio, cases.first().event.
          expression, cases.first().statement)
   )
;
Void handlePresentCaseSingle(Region parent, State previous, Integer prio,
    Expression e, Statement st):
   let newS = new State:
   let t = new Transition:
   newS.setIsFinal(true) ->
   parent.states.add(newS) ->
   t.setPriority(prio) ->
   t.setTrigger(convertEsterelExpression((Expression) clone(e))) ->
   t.connectTransition(previous, newS) ->
   finalizeRule(newS, st)
;
```

Listing 5.35: `present`'s transformation snippet

5.2.18 `call`

Listing 5.36: `call`'s grammar snippet

```
ProcCall:
   "call" proc=[Procedure|ID] "(" (varList+=[kexpressions::IVariable|ID]
   ("," varList+=[kexpressions::IVariable|ID])*)? ")"
   "(" (kexpressions+=Expression ("," kexpressions+=Expression)*)? ")";
```

Statement Description

The `call` statement calls an externally defined procedure. Several variables can be passed as parameters.

Equivalent macro-state

The procedure call is added as an effect of a transition connecting an initial with a final state [Küh06, p. 68].

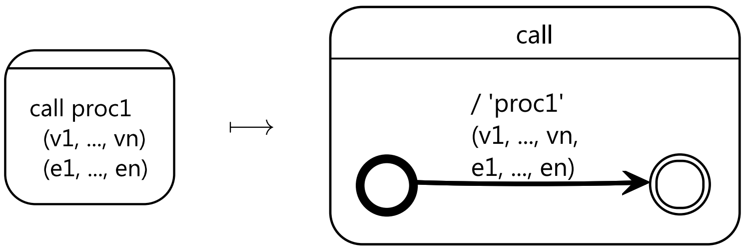


Figure 5.19: `call`'s transformation

Transformation

```
Void rule(State s, ProcCall c):
   let r = new Region:
   let initS = new State:
   let finalS = new State:
   let t = new Transition:
   let textEffect = new TextEffect:
   initializeRule(s, c) ->
   // setup
   s.regions.add(r) ->
   r.states.add(initS) ->
   r.states.add(finalS) ->
   initS.setIsInitial(true) ->
   finalS.setIsFinal(true) ->
   t.connectTransition(initS, finalS) ->

   // first add variables
   textEffect.subExpressions.addAll(c.varList.convertToReferences()) ->
   // then add expressions
   textEffect.subExpressions.addAll(c.kexpressions) ->

   // create call statement
   textEffect.setCode(c.proc.name) ->
   t.effects.add(textEffect) ->
   initS
;

List[Expression] convertToReferences(List[IVariable] vars):
   let list = {}:
   list.addAll(vars.createValObjReference()) ->
   list
;
```

Listing 5.37: `call`'s transformation snippet

5.2.19 sequence

Listing 5.38: `sequence`'s grammar snippet

```
Sequence returns Statement:
  AtomicStatement ({Sequence.list+=current} ";" list+=AtomicStatement)* ";"?;
```

Statement Description

The first statement of a sequence is started instantaneously. Upon termination it passes the control immediately to the successive statement. In case all statements are finished the sequence terminates itself.

Equivalent macro-state

Each sequential statement is represented as a macro state with the respective statement as its body. Those macro states are connected by normal terminations according to their textual order [Küh06, p. 69].

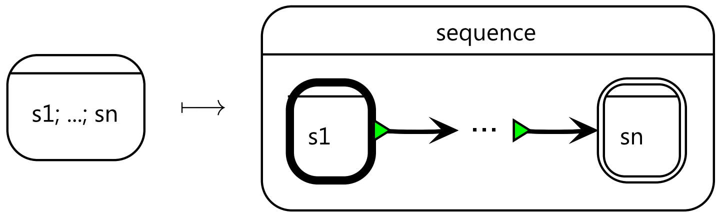


Figure 5.20: `sequence`'s transformation

Transformation

```
 Void rule(State s, Sequence seq):
   let r = new Region:
   let initial = new State:
   let list = (List[Statement]){}:
   initializeRule(s, seq) ->
   s.regions.add(r) ->
   initial.setIsInitial(true) ->
   r.states.add(initial) ->
   // as the grammar generates nested sequences, we flatten it first
   // this is important to conserve order!
   seq.flattenSequence(list) ->
   // call recursively
   ruleSequenceRecursive(r, initial, list.withoutFirst()) ->

   // process body of first sequence state
   finalizeRule(initial, list.first())
;

Void flattenSequence(Statement s, List list):
   Sequence.isInstance(s) ? (
      (Sequence)s).list.flattenSequence(list)
      :
      (list.add(s))
;

Void ruleSequenceRecursive(Region region, State previous, List[Statement]
    statements):
   let s = new State:
   let t = new Transition:
   // create the new state and add to the sequence chain
   region.states.add(s) ->
   t.setType(TransitionType::NORMALTERMINATION) ->
   t.connectTransition(previous, s) ->

   (((statements.size > 1) ?
   // if more than 1 element we have to handle another one
   ruleSequenceRecursive(region, s, statements.withoutFirst())
   :
   // else finished and the last one is a final state!
   s.setIsFinal(true))) ->
   finalizeRule(s, statements.first())
;
```

Listing 5.39: sequence's transformation snippet

5.2.20 `suspend`

Listing 5.40: `suspend`'s grammar snippet

```
Suspend:
   "suspend" statement=Statement "when" delay=DelayExpr;
```

Statement Description

The `suspend` statement pauses its execution every time a certain signal expression is true.

Equivalent macro-state

A new macro state is created with the signal expression as a suspension trigger [Küh06, p. 70].

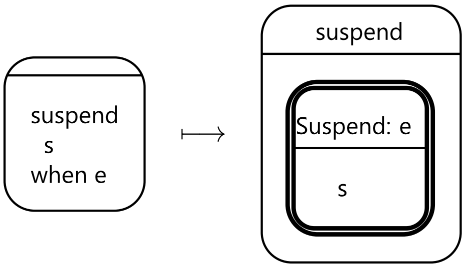


Figure 5.21: `suspend`'s transformation

Transformation

```
Void rule(State s, Suspend sus):
   let r = new Region:
   let susState = new State:
   let act = new synccharts::Transition:
   initializeRule(s, sus) ->
   s.regions.add(r) ->
   r.states.add(susState) ->
   // setup
   susState.setIsInitial(true) ->
   susState.setIsFinal(true) ->
   // add suspension
   act.addDelayToTrigger(sus.delay) ->
   susState.setSuspensionTrigger(act) ->
   // recursive
   finalizeRule(susState, sus.statement)
;
```

Listing 5.41: `suspend`'s transformation snippet

5.2.21 sustain

Listing 5.42: sustain's grammar snippet

```
Sustain:
   "sustain" ((signal=[kexpressions::ISignal|ID]) | tick=Tick) ("(" expression=
       Expression")")?;
```

Statement Description

The sustain statement emits the specified signal in every instant.

Equivalent macro-state

A simple initial state is created and a weakly aborting self transition is added. As an effect of the transition the emission of the specified signal is added. The simple state is not final. Therefore, it never stops emitting [Küh06, p. 71ff].

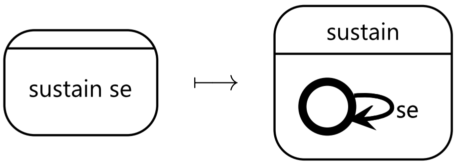


Figure 5.22: sustain's transformation

Transformation

```
Void rule(State s, Sustain sus):
   let r = new Region:
   let initS = new State:
   let finalS = new State:
   let initT = new Transition:
   let sustT = new Transition:
   let emission = new Emission:
   initializeRule(s, sus) ->
   s.regions.add(r) ->
   r.states.add(initS) ->
   r.states.add(finalS) ->
   initS.setIsInitial(true) ->
   // setup transitions
   initT.connectTransition(initS, finalS) ->
   sustT.connectTransition(finalS, finalS) ->
   emission.setSignal(sus.signal) ->
   emission.setNewValue(convertEsterelExpression((Expression) clone(sus.expression
       ))) ->
   initT.effects.add(emission) ->
   sustT.effects.add((Emission) clone(emission))
;
```

Listing 5.43: `sustain`'s transformation snippet

5.2.22 trap

Listing 5.44: trap's grammar snippet

```
Trap:
  "trap" trapDeclList=TrapDeclList "in" statement=Statement
  (trapHandler+=TrapHandler)* "end" (optEnd="trap")?;

TrapDeclList:
  trapDecls+=TrapDecl ("," trapDecls+=TrapDecl)*;

TrapDecl returns kexpressions::ISignal:
  {TrapDecl} name=ID channelDescr=(ChannelDescription)?;

TrapHandler:
  "handle" trapExpr=TrapExpr "do" statement=Statement;
```

Statement Description

The `trap` statement terminates the execution of its body statement upon occurrence of a specified trap expression. It is possible to specify certain exception statements that have to be executed if a trap expression evaluates successfully. In case several expressions evaluate to `true` the specified statements are executed in parallel.

Equivalent macro state

In SyncCharts traps are modeled as usual signal. Therefore, a new signal is introduced for each declared trap. An additional `traphalt` is created to stop the execution of the trap. All new signals are added to the current macro state.

An initial macro state contains the body statement of the trap. A normally terminating transition leads to a simple, final state, another weakly aborting transition to a simple, non-final state with the `traphalt` signal as the trigger.

All specified trap expressions are connected with `or`. The created `or` expression is added as the trigger to an immediate, weakly aborting transition that leads to a new final macro state. The latter contains parallel regions for each trap handler. Each region contains three states: First, an initial state, second, a simple, final state reached by a weakly aborting transition, and last, a final macro state containing the exception statement. The state is reached by a weakly aborting transition with the corresponding trap expression as a trigger [Küh06, p. 75ff].

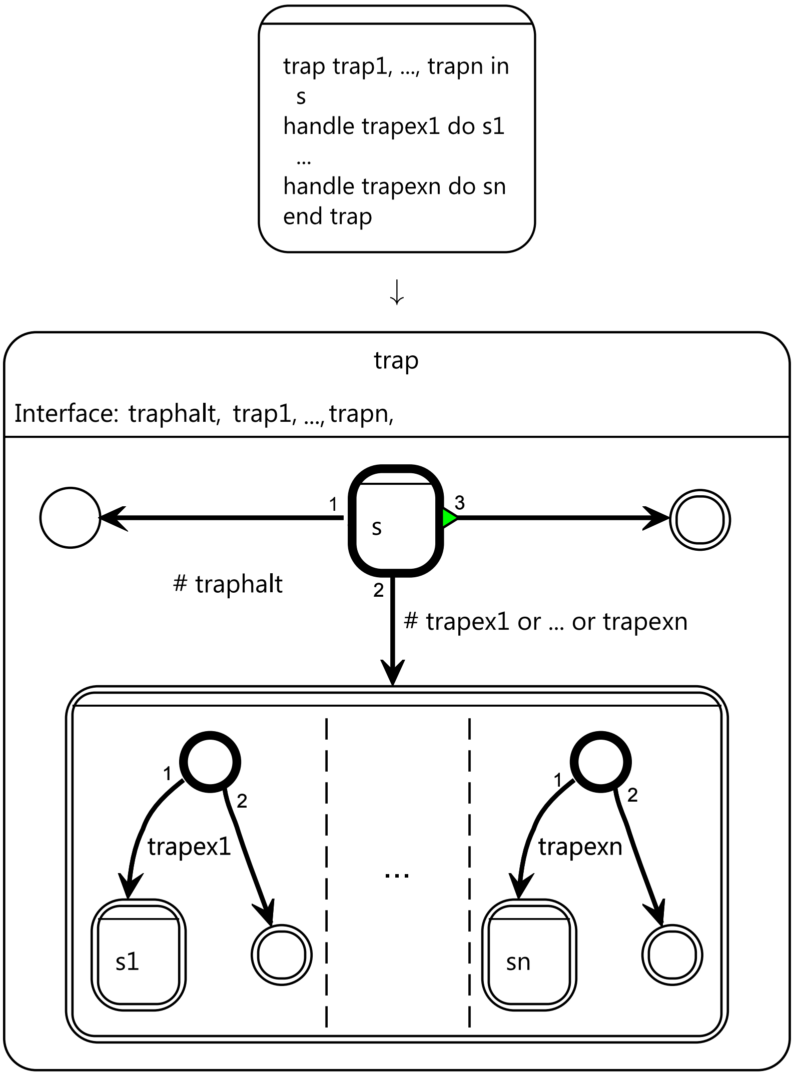


Figure 5.23: `trap`'s transformation

Transformation

```
Void rule(State s, Trap t):
   let r = new Region:
   let trapS = new State:
   let finalS = new State:
   let normalT = new Transition:
   let haltS = new State:
   let haltT = new Transition:
   let haltSig = new ISignal:
   initializeRule(s, t) ->
   s.regions.add(r) ->
   r.states.add(trapS) ->
   r.states.add(finalS) ->
   r.states.add(haltS) ->
   // normal termination
   finalS.setIsFinal(true) ->
   normalT.setType(TransitionType::NORMALTERMINATION) ->
   normalT.setPriority(3) ->
   normalT.connectTransition(trapS, finalS) ->
   // halt due to higher trap
   trapS.setIsInitial(true) ->
   // new signal to halt execution if higher trap fires
   haltSig.setName("traphalt" + getNumberOfTraphalts(s)) ->
   haltSig.addSignalToState(s) ->
   haltT.connectTransition(trapS, haltS) ->
   // immediate transition with traphalt signal
   haltT.setIsImmediate(true) ->
   haltT.setPriority(1) ->
   (let vo = new ValuedObjectReference:
    vo.setValuedObject(haltSig) ->
    haltT.setTrigger(vo)) ->
   // setup exit state
   (let handleState = new State:
    let handleExpr = new OperatorExpression:
    let handleT = new Transition:
    // setup state
    handleState.setLabel("Trap Handler State") ->
    handleState.setIsFinal(true) ->
    r.states.add(handleState) ->
    // copy all traps as signals
    t.trapDeclList.trapDecls.addTrapSignalToState(s) ->
    // transition
    handleT.setIsImmediate(true) ->
    handleT.setPriority(2) ->
    handleT.connectTransition(trapS, handleState) ->
    // collect traps
    (t.trapDeclList.trapDecls.size > 1 ?
      (t.trapDeclList.collectTraps(handleExpr) ->
```

```
      handleExpr.setOperator(OperatorType::OR) ->
      handleT.setTrigger(handleExpr))
    : (let valObjRef = new ValuedObjectReference:
      valObjRef.setValuedObject(t.trapDeclList.trapDecls.first()) ->
      handleT.setTrigger(valObjRef))
    ) ->
    // create handler if existing
    if !t.trapHandler.isEmpty then
      handleTrapHandler(handleState, t.trapHandler)
   ) ->

   finalizeRule(trapS, t.statement)
;
Void handleTrapHandler(State handleState, List[TrapHandler] handler):
   handleTrapHandlerSingle(handleState, handler.first()) ->
   if(handler.size > 1) then
      handleTrapHandler(handleState, handler.withoutFirst())
;
Void handleTrapHandlerSingle(State handleState, TrapHandler handler):
   let r = new Region:
   let initS = new State:
   let macroS = new State:
   let macroT = new Transition:
   let finalS = new State:
   let finalT = new Transition:
   handleState.regions.add(r) ->
   r.states.add(initS) ->
   r.states.add(macroS) ->
   r.states.add(finalS) ->
   initS.setIsInitial(true) ->
   // finalState setup
   finalS.setIsFinal(true) ->
   finalT.setIsImmediate(true) ->
   finalT.setPriority(2) ->
   finalT.connectTransition(initS, finalS) ->
   // macroState setup
   macroS.setIsFinal(true) ->
   macroT.setIsImmediate(true) ->
   macroT.setPriority(1) ->
   macroT.connectTransition(initS, macroS) ->
   macroT.setTrigger(convertEsterelExpression((Expression) clone(handler.trapExpr)
       )) ->
   // recursive
   macroS.setJavaBodyReference(handler.statement) ->
   macroS.recursiveRule(handler.statement)
;
```

Listing 5.45: `trap`'s transformation snippet

5.2.23 exit

Listing 5.46: exit's grammar snippet

```
Exit:
   "exit" trap=[TrapDecl|ID] ("(" expression=Expression ")")?;
```

Statement Description

The exit statement triggers a specified trap. Possible traps defined in a hierarchy level between the exit and the triggered trap are terminated.

Equivalent macro-state

The equivalent SyncChart is similar to the emit statement presented in Section 5.2.9. As the signal the specified trap is emitted. To terminate traps defined on a hierarchy level between exit and the corresponding trap their traphalt signal is emitted [Küh06, p. 84].

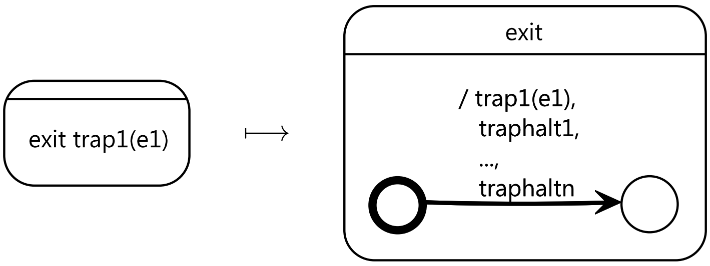


Figure 5.24: exit's transformation

Transformation

```
Void rule(State s, Exit e):
   let initS = new State:
   let finalS = new State:
   let r = new Region:
   let emitTrans = new Transition:
   let emission = new Emission:
   initializeRule(s, e) ->
   s.regions.add(r) ->
   initS.setIsInitial(true) ->
   // add new states to region
   r.states.add(initS) ->
   r.states.add(finalS) ->

   // add the effect
   emitTrans.setIsImmediate(true) ->
   emission.setSignal(e.trap) ->
   emission.setNewValue(convertEsterelExpression((Expression)clone(e.expression)))
        ->
   emitTrans.effects.add(emission) ->
   // find and add corresponding traphalts
   findAndAddCorrespondingTraphalts(e.trap, s, emitTrans) ->
   // add transition to state
   emitTrans.connectTransition(initS, finalS) ->
   initS
;
```

Listing 5.47: `exit`'s transformation snippet

5.3 Optimization of SyncCharts

As explained in Section 1.2 an optimization of a transformed Esterel program is essential to obtain a reasonable SyncCharts diagram.

A SyncCharts optimization can be used for removing obsolete elements of any arbitrary SyncCharts diagram. Therefore, the presented transformations are also applicable beyond the scope of this thesis.

The optimization rules were not proven formally, in contrast to the Esterel to SyncCharts transformation rules. So far, they are results of logical reasoning and yield correct results for several small test examples. For this reason, it is not guaranteed that they preserve semantics.

5.3.1 Concept

In the following sections the concept of each optimization rule is presented separately. An optimization rule is applied to a state that meets certain conditions. These conditions are enumerated and the modifications that are made are described with the help of a short text.

Furthermore, two representative SyncCharts are presented illustrating the modifications. The left hand diagram contains a state having some optimization potential, the right hand diagram is the optimized result.

(a) arbitrary macro state (b) macro state without final state

Figure 5.25: Notations of representative optimization diagrams

Some notations are used to keep these representatives as compact as possible but still conserve generality. The state seen in Figure 5.25a exemplifies an arbitrary macro state or a simple state. Figure 5.25b expresses a macro state that does not contain any final child state within its first hierarchically layer. Just as in Section 5.2 the two notations below are used.

`e1, en, ex:` an arbitrary effect (e. g., `/ 0`).

`t1, tn, tx:` an arbitrary trigger (e. g., `1 < 5`).

Additionally, the Xtend implementation of each rule is presented. First, a predicate testing the conditions is shown. Second, the rule applying the stated modifications is listed.

5.3.2 Optimization Rule 1 : Removal of Unessential Conditional Pseudostates

Conditions

1. The state is a conditional pseudostate with just one outgoing transition.
2. The outgoing transition contains no triggers.

Modifications

All of the pseudostate's incoming transitions are bent to the target state of the outgoing transition. The pseudostate and its outgoing transition are removed.

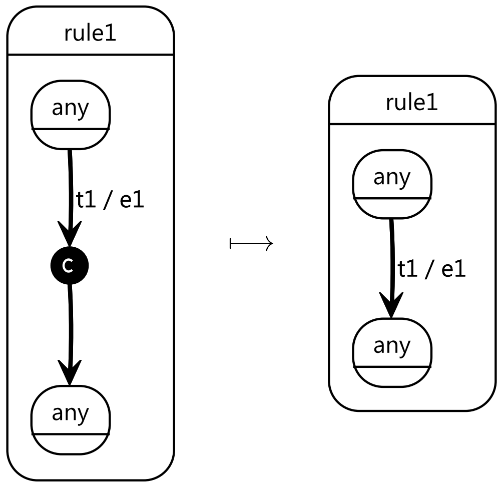


Figure 5.26: Removal of Unessential Conditional Pseudostates

Transformation

```
Boolean rule1applies(State s):
   switch {
      case isConditional(s) && hasNumberOfOutgoingTrans(s, 1) :
         (isTransitionWithoutTaE(s.outgoingTransitions.get(0)) ? true : false)
      default : false
   }
;

Void rule1(State s):
   let targetState = s.outgoingTransitions.get(0).targetState:
   let incomingTrans = s.incomingTransitions.copyListTrans():
   // bend transition to new target
   incomingTrans.setTargetState(targetState) ->
   s.outgoingTransitions.get(0).removeTransition() ->
   // remove conditional state
   s.removeStateFromRegion()
;
```

Listing 5.48: Transformation snippet of `rule1`

5.3.3 Optimization Rule 2 : Removal of Unessential Simple States (1)

Conditions

1. The state is a simple state, which is neither final nor initial.
2. The state has only one outgoing transition, which is immediate.
3. The transition has no further triggers.

Modifications

All of the simple state's incoming transitions are bent to the target state of the outgoing transition. The effects of the outgoing transition are added to each incoming transition. The simple state and its outgoing transition are removed.

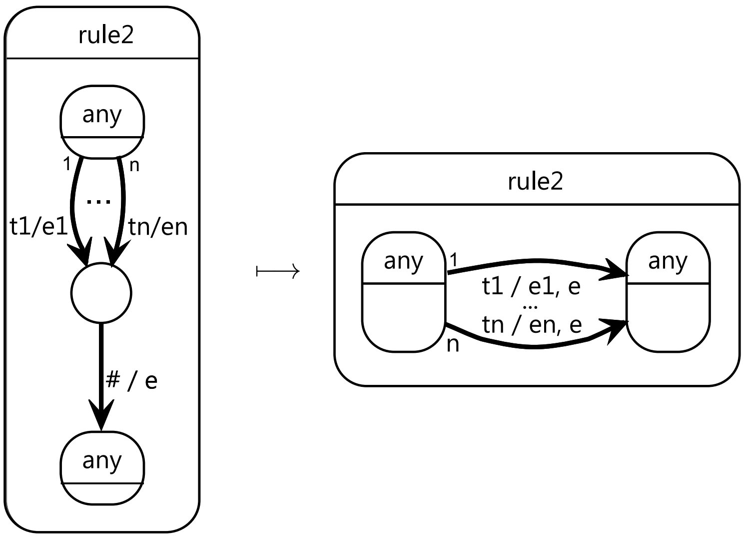


Figure 5.27: Removal of Unessential Simple States (1)

Transformation

```
Boolean rule2applies(State s):
   switch {
      case isSimpleState(s) && !s.isFinal && !s.isInitial &&
         hasNumberOfOutgoingTrans(s, 1) && isImmediateTransition(s.
             outgoingTransitions.get(0)) &&
         isTransitionWithoutT(s.outgoingTransitions.get(0)):
            true
      default : false
   }
;

Void rule2(State s):
   let incomingTrans = s.incomingTransitions.copyListTrans():
   let outgoingTrans = s.outgoingTransitions.first():
   let effects = outgoingTrans.effects:
   // bend transitions
   incomingTrans.setTargetState(outgoingTrans.targetState) ->
   // append effects
   incomingTrans.addEffects(effects) ->

   // if s is initial .. the new target needs to be initial
   if s.isInitial then
      outgoingTrans.targetState.setIsInitial(true) ->

   // remove state and outgoing transition
   outgoingTrans.removeTransition() ->
   s.parentRegion.states.remove(s)
;
```

Listing 5.49: Transformation snippet of `rule2`

5.3.4 Optimization Rule 3 : Removal of Unessential Simple States (2)

Conditions

1. The state is a simple state, which is neither final nor initial.
2. It exists exactly one incoming and one outgoing transition with the same trigger.
3. The transition is not immediate.

Modifications

The delay counters of each trigger are summed up and set as the delay of the incoming transition. The incoming transition is bent to the target state of the outgoing transition. The simple state and the outgoing transition are removed.

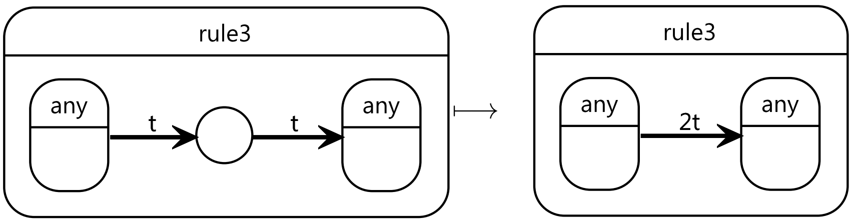


Figure 5.28: Removal of Unessential Simple States (2)

Transformation

```
Boolean rule3applies(State s):
   switch {
      case isSimpleState(s) && s.hasNumberOfIncomingTrans(1) && s.
          hasNumberOfOutgoingTrans(1)
      && s.hasOnlyMatchingTriggerTrans() :
          true
      default : false
   }
;

Void rule3(State s):
   let in = s.incomingTransitions.get(0):
   let out = s.outgoingTransitions.get(0):
   out.setSourceState(in.sourceState) ->
   out.setDelay(in.delay + out.delay) ->
   if s.isInitial then
      out.targetState.setIsInitial(true) ->
   removeTransition(in) ->
   s.removeStateFromRegion()
;
```

Listing 5.50: Transformation snippet of `rule3`

5.3.5 Optimization Rule 4 : Merging of Simple Final States

Conditions

1. The state is a macro state containing several simple final states.

Modifications

One of the simple final states is chosen to remain. The incoming transitions of all other simple final states are bent to this one final state, and the states themselves are removed.

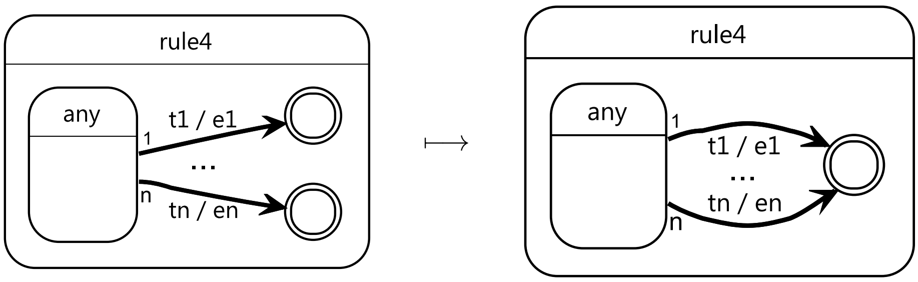


Figure 5.29: Merging of Simple Final States

Transformation

```
Boolean rule4applies(State s):
   switch {
      case hasMultipleSimpleFinalSubStates(s) :
         true
      default: false
   }
;

Void rule4(State s):
   let regions = (List[Region]) s.regions.select(e | e.states.select(e|e.isSimpleState() && e.isFinal).size > 1):
   regions.handleRule4()
;

Void handleRule4(Region r):
   let simpleFinals = r.states.select(e|e.isSimpleState() && e.isFinal):
   // keep the first one and bend transitions there
   let firstFinal = simpleFinals.first():
   let simpleWithoutFirst = simpleFinals.withoutFirst():
   simpleWithoutFirst.handleRule4rec(firstFinal)
;

Void handleRule4rec(State s, State to):
   let incomings = s.incomingTransitions.copyListTrans():
   incomings.bendAndRemove(to)
;

Void bendAndRemove(Transition t, State to):
   let oldTarget = t.targetState:
   t.setTargetState(to) ->
   oldTarget.removeStateFromRegion()
;
```

Listing 5.51: Transformation snippet of `rule4`

5.3.6 Optimization Rule 5 : Removal of Unessential Normal Terminations

Conditions

1. The state is a macro state containing no final state.
2. The state has an outgoing normally terminating transition.

Modifications

The normally terminating transition is removed. If this removal leaves a state without any incoming transition, this state is removed as well.

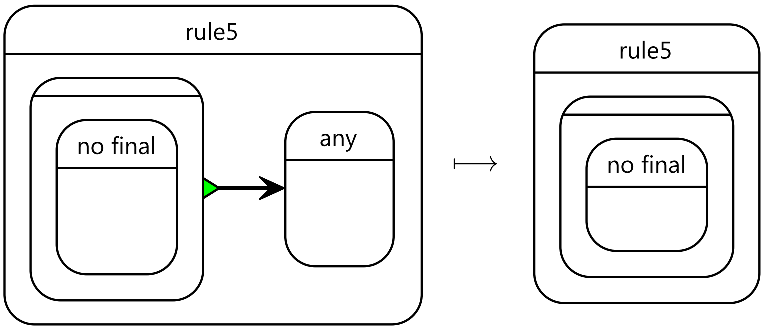


Figure 5.30: Removal of Unessential Normal Terminations

Transformation

```
Boolean rule5applies(State s):
   switch {
      case !isSimpleState(s) && hasOutNormalTransitions(s) && !hasFinalSubState(s)
           && !hasOnlySelfLoop(s):
            true
      default: false
   }
;

Void rule5(State s):
   let outTrans = s.outgoingTransitions.select(e|e.type == TransitionType::
       NORMALTERMINATION
      && (s.isInitial || e.sourceState != e.targetState)).copyListTrans():
   outTrans.removeTransAndPossiblyState()
;

Void removeTransAndPossiblyState(Transition t):
   let target = t.targetState:
   t.removeTransition() ->
   if(target.incomingTransitions.isEmpty) then
      removeStateFromRegion(target)
;
```

Listing 5.52: Transformation snippet of `rule5`

5.3.7 Optimization Rule 6 : Removal of Unessential Macro States

Conditions

1. The state is a macro state without defined signals or variables.
2. The state has no outgoing weakly or strongly aborting transitions.
3. It is not a parallel macro state and has a parent macro state.

Modifications

All of the macro state's incoming transitions are bent to the contained initial state.

The contained final macro states are not final anymore and, in case a normal termination exists, get a copy of the normal termination. The final simple states are not final anymore as well and, in case a normal termination exists, get a new immediate weakly aborting transition, leads to the target of the normal termination. Any effect is copied.

The macro state and a possible normal termination are removed. The contained elements are added to the higher hierarchy level.

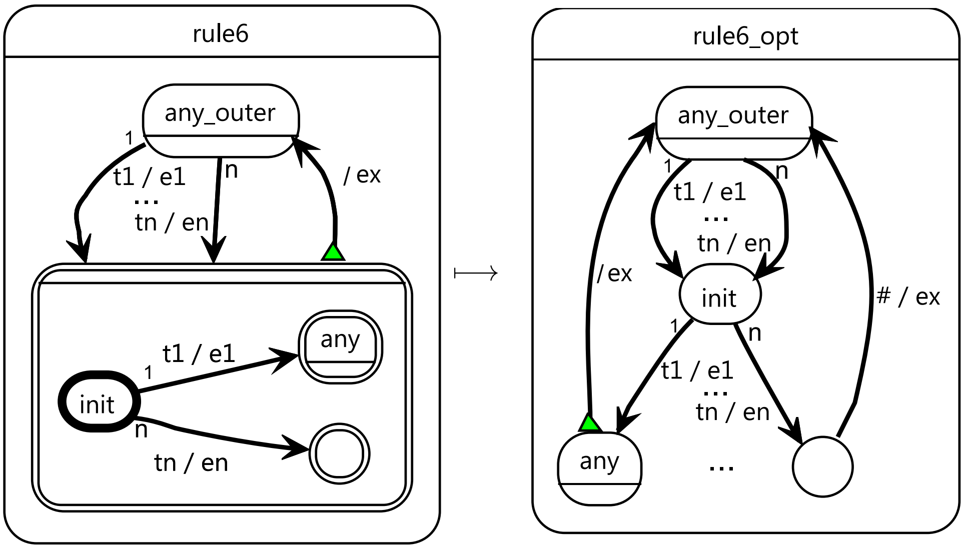


Figure 5.31: Removal of Unessential Macro States

Transformation

```
Boolean rule6applies(State s):
   switch {
      case !hasOutWeakTransitions(s) && !hasOutStrongTransitions(s) && !
          isSimpleState(s) &&
         !hasSignalsVariables(s) && hasParentMacroState(s) && !
             isParallelMacroState(s):
            true
      default: false
   }
;
Void rule6(State s, List[State] states):
   let r = s.regions.first():
   let initial = findInitialState(r):
   let finalMacros = r.states.select(e|!isSimpleState(e) && e.isFinal):
   let finalSimples = r.states.select(e|isSimpleState(e) && e.isFinal):
   let incomingTrans = s.incomingTransitions.copyListTrans():
   let normalTerms = s.outgoingTransitions.select(e|e.type == TransitionType::
       NORMALTERMINATION):
   // do not make states without incoming transitions or only one selfloop non-
       initial
   if (!incomingTrans.isEmpty) then
      (if (!s.hasOnlySelfLoop()) then
         initial.setIsInitial(false)) ->
   // reroute all incoming transitions to the initial state
   incomingTrans.setTargetState(initial) ->
   initial.incomingTransitions.addAll(incomingTrans) ->
   // if normal termination exist
   (s.outgoingTransitions.size == 1) ?
   (finalMacros.setIsFinal(false) -> // final macros become non-final
   finalSimples.setIsFinal(false) -> // final simples become non-final
   finalSimples.createImmediateWeakAbortTo(s.outgoingTransitions.get(0).
       targetState,
      s.outgoingTransitions.get(0)) ->
   if !normalTerms.isEmpty then
      finalMacros.copyNormalTransitionFrom(normalTerms.get(0))) : s ->
   (let copyStates = r.states.copyListTrans(): // copy whole stuff
   s.parentRegion.states.addAll(copyStates) ->
   // add the inner states to the list, as there might be new optimization
       potential
   states.addToFrontOfList(copyStates)) ->
   // remove old normal termination and old state
   if (!normalTerms.isEmpty) then
      normalTerms.removeTransition() ->
   s.removeStateFromRegion()
;
```

Listing 5.53: Transformation snippet of `rule6`

5.3.8 Optimization Rule 7 : Removal of Macro States with Only One Sub-State

Conditions

1. The state is a macro state with just one sub-state.

2. The state is not a parallel macro state and has a parent macro state.

Modifications

The modification of the sub-state is analogous to the one presented in Section 5.3.7. Additionally, possible weakly and strongly aborting transitions are added as outgoing transitions. The signal and the variable declarations of the sub-state can be moved into the macro state.

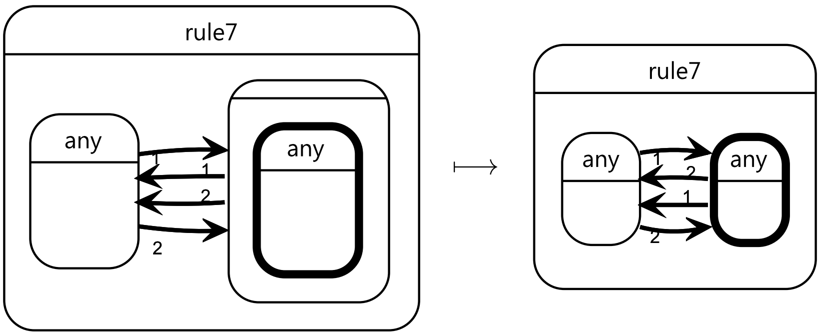


Figure 5.32: Removal of Macro States with Only One Sub-State

Transformation

```
Boolean rule7applies(State s):
   switch {
      case !isSimpleState(s) && s.hasParentMacroState() && s.hasNumberOfSubStates
          (1)
         !hasSignalsVariables(s):
            true
      default: false
   }
;

Void rule7(State s):
   let parentReg = s.parentRegion:
   let incomingT = s.incomingTransitions.copyListTrans():
   let outgoingT = s.outgoingTransitions.copyListTrans():
   let states = (List[State]) {}:
   s.regions.collectStates(states) ->
   (let found = states.get(0):
    parentReg.states.add(found) ->
    incomingT.setTargetState(found) ->
    outgoingT.setSourceState(found)
   ) ->
   s.removeStateFromRegion()
;
```

Listing 5.54: Transformation snippet of `rule7`

5.3.9 Optimization Rule 8 : Checking of a State's Final Character

Conditions

1. The state is a final macro state containing no final state.

Modifications

The state is no longer marked as final.

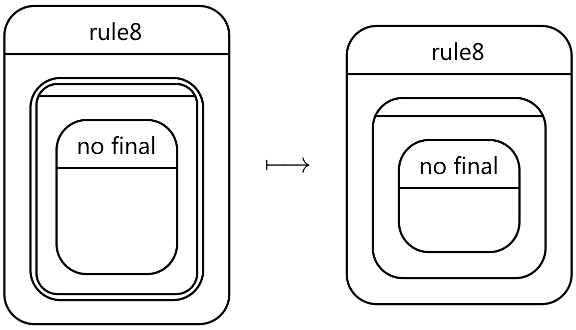


Figure 5.33: Checking of a State's Final Character

Transformation

```
Boolean rule8applies(State s):
   switch {
      case !isSimpleState(s) && s.isFinal && !s.hasFinalSubState():
            true
      default: false
   }
;

Void rule8(State s):
   s.setIsFinal(false)
```

Listing 5.55: Transformation snippet of `rule8`

6 Implementation

First, the requirements stated for the implementation are reconsidered.

1. The user should be able to process steps, to execute the whole transformation at once, and to execute the transformation and optimize the transformed result at the same time. Also, back steps have to be supported.

2. The changes of one single transformation have to be presented in a way that is clearly understandable for the user.

3. A selection of the context, in which the next transformation is performed, has to be applicable by the user.

The KIELER project serves as a basis for the whole implementation. KiVi and KIEM are used to satisfy the demands of good usability and different kinds of execution modes. As mentioned in Section 5.2 Xtend serves as the transformation language.

Figure 6.2 shows a class diagram of the core elements. In the diagram a class `TransformationDescriptor` and an interface `TransformationContext` can be seen. Both refer to the previously introduced specification of a transformation. The actual implementation of the latter depends on the technology used. Hence, just one method named `execute` is specified. The `execute` method demands a `TransformationDescriptor` as the argument and is supposed to perform the actual transformation for the chosen technology and the chosen context. After the transformation is executed any result is stored in the `TransformationDescriptor` and can be retrieved.

KiVi is used to execute an arbitrary transformation. A `Combination` contributes the control buttons, mentioned above, to Eclipse's user interface. As soon as a button is clicked by the user the `Combination` sets up the `TransformationContext`, passes it to a `TransformationEffect`, and schedules the effect. The `TransformationEffect` is executed by KiVi concurrently, it calls the `execute` method of the context, and stores the result. See Figure 6.1 for a sequence diagram.

6.0.10 Creation of a `TransformationContext`

To execute a transformation the `TransformationContext` has to be assembled. Because Xtend and Eclipse are used the following information has to be gathered. It is combined in the `XtendTranformationContext` seen in Figure 6.3.

Model: It specifies the model which should be transformed (e. g., the root element of a SyncChart).

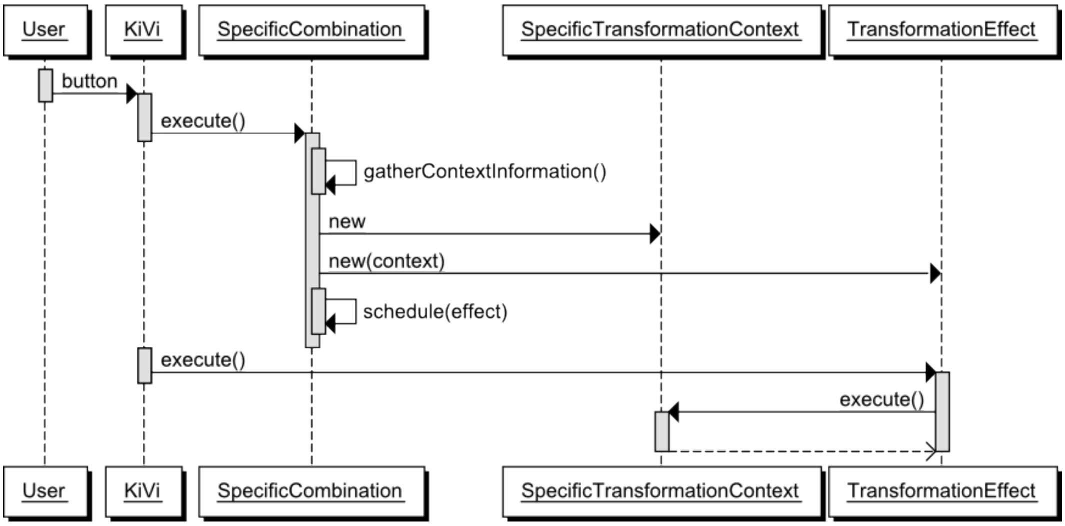


Figure 6.1: Sequence diagram of the interaction with KiVi

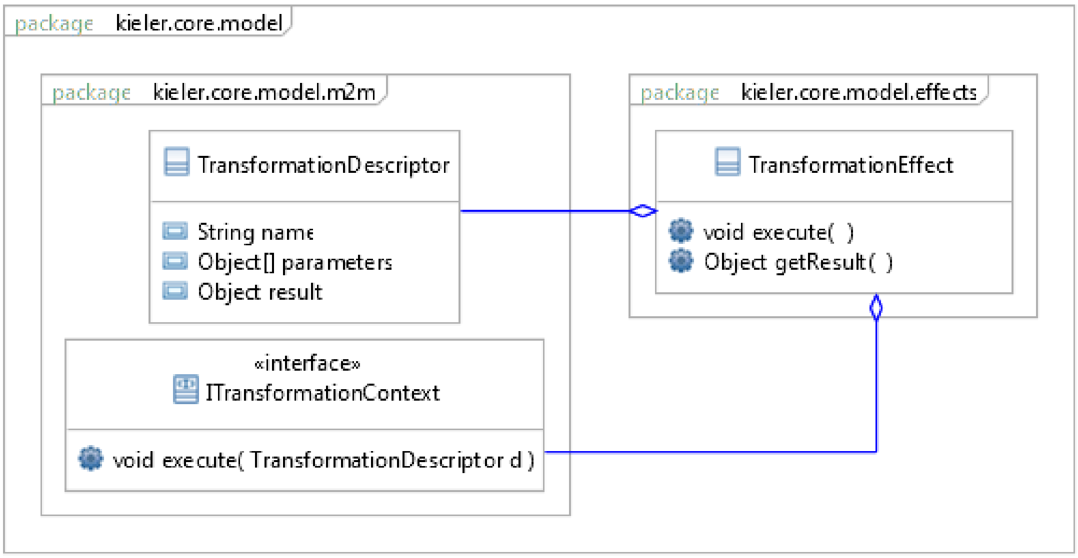


Figure 6.2: Class diagram of the core package

Transactional Editing Domain: The `TransactionalEditingDomain` is part of EMF and manages commands that modify a model. It also holds an `CommandStack` which can be used to provide *undo/redo* functionality. For further information see the EMF documentation[1].

[1]http://help.eclipse.org/helios/nav/23

Base Packages: A base package can be seen as the Java representation of a metamodel. It contains the information how to access a metamodel's objects (e. g., the `EsterelPackage`).

Extension File: The file containing the Xtend extensions needs to be specified.

Global Variables: Global variables used by the implemented Xtend methods have to be available.

Xtend Facade: The latter three points can be combined in terms of an `XtendFacade`.

6.0.11 Generic Execution

In terms of preserving the unity of the Esterel to SyncCharts implementation and the SyncCharts optimization implementation an abstract class is provided. The class `AbstractTransformationDataComponent` can be seen in Figure 6.3. It serves as the common base for the `EsterelToSyncChartsDataComponent` and the `SyncChartsDataComponent`.

```
step()
   descriptor = getNextTransformation()
   if (descriptor != null)
      facade = new XtendFacade(getBasePackages(), getTransformationFile())
      context = new XtendTransformationContext(facade, descriptor,
          getTransactionalEditingDomain())

      if (kiemMode)
         // in kiemMode execute directly without the use of KiVi
         effect = new TransformationEffect(context.execute())
         effect.execute()
      else
         // remember the current context
         this.currentContext = context

   else
      doPostTransformation()
```

Listing 6.1: Java pseudo code for the `step` method

The `AbstractTransformationDataComponent` extends the `DataComponent` class specified by KIEM. In this way a simple mechanism for step-wise execution is contributed because of the natural functionality of KIEM. The `step()` method implements the retrieval of the necessary information and execution of the transformation, as described below.

Because a `Combination` is supposed to be the controlling piece and the execution should be processed as a viewmanagement effect the `DataComponent` can operate in two different modes. Either it can be used stand-alone, or it can serve as an *information-collector* providing the created `TransformationContext` to a `Combination`. Listing 6.1 shows the implementation of the `step` method in pseudo code.

In Figure 6.3 the class `XtendTransformationCommand` can be seen. It extends the `RecordingCommand` class provided by EMF. An `RecordingCommand` records all changes, which are made concerning the underlying model, during the command's execution. This allows the applied changes to be undone. For this reason, the `XtendTransformationContext` uses an `XtendTransformationCommand` for execution to support *undo* functionality. Figure 6.4 depicts the creation of such a context as a schematic. Listing 6.1 lists Java pseudo code for the `step` method.

Figure 6.3: Class diagram of the Java implementation

In the following, the methods demanded by the abstract class `AbstractDataComponent` are discussed in further detail.

AbstractTransformationDataComponent(globVars): A map with global variables can be passed to the constructor to make them available for the use within Xtend.

getBasePackages(): The extending class has to provide all metamodels that are required by the transformation.

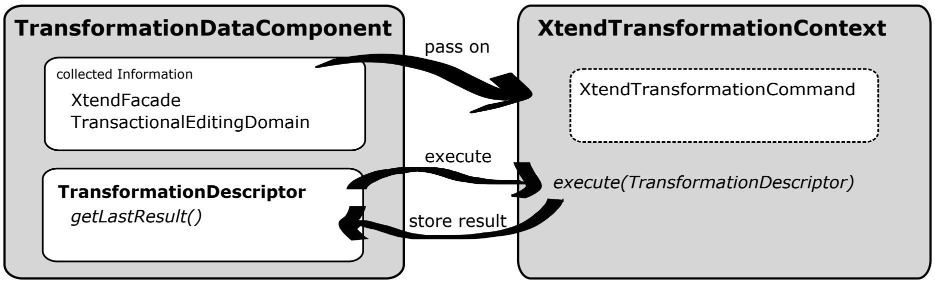


Figure 6.4: Combining of a `TransformationContext`

getNextTransformation(): The next transformation that should be performed has to be passed as a `TransformationDescriptor`.

getTransformationFile(): The extension file (`.ext`) containing the used extensions has to be defined.

doPostTransformation(): Anything that has to be processed after the overall transformation has finished should be done in this method.

6.1 Implementation of the Esterel to SyncCharts Transformation

In the first part of this section the Xtend implementation of the transformation rules is looked at in further detail. Afterwards, the created Java classes are presented.

6.1.1 Initial Transformation

The first thing to consider is the fact that an Esterel source code file has to be transferred into a SyncChart initially. On the left side of Figure 6.5 an Esterel source file with the `ABRO` module can be seen. On the right side the initially transformed SyncChart is presented. It consists of one single macro state called *Esterel State*, which has the Esterel module as body text.

This initial step is implemented in Java without the use of Xtend. First, a new SyncChart is created programmatically. Second, the Esterel `module` is added as body text to the SyncChart's root state. Also, the actual model is referenced by using the `bodyReference` field of each SyncChart state, which can be seen in the corresponding metamodel (Figure 3.7). Finally, the new SyncChart is opened in the editor.

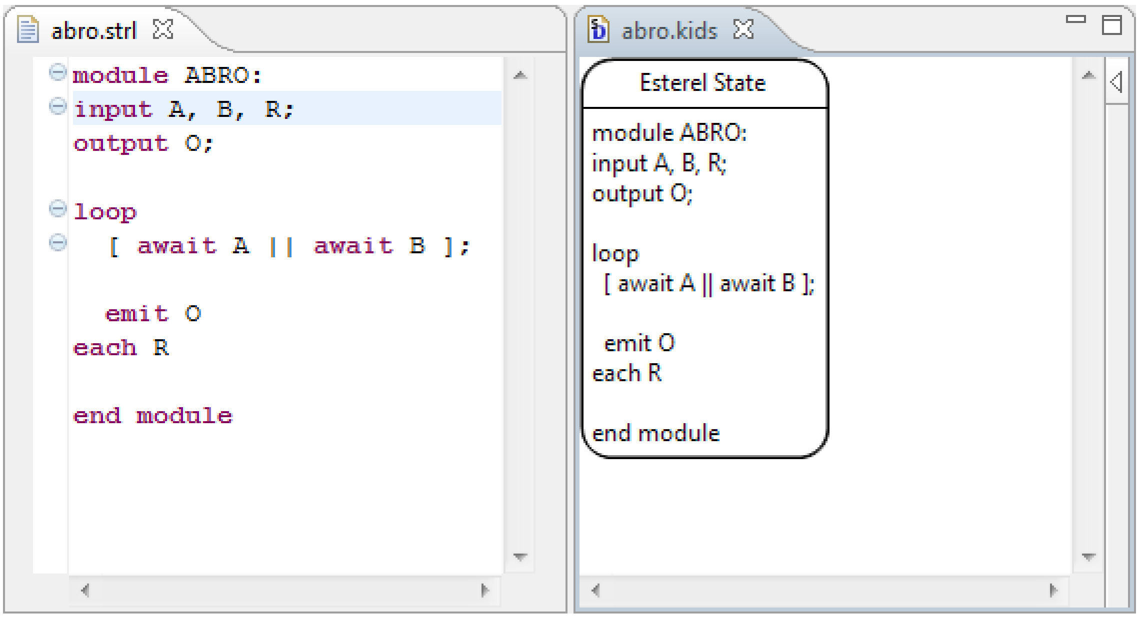


Figure 6.5: Initial transformation of an Esterel `module`

6.1.2 Xtend Implementation

The Xtend implementations of the concrete transformation rules are already presented in each respective subsection of Section 5.2. In the following, the additional methods necessary to process a transformation are explained. All used utility methods are listed in Listing A.2 in the appendix.

To achieve an efficient implementation the decision whether to finish the transformation, or to do just one single step is passed to the Xtend via a global variable. Otherwise the `XtendFacade` would have to be called separately for each transformable Esterel element by Java.

For this reason a so-called `recursiveRule` is introduced. It is listed in Listing 6.2 and is supposed to be called at the end of each transformation rule with further child Esterel elements. In line 2 the decision is made whether to execute a further transformation rule or not to depend on the global variable `recursive`.

```
1 Void recursiveRule(State s, emf::EObject e):
2    if ((boolean) GLOBALVAR recursive) then
3       rule(s, e)
4 ;
```

Listing 6.2: Xtend method `recursiveRule`

Listing 6.3 presents the `initializeRule` method. In line 2 any body text is removed from the current state. In line 3 the label of `s` is adapted according to the passed `esterelObject`. E.g., The `abort` statement would yield a state labeled *abort State*.

```
1 Void initializeRule(State s, emf::EObject esterelObject):
2    removeBodyText(s) ->
3    s.setLabelIfEmpty(esterelObject.metaType.name.replaceAll("esterel::", "") + " State")
4 ;
```

Listing 6.3: The `initializeRule` method

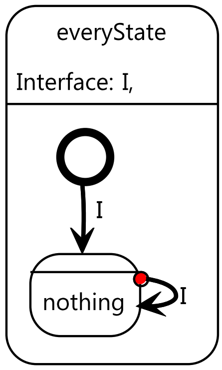
Listing 6.4 shows the `finalizeRule` method, which is called at the end of each transformation rule. In line 2 the passed `esterelObject` is set as `BodyReference` of the current state `s`. In line 3 the `recursiveRule` is called and determines whether the transformation stops at this point or not.

```
1 Void finalizeRule(State s, emf::EObject esterelObject):
2    setJavaBodyReference(s, esterelObject) ->
3    recursiveRule(s, esterelObject)
4 ;
```

Listing 6.4: The `finalizeRule` method

```
1    Void rule(State s, EveryDo e):
2    let r = new Region:
3    let initS = new State:
4    let everyS = new State:
5    let initT = new Transition:
6    let everyT = new Transition:
7    initializeRule(s, e) ->
8    // setup states
9    s.regions.add(r) ->
10   r.states.add(initS) ->
11   r.states.add(everyS) ->
12   initS.setIsInitial(true) ->
13   // init transitions
14   initT.setType(TransitionType::WEAKABORT) ->
15   everyT.setType(TransitionType::WEAKABORT) ->
16   initT.connectTransition(initS, everyS) ->
17   everyT.connectTransition(everyS, everyS) ->
18   // add delays
19   initT.addTriggerToTransition(e.delay) ->
20   everyT.addTriggerToTransition(e.delay) ->
21   // recursive
22   finalizeRule(everyS, e.statement)
23 ;
```

(a) Input



(b) Transformed

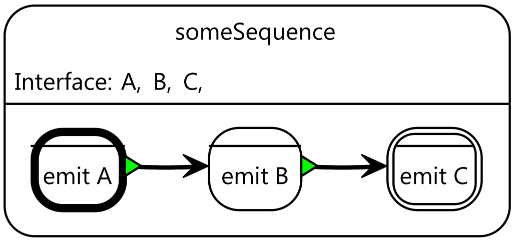
Figure 6.6: Transformation of the `every` statement

In the following, the implementation of the `every` statement, presented in Section 5.2.10, is explained in further detail. It should serve as the representative for all of the other rules.

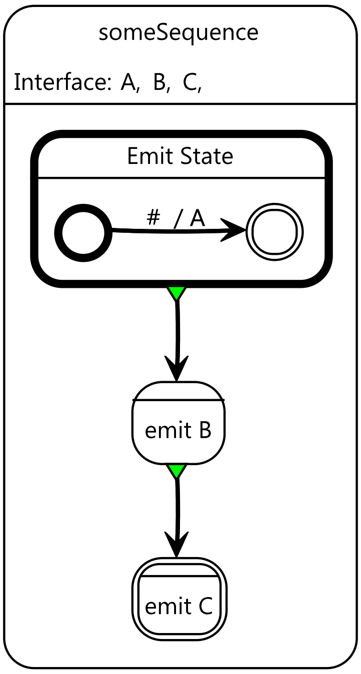
As seen in Figure 6.6 two new states and two new transitions have to be created. This is done in line 3–6 by using the `let` expression. In line 7 the `initializeRule` is called, it changes the state's name into *every State*, and removes the body text. The statement specific transformation takes place in line 9–20. The first transition is marked as a `WEAKABORT`, the second one as a `STRONGABORT`, and both transitions are connected to their respective states. The delay expression specified by the `every` statement is set as the trigger of each transition. Finally, the `finalizeRule` is called with `every`'s body statement and the new macro state `everyS`.

6.1.3 Java Implementation

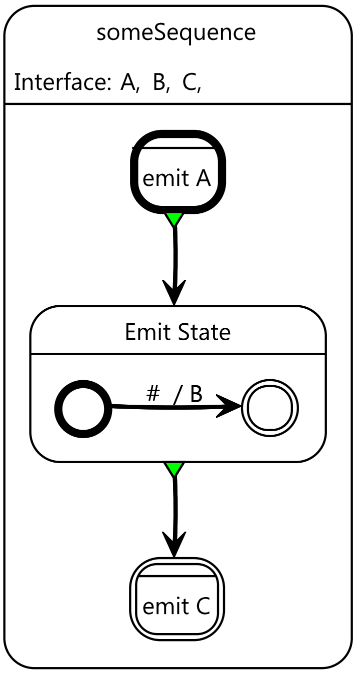
In the following, the implementation of the `EsterelToSyncChartsDataComponent`, seen in Figure 6.3, is described. As it extends the `AbstractTransformationDataComponent` all demanded methods are discussed in further detail.



(a) Initial SyncChart



(b) No state was selected



(c) The `emit B` state was selected

Figure 6.7: The effect of pre-transformation state selection

Global Variables

For this transformation just one global variable is specified. The `recursive` variable determines whether Xtend should stop after the transformation of one Esterel element or if it should transform all available elements.

Base Packages

- `KExpressionsPackage`: The `KExpressions` metamodel as introduced in Section 4.1.1.
- `EsterelPackage`: The Esterel metamodel.

- `SyncChartsPackage`: The SyncCharts metamodel, see Figure 3.7.
- `EcorePackage`: The base metamodel for all of the other metamodels.

Retrieval of the Next `TransformationDescriptor`

```
State getNextTransformableState(State parent)
   if parent.isTransformable then
      return parent

   foreach child in parent.childrenStates
      if child.isTransformable then
         return child

   foreach child in parent.childrenStates
      next = getNextTransformableState(child)
      if next != null then
         return next

   return null
```

Listing 6.5: Retrieving the next transformable state

```
List[State] getAllTransformableStates(State parent)
   foundStates = List[State]
   if parent.isTransformable then
      foundStates.add(parent)
      return foundStates

   foreach child in parent.childrenStates
      if child.isTransformable then
         foundStates.add(child)
      else
         foundStates.addAll(getAllTransformableStates(child)

   return foundStates
```

Listing 6.6: Retrieving all transformable states

Listing 6.5 and Listing 6.6 show pseudo code of the methods used for the retrieval of the next transformable state or in case of the latter one all possible transformable states.

The `getNextTransformableState` method is used for regular step-wise execution. As the `parent` state either the root state, or a state selected by the user is passed to allow the transformation in a certain context. The difference originating from such a selection is depicted in Figure 6.7. No selection within the initial SyncChart would yield the lower left result. Selecting the state that contains the `emit B` statement yields the lower right result.

The method returns the first state of `parent`'s hierarchy that is transformable or `null`. To do so the passed `parent` state is checked for transformability in line 2. In line 5–7 the same is done for the child states of the current `parent`. The children of the child states are tested in line 9–12.

The `getAllTransformableStates` method is used to determine all states that are transformable and it therefore returns a list of states. This method is always called with the model's root state as `parent` parameter. If the root element still needs to be transformed, it is returned immediately, as seen in line 5. If it is already transformed, all child states are scanned and, in case any state is transformable, added to the list to be returned. In line 10 this strategy is continued recursively. If everything has already been transformed, an empty list is returned.

The `TransformationDescriptor` can be put together with the help of the information about the next transformable state. Either the next transformable state or the list with all transformable states is used as `parameters`. As mentioned earlier, all rules are named *rule*, which is passed as the `name` to the `TransformationDescriptor`.

In case a list is passed an additional Xtend entry rule has to be specified which processes each transformable state sequentially.

Post Transformation

Because that expressions used in the Esterel program are just copied into the SyncChart the signal and variable references used in it still point to the signals and variables defined in the Esterel program. This problem is solved by using the `ActionLabelProcessorWrapper` created for SyncCharts, which is basically a parser and serializer for the triggers and effects of a transition. It is applied in the scope of one single SyncCharts diagram.

6.2 Implementation of the SyncCharts Optimization

This section is split into two parts. First, the Xtend implementations are presented. Second, the controlling Java classes are discussed.

6.2.1 Xtend Implementation

For each optimization rule two Xtend methods are implemented. First, a predicate is created determining whether all conditions for that specific rule are met. Second, the method applying the actual transformation is provided. Listing 6.7 shows an example of such a predicate for `rule1`, which is presented in Section 5.3.2. In line 3 and 4 it is checked if `s` is a conditional state if it has one single outgoing transition, and whether the outgoing transition has neither a trigger nor an effect.

In Listing 6.8 the actual appliance of the optimization is listed. In line 5 the target state of the incoming transition is replaced. The superfluous transition is removed in line 6. The needless conditional state is removed too, as seen in line 8. A recursive

```
1 Boolean rule1applies(State s):
2    switch {
3       case isConditional(s) && hasNumberOfOutgoingTrans(s, 1) :
4          (isTransitionWithoutTaE(s.outgoingTransitions.get(0)) ? true : false)
5       default : false
6    }
7 ;
```

Listing 6.7: Predicate for `rule1`

```
1 Void rule1(State s):
2    let targetState = s.outgoingTransitions.get(0).targetState:
3    let incomingTrans = s.incomingTransitions.copyListTrans():
4    // bend transition to new target
5    incomingTrans.setTargetState(targetState) ->
6    s.outgoingTransitions.get(0).removeTransition() ->
7    // remove conditional state
8    s.removeStateFromRegion()
9 ;
```

Listing 6.8: Performing the optimization of `rule1`

rule is used in the same way as it is described in Section 6.1.

Furthermore, the optimization implementation allows the selection of a subset of rules, which is done by using global variables. In contrast to the Esterel to SyncCharts transformation the decision whether a state is optimizable or not cannot be made on the side of Java as the predicates are implemented in Xtend. Therefore, a list with all states of a SyncChart has to be passed to Xtend. Listing 6.9 shows the method `ruleAll`. It is the method that is called by the `XtendFacade` and tests for each existing optimization rule if this rule should be applied, depending on a global variable, and if all conditions are met for this explicit rule. This can be seen in line 7–22. In line 24 a processed state is removed from the `states` list. The `recursiveRule` is called in line 28, and in line 29 the size of the `states` list is returned. Hence, if the method returns `0`, all optimization rules have already been processed for all states.

```
1  Integer ruleAll(List[State] states):
2  
3     if states.size > 0 then
4     (
5        let s = states.first():
6        switch{
7           case ((boolean) GLOBALVAR rule1) && rule1applies(s) :
8              rule1(s)
9           case ((boolean) GLOBALVAR rule2) && s.isSimpleState() && rule2applies(s)
                :
10             rule2(s)
11          case ((boolean) GLOBALVAR rule3) && s.isSimpleState() && rule3applies(s)
                :
12             rule3(s)
13          case ((boolean) GLOBALVAR rule4) && rule4applies(s):
14             rule4(s)
15          case ((boolean) GLOBALVAR rule5) && rule5applies(s) :
16             rule5(s)
17          case ((boolean) GLOBALVAR rule6) && rule6applies(s) :
18             rule6(s, states)
19          case s.parentRegion != null && ((boolean) GLOBALVAR rule7) &&
                rule7applies(s) :
20             rule7(s)
21          case s.parentRegion != null && ((boolean) GLOBALVAR rule8) &&
                rule8applies(s) :
22             rule8(s)
23          default :
24          // this state is finished remove state
25          (states.remove(states.first()) -> ruleAll(states))
26       }
27    ) ->
28 
29    recursiveRule(states)
30    -> states.size
31 ;
```

Listing 6.9: Root rule for the SyncCharts optimization

6.2.2 Java Implementation

In the following, the implementation of the `SyncChartsOptimizationDataComponent` seen in Figure 6.3 is described. The demanded methods of the `AbstractTransformation-DataComponent` are discussed in further detail.

Global Variables

The global variable `recursive` is used the same way as mentioned in Section 6.1.3. Furthermore, a global variable is introduced for each optimization rule to specify whether the specific rule should be applied or not. Currently eight optimization rules are implemented. Hence, the global variables `rule1` to `rule8` are supplied.

Base Packages

- `KExpressionsPackage`: The `KExpressions` metamodel as introduced in Section 4.1.1.
- `SyncChartsPackage`: The SyncCharts metamodel seen in Figure 3.7.
- `EcorePackage`: The base metamodel for all of the other metamodels.

Retrieval of the Next Transformation

As mentioned earlier it is necessary to pass all available states to Xtend as the conditions for an optimization are tested by a predicate implemented in Xtend. Again, the context of the current optimization can be adapted by passing the currently selected SyncCharts state.

Listing 6.10 shows the `collectHierarchically` method in pseudo code. In line 3 the `parent` state is added to the specified `level` of an internally held data structure. In line 5 and 6 all child states are collected in the same way. However, it is more efficient to start the optimization on the lowermost hierarchical level as the result of one optimization rule can yield new optimization potential. Therefore, after the collection of the hierarchy it has to be flattened to a list and passed inversely to the `ruleAll` method.

```
1 collectHierarchically(State parent, Integer level)
2
3   addToHierarchy(parent, level)
4
5   foreach child in parent.states
6     collectHierarchically(child, level + 1);
```

Listing 6.10: Pseudo code collecting all states hierarchically ordered

Post Transformation

No post processing is required after a SyncCharts optimization.

6.3 Implementation of the Controlling `Combination`

The `E2STransformationCombination` seen in Figure 6.3 is a KiVi `Combination`. It contributes the buttons introduced in Section 5.1.1 to Eclipse's user interface and contains the logic connecting user inputs with internal actions.

The `E2STransformation` listens to three different events. In terms of KiVi such events are passed as *state*, e. g., `ButtonState`.

ActiveEditorState: The transformation can be performed in the context of two Eclipse editors. First, an open Esterel editor allows the initial transformation of an Esterel file to a SyncChart. Second, a SyncCharts editor allows either execution of the transformation of Esterel elements or the optimization of the currently opened SyncChart.

ButtonState: The `Combination` listens to clicks of the contributed buttons.

EffectTriggerState: Scheduled `TransformationEffects` are executed concurrently. Therefore, post transformation actions, e. g., automatic layout, have to be performed after the effect has been executed. An `EffectTriggerState` contains the information about an executed effect.

In Listing 6.11 pseudo code for the `execute` method is listed. In line 2, 6 and 14 the current event is determined. In case it is an `activeEditorState` the currently active editor is remembered as seen in line 3.

If a button is clicked while the XtextEditor is opened, an initial transformation will be processed. Otherwise, the pressed button is retrieved and the `process` method is called to execute the correct transformation.

An `effectState` either yields the execution of a further transformation or the appliance of automatic layout, see line 16–18. A further transformation will be needed if the user presses the button triggering the complete transformation and optimization of the currently opened SyncChart. In this case, all non-transformed Esterel elements have to be handled by a `TransformationEffect` first. The effect has to be executed, afterwards the optimization can be processed.

The `process` method is described in Listing 6.12. In line it 2 can be seen that a `DataComponent` is created in `kiviMode` and respective to the passed `button`. The `DataComponent` is initialized and one single step is performed to set up a `TransformationContext`. This context is retrieved in line 5, and a new `TransformationContext` is scheduled in line 7.

```
execute(buttonState, activeEditorState, effectState)
   if activeEditorState
      this.currentlyActiveEditor = activeEditorState.getEditor()
      return

   if buttonState
      if currentlyActiveEditor == XtextEditor
         initializeTransformation()
      else
         b = buttonState.getButton()
         process(b)
      return

   if effectState
      if furtherTransformationNecessary
         process(furtherTrans)
      else
         applyLayout()
```

Listing 6.11: Pseudo code of the `execute` method

```
process(button)
   dataComponent = createRespectiveDataComponent(button, kiviMode)
   dataComponent.initialize()
   dataComponent.step()
   context = dataComponent.getContext()

   schedule(new TransformationEffect(context))
```

Listing 6.12: Pseudo code of the `process` method

7 Validation and Experimental Results

This chapter presents the used testing approaches to validate the created Esterel grammar, which was presented in Chapter 4 and the transformation implementation presented in Chapter 6. Also, some experimental results are shown and several measurements of execution times are discussed.

7.1 Testing the Esterel Grammar

The created Xtext grammar has to be tested with respect to its correctness. However, a complete formal analysis would be beyond the scope of this thesis. Hence, a proper testing strategy covering the overall expressiveness of Esterel has to be established.

A set of Esterel programs provided by the CEC[1] serves as a test case. It includes authentic programs and programs that cover various of possible expressions to check the whole range of possible inputs.

Testing is done by iterating over all test files, parsing, and serializing them. In case a program is not recognized correctly Xtext's generated parser and serializer, respectively, present readable error messages.

7.2 Testing the Transformation Implementation

In the following, an approach to test the implementations presented in Section 6.1 and Section 6.2 is discussed. Both are tested in the same way despite of the fact that different input modules are used. For each Esterel to SyncCharts transformation rule a representative Esterel module is constructed. The same is done for each optimization rule by using a suitable diagram. They are supposed to cover as much expressiveness as possible for the specific rule. In combination with this another diagram is created by hand, which equals the correct result of the transformation rule. Figure 7.1 illustrates this testing procedure.

For each transformation rule a JUnit test is written. The test transforms the input automatically and compares the result to the expected diagram. The comparison is done by using *EMF Compare*[2]. EMF Compare is able to compare two arbitrary models, which have to base on an EMF metamodel, and to report differences. If there are no differences, the transformation yields the expected result and is considered correct.

[1] http://www.cs.columbia.edu/~sedwards/cec/
[2] http://wiki.eclipse.org/EMF_Compare

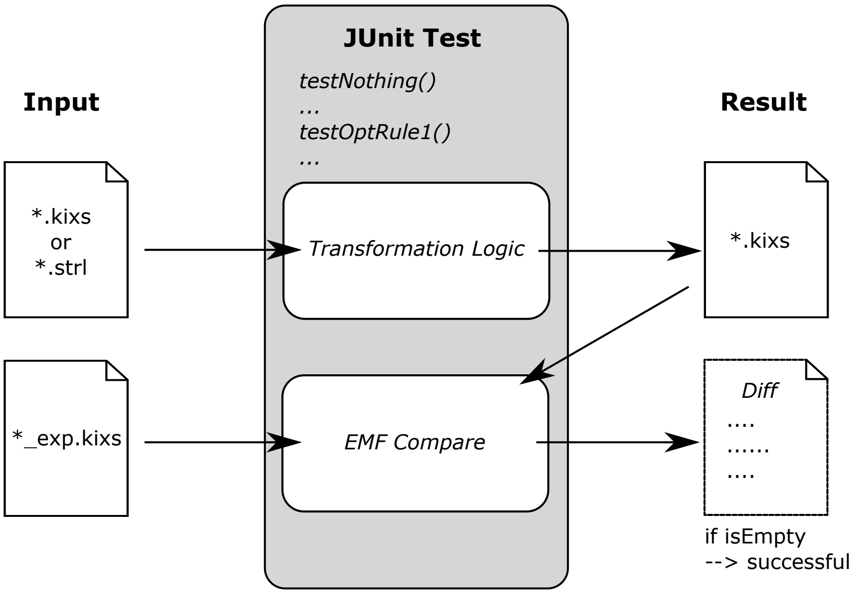


Figure 7.1: Testing of the transformation rules

```
1 performTest(inputFile, expectedFile) {
2   input = loadInput(inputFile);
3   expected = loadExpectedResult(expectedFile);
4 
5   result = transform(input);
6 
7   compare(result, expected);
8 }
```

Listing 7.1: Testing a transformation rule

Listing 7.1 shows the testing procedure in pseudo code which corresponds to the gray box in Figure 7.1. In line 2 and 3 the input model and the expected diagram are loaded. The input is either transformed to a SyncChart or optimized depending on the input in line 5. In line 7 the result of the transformation is compared to the expected result.

Listing 7.2 presents the JUnit test implementation for the `nothing` statement. The first parameter of the `performTest` method specifies the input file, in this case an Esterel file. The second parameter specifies the name extension of the diagram with the expected result. In this case it would be named `02-nothing_exp`.

This testing method has several weaknesses. For instance, EMF Compare fails if the order of signals is changed. Not the whole expressiveness of a statement is tested and the expected diagrams are created by humans. Hence, these diagrams can contain errors as well.

A better way to test the correctness would be the simulation of an Esterel program prior to the transformation and the simulation of the transformed SyncChart. Both simulation results could be compared to each other afterwards.

```
@Test
public void testNothing() throws Exception {
    performTest("02-nothing.strl", "_exp");
}
```

Listing 7.2: Pseudo code for testing a transformation rule

7.3 Experimental Results

After the implementation details have been given this section presents an exemplary transformation and studies execution times of the transformation as well as the quality of the presented optimizations.

Figure 7.2 shows the `ABRO` program. In the leftmost diagram the initially created SyncChart containing the Esterel module as the body text can be seen. The middle diagram represents the fully transformed SyncChart. As one can see a lot of new and unnecessary hierarchy levels have been introduced. These were removed in the optimized SyncChart, which can be seen on the right hand side of the figure.

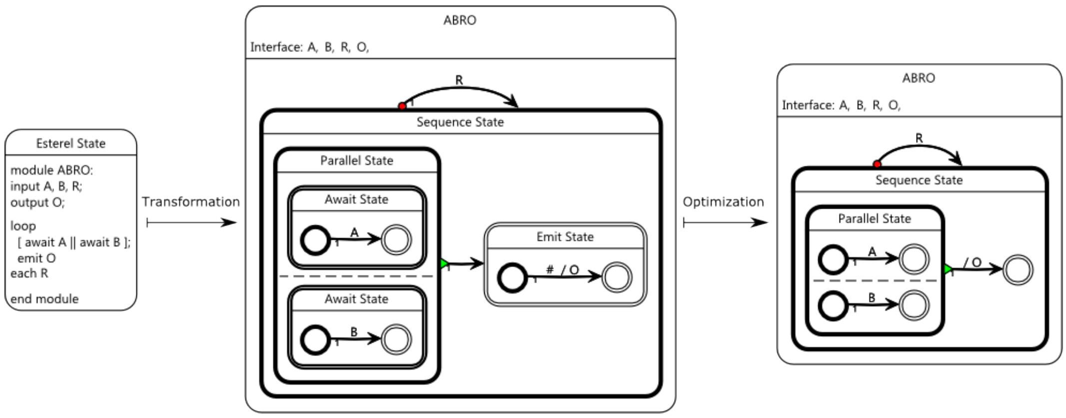


Figure 7.2: Transformation and optimization of `ABRO`

7.3.1 Transformation Durations

The Esterel programs provided by the CEC, mentioned in Section 7.1, were transformed by using different measuring setups. This aims at getting an overview of the implementation's time consumption. The different measuring setups are introduced in the following.

1. `Headless`: The duration of a complete headless transformation is measured as it is necessary to transform an Esterel file to SyncChart prior to opening the SyncChart in the editor. This includes loading the resource and initializing all needed classes, e. g., the `EsterelToSyncChartsDataComponent`.

2. `Recursive`: The duration of the recursive transformation itself is measured. This is only the time the call of the `XtendFacade` and the Xtend execution need.

3. `Recursive+Setup`: It is measured how long the creation of a new `EsterelToSyncChartsDataComponent` and the recursive transformation takes.

4. `Stepwise`: The time which is needed to transform an Esterel program completely by using the *step* functionality programmatically is measured. This means that the `XtendFacade` is called with a new `TransformationDescriptor` and a new `TransformationContext` as long as any further state can be transformed.

5. `Stepwise+Setup`: The duration of the latter including the initializing of a new data component per step is measured.

The results can be seen in Figure 7.4 and Figure 7.5. For both figures the x-axis represents different diagrams and is ordered by the recursive execution's duration. No correlation to the diagram size can be made. Furthermore, in Figure 7.3 the calculated differences of several average values are presented.

This figure shows that the differences between points 2–4 are marginal. Sometimes, the `Recursive+Setup` execution took less time than the sole `Recursive` execution. This can be explained with the usual fluctuation in thread activity and processor time. Therefore, the overhead of the creation of an `XtendFacade` and calling it for every transformation rule, instead of using recursive execution, is negligible.

The `Headless` execution takes constantly $\sim$ 250ms longer than the `Recursive` one. The additional time is needed to fetch the resource and set up all necessary classes, which in a non-headless mode would be provided by the Eclipse editor before the time measurement is started. Considering that this execution mode is triggered only once for an Esterel program to transform it completely prior to opening it in the editor $\sim$ 250ms is an acceptable execution time.

A critical increase in execution time shows the `Stepwise+Setup` execution, especially by using larger diagrams. For small diagrams the difference between the latter and the `Stepwise` execution is very small but increases constantly for diagrams with more hierarchy levels. The two measured values only differ in the creation of the data component. Therefore, the increase in duration can be explained by the adding up of the additional time needed for the instantiation of a new data component and the creation of a new `XtendFacade`.

Further measurements showed that the average execution time of a single step is lower than 10ms and therefore negligible compared to the time a user needs to press a button.

`Headless — Recursive`	$\sim$ 250ms
`Recursive — Stepwise`	$\sim$ 10ms
`Recursive+Setup — Recursive`	$\sim$ 10ms
`Stepwise — Stepwise+Setup`	$\sim$ 1000ms

Figure 7.3: Differences of the measured average values

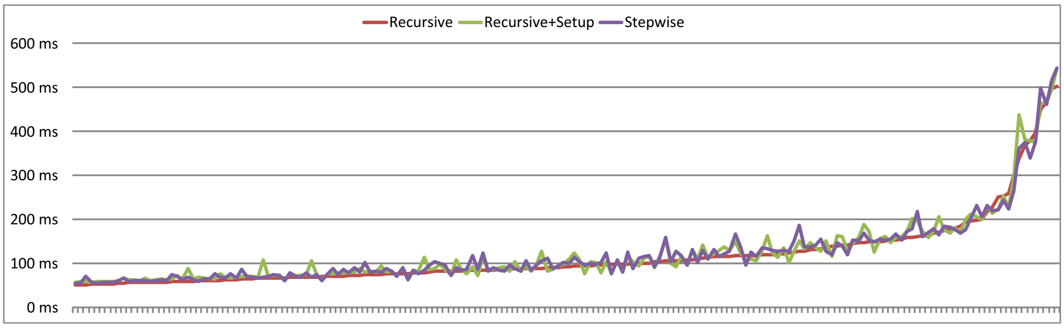


Figure 7.4: Measured times for `Recursive`, `Recursive+Setup`, and `Stepwise`

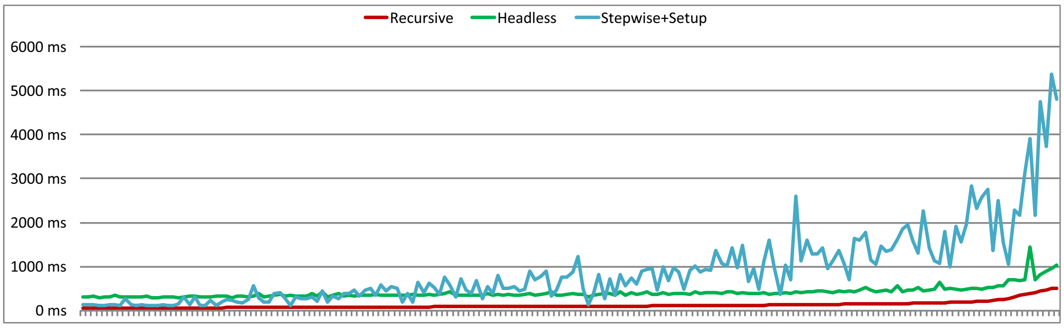


Figure 7.5: Measured times for `Headless`, `Recursive`, and `Stepwise+Setup`

7.3.2 Optimization Quality

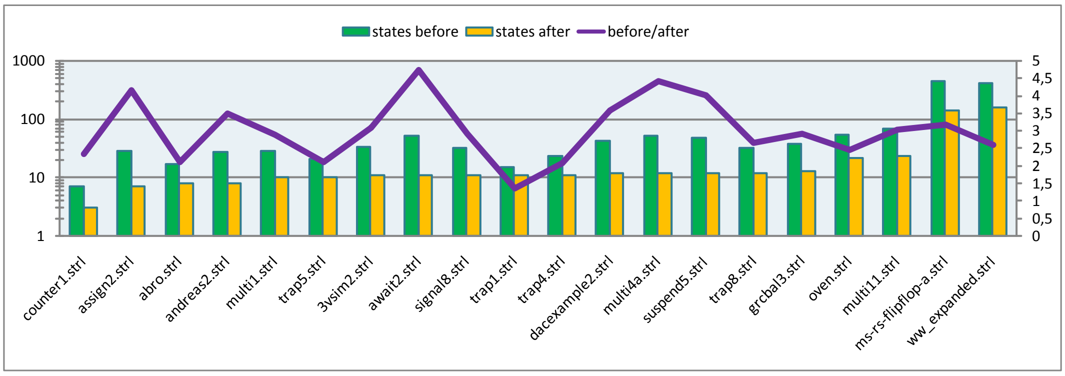


Figure 7.6: Decrease of the number of states due to optimization

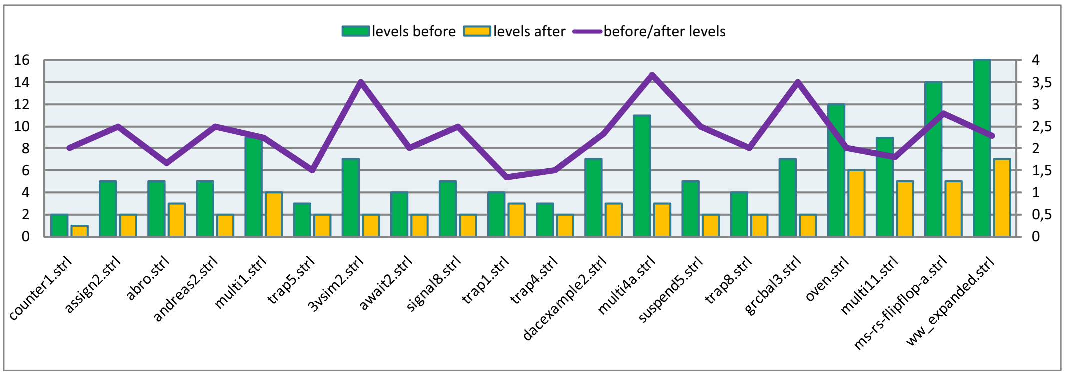


Figure 7.7: Decrease of the number of hierarchy levels due to optimization

To evaluate the quality and importance of the SyncCharts optimization some measurements concerning the number of states were done, which can be seen in Figure 7.6 and Figure 7.7.

Again, the CEC Esterel programs were used. They were transformed and optimized and the number of states and hierarchy levels were measured prior to and after the optimization. In the two figures just an extract of the results is presented for reasons of clarity and comprehensibility.

The left axis refers to the number of states and hierarchy levels, respectively. Green bars indicate the values prior to optimization, the orange ones the values after optimization. The purple line is the ratio of the two mentioned values and is quantified by the right axis.

Considering all results it shows that the number of states decreases on average by a factor of 3, the number of hierarchy levels by a factor of 2.5.

8 Concluding Results

In this chapter the work of this thesis is summarized. Some conclusions are made and future work is discussed.

8.1 Summary

In Chapter 1 some motivation for the step-wise execution and visualization of transformations was given. Also, the languages Esterel and SyncCharts were introduced as a transformation from Esterel to SyncCharts served as the primary example.

Two issues were addressed in particular. First, tools to work with Esterel, such as the CEC or Esterel studio, were introduced. Second, the topic of model transformations was regarded. Transformation languages were discussed, other KIELER M2M transformations were presented, and different approaches that do not use EMF were depicted, such as TGGs.

In the following, the tasks of this thesis as stated in Section 1.2 are reconsidered.

> *Provide facilities to handle Esterel code in the context of KIELER.*

An Esterel editor was created and embedded into KIELER. It bases on an Xtext grammar and provides sophisticated tooling such as *code completion*, *syntax highlighting*, and *code formatting*. The adaption of an existing Esterel grammar was presented in Chapter 4 aiming at meeting Xtext's requirements.

> *Point out an approach to handle visual transformations.*

In Section 5.1 criteria for the description of general transformations were specified. Based on this a generic implementation was developed. In combination with KiVi this implementation served as a framework to perform arbitrary transformations as a view management effect.

> *Implement both, the SyncCharts to Esterel transformation and the SyncCharts optimization.*

Section 5.2 and Section 5.3 reviewed the theoretical basis of the Esterel to SyncCharts transformation and the SyncCharts optimization as they were presented by Kuehl [Küh06]. They are intended to serve as a reference for these specific transformation rules. For this reason the actual implementation in the Xtend language was listed as well.

A graphical user interface was developed. It provides different execution modes. The user is able to perform a headless transformation and an optimization at once. He can carry out the process in certain steps to understand the transformation's functionality.

The actual implementation of the two transformations was presented in Chapter 6.

Define proper testing criteria and provide rudimentary tests.

In Section 7.1 and Section 7.2 testing possibilities were depicted. Esterel files and SyncChart diagrams covering as much expressiveness as possible were created. JUnit served as an automatic testing facility and validated the provided test diagrams by using EMF Compare.

8.2 Conclusions

The approaches presented in this thesis would not be practical without KIELER as their basis. A step-wise execution with visualization of intermediate steps is made possible by using *automatic layout*. Without automatic layout it would be very hard to present intermediate steps in a structured and understandable form. Also, KIEM and KiVi are sophisticated means to allow continuous user interaction.

The adaption of the Esterel grammar turned out to be more difficult than expected. The problem of Xtext's expressiveness having several limits had to be solved, and some weaknesses concerning the performance of the serialization showed up for deeply nested constructs.

The choice to base the Esterel grammar on the `KExpressions` grammar was useful. Particularly, the transformation of Esterel files to SyncCharts is simplified. This is because the majority of signals, variables, and expressions of an Esterel file can be transferred to a SyncChart without the need of further adaptations.

The atomicity of both, the Esterel to SyncCharts transformation rules and the SyncCharts optimization rules, makes them easy to use. Short and precise rules are well comprehensible and straightforward. Hence, the implementation in Xtend was simple and errors could be located quickly.

Furthermore, the presented optimization rules turned out to be essential to obtain readable SyncCharts. As mentioned in Chapter 7 first results show a reduction of states and hierarchy levels by the factor of 3 and 2.5, respectively.

The execution of an arbitrary transformation is implemented by using a KiVi `Effect`. This allows a seamless integration into the view management, which simplifies the adding of additional visualizations, e. g., focus and context as discussed in the next section.

To provide an easy user interface a controlling KIEM `DataComponent` is provides as well as a KiVi `Combination`. However, the `DataComponent` results from first implementations. Because KIEM and KiVi base on different ideas the maintenance of both controlling logics yields additional efforts to be performed.

To sum up, it can be stated that an easy and pleasant way to execute transformations was presented while preserving expandability with relation to visualization and user interaction.

8.3 Future Work

During this work several ideas to enhance the Esterel integration into KIELER (Chapter 4), to extend the theoretical background of the Esterel to SyncCharts transformation (Chapter 5), and to improve the transformation execution itself emerged. Most of the ideas are not an essential part of the presented implementation but their realization would improve the usability of the tooling. Therefore, these ideas are described in the following.

Esterel Code Formatting

Xtend comes with the possibility to define formatting information for its grammar elements. This information can be applied to a source file via the generated tooling, which then formats the source file automatically according to the defined rules.

Proper code formatting eases the fast understanding of pieces of code especially for those people who did not write the code themselves. Furthermore, formatting produces structured and well readable code. Automatic tooling support saves time and generates a consistent formatting amongst all developers and source files. Therefore, it is particularly important while writing source code in a collaborative environment.

Integrating Esterel v7

There are two facets of this point. First, it would seem natural to extend the v5 implementation and integrate v7 into the tooling. This includes extending the existing grammar (Chapter 4) to meet the requirements of Esterel v7, for instance, v7 allows *arrays*, which are not considered in the current implementation.

Second, a transformation of Esterel v7 source code to SyncCharts can be attempted, which is a complex problem. Initially, it needs to be evaluated whether such a transformation is possible. It is necessary to verify that the expressiveness of SyncCharts is powerful enough to meet all of the new constructs. Furthermore, transformation rules have to be declared and proven.

For both points it has to be validated that all other technologies used meet all requirements.

Correctness of Optimization Rules

As mentioned in Section 5.3 the SyncCharts optimization rules that were presented in this thesis have not been proven to maintain semantics. This should be done in the future, or proper testing facilities should be provided. Such testing facilities could simulate the behavior of a SyncChart. KIELER comes with tools to simulate a

SyncChart and to retrieve information about the results, e. g., by using SC or KlePto. Both, the non-optimized and the optimized SyncChart could be simulated with all possible inputs. These might not be possible strategies to use for larger diagrams, so in case proper inputs have to be developed, e. g., by selecting equivalence classes, see *black-box testing*. Afterwards, the results of both simulations would be compared.

Further Optimization Rules

The optimization rules presented in Section 5.3 were developed in the context of the Esterel to SyncCharts transformation. They aim at reducing the number of hierarchy levels and produce a more compact SyncChart. Hence, it is just a set of intuitive rules that does not cover the whole optimization potential. An additional optimization rule might be one that removes unreachable transitions. Consider two signals `A` and `B` and a transition with the trigger `A or B` and the highest priority. This transition leaves a state with another transition, which has a lower priority and the trigger `B`. The second transition will never be taken and can therefore be removed.

A further rule might be one eliminating unnecessary signals, for instance, signals that are declared or emitted but never tested and not specified as output signal.

Selection of Optimization Rules

Sometimes, the user wants to apply only a subset of the possible optimization rules. This becomes especially useful during the process of testing. The current implementation already offers such a selection, as mentioned in Section 6.2.2. The user interface could be adapted to provide functionality to activate or to deactivate certain rules.

Handling of Multiple Modules

Currently, the implemented Esterel to SyncCharts transformation is restricted to one module within the Esterel file.

The CEC provides functionality to expand an Esterel program to reduce the number of modules to one. This could be implemented as an automated process that is performed prior to the actual transformation.

Also, different strategies to handle multiple modules might be interesting. Each module could be transformed into a separate SyncChafARRRRRf, or all modules could be placed as an individual macro state within one SyncChart.

An approach to use SyncCharts with *reference macro states* to associate a model with a macro state was presented by Bleidiessel [Ble10]. A SyncChart containing such reference states can be expanded by replacing each reference macro state with the actual model.

Focus and Context during Transformations

A pragmatic aspect presented in [FvH10b] is *Focus and Context*. Focus and Context proposes to apply automatic layout depending on the current focus and context. The focus refers to elements of temporal interest, for instance, the currently transformed elements during an overall transformation. The context includes, for instance, elements on the same hierarchy level.

Such a functionality would yield better clarity and comprehensibility, especially concerning the transformation of large diagrams.

Side by Side Esterel and SyncCharts

In Eclipse it is possible to open and to show two editors simultaneously. Hence, an Esterel file, which is being transformed, could be presented alongside of the currently transformed SyncChart. To improve the comprehensibility it might be desirable to highlight the currently handled Esterel statement and the corresponding SyncChart element. Similar work is presented by [Sch11].

Xtend2

Currently, the implementation presented in Chapter 6 bases on the Xtend technology introduced in Section 3.1.4. Xtend will be succeeded by *Xtend2*[1], which integrates seamlessly into the Java code, bases on the *Xbase*[2] language, and provides further features that will not be described in further detail here.

As Xtend code is not fully compatible with Xtend2 some refactoring might be necessary to use the implementations presented in this thesis with Xtend2. The benefits and efforts of such an adaption should be examined as soon as Xtend2 is released in a final version.

[1] http://blog.efftinge.de/2010/12/xtend-2-successor-to-xpand.html
[2] http://blog.efftinge.de/2010/09/xbase-new-programming-language.html

A Sources

Listing A.1: The Esterel grammar

```
grammar de.cau.cs.kieler.kies.Esterel with de.cau.cs.kieler.core.kexpressions.
    KExpressions

generate esterel "http://www.cau.de/cs/kieler/kies/Esterel"
import "platform:/resource/de.cau.cs.kieler.core.kexpressions/model/kexpressions.
    ecore" as kexpressions
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

// root rule. an esterel file can contain multiple modules
Program hidden(Esterel_SL_Comment, Esterel_ML_Comment, WS):
   (modules+=Module)*;

   // a module consists of an interface and a body
Module:
   "module" name=ID ":" (interface=ModuleInterface)? body=ModuleBody end=EndModule
       ;

EndModule:
   "end" "module"
   | "."; //deprecated
ModuleBody:
   statements+=Statement;

//            Interface Declarations
// ---------------------------------------------
ModuleInterface:
   (intSignalDecls+=InterfaceSignalDecl
   | intTypeDecls+=TypeDecl
   | intSensorDecls+=SensorDecl
   | intConstantDecls+=ConstantDecls
   | intRelationDecls+=RelationDecl
   | intTaskDecls+=TaskDecl
   | intFunctionDecls+=FunctionDecl
   | intProcedureDecls+=ProcedureDecl)+;

   // overwrite to add the EsterelTypeIdentifier
ChannelDescription:
   (":" type=EsterelTypeIdentifier)
   | ("(" type=EsterelTypeIdentifier ")")
   | (":=" expression=Expression ":" type=EsterelTypeIdentifier);
```

```
   // overwrite to allow function references for signal declarations
EsterelTypeIdentifier returns kexpressions::TypeIdentifier:
   type=ValueType
   | typeID=ID
   | {EsterelTypeIdentifier} ("combine" (type=ValueType | typeID=ID) "with" (func
       =[Function|ID] |
   operator=CombineOperator));

   // overwrite to allow type definitions in a specific module
TypeIdentifier:
   type=ValueType
   | typeID=ID
   | ("combine" (type=ValueType | typeID=ID) "with" operator=CombineOperator)
   | {EsterelType} estType=[Type|ID];

   // ==> Local Signal Declaration
LocalSignalDecl:
   "signal" signalList=LocalSignalList "in" statement=Statement "end" (optEnd="
       signal")?;

LocalSignalList:
   {LocalSignal} signal+=ISignal
   ("," signal+=ISignal)*;

// ==> Sensor
// -------------------------------------
SensorDecl:
   "sensor" sensors+=SensorWithType ("," sensors+=SensorWithType)* ";";

SensorWithType:
   (sensor=Sensor (":" type=TypeIdentifier)) | (sensor=Sensor "(" type=
       TypeIdentifier ")");

Sensor returns kexpressions::ISignal:
   name=ID;

   // ==> Relations
// -------------------------------------
RelationDecl:
   {Relation} "relation" relations+=RelationType ("," relations+=RelationType)* ";
       ";

RelationType:
   RelationImplication | RelationIncompatibility;

RelationImplication:
   first=[kexpressions::ISignal|ID] type="=>" second=[kexpressions::ISignal|ID];

```

```
RelationIncompatibility:
   incomp+=[kexpressions::ISignal|ID] type="#" incomp+=[kexpressions::ISignal|ID]
       ("#"
   incomp+=[kexpressions::ISignal|ID])*;

   // ==> Types
// ------------------------------------
TypeDecl:
   "type" types+=Type ("," types+=Type)* ";";

Type:
   name=ID;

   // ==> Constants
// ------------------------------------
ConstantDecls:
   "constant" constants+=OneTypeConstantDecls ("," constants+=OneTypeConstantDecls
       )* ";";

OneTypeConstantDecls:
   constants+=ConstantWithValue ("," constants+=ConstantWithValue)* ":" type=
       TypeIdentifier;

ConstantWithValue:
   constant=Constant ("=" value=ConstantAtom)?;

Constant returns kexpressions::ValuedObject:
   {Constant} name=ID;

ConstantAtom:
   INT | ConstantLiteral;

ConstantLiteral:
   Float | Boolean | ID | STRING;

   // ==> Functions
// ------------------------------------
FunctionDecl:
   "function" functions+=Function ("," functions+=Function)* ";";

Function:
   name=ID "(" (idList+=TypeIdentifier ("," idList+=TypeIdentifier)*)? ")" ":"
       type=TypeIdentifier;

   // ==> Procedures
ProcedureDecl:
   "procedure" procedures+=Procedure ("," procedures+=Procedure)* ";";

Procedure:
```

```
    name=ID "(" (idList1+=TypeIdentifier ("," idList1+=TypeIdentifier)*)? ")" "(" (
        idList2+=TypeIdentifier (","
    idList2+=TypeIdentifier)*)? ")";

    // ==> Tasks
TaskDecl:
    "task" tasks+=Task ("," tasks+=Task)* ";";

Task:
    name=ID "(" (idList1+=TypeIdentifier ("," idList1+=TypeIdentifier)*)? ")" "(" (
        idList2+=TypeIdentifier (","
    idList2+=TypeIdentifier)*)? ")";

    // ==============================================
// ===            B.4 Statements              ===
// ==============================================
Statement:
    Sequence ({Parallel.list+=current} "||" list+=Sequence)*;

AtomicStatement returns Statement:
    Abort | Assignment | Await | Block | ProcCall | Do | Emit | EveryDo | Exit |
        Exec | Halt | IfTest | LocalSignalDecl |
    Loop | Nothing | Pause | Present | Repeat | Run | Suspend | Sustain | Trap |
        LocalVariable | VarStatement | WeakAbort;

    // --> B.4.1 Control Flow Operators <--
Sequence returns Statement:
    AtomicStatement ({Sequence.list+=current} ";" list+=AtomicStatement)* ";"?;

Block:
    "[" statement=Statement "]";

VarStatement returns Statement:
    vardecl=IVariable;

    // Assignment
// -------------------------------------
Assignment:
    var=[kexpressions::IVariable|ID] ":=" expr=Expression;

// --> B.4.2 abort: Strong Preemption
// -------------------------------------
Abort:
    "abort" statement=Statement "when" body=AbortBody;

AbortBody:
    AbortInstance | AbortCase;

AbortInstance:
```

```
   delay=DelayExpr ("do" statement=Statement "end" (optEnd="abort")?)?;

AbortCase:
   cases+=AbortCaseSingle (cases+=AbortCaseSingle)* "end" (optEnd="abort")?;

AbortCaseSingle:
   "case" delay=DelayExpr ("do" statement=Statement)?;


// --> B.4.25 weak abort: Weak Preemption
// -------------------------------------
WeakAbort returns Abort:
   {WeakAbort} "weak" "abort" statement=Statement "when" body=WeakAbortBody;

WeakAbortBody:
   WeakAbortInstance | WeakAbortCase;

WeakAbortEnd:
   {WeakAbortEnd} "end" (optEnd=WeakAbortEndAlt)?;

WeakAbortEndAlt:
   (end="weak")? endA="abort";

WeakAbortInstance returns AbortInstance:
   {WeakAbortInstance} delay=DelayExpr ("do" statement=Statement end=WeakAbortEnd)?;

WeakAbortCase returns AbortCase:
   {WeakAbortCase} cases+=AbortCaseSingle (cases+=AbortCaseSingle)* end=WeakAbortEnd;

   // --> B.4.3 await: Strong Preemption
// -------------------------------------
Await:
   "await" body=AwaitBody;

AwaitBody:
   AwaitInstance | AwaitCase;

AwaitInstance:
   delay=DelayExpr ("do" statement=Statement end=AwaitEnd)?;

AwaitCase:
   cases+=AbortCaseSingle (cases+=AbortCaseSingle)* end=AwaitEnd;

AwaitEnd:
   "end" "await"?;

// --> B.4.4 call: Procedure Call
```

```
// --------------------------------------
ProcCall:
   "call" proc=[Procedure|ID] "(" (varList+=[kexpressions::IVariable|ID] ("," varList+=[kexpressions::IVariable|ID])∗)?
   ")"
   "(" (kexpressions+=Expression ("," kexpressions+=Expression)∗)? ")";

   // --> B.4.5 do-upto: Conditional Iteration (deprecated)
// --> B.4.6 do-watching: Strong Preemption (deprecated)
// --------------------------------------
Do:
   "do" statement=Statement (end=DoUpto | end=DoWatching);

DoUpto:
   "upto" expr=DelayExpr;

DoWatching:
   "watching" delay=DelayExpr (end=DoWatchingEnd)?;

DoWatchingEnd:
   "timeout" statement=Statement "end" (optEnd="timeout")?;

// --> B.4.7 emit: Signal Emission <--
// --------------------------------------
Emit:
   "emit" ((signal=[kexpressions::ISignal|ID]) | tick=Tick) ("(" expr=Expression ")")?;

// --> B.4.8 every-do: Conditional Iteration
// --------------------------------------
EveryDo:
   "every" delay=DelayExpr "do" statement=Statement "end" (optEnd="every")?;

// --> B.4.10 exit: Trap Exit
// --------------------------------------
Exit:
   "exit" trap=[TrapDecl|ID] ("(" expression=Expression ")")?;

// --> B.4.11 halt: Wait Forever
// --------------------------------------
Halt:
   {Halt} "halt";

// --> B.4.12: if: Conditional for Data
// --------------------------------------
IfTest:
   "if" expr=Expression (thenPart=ThenPart)? (elsif+=ElsIf)∗ (elsePart=ElsePart)? "end" (optEnd="if")?;

```

```
ElsIf:
   "elsif" expr=Expression (thenPart=ThenPart)?;

ThenPart:
   "then" statement=Statement;

ElsePart:
   "else" statement=Statement;

// --> B.4.13 loop: Infinite Loop
// --> B.4.14 loop-each: Condition Iteration
// ------------------------------------
Loop:
   "loop" body=LoopBody (end1=EndLoop | end=LoopEach);

EndLoop:
   "end" "loop"?;

LoopEach:
   "each" LoopDelay;

LoopDelay:
   delay=DelayExpr;

LoopBody:
   statement=Statement;

// --> B.4.15 nothing: No Operation
// ------------------------------------
Nothing:
   "nothing" {Nothing};

// --> B.4.16 pause: Unit Delay
// ------------------------------------
Pause:
   "pause" {Pause};

// --> B.4.17 present: Conditional for Signals
// ------------------------------------
Present:
   "present" body=PresentBody (elsePart=ElsePart)? "end" (optEnd="present")?;

PresentBody:
   PresentEventBody | PresentCaseList;

PresentEventBody:
   event=PresentEvent (thenPart=ThenPart)?;

PresentCaseList:
```

```
   cases+=PresentCase (cases+=PresentCase)*;

PresentCase:
   "case" event=PresentEvent ("do" statement=Statement)?;

PresentEvent:
   expression=SignalExpression | "[" expression=SignalExpression "]" | tick=Tick;


// --> B.4.18 repeat: Iterate a Fixed Number of Times
// -------------------------------------
Repeat:
   (positive?="positive")? "repeat" expression=Expression "times" statement=Statement "end" (optEnd="repeat")?;

   // --> B.4.19 run: Module Instantiation
// -------------------------------------
Run:
   "run" module=ModuleRenaming ("[" list=RenamingList "]")? | "copymodule" module=ModuleRenaming ("[" list=RenamingList
   "]")?; //deprecated


// Renamings
// -------------------------------------
ModuleRenaming:
   module=[Module|ID] | (newName=ID "/" module=[Module|ID]);

RenamingList:
   list+=Renaming (";" list+=Renaming)*;

Renaming:
   "type" renamings+=TypeRenaming ("," renamings+=TypeRenaming)*
   | "constant" renamings+=ConstantRenaming ("," renamings+=ConstantRenaming)*
   | "function" renamings+=FunctionRenaming ("," renamings+=FunctionRenaming)*
   | "procedure" renamings+=ProcedureRenaming ("," renamings+=ProcedureRenaming)*
   | "task" renamings+=TaskRenaming ("," renamings+=TaskRenaming)*
   | "signal" renamings+=SignalRenaming ("," renamings+=SignalRenaming)*;

TypeRenaming:
   (newName=[Type|ID] | newType=ValueType) "/" oldName=[Type|ID];

ConstantRenaming:
   (newName=[kexpressions::ValuedObject|ID] | newValue=ConstantAtom) "/" oldName=[kexpressions::ValuedObject|ID];

FunctionRenaming:
   (newName=[Function|ID] | newFunc=BuildInFunction) "/" oldName=[Function|ID];

```

```
ProcedureRenaming:
   newName=[Procedure|ID] "/" oldName=[Procedure|ID];

TaskRenaming:
   newName=[Task|ID] "/" oldName=[Task|ID];

SignalRenaming:
   (newName=[kexpressions::ISignal|ID] | "tick") "/" oldName=[kexpressions::
       ISignal|ID];

   // renamings can also rename build in types and functions
BuildInFunction:
   "*" | "/" | "+" | "-" | "mod" | "=" | "<>" | ">" | "<" | "<=" | ">=" | "not" |
       "and" | "or";

   // --> B.4.21 suspend: Preemption with State Freeze
// -------------------------------------
Suspend:
   "suspend" statement=Statement "when" delay=DelayExpr;

// --> B.4.22 sustain: Emit a Signal Indefinitely
// -------------------------------------
Sustain:
   "sustain" ((signal=[kexpressions::ISignal|ID]) | tick=Tick) ("(" expression=
       Expression ")")?;

// --> B.4.23 trap: TrapDeclaration and Handling
// -------------------------------------
Trap:
   "trap" trapDeclList=TrapDeclList "in" statement=Statement
   (trapHandler+=TrapHandler)* "end" (optEnd="trap")?;

TrapDeclList:
   trapDecls+=TrapDecl ("," trapDecls+=TrapDecl)*;

TrapDecl returns kexpressions::ISignal:
   {TrapDecl} name=ID channelDescr=(ChannelDescription)?;

TrapHandler:
   "handle" trapExpr=TrapExpr "do" statement=Statement;

// --> B.4.24 var: Local Variable Declaration
// -------------------------------------
LocalVariable:
   var=InterfaceVariableDecl "in" statement=Statement "end" (optEnd="var")?;

// ==============================================
// ===           B.3 Expressions             ===
// ==============================================
```

```

// esterel is a bit richer than what is provided by kexpressions. These rules are
    introduced here
// care about order of the rules!
AtomicExpression returns kexpressions::Expression:
   FunctionExpression
   | TrapExpression
   | BooleanValue
   | ValuedObjectTestExpression
   | TextExpression
   | '(' BooleanExpression ')'
   | ConstantExpression;

TrapExpression returns kexpressions::Expression:
   {TrapExpression} "??" trap=[kexpressions::ISignal|ID];

FunctionExpression returns kexpressions::Expression:
   {FunctionExpression} function=[Function|ID] "(" (kexpressions+=Expression (","
       kexpressions+=Expression)*)? ")";

ConstantExpression returns kexpressions::Expression:
   {ConstantExpression} (constant=[Constant|ID] | value=ConstantAtom);

   // --> B.3.5 Trap Expressions <--
// -------------------------------------
TrapExpr returns kexpressions::Expression:
   SignalExpression;

   // --> B.3.3 Signal Expressions <--
// -------------------------------------
SignalExpression returns kexpressions::Expression:
   SignalAndExpression ({kexpressions::OperatorExpression.subExpressions+=current}
        operator=OrOperator
   subExpressions+=SignalAndExpression)*;

SignalAndExpression returns kexpressions::Expression:
   SignalNotExpression ({kexpressions::OperatorExpression.subExpressions+=current}
        operator=AndOperator
   subExpressions+=SignalNotExpression)*;

SignalNotExpression returns kexpressions::Expression:
   {kexpressions::OperatorExpression} operator=NotOperator subExpressions+=(
       SignalNotExpression) |
   SignalAtomicExpression;

SignalAtomicExpression returns kexpressions::Expression:
   SignalReferenceExpr
   | "(" SignalExpression ")"
   | SignalPreExpr
```

```
    | TrapReferenceExpr // maybe place this somewhere else
;

SignalReferenceExpr returns kexpressions::ValuedObjectReference:
    valuedObject=[kexpressions::ISignal|ID];

SignalPreExpr returns kexpressions::Expression:
    {kexpressions::OperatorExpression} operator=PreOperator '(' subExpressions+=
        SignalReferenceExpr ')';

TrapReferenceExpr returns kexpressions::ValuedObjectReference:
    {TrapReferenceExpr} valuedObject=[TrapDecl|ID];

    // --> B.3.4 Delay Expressions <--
// -------------------------------------
DelayExpr:
    (expr=Expression event=DelayEvent) | event=DelayEvent | (isImmediate?="
        immediate" event=DelayEvent);

DelayEvent:
    tick=Tick | expr=SignalReferenceExpr | "[" expr=SignalExpression "]";

    // --> Exec
// -------------------------------------
Exec:
    ("exec" task=[Task|ID] body=ExecBody "return" retSignal=[kexpressions::ISignal]
         ("do" statement=Statement)? | "exec"
    execCaseList+=ExecCase (execCaseList+=ExecCase)*) "end" (optEnd="exec")?;

ExecBody:
    {ExecBody} "(" (vars+=[kexpressions::IVariable|ID] ("," vars+=[kexpressions::
        IVariable|ID])*)? ")" "("
    (kexpressions+=Expression ("," kexpressions+=Expression)*)? ")";

ExecCase:
    "case" task=[Task|ID] body=ExecBody "return" retSignal=[kexpressions::ISignal]
        ("do" statement=Statement)?;

    // =============================================
// === B.2 Namespaces and Predefined Objects  ===
// =============================================
Tick:
    "tick";

terminal Esterel_SL_Comment:
    '%' !('\n' |
    '\r')* ('\r'? '\n')?;

terminal Esterel_ML_Comment:
```

```
    ('%' '{')->('}' '%');

    // allow escaping by double quotes ( "this is a ""quote"", how nice." ) -
        esterelstyle
terminal STRING returns ecore::EString:
    '"' (!('"') | ('"' '"'))* '"';
```

Listing A.2: The KiesUtil.ext extension file

```
import ecore;
import annotations;
import kexpressions;
import synccharts;
import utilities;
import esterel;

extension feature;
extension org::eclipse::xtend::util::stdlib::cloning; // provides clone
    functionality

 /*
  * Convenient methods used by the Esterel to SyncCharts transformation.
  */

/**
 * extracts the expression and immediate information of a esterel::DelayExpr
 * and adds them to the synccharts::Transition
 */
Void addTriggerToTransition(synccharts::Transition t, DelayExpr delay):
   if delay.isImmediate then
      t.setIsImmediate(true) ->
   // clone the expression as in should stay in esterel model too
   t.setTrigger((Expression)clone(delay.event.expr)) ->
   if delay.expr != null then
      t -> // not yet supported, as synccharts do not offer a delay expression!
   t
;

/**
 * extracts all signals from the InterfaceSignalDecl and adds them to the state
 */
Void extractSignals(InterfaceSignalDecl decl, State s):
   let clonedDecl = (InterfaceSignalDecl) clone(decl):
   let copy = (List[Signal]) copyList(clonedDecl.signals):
   switch {
      // clone loses the information
      case Input.isInstance(decl) : (copy.setIsInput(true) ->  copy.setIsOutput(
          false))
      case Output.isInstance(decl) : (copy.setIsInput(false) ->  copy.setIsOutput(
          true))
      case InputOutput.isInstance(decl) : (copy.setIsInput(true) ->  copy.
          setIsOutput(true))
         default : decl //no information about input/output
   } ->
   if clonedDecl.signals.size > 0 then
      copy.addSignalToState(s)
;
```

```

Void addSignalToState(Signal sig, State st):
   st.signals.add(sig)
;

/**
 * extract all variables
 */
Void extractLocalVariables(VariableDecl decl, State s):
   let vars = (List[IVariable]) clone(decl.variables):
   vars.addVariableToState(s)
;

Void addVariableToState(kexpressions::Variable v, State s):
   s.variables.add(v)
;

/**
 * extracts all local signals
 */
Void extractLocalSignals(LocalSignalList lsl, State s):
   if LocalSignal.isInstance(lsl) then
      (let copy = (List[ISignal]) clone(((LocalSignal)lsl).signal):
       copy.addSignalToState(s)
      )
;

/**
 * connects a transition with the two passed states
 */
Void connectTransition(Transition t, State source, State target):
   t.setSourceState(source) ->
   t.setTargetState(target) ->
   source.outgoingTransitions.add(t)
;

/**
 * removes EVERYTHING !! body text, sets state type to NORMAL
 */
Void removeBodyText(State s):
   s.bodyText.removeAll(s.bodyText) ->
   s.setBodyReference(null) ->
   s.setType(StateType::NORMAL)
;

/**
 * clears bodycontents of ALL child states of State s.
 */
Void clearBodyReferences(State s):
```

```
   let states = (Set[State]) s.eAllContents.select(e| State.isInstance(e)
      \&\& ((State)e).bodyReference != null):
   states.add(s) ->
   states.setBodyReference(null)
;

/**
 * only sets the label if there's no previous label
 */
Void setLabelIfEmpty(State s, String label):
   (s.label == null || s.label.trim().length == 0) ?
      s.setLabel(label) : s
;

// collect all traps in an "or" expression
Void collectTraps(TrapDeclList traps, OperatorExpression expr):
   traps.trapDecls.addTrapToExpression(expr)
;
// as trap just extends ISignal .. TrapDecl is unknown here
Void addTrapToExpression(ISignal trap, OperatorExpression expr):
   let ref = new ValuedObjectReference:
   ref.setValuedObject(trap) ->
   expr.subExpressions.add(ref)
;

/*
 * esterel provides additional expression constructs which need to be converted
 * into a synccharts adequate form.
 * These are: - FunctionExpression (replaced by hostcode)
 *            - ConstantExpression (replaced by corresponding primitive type)
 *
 */
Expression convertEsterelExpression(Expression e):
   if e != null then
       convertEsterelExpressionRec(e) ->  e
;
// in case there is no expression at all, null is no problem here
Expression convertEsterelExpression(Void v):
   null
;
Void replaceWithCorrespondingExpression(Expression e):
   let parent = (ComplexExpression) e.eContainer:
   parent.subExpressions.add(e.convertEsterelExpression()) ->
   parent.subExpressions.remove(e)
;
// apply conversion on children
Expression convertEsterelExpressionRec(Expression e):
   if ComplexExpression.isInstance(e) then
      ((ComplexExpression) e).subExpressions.convertEsterelExpression()
```

```
;
// Function expression
Expression convertEsterelExpressionRec(FunctionExpression fe):
   let textExpr = new TextExpression:
   textExpr.subExpressions.addAll(fe.kexpressions) ->
   textExpr.setCode(fe.function.name) ->
   textExpr
;
// Constant expression
Expression convertEsterelExpression(ConstantExpression te):
    convertConstantExpressionJava(te)
;

/**
 * finds and returns the initial state of the passed region.
 * In case an inconsistency occurs, a dummy state is created giving feedback about
     the problem.
 */
State findInitialState(Region r):
   let initials = r.states.select(e|e.isInitial):
   if initials.isEmpty then
      (let s = new State:
       s.setLabel("Inconsistency! Could not find initial state in this region.")
           ->
       r.states.add(s) ->
       initials.add(s)) ->
   initials.get(0)
;

/**
 * creates an immediate weakly aborting transition between the two passed states.
 */
Void createImmediateWeakAbortTo(State from, State to, Transition original):
   let t = new Transition:
   from.outgoingTransitions.add(t) ->
   t.setSourceState(from) ->
   t.setTargetState(to) ->
   t.addEffects(original.effects) ->
   to.incomingTransitions.add(t) ->

   t.setType(TransitionType::WEAKABORT) ->
   t.setIsImmediate(true)
;

/**
 * copies a normal transition with "from" as the new source state.
 */
Void copyNormalTransitionFrom(State from, Transition t):
   let newT = new Transition:
```

```
   newT.setSourceState(from) ->
   newT.setType(t.type) ->
   newT.setTrigger((Expression) clone(t.trigger)) ->
   newT.setEffects((List[Effect]) clone(t.effects)) ->
   newT.setTargetState(t.targetState) ->
   from.outgoingTransitions.add(newT) ->
   t.targetState.incomingTransitions.add(newT)
;

/**
 * adds all the passed effects to the transition (clones them previously)
 */
Void addEffects(Transition t, List[Effect] effects):
   // make sure to clone! as it's containment
   t.effects.addAll(clone(effects))
;

/**
 * removes the passed transition from all containments
 */
Void removeTransition(Transition t):
   let source = t.sourceState:
   let target = t.targetState:
   t.setTargetState(null) ->
   t.setSourceState(null) ->
   source.outgoingTransitions.remove(t) ->
   target.incomingTransitions.remove(t)
;

/**
 * remove a state from its parent region if the state was the last one within that region.
 * the region is removed as well.
 */
Void removeStateFromRegion(State state):
   let parent = state.parentRegion:
   state.outgoingTransitions.removeAll(state.outgoingTransitions) ->
   state.incomingTransitions.removeAll(state.incomingTransitions) ->
   parent.states.remove(state) ->
   if parent.states.isEmpty then
      (if parent.parentState != null then
         parent.parentState.regions.remove(parent))
;

/**
 * adds all states of r to the list
 */
Void collectStates(Region r, List[State] states):
   states.addAll(r.states)
```

```
;

/**
 * collects the number of already defined traphalt signals
 */
Integer getNumberOfTraphalts(State s):
   s.parentRegion.parentState != null ?
      getNumberOfTraphalts(s.parentRegion.parentState) :
      getNumberOfTraphaltsFromRoot(s)
;
Integer getNumberOfTraphaltsFromRoot(State root):
   let signals = root.eAllContents.select(e|Signal.isInstance(e)) :
   let traphalts = signals.select(e|((Signal)e).name.contains("traphalt")):
   traphalts.size
;

/**
 * adds the trap as signal to the specified state.
 */
Void addTrapSignalToState(ISignal trap, State s):
   addSignalToState((ISignal) clone(trap), s)
;

/**
 * collect all traphalts in between an exit and the fired trap
 */
List[ISignal] findAndAddCorrespondingTraphalts(ISignal trap, State s, Transition t
    ):
   let found = (Collection[ISignal]) {}:
   collectTrapHalts(trap, s, found) ->
   found.addTrapHalt(t) ->
   found
;
Void collectTrapHalts(ISignal trap, State s, Collection[ISignal] found):
   if !(s.signals.containsSignalWithSameName(trap)) then
      (let currentHalts = s.signals.select(e|e.name.contains("traphalt")):
       found.addAll(currentHalts) ->
       collectTrapHalts(trap, s.parentRegion.parentState, found))
;
Void addTrapHalt(Signal traphalt, Transition t):
   let emission = new Emission:
   emission.setSignal(traphalt) ->
   t.effects.add(emission)
;
Boolean containsSignalWithSameName(List[Signal] signals, Signal s2):
   let sameName = signals.select(e|(e.name.matches(s2.name))):
   !sameName.isEmpty
;

```

```
/**
 * creates a new valued object reference for the passed valued object.
 */
ValuedObjectReference createValObjReference(ValuedObject obj):
   let ref = new ValuedObjectReference:
   ref.setValuedObject(obj) ->
   ref
;

/**
 * returns a copy of the passed list, just the list is new, all elements
 * remain the same
 */
List copyList(List list):
   let copy = {}:
   copy.addAll(list)
;

// to avoid casting
List[Transition] copyListTrans(List[Transition] list):
   let copy = {}:
   copy.addAll(list)
;

/**
 *        Predicates used to determine the optimization capacity of a state.
 */

Boolean isConditional(State s):
   s.type == StateType::CONDITIONAL
;

Boolean isTransitionWithoutTaE(Transition t):
   t.trigger == null && t.effects.size == 0
;

Boolean isTransitionWithoutT(Transition t):
   t.trigger == null
;

Boolean isImmediateTransition(Transition t):
   t.isImmediate
;

Boolean isParallelMacroState(State s):
   s.regions.size > 1
;

Boolean isSimpleState(State s):
```

```
   !s.hasSignalsVariables()
    && s.regions.isEmpty
    && s.entryActions.isEmpty
    && s.innerActions.isEmpty
    && s.exitActions.isEmpty
    && s.suspensionTrigger == null
;

Boolean hasSignalsVariables(State state):
   scopeHasSignalsVariables(state)
;

Boolean hasOnlySelfLoop(State s):
   let self = s.incomingTransitions.select(e|e.targetState == e.sourceState):
   self.size > 0 && self.size == s.outgoingTransitions.size
;

Boolean hasNumberOfSubStates(State s, Integer number):
   let states = (List[State]) {}:
   s.regions.collectStates(states) ->
   states.size == number
;

Boolean hasNumberOfOutgoingTrans(State s, Integer n):
   s.outgoingTransitions.size == n
;

Boolean hasNumberOfIncomingTrans(State s, Integer n):
   s.incomingTransitions.size == n
;

Boolean hasOutTransitions(State s):
   !s.outgoingTransitions.isEmpty
;

Boolean hasOutWeakTransitions(State s):
   let weaks = s.outgoingTransitions.select(e|e.type == TransitionType::WEAKABORT)
       :
   weaks.size > 0
;

// only use after checking for existing transitions
Boolean hasOnlyMatchingTriggerTrans(State s):
   let in = s.incomingTransitions.get(0):
   let out = s.outgoingTransitions.get(0):
   // incoming trans may not have any effect
   (in.effects.isEmpty) ?
      in.compareTrigger(out)
      :
```

```
      false
;

Boolean hasOutNormalTransitions(State s):
   let normals = s.outgoingTransitions.select(e|e.type == TransitionType::
       NORMALTERMINATION):
   normals.size > 0
;

Boolean hasOutStrongTransitions(State s):
   let strongs = s.outgoingTransitions.select(e|e.type == TransitionType::
       STRONGABORT):
   strongs.size > 0
;

Boolean hasFinalSubState(State s):
   let states = (List[State]) {}:
   s.regions.collectStates(states) ->
   !states.select(e|e.isFinal).isEmpty
;

Boolean hasMultipleSimpleFinalSubStates(State s):
   let regions = s.regions.select(e | e.states.select(e|e.isSimpleState() && e.
       isFinal).size > 1):
   !regions.isEmpty
;

Boolean hasParentMacroState(State s):
   s.parentRegion.parentState != null
;


/**
 *       Methods calling Java
 */

/**
 * converts a ConstantExpression into a kexpressions valid form. See javadoc for
     further information.
 */
Expression convertConstantExpressionJava(ConstantExpression e):
   JAVA de.cau.cs.kieler.kies.transformation.util.TransformationUtil.
       convertConstantExpression(de.cau.cs.kieler.kies.esterel.ConstantExpression)
;

/**
 * sets the state's body reference and adds TextualCode for the specific esterel
     element.
 */
```

```
Void setJavaBodyReference(State s, emf::EObject esterelElement):
   JAVA  de.cau.cs.kieler.kies.transformation.util.TransformationUtil.
       setBodyReference(de.cau.cs.kieler.synccharts.State, org.eclipse.emf.ecore.
       EObject)
;

/**
 * adds all elements of list2 to the front of list1.
 */
Void addToFrontOfList(List list1, List list2):
   JAVA  de.cau.cs.kieler.kies.transformation.util.TransformationUtil.
       addToFrontOfList(java.util.List, java.util.List)
;

/**
 * compares the two passed triggers and returns wheter they are equivalent.
     Ignores possible delays and t2's effects.
 */
Boolean compareTrigger(Action t1, Action t2):
   JAVA de.cau.cs.kieler.kies.transformation.util.TransformationUtil.
       compareTrigger(de.cau.cs.kieler.synccharts.Action,
   de.cau.cs.kieler.synccharts.Action)
;

Void debug(Object obj):
   JAVA  de.cau.cs.kieler.kies.transformation.util.TransformationUtil.debug(java.
       lang.Object)
;
```

Bibliography

[Ame10] Torsten Amende. Synthese von SC-Code aus SyncCharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf. xi, 11

[And96] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996. 1

[And03] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France, April 2003. http://www.esterel-technologies.com. 6, 10

[BC84] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 389–448. Springer-Verlag, 1984. 4

[Ber00] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000. ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf. xiii, 1, 5, 31, 35

[Ble10] Joachim Bleidiessel. A domain specific language for railway control. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2010. 140

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

[Est06] Esterel Technologies. *SCADE Technical Manual*, 5.1 edition, February 2006. 1

[EZ07] Stephen A. Edwards and Jia Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, Article ID 52651, 31 pages, 2007. 9

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? June 2005. `http://martinfowler.com/articles/languageWorkbench.html`. 16

[Fuh11] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling.* Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011. xi, 2, 24, 26

[FvH10a] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, 2010. 2

[FvH10b] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, LNCS, Oslo, Norway, October 2010. Springer. 2, 26, 141

[GGBM91] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), September 1991. 1

[GW06] Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 543–557. Springer Berlin / Heidelberg, 2006. 10.1007/11880240_38. 12

[HR01] Nicolas Halbwachs and Pascal Raymond. *A Tutorial of Lustre*, 2001. 1

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST). 11

[JM03] Timothy Jacobs and Benjamin Musial. Interactive visual debugging with UML. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis 03, pages 115–122, New York, NY, USA, 2003. ACM. 12

[JS09] Angelika Kusel Werner Retschitzegger Wieland Schwinger Manuel Wimmer Johannes Schönböck, Gerti Kappel. Catch me if you can – debugging support for model transformations. In *Models in Software Engineering Workshops and Symposia at MODELS 2009*, LNCS 6002, pages 5–20, 2009. 12

[Küh06] 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. 10, 39, 45, 46, 47, 48, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 90, 137

[KW07] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007. 12

[LS11] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 2011. http://LeeSeshia.org. 1

[Luk10] Adriana Lukaschewitz. Transformation von Esterel nach SyncCharts in KIELER. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/adl-bt.pdf. 10

[Mat10] Michael Matzen. A generic framework for structure-based editing of graphical models in Eclipse. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf. 11, 42

[MFvH09] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. Technical Report 0923, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. 24

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). 18

[Mot09] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf. xi, 10, 24, 25

[Mot10] Christian Motika. Executing synccharts with ptolemy. Presentation at the 17th International Open Workshop on Synchronous Programming (SYNCHRON'10), Frejus, France, December 2010. 11

[MSF+11] Christian Motika, Miro Spönemann, Hauke Fuhrmann, Christoph Krüger, John Julian Carstens, and Reinhard von Hanxleden. KIELER

Actor Oriented Modeling (KAOM). Poster presented at 9th Biennial Ptolemy Miniconference (PTCONF'11), Berkeley, CA, USA, February 2011. 11

[Mül10] Martin Müller. View management for graphical models. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2010. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mmu-mt.pdf`. 26

[Nat08] National Instruments. LabVIEW, visited 03/2008. `http://www.ni.com/labview/`. 1

[PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007. 4, 31

[PTvH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006. 1, 4, 10

[PvH07] Steffen Prochnow and Reinhard von Hanxleden. Enhancements of Statechart-modeling—the KIEL environment. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE'07)*, Nice, France, April 2007. With accompanying poster. 10

[Sch08] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf`. 1

[Sch09] Matthias Schmeling. ThinKCharts—the thin KIELER SyncCharts editor. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf`. 27

[Sch11] Christian Schneider. On integrating graphical and textual modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2011. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/chsch-dt.pdf`. xi, 13, 16, 141

[SSBD99] Sanjit A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar. A translation of Statecharts to Esterel. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 volume 2— World Congres on Formal Methods*, volume 1709 of *LNCS*, pages 983–1007. Springer-Verlag, 99. 9

[TAvH11] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe (DATE'11)*, Grenoble, France, March 2011. 11

[Tra10] Claus Traulsen. *Reactive Processing for Synchronous Language and its Worst Case Reaction Time Analysis*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2010. `http://eldiss.uni-kiel.de/macau/servlets/MCRFileNodeServlet/dissertation_derivate_00003253/ClausTraulsen.pdf`. 4, 5, 6

[vH09] 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. 11

[vHF10] Reinhard von Hanxleden and Hauke Fuhrmann. Taming graphical modeling. Presentation at the 17th International Open Workshop on Synchronous Programming (SYNCHRON'10), Frejus, France, December 2010. 1