Transformation von Esterel nach SyncCharts in KIELER

Adriana Lukaschewitz

Überarbeitete Version der gleichnamigen Bachelorarbeit,

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel,
März 2010

Datum der Überarbeitung: 10. Mai 2010

Zusammenfassung

Für die Beschreibung von reaktiven Systemen gibt es unterschiedliche Darstellungsformen. Zwei davon sind Esterel und SyncCharts. Während Esterel eine textuelle Darstellung ist, handelt es sich bei SyncCharts um eine grafische Darstellung. Beide haben ihre Vorteile und Nachteile.
Grafische Darstellungsformen erlauben es dem Menschen, Zusammenhänge schneller und leichter zu verstehen. Jedoch ist es meist schwierig, Veränderungen vorzunehmen.
Textuelle Darstellungsformen erlauben schnelle Veränderung, sind jedoch meist deutlich schwieriger zu verstehen und auf Fehler zu überprüfen.
Durch die Transformation von Esterel nach SyncChart sollen die Vorteile beider Darstellungsformen vereint werden. Während ein Programm verhältnismäßig schnell in Esterel *geschrieben* ist, fällt die *Analyse* nach der Transformation in ein SyncChart leichter.
Diese Arbeit ist als Teil des KIELER-Projekts implementiert worden, welches es sich zur Aufgabe gemacht hat, den Entwickler beim modellbasierten Entwurf komplexer Systeme zu unterstützen.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Esterel . . . 1
- 1.2 SyncCharts . . . 2
- 1.3 Verwendete Werkzeuge . . . 3
  - 1.3.1 Eclipse . . . 3
  - 1.3.2 EMF und GMF . . . 3
  - 1.3.3 Xtext . . . 4
  - 1.3.4 Xpand und Xtend . . . 5
  - 1.3.5 Kiel Integrated Environment for Layout Eclipse Rich Client . . . 5
  - 1.3.6 ThinKCharts . . . 6
- 1.4 Verwandte Arbeiten . . . 6

2 Konzept **7**
- 2.1 Konzept in KIEL . . . 7
- 2.2 Konzept in KIELER . . . 8

3 Umsetzung von Esterel in Eclipse **11**
- 3.1 Module . . . 11
- 3.2 Module Interface . . . 12
  - 3.2.1 Signale und Sensoren . . . 12
  - 3.2.2 Typen . . . 13
  - 3.2.3 Konstanten . . . 13
  - 3.2.4 Relationen . . . 14
  - 3.2.5 Funktionen und Prozeduren . . . 14
- 3.3 Module Body . . . 15
  - 3.3.1 Sequence, Parallel und Block . . . 16
  - 3.3.2 Abort . . . 17
  - 3.3.3 Await . . . 17
  - 3.3.4 Do . . . 18
  - 3.3.5 Emit . . . 18
  - 3.3.6 Every-do . . . 18
  - 3.3.7 Halt . . . 19
  - 3.3.8 If . . . 19
  - 3.3.9 Loop . . . 20
  - 3.3.10 Nothing . . . 20
  - 3.3.11 Pause . . . 20

3.3.12 Present . . . 21
3.3.13 Repeat . . . 21
3.3.14 Run . . . 21
3.3.15 Local signal . . . 22
3.3.16 Suspend . . . 23
3.3.17 Sustain . . . 23
3.3.18 Trap . . . 24
3.3.19 Local variable . . . 24
3.3.20 Assignment . . . 25
3.3.21 Weak abort . . . 25

4 Transformation von Esterel nach SyncCharts 27
4.1 Module . . . 27
4.2 Sequence . . . 28
4.3 Parallel . . . 28
4.4 Abort . . . 28
4.5 Await . . . 29
4.6 Do-Upto . . . 30
4.7 Do-Watching . . . 30
4.8 Emit . . . 31
4.9 Every . . . 31
4.10 Halt . . . 32
4.11 If . . . 32
4.12 Loop . . . 32
4.13 Loop-Each . . . 33
4.14 Nothing . . . 34
4.15 Pause . . . 34
4.16 Present . . . 35
4.17 Local Signal . . . 35
4.18 Suspend . . . 35
4.19 Sustain . . . 35
4.20 Trap . . . 35
4.21 Exit . . . 37
4.22 Local Variable . . . 38
4.23 Assignment . . . 38
4.24 Weak abort . . . 38

5 Zusammenfassung und Ausblick 43
5.1 Optimierung von SyncCharts . . . 43
5.2 Erweiterung auf Esterel v7 . . . 44

Literaturverzeichnis 45

Abbildungsverzeichnis

1.1 ABRO in Esterel und als SyncChart . . . 3
1.2 Klassenstruktur eines Adressbuchs . . . 5

2.1 Ein durch Xtext erzeugter Editor . . . 9
2.2 SyncChart Metamodell in Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) . . . 10

3.1 Modell von Module . . . 12
3.2 Modell von Module Interface . . . 15
3.3 Eindruck über den Umfang des Modells von Esterel . . . 26

4.1 Transformation von Module . . . 28
4.2 Transformation von Sequence . . . 28
4.3 Transformation von Parallel . . . 29
4.4 Transformation von Abort . . . 29
4.5 Transformation von Await . . . 30
4.6 Transformation von Do-Upto . . . 30
4.7 Transformation von Do-Watching . . . 31
4.8 Transformation von Emit . . . 31
4.9 Transformation von Every . . . 32
4.10 Transformation von Halt . . . 32
4.11 Transformation von If . . . 33
4.12 Transformation von Loop . . . 33
4.13 Transformation von Loop-Each . . . 33
4.14 Transformation von Nothing . . . 34
4.15 Transformation von Pause . . . 34
4.16 Transformation von Present . . . 35
4.17 Transformation von Local Signal . . . 36
4.18 Transformation von Suspend . . . 36
4.19 Transformation von Sustain . . . 36
4.20 Transformation von Trap . . . 37
4.21 Transformation von Exit . . . 37
4.22 Transformation von Local Variable . . . 38
4.23 Transformation von Assignment . . . 38
4.24 Transformation von Weak Abort . . . 39
4.25 Transformationsergebnis von ABRO . . . 41

Verzeichnis der Abkürzungen

CEC	Columbia Esterel Compiler
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GMF	Eclipse Graphical Modeling Framework
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
ThinKCharts	Thin Kieler SyncCharts Editor

1 Einführung

Um Zusammenhänge zu beschreiben, kann man auf zwei wesentliche Arten der Beschreibung zurückgreifen, nämlich auf textuelle und grafische Beschreibungen.
Dabei sind textuelle Beschreibungen in der Regel leichter und schneller verfasst. Weiterhin bereitet es kaum Schwierigkeiten, Veränderungen im Nachhinein einzubauen. Es ist jedoch schwieriger, eine textuelle Beschreibung zu verstehen oder Fehler zu finden.
Grafische Beschreibungen sind verhältnismäßig leicht zu verstehen. Fehler lassen sich recht schnell erkennen. Es ist jedoch deutlich umständlicher eine grafische Beschreibung zu verfassen. Nachträgliche Veränderungen lassen sich meist nur mit großem Aufwand einbauen.
Esterel und SyncCharts sind zwei unterschiedliche Beschreibungsformen für reaktive Systeme. Esterel ist eine textuelle Beschreibung während ein SyncChart eine grafische Beschreibung ist. Durch die Transformation von Esterel nach SyncChart sollen die Vorteile beider Beschreibungsformen genutzt werden. Es ist dadurch möglich ein System in vergleichsweise kurzer Zeit in Esterel zu beschreiben oder auch zu ändern. Dank der Transformation muß man allerdings nicht auf die leichtere Analyse im SyncChart verzichten.
Die Implementierung der Transformation ist Teil des KIELER Projekts, das Entwickler beim modellbasierten Entwurf unterstützen soll.

Im Folgenden möchte ich die verwendeten Sprachen und Werkzeuge vorstellen. Dabei handelt es sich nur um eine kurze Beschreibung, die lediglich einen groben Überblick bieten soll. Für genauere Details verweise ich auf die entsprechende Literatur.

1.1 Esterel

Bei Esterel handelt es sich um eine von G. Berry entwickelte synchrone Sprache [2], die speziell für die Beschreibung eingebetteter, reaktiver Systeme geeignet ist. Das in einem Esterel-Programm beschriebene Verhalten kann sowohl in Hardware als auch in einer Mischung aus Hard- und Software umgesetzt werden.
Die Ausführung des Programms wird in einzelne Zeitabschnitte, sogenannte Instanzen oder Ticks, unterteilt. Es können mehrere Befehle innerhalb einer Instanz ausgeführt werden. Befehle, die innerhalb derselben Instanz ausgeführt werden, gelten als gleichzeitig und verbrauchen somit keine Zeit. Dies hat insbesondere zur Folge, dass ein Signal, das innerhalb einer Instanz auf *present* gesetzt wird (ausgesendet wird), für die gesammte Instanz als *present* gilt.

Esterel besteht aus einigen wenigen Grundbefehlen, den *Kernel-Statements*, aus denen alle weiteren Befehle gebildet werden können. Tabelle 1.1 zeigt eine Übersicht der Kernel-Statements.

nothing	leerer Ausdruck
pause	Trennung von zwei Makroschrittten
emit S	Signal Emission
present S then p else q end	bedingte Anweisung
suspend p when S	Suspendierung eines Prozesses
p;q	Sequenz
p || q	synchrone Parallelität
loop p end	unendliche Schleife
trap T in p end	Ausnahmebehandlung
exit T	Auslösen einer *exception*
signal S in p end	lokales Signal

Tabelle 1.1: Esterel Kernel-Statements

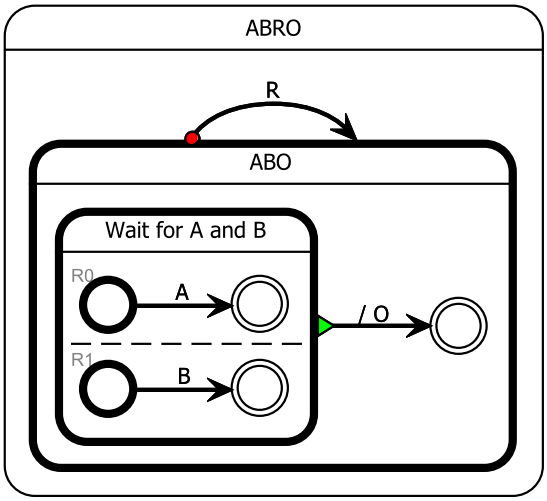
1.2 SyncCharts

Statecharts wurden von David Harel [4] entwickelt und werden genutzt, um komplexe reaktive Systeme zu spezifizieren. Sie sind eine Form von endlichen Automaten, die um Hierarchie, Parallelität, Daten und Broadcasts erweitert wurden.
Die Hierarchie erlaubt es, das System in unterschiedlichen Schichten zu betrachten. So mag die oberste Hierarchie-Ebene das Zusammenspiel zwischen großen Komponenten beschreiben, während tiefere Ebenen das Verhalten der einzelnen Komponenten darstellen. Die Hierarchie dient folglich der Lesbarkeit und verringert die Anzahl der benötigten Transitionen.
Parallelität dient vor allem der Übersicht und dem Verständnis. Insbesondere wird die Zustandsexplosion, die bei hoher Parallelität in endlichen Automaten auftritt, vermieden.
SyncCharts sind eine Abwandlung der *Statecharts*, die von Charles André [1] entwickelt wurde. *SyncCharts* orientieren sich stark an Esterel. SyncCharts sind insbesondere, ebenso wie Esterel, eine synchrone, deterministische Sprache. SyncCharts lassen sich automatisch in Esterel-Programme übersetzen.

In Abbildung 1.1 ist das Programm ABRO sowohl in Esterel als auch als SyncChart zu sehen. Das Programm wartet darauf, dass die Signale A und B auftreten und emittiert dann das Signal O. Dieser Vorgang wird sofort abgebrochen und neu gestartet, wenn das Signal R auftritt.

```
1  module ABRO:
2  input A, B, R;
3  output O;
4  loop
5  [
6    await A
7    ||
8    await B
9  ];
10 emit O;
11 each R
12 end module
```

(a) Esterel



(b) SyncChart

Abbildung 1.1: ABRO in Esterel und als SyncChart

1.3 Verwendete Werkzeuge

Bei der Erstellung dieser Arbeit kamen einige Werkzeuge zum Einsatz, die ich im Folgenden kurz vorstellen möchte.

1.3.1 Eclipse

Eclipse[1] ist eine Entwicklungsumgebung, die sich durch große Erweiterbarkeit auszeichnet. Ursprünglich als Entwicklungsumgebung für JAVA gedacht, ist Eclipse dank diverser Plugins inzwischen für weit mehr unterschiedliche Programmieraufgaben nutzbar.
Bei Eclipse handelt es sich um ein Open Source Projekt.

1.3.2 EMF und GMF

Das Eclipse Modeling Framework (EMF)[2] ist ein Plugin für Eclipse. Es kann aus einem Modell, das in einer XMI-Datei beschrieben wird, Java-Code erzeugen. Basierend auf diesem Code können Instanzen des Modells erzeugt werden.
Das Eclipse Graphical Modeling Framework (GMF)[3] bietet Werkzeuge, die es einem

[1] http://www.eclipse.org/
[2] http://www.eclipse.org/modeling/emf/
[3] http://www.eclipse.org/gmf/

erlauben grafische Editoren auf Basis von EMF zu erstellen.

1.3.3 Xtext

Xtext[4] ist ein Plugin für Eclipse, das für die Einbindung von Domain Specific Languages (DSLs) in Eclipse gedacht ist.
Es erfordert die Eingabe der Grammatik der entsprechenden Sprache und liefert im Gegenzug einen Editor und einen Parser, der auf dem mitgenerierten EMF-Modell der Sprache basiert.
Die Grammatik muss in Erweiterter Backus-Naur-Form (EBNF)-Syntax angegeben werden. Das Besondere ist, dass neben der konkreten Syntax auch die abstrakte Syntax in die Grammatik einfließt.
Ich will dies an einem Beispiel verdeutlichen. Nehmen wir an, dass eine DSL zur Beschreibung von Adressbucheinträgen erstellt werden soll. Ein Eintrag in der Grammatik könnte wie in Auflistung 1.1 lauten. Dieser Grammatikeintrag bedeutet für die konkrete Syntax lediglich, dass nach dem Eingeben des Wortes *name* zwei Strings erwartet werden. Für die abstrakte Syntax bedeutet der Eintrag, dass ein Objekt *Person* im Modell vorkommt. Dieses hat zwei Attribute, nämlich *firstname* und *lastname*, unter denen der erste und zweite String abgelegt werden.

Auflistung 1.1: Teilgrammatik einer Adresssbuchsprache in Xtext

```
1 Person:
2 "name" firstname=STRING lastname=STRING;
```

Basierend auf der abstrakten Syntax wird ein Parser erstellt. Um die Funktion zu verdeutlichen, erweitern wir unsere Grammatik aus Auflistung 1.1, so dass ein Adressbuch mit mehreren Personen erstellt wird. Die neue Grammatik ist in Auflistung 1.2 zu sehen.

Auflistung 1.2: Grammatik einer Adressbuchsprache in Xtext

```
1  Addressbook:
2      person+=Person+;
3
4  Person:
5      name=Name ("," address=Address)? ";";
6
7  Name:
8      "name" firstname=STRING lastname=STRING ;
9
10 Address:
11     address=STRING;
```

[4] http://www.eclipse.org/Xtext/

Demnach enthält ein Adressbuch nun eine oder mehrere Personen. Eine Person hat einen Namen, bestehend aus Vor- und Nachname, und optional eine Adresse, die lediglich als String angegeben wird.
`Xtext` generiert dazu ein passendes Metamodell, anhand dessen Informationen beim Parsen abgelegt werden. Das Metamodell zu Auflistung 1.2 ist in Abbildung 1.2 zu sehen.

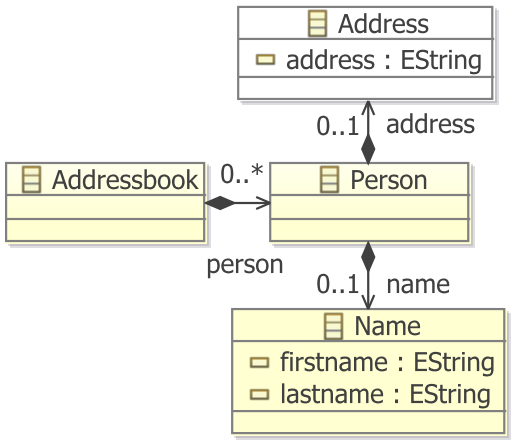


Abbildung 1.2: Klassenstruktur eines Adressbuchs

1.3.4 Xpand und Xtend

Xtend[5] ist eine funktionale Sprache, die unter anderem genutzt wird, um bestehende Modelle mit weiterer Logik auszustatten. Weiterhin kann Xtend jedoch auch für Model-zu-Model-Transformationen genutzt werden. Xtend kann sowohl für Transformationen zwischen unterschiedlichen Modellen also auch für Transformationen mit dem selben Ausgangs- und Zielmodell genutzt werden.
Xtend ist ein Teil des Xpand-Projekts, welches den Benutzer beim modellbasierten Entwurf unterstützen soll und häufig auch für die Codegenerierung genutzt wird.

1.3.5 Kiel Integrated Environment for Layout Eclipse Rich Client

Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[6] ist ein Forschungsprojekt am Lehrstuhl für Echtzeit und Eingebettete Systeme an der Christian-Albrechts-Universität zu Kiel. Ziel des Projekts ist es, den modellbasierten Entwurf

[5] `http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01s05.html`
[6] `http://www.informatik.uni-kiel.de/rtsys/kieler/`

von komplexen Systemen zu unterstützen und zu vereinfachen [3]. Automatisches Layout ist dabei der Anfang und Grundstein der folgenden Arbeiten [11]. Diese umfassen Erleichterungen im Bereich des Editierens, wie z.B. strukturbasiertes Editieren, als auch Hilfsmittel in der Visualisierung, z.B. zum Vergleich von Modellen [9].
Dabei soll sich KIELER nicht auf eine bestimmte Sprache einschränken, sondern soll eine Vielfalt an leicht integrierbaren Werkzeugen bereitstellen.

1.3.6 ThinKCharts

Der Thin Kieler SyncCharts Editor (ThinKCharts) [10] ist Teil des KIELER-Projekts. Es handelt sich um einen SyncChart-Editor, der die Features von KIELER, wie z.B. automatisches Layout, nutzt.
ThinKCharts basiert zum großen Teil auf EMF und GMF.

1.4 Verwandte Arbeiten

Das Kiel Integrated Environment for Layout (KIEL) [8] ist das Vorgängerprojekt zu KIELER.
KIEL wurde von Prochnow et al. speziell für den Umgang mit Statecharts entwickelt. Unter anderem ermöglicht KIEL die Transformation von Esterel nach SyncCharts. Die Transformationen wurden von Lars Kühl im Rahmen seiner Diplomarbeit [5] in KIEL implementiert. Vorgestellt wurden die Transformationstheorien von Prochnow et al. [7]
Die Neuimplementierung in KIELER soll im Gegensatz zur KIEL-Version von der Erweiterbarkeit und Modularität von Eclipse profitieren.

2 Konzept

In diesem Kapitel möchte ich nicht nur das aktuell angewandte Konzept erläutern, sondern auch auf vorhergehende Ideen eingehen.

2.1 Konzept in KIEL

Die Möglichkeit, Esterel nach SyncCharts zu transformieren, war bereits in KIEL gegeben, weshalb es naheliegend ist, den alten Code so weit wie möglich wiederzuverwenden.
Die Transformation ist in KIEL in drei Schritte unterteilt. Im ersten Schritt wird das Esterel Programm geparsed. Im nächsten Schritt findet die eigentliche Transformation statt. Da die Transformationsregeln allerdings sehr allgemein gehalten sind, ist das resultierende SyncChart sehr umfangreich. Daher findet ein letzter Schritt statt, in dem das SyncChart optimiert wird. Ziel der Transformationen ist dabei immer das SyncChart-Modell, das die grafischen Editoren in KIEL für die Darstellung von SyncCharts nutzen, welches manuell in Java implementiert wurde.
Für das Parsen kommt der Columbia Esterel Compiler (CEC)[1] zum Einsatz, der in seiner derzeitigen Form Esterel v5 unterstützt. Der CEC erzeugt C-Code aus Esterel-Code und hinterlässt dabei eine XML-Datei, in der der Abstrakte Syntaxbaum des komplett expandierten Esterel Programms enhalten ist. Die Transformationen finden im Folgenden auf Basis dieser XML-Datei statt.
Die Transformationen sind komplett in JAVA geschrieben und bestehen aus zwei Teilen. Der erste Teil ist für das Einlesen bzw. Erkennen des Esterelbefehls aus der XML-Datei heraus zuständig. Im zweiten Teil wird die eigentliche Transformation des erkannten Befehls in ein äquivalentes SyncChart, basierend auf dem SyncChart-Modell von KIEL, durchgeführt.
Die Optimierung findet ebefalls komplett in JAVA statt. Ursprung und Ziel der Optimierungstransformationen ist ein SyncChart basierend auf dem SyncChartmodell von KIEL.

Da KIEL und KIELER unterschiedliche Modelle für SyncCharts verwenden, ist es nicht möglich, den komplette KIEL-Code zu übernehmen. Durch die Aufteilung der Transformation in zwei Teile, bietet sich jedoch die Möglichkeit, das Parsen und Einlesen der XML-Datei aus KIEL zu übernehmen und lediglich das Ziel der Transformation auf das in KIELER verwendete SyncChartmodell zu ändern. Die Optimierung müsste

[1] `http://www1.cs.columbia.edu/~sedwards/cec/`

für KIELER komplett neu geschrieben werden.
Diese Lösung hat allerdings einen Nachteil. Wie KIEL würde dadurch auch KIELER vom CEC abhängig sein. Der CEC ist komplett in C geschrieben und erfordert daher, dass man den Code für jede Zielplatform individuell kompiliert, was zusätzlichen Aufwand bedeutet.
Weiterhin unterstützt der CEC bisher lediglich Esterel v5. Sollte in KIELER irgendwann die Unterstützung für Esterel v7 implentiert werden sollen, so wäre man darauf angewiesen, dass der CEC entsprechend erweitert wird. Andernfalls wäre eine Neuentwicklung ohne den CEC nötig.

2.2 Konzept in KIELER

Um späteren Problemen mit dem CEC aus dem Weg zu gehen, soll die Implementierung der Transformationen in KIELER ohne den CEC erfolgen.
Der wiederverwendbare Code aus KIEL basiert allerdings komplett auf dem CEC, wodurch auch dieser aus der Neuimplementierung entfällt.
Da der CEC als Parser wegfällt, wird für KIELER ein neuer Parser mit Hilfe von Xtext geschrieben. Damit Xtext einen automatisch generierten Editor mit den üblichen Features ausstattet, erfordert es die Eingabe der kompletten Esterel-Grammatik. Basierend auf der Grammatik werden automatisch ein Parser und ein Editor generiert. Der Editor enthält dabei die üblichen oben genannten Features, die man von einem auf eine spezielle Programmiersprache zugeschnittenen Editor erwartet.
Abbildung 2.1 zeigt einen Editor, der durch Xtext aus einer Grammatik generiert wurde.

Für die Transformationen ins SyncChart wird XTend verwendet. Das Ziel der Transformationen ist das Modell in Abbildung 2.2, das in ThinKCharts verwendet wird. Auf diese Weise können die SyncCharts mit Hilfe des ThinKCharts-Editors angezeigt werden, der bereits in KIELER integriert ist.

Alle Transformationen beruhen dabei auf den bereits in KIEL genutzten Transformationstheorien.

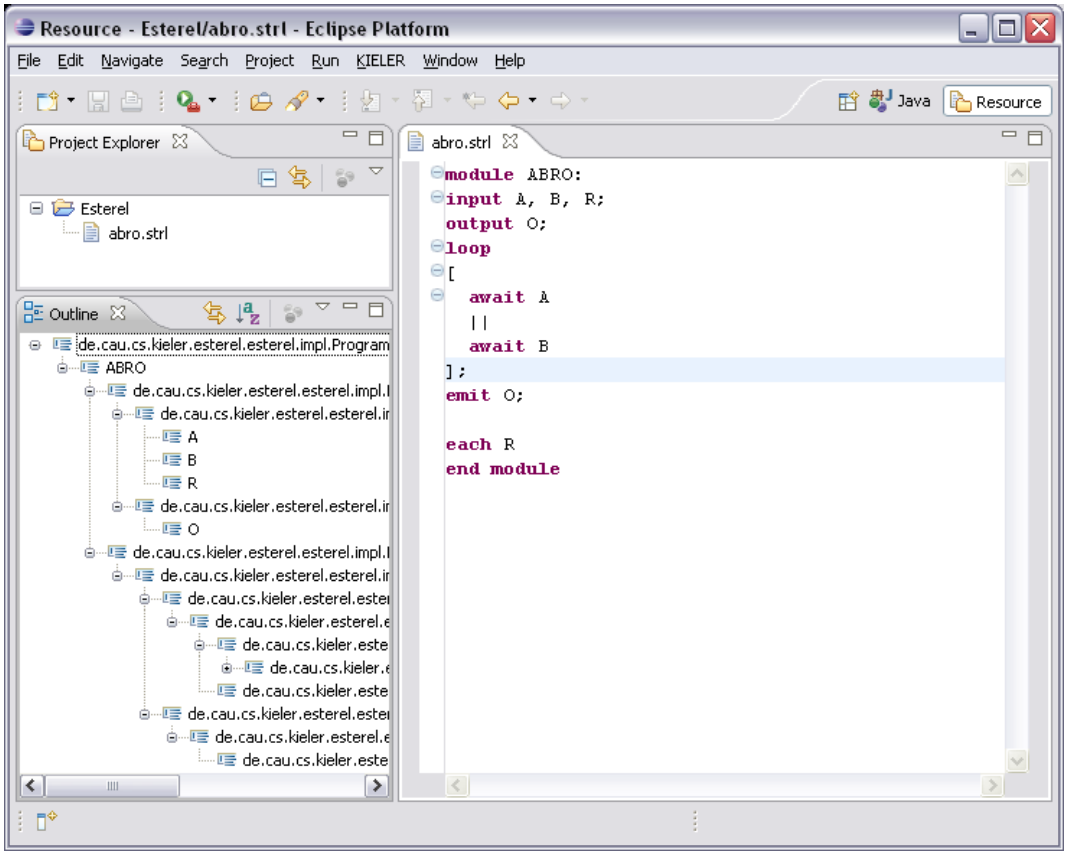


Abbildung 2.1: Ein durch Xtext erzeugter Editor

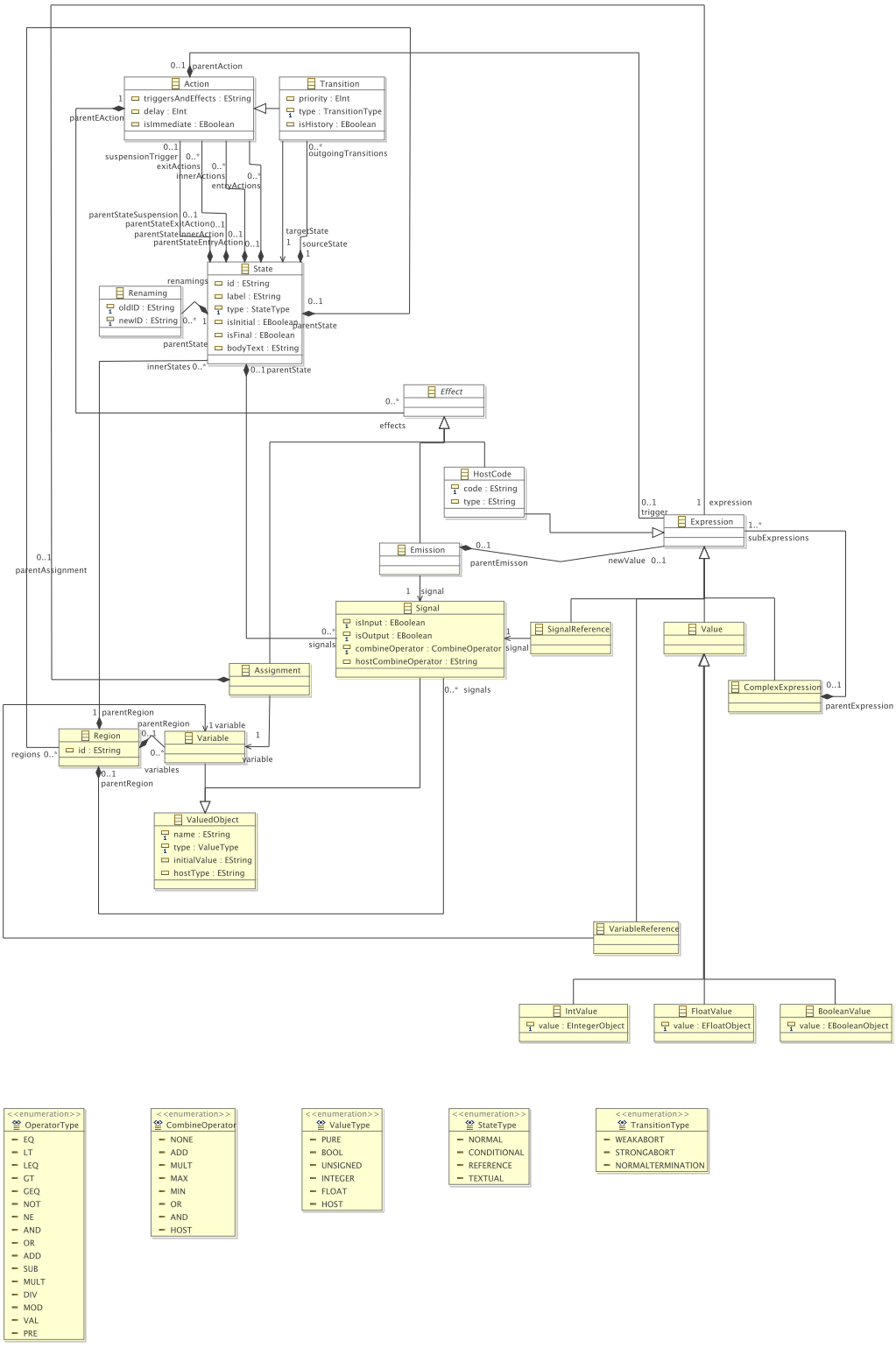


Abbildung 2.2: SyncChart Metamodell in KIELER

3 Umsetzung von Esterel in Eclipse

In der Grundinstallation bringt Eclipse keinen Editor oder Parser für Esterel mit. Weiterhin existieren zum Zeitpunkt dieser Arbeit auch keine Plugins, die Eclipse um diese Funktionen erweitern. Um Transformationen von Esterel nach Statecharts zu ermöglichen, ist es daher nötig, Eclipse um einen Estereleditor und -parser zu erweitern.
Zur Implementierung eben dieser Funktionen gibt es für Eclipse das Plugin Xtext, dass bereits in Unterabschnitt 1.3.3 vorgestellt wurde. Xtext erfodert die Eingabe der Grammatik der DSL, auf der der Editor und Parser basieren sollen. Im Fall dieser Arbeit handelt es sich folglich um die Esterel-Grammatik. Der Großteil der Funktionen, so auch der für diese Arbeit wichtige Parser, werden auf Basis der Grammatik automatisch generiert.

Dieser Abschnitt soll die Umsetzung der Esterel-Grammatik in Xtext demonstrieren. Die vollständige Esterel-Grammatik steht in „Compiling Esterel“ von Dumitru Potop-Butucaru et al. [6]
Zur Benennung von Objekten, wie z.B. Variablen, werden in Esterel *Identifier* genutzt. In „Compiling Esterel“ wird ein *Identifier* als Sequenz von Buchstaben, Ziffern und Unterstrichen, die mit einem Buchstaben beginnen muss, beschrieben. Dabei wird zwischen Groß- und Kleinschreibung unterschieden. Die Entsprechung dazu in Xtext steht in Auflistung 3.1.

Auflistung 3.1: Identifier in Xtext

```
1 terminal EsterelID: ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

3.1 Module

Innerhalb von Esterel bildet *Module* die Basis des Programms. Innerhalb von *Module* werden die verwendeten Signale deklariert und der eigentliche Programmablauf festgehalten.

Auflistung 3.2: Module in Xtext

```
1 Module:
2     "module" name=EsterelID ":" (modInt=ModuleInterface)? modBody=ModuleBody "end"
          "module";
```

Wie in Auflistung 3.2 zu sehen, hat ein *Module* einen Namen, ein optionales *Module Interface*, in dem die Deklarationen stattfinden, und einen *Module Body*, der den Programmablauf enthält. Dies entspricht dem Modell in Abbildung 3.1, das anhand des Grammatik-Eintrags von Xtext generiert wurde.

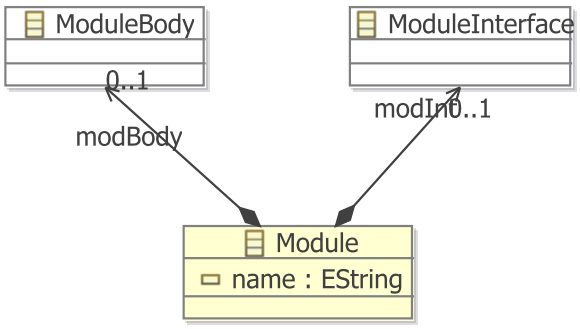


Abbildung 3.1: Modell von Module

3.2 Module Interface

Innerhalb des *Module Interface* (Auflistung 3.3) werden sämtliche Signale (Auflistung 3.4), Typen (Auflistung 3.6), Sensoren (Auflistung 3.5), Konstanten (Auflistung 3.7), Relationen (Auflistung 3.8), Funktionen (Auflistung 3.9) und Prozeduren (Auflistung 3.10) deklariert, die modulweit verfügbar sein sollen.

Auflistung 3.3: Module Interface in Xtext

```
1 ModuleInterface:
2     (signalDecl+=SignalDecl
3     | typeDecl+=TypeDecl
4     | sensorDecl+=SensorDecl
5     | constantDecl+=ConstantDecl
6     | relationDecl+=RelationDecl
7     | functionDecl+=FunctionDecl
8     | procedureDecl+=ProcedureDecl)+;
```

3.2.1 Signale und Sensoren

Signale und Sensoren sind fundamentale Objekte in Esterel. Dabei wird zwischen drei Typen unterschieden. *Pure signals* enthalten nur die Information, ob das Signal *present* oder *absent* ist. *Valued signals* enthalten zusätzlich noch einen Datenwert. *Sensors* enthalten lediglich einen Datenwert.

Die Information, ob ein Signal *present* oder *absent* ist, ist innerhalb einer Zeitinstanz persistent. Ein Signal ist nur *present* in einer Instanz, wenn in dieser ein *emit-Statement* ausgeführt wurde. Andernfalls gilt es als *absent*. Die einzige Ausnahme von dieser Regel bildet das Signal tick, welches immer *present* ist.
Der Wert eines *sensors* wird durch die Umgebung vorgegeben und kann sich zu Beginn jeder Instanz verändern.

Auflistung 3.4: Signal Declaration in Xtext

```
1 SignalDecl:
2     {Input} "input" signal+=Signal ("," signal+=Signal)* ";"
3     | {Output} "output" signal+=Signal ("," signal+=Signal)* ";"
4     | {InputOutput} "inputoutput" signal+=Signal ("," signal+=Signal)* ";"
5     | {Return} "return" signal+=Signal ("," signal+=Signal)* ";";
```

Auflistung 3.5: Sensor Declaration in Xtext

```
1 SensorDecl:
2     "sensor" sensor+=Sensor ("," sensor+=Sensor)* ";";
```

3.2.2 Typen

Esterel enthält fünf Typen. Dies sind boolean, integer, float, double und string. Weiterhin erlaubt Esterel benutzerdefinierte Typen.

Auflistung 3.6: Type Declaration in Xtext

```
1 TypeDecl:
2     "type" type+=Type ("," type+=Type)* ";";
3
4 Type:
5     name=EsterelID;
```

3.2.3 Konstanten

Konstanten sind Objekte, die einen Wert beinhalten. Der Wert einer Konstante wird zum Programmstart gesetzt und während des Programmablaufs nicht verändert. Konstanten werden über einen Namen identifiziert.

Auflistung 3.7: Constant Declaration in Xtext

```
1 ConstantDecl:
2     "constant" constant+=OneTypeConstantDecl ("," constant+=OneTypeConstantDecl)* ";
          ";
3
4 OneTypeConstantDecl:
5     constant+=Constant ("," constant+=Constant)* ":" (type=EsterelID | type=BaseType);
```

```
6
7  Constant:
8      name=EsterelID ("=" value=ConstantValue)?;
```

3.2.4 Relationen

Relationen geben einen Zusammenhang zwischen dem Zustand von Signalen (*present*/*absent*) an.
relation A => B, C # D # E
Die obige Relation besagt, dass B *present* sein muss, wenn A *present* ist, und dass C, D und E sich gegenseitig ausschließen.
Es wird davon ausgegangen, dass diese Zusammenhänge durch die Umgebung bereitgestellt werden.

Auflistung 3.8: Relation Declaration in Xtext

```
1  RelationDecl:
2      {Relation} "relation" relation+=RelationType ("," relation+=RelationType)* ";";
3
4  RelationType:
5      RelationImplication|RelationIncompatibility;
6
7  RelationImplication :
8      first=[Signal|EsterelID] type="=>" second=[Signal|EsterelID];
9
10 RelationIncompatibility :
11     incomp+=[Signal|EsterelID] type="#" incomp+=[Signal|EsterelID] ("#" incomp+=[
           Signal|EsterelID])*;
```

3.2.5 Funktionen und Prozeduren

Funktionen und Prozeduren werden durch den Benutzer definiert. Es wird vorrausgesetzt, dass sie in der Ausführungsinstanz terminieren, in der sie aufgerufen wurden.

Auflistung 3.9: Function Declaration in Xtext

```
1  FunctionDecl:
2      "function" function+=Function ("," function+=Function)* ";";
3
4  Function:
5      name=EsterelID "(" (idList+=(EsterelID|BaseType) ("," idList+=(EsterelID|BaseType))*)
           ? ")" ":" (type=EsterelID | type=BaseType);
```

Auflistung 3.10: Procedure Declaration in Xtext

```
1 ProcedureDecl:
2     "procedure" procedure+=Procedure ("," procedure+=Procedure)* ";";
3
4 Procedure:
5     name=EsterelID
6     "(" (idList1+=(EsterelID|BaseType) ("," idList1+=(EsterelID|BaseType))*)? ")"
7     "(" (idList2+=(EsterelID|BaseType) ("," idList2+=(EsterelID|BaseType))*)? ")";
```

Das aus diesen Deklarationen resultierende Modell für das *Module Interface* ist in Abbildung 3.2 zu sehen.

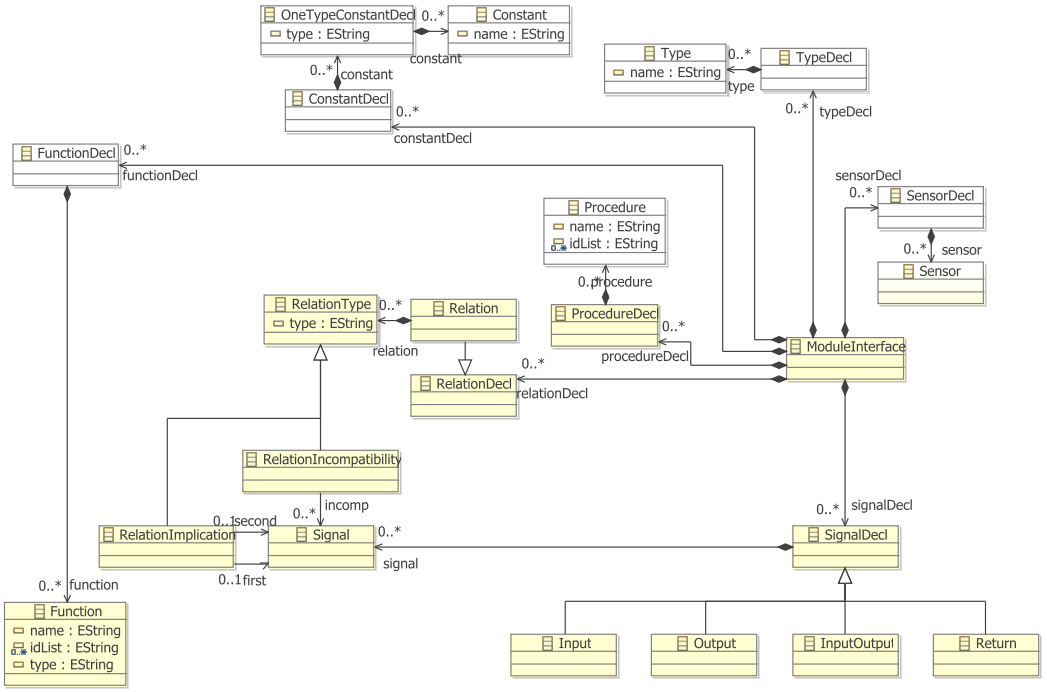


Abbildung 3.2: Modell von Module Interface

3.3 Module Body

Innerhalb des *Module Body* wird der eigentliche Programmablauf deklariert. Dieser setzt sich aus unterschiedlichen *Statements* zusammen, die im Folgenden erläutert werden.

3.3.1 Sequence, Parallel und Block

Durch „ || “ getrennte Objekte werden in Esterel gleichzeitig ausgeführt. Das entsprechende *Statement* wird *Parallel* (Auflistung 3.11) genannt. Unter *Sequence* (Auflistung 3.12) versteht man die Nacheinanderausführung von *Statements*, die durch ein „ ; “ getrennt werden.
Der *Block* (Auflistung 3.13) wirkt als Klammerung und legt somit eine bestimmte Ausführungsreihenfolge fest.

Auflistung 3.11: Parallel in Xtext

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

Auflistung 3.12: Sequence in Xtext

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

Auflistung 3.13: Block in Xtext

```
1 Block:
2     "["statement=Statement"]";
```

Neben diesen *Statements* gibt es noch atomare *Statements*, deren Aufruf breits in *Sequence* zu sehen ist.
Dies sind z.B. *Loop*, *Abort* und *Pause*. In Auflistung 3.14 sieht man den Aufruf der atomaren *Statements*.

Auflistung 3.14: Atomic Statement in Xtext

```
1  AtomicStatement:
2      Abort
3      | Await
4      | Do
5      | Emit
6      | EveryDo
7      | Exit
8      | Halt
9      | IfTest
10     | LocalSignalDecl
11     | Loop
12     | Nothing
13     | Pause
14     | Present
15     | Repeat
16     | Run
17     | Suspend
18     | Sustain
```

```
19      | Trap
20      | Variable
21      | WeakAbort;
```

3.3.2 Abort

Die Grundform von *Abort* (Auflistung 3.15) lautet:
abort p when d
Dabei wird bei Erreichen von *Abort* das Programm p ausgeführt. Die Ausführung von p wird sofort abgebrochen, wenn die Bedingung in d erfüllt ist. Das *Abort-Statement* terminiert, wenn p terminiert oder wenn d erfüllt ist und, sofern vorhanden, der optionale zu einer Bedingung gehörende do-Teil terminiert.

Auflistung 3.15: Abort in Xtext

```
1  Abort:
2      "abort" statement=Statement "when" body=AbortBody;
3
4  AbortBody:
5      AbortInstance | AbortCase;
6
7  AbortInstance:
8      delay=DelayExpr ("do" statement=Statement "end" "abort"?)?;
9
10 AbortCase:
11     cases+=AbortCaseSingle (cases+=AbortCaseSingle)* "end" "abort"?;
12
13 AbortCaseSingle:
14     "case" delay=DelayExpr ("do" statement=Statement)?;
```

3.3.3 Await

Das *Await-Statement* (Auflistung 3.16) wartet darauf, dass eine bestimmte Bedingung erfüllt wird. Wird diese Bedingung erfüllt, terminiert das *Statement* oder es wird, sofern vorhanden, der zur erfüllten Bedingung gehörende do-Teil ausgeführt.

Auflistung 3.16: Await in Xtext

```
1  Await:
2      "await" body=AwaitBody;
3
4  AwaitBody:
5      AwaitInstance | AwaitCase;
6
7  AwaitInstance:
8      delay=DelayExpr ("do" statement=Statement "end" "await"?)?;
```

```
9
10 AwaitCase:
11     cases+=AbortCaseSingle (cases+=AbortCaseSingle)* "end" "await"?;
```

3.3.4 Do

Das *Do-Statement* (Auflistung 3.17) ist inzwischen veraltet und sollte nicht länger genutzt werden. Es lässt sich allerdings durch andere *Statements* ausdrücken. Dabei gibt es zwei unterschiedliche *Do-Statements*, nämlich *Do-Upto* und *Do-Watching*.
Der äquivalente Ausdruck zu „do p upto d“ ist „abort p ; halt when d“.
Bei *Do-Watching* handelt es sich um eine veraltete Form des *Abort-Statement*. Dabei ist „do p watching d timeout q end“ äquivalent zu „abort p when d do q end“. Die kürzere Version von *Do-Watching* ohne den timeout-Teil entspricht der kürzeren Version von *Abort* ohne den do-Teil.

Auflistung 3.17: Do in Xtext

```
1  Do:
2      "do" statement=Statement (endUp=DoUpto | endWatch=DoWatching);
3
4  DoUpto:
5      "upto" expr=DelayExpr;
6
7  DoWatching:
8      "watching" delay=DelayExpr (end=DoWatchingEnd)?;
9
10 DoWatchingEnd:
11     "timeout" statement=Statement "end" "timeout"?;
```

3.3.5 Emit

Emit (Auflistung 3.18) setzt das angegebene Signal auf *present* in der entsprechenden Instanz, in der es aufgerufen wurde. *Emit* ist ein instantanes *Statement*, d.h. es terminiert in derselben Instanz, in der es aufgerufen wurde.

Auflistung 3.18: Emit in Xtext

```
1  Emit:
2      "emit" signal=[Signal|EsterelID] ("(" expr=DataExpr")")?;
```

3.3.6 Every-do

Every-Do (Auflistung 3.19) ist dem *Loop-Each* sehr ähnlich. Jedes Mal, wenn die Bedingung wahr wird, wird die Ausführung der Schleife neu gestartet. Anders als

beim *Loop-Each* wird die erste Ausführung allerdings nicht sofort gestartet, sondern erst, nachdem die Bedingung das erste Mal erfüllt wurde.

Auflistung 3.19: Every-do in Xtext

```
1 EveryDo:
2     "every" delay=DelayExpr "do" statement=Statement "end" "every"?;
```

3.3.7 Halt

Durch *Halt* (Auflistung 3.20) wartet das Programm auf unbestimmte Zeit, es sei denn, der Zustand wird durch *preemption* abgebrochen. Der Ausdruck „halt“ ist äquivalent zu „loop pause end“.

Auflistung 3.20: Halt in Xtext

```
1 Halt:
2     "halt";
```

3.3.8 If

Das *If* (Auflistung 3.21) prüft, ob eine Bedingung erfüllt ist. Trifft diese zu, wird der then-Teil ausgeführt, andernfalls der else-Teil. Es kann auch der then- oder else-Teil fehlen, was äquivalent zu einem nothing an der entsprechenden Stelle ist. Weiterhin gibt es noch den optionalen elsif-Teil, der das Prüfen auf weiterer Bedingungen ermöglicht, sollten die vorherigen nicht zutreffen.

Auflistung 3.21: If in Xtext

```
1  IfTest:
2      "if" expr=DataExpr (thenPart=ThenPart)?
3      (elseIf=ElsIfPart)? (elsePart=ElsePart)?
4      "end" "if"?;
5
6  ElsIfPart:
7      elsif+=ElsIf (elsif+=ElsIf)*;
8
9  ElsIf:
10     "elsif" expr=DataExpr (thenPart=ThenPart)?;
11
12 ThenPart:
13     "then" statement=Statement;
14
15 ElsePart:
16     "else" statement=Statement;
```

3.3.9 Loop

Es gibt zwei Arten von *Loops* in Esterel (Auflistung 3.22).
Das einfache *Loop* wiederholt unendlich oft den *Loop-Body*. Der *Loop-Body* darf nicht instantan sein.
Das *Loop-Each* führt beim start den *Loop-Body* aus. Jedes Mal, wenn die Bedingung nach dem each zutrifft, wird die aktuelle Ausführung des *Loop-Body* abgebrochen und der *Loop-Body* neu gestartet.

Auflistung 3.22: Loop in Xtext

```
1  Loop:
2      "loop" body=LoopBody (EndLoop | end=LoopEach);
3
4  EndLoop:
5      "end" "loop"?;
6
7  LoopEach:
8      "each" LoopDelay;
9
10 LoopDelay:
11     delay=DelayExpr;
12
13 LoopBody:
14     statement=Statement;
```

3.3.10 Nothing

Bei *Nothing* (Auflistung 3.23) wird die Kontrolle innerhalb eine *Sequence* sofort weitergegeben. *Nothing* verbraucht keine Zeit.

Auflistung 3.23: Nothing in Xtext

```
1  Nothing:
2      "nothing";
```

3.3.11 Pause

Pause (Auflistung 3.24) gibt die Kontrolle innerhalb einer *Sequence* nicht sofort weiter, sondern erst in der nächsten Instanz nach dem Aufruf.

Auflistung 3.24: Pause in Xtext

```
1  Pause:
2      "pause";
```

3.3.12 Present

Present (Auflistung 3.25) ist dem *If* ähnlich. Es wird auf eine Bedingung geprüft und entsprechend der then- oder else-Teil ausgeführt. Es kann einer der beiden Teile fehlen, was einem nothing an der entsprechenden Stelle entspricht. Die Abfrage auf mehrere Bedingungen nacheinander ist durch case-Ausdrücke möglich.

Auflistung 3.25: Present in Xtext

```
1  Present:
2      "present" body=PresentBody
3      (elsePart=ElsePart)? "end" "present"?;
4
5  PresentBody:
6      PresentEventBody
7      | PresentCaseList;
8
9  PresentEventBody:
10     event=PresentEvent (thenPart=ThenPart)?;
11
12 PresentCaseList:
13     case+=PresentCase (case+=PresentCase)*;
14
15 PresentCase:
16     "case" event=PresentEvent ("do" statement=Statement)?;
17
18 PresentEvent:
19     expression=SigExpr
20     | "[" expression=SigExpr "]";
```

3.3.13 Repeat

Repeat (Auflistung 3.26) wiederholt den *Statement*-Teil eine bestimmte Anzahl. Wie oft die Wiederholung stattfinden soll, wird durch den Ausdruck in dateExpr festgelegt. Durch Voranstellen des Wortes positive erzwingt man, dass der statement-Teil mindestens einmal ausgeführt wird, selbst wenn dataExpr null oder negativ ist.

Auflistung 3.26: Repeat in Xtext

```
1  Repeat:
2      (positive?="positive")? "repeat" dataExpr=DataExpr "times" statement=Statement "end
           " "repeat"?;
```

3.3.14 Run

Run (Auflistung 3.27) erlaubt es innerhalb eines *Module* ein anderes *Module* zu instanziieren. Dabei wird das *Run* durch den *Module-Body* des aufgerufenen *Module*

ersetzt. Es ist möglich, eine Liste an Umbenennungen mit anzugeben. Im Code, der an der Stelle von *Run* eingesetzt wird, werden die entsprechenden Umbenenungen vollzogen.
Der Befehl copymodule ist veraltet und sollte durch run ersetzt werden. Das Verhalten beider Befehle ist äquivalent.

Auflistung 3.27: Run in Xtext

```
1  Run:
2      "run" module=ModuleRenaming ("[" list=RenamingList"]")?
3      | "copymodule" module=ModuleRenaming ("[" list=RenamingList"]")?;
4
5  ModuleRenaming:
6      module=[Module|EsterelID] (renamed?="/" newName=EsterelID)?;
7
8  RenamingList:
9      list+=Renaming (";" list+=Renaming)*;
10
11 Renaming:
12     "type" renaming+=TypeRenaming ("," renaming+=TypeRenaming)*
13     | "constant" renaming+=ConstantRenaming ("," renaming+=ConstantRenaming)*
14     | "function" renaming+=FunctionRenaming ("," renaming+=FunctionRenaming)*
15     | "signal" renaming+=SignalRenaming ("," renaming+=SignalRenaming)*;
16
17 TypeRenaming:
18     newName=[Type|EsterelID] "/" oldName=[Type|EsterelID];
19
20 ConstantRenaming:
21     newName=[Constant|EsterelID] "/" oldName=[Constant|EsterelID];
22
23 FunctionRenaming:
24     newName=[Function|EsterelID] "/" oldName=[Function|EsterelID];
25
26 ProcedureRenaming:
27     newName=[Procedure|EsterelID] "/" oldName=[Procedure|EsterelID];
28
29 SignalRenaming:
30     newName=[Signal|EsterelID] "/" oldName=[Signal|EsterelID];
```

3.3.15 Local signal

Mit *Local Signal* (Auflistung 3.28) ist es möglich ein oder mehrere Signale zu deklarieren, die nur lokal im statement-Teil gültig sind.

Auflistung 3.28: Local signal in Xtext

```
1  LocalSignalDecl:
```

```
2     "signal" signalList=LocalSignalList "in" statement=Statement "end" "signal"?;
3
4  LocalSignalList:
5     {LocalSignal} signal+=Signal
6     ("," signal+=Signal)*;
```

3.3.16 Suspend

Suspend (Auflistung 3.29) pausiert die Ausführung des statement-Teils in jeder Instanz, in der die delayExpr zutrifft. Der Fortschritt in der Ausführung von statement wird dabei nicht gelöscht. Die Ausführung wird an der Stelle, an der sie unterbrochen wurde, in der nächsten Instanz, in der die Bedingung nicht zutrifft, fortgesetzt.

Auflistung 3.29: Suspend in Xtext

```
1  Suspend:
2     "suspend" statement=Statement "when" delayExpr=DelayExpr;
```

3.3.17 Sustain

Sustain (Auflistung 3.30) emittiert in jeder Instanz das entsprechende Signal. Die Ausführung kann nur durch *preemption* abgebrochen werden, wie z.B. durch ein umschließendes *Suspend*.

Auflistung 3.30: Sustain in Xtext

```
1 Sustain:
2     "sustain" signal=[Signal|EsterelID] ("(" dataExpr=DataExpr ")")?;
```

3.3.18 Trap

Mit Hilfe einer *Trap* (Auflistung 3.31) kann ein Codeblock an einer bestimmten Stelle verlassen werden. Dies geschieht mittels exit.

Auflistung 3.31: Trap in Xtext

```
1  Trap:
2      "trap" trapDeclList=TrapDeclList "in" statement=Statement (trapHandlerList=
           TrapHandlerList)? "end" "trap"?;
3
4  TrapDeclList:
5      trapDecl+=TrapDecl ("," trapDecl+=TrapDecl)*;
6
7  TrapDecl:
8      name=EsterelID (channelDesc=ChannelDescription)?;
9
10 TrapHandlerList:
11     trapHandler+=TrapHandler (trapHandler+=TrapHandler)*;
12
13 TrapHandler:
14     "handle" trapExpr=TrapExpr "do" statement=Statement;
15
16 Exit:
17     "exit" trap=[TrapDecl|EsterelID] ("(" dataExpr=DataExpr ")")?;
```

3.3.19 Local variable

Mit Hilfe von *Variable* (Auflistung 3.32) ist es möglich eine lokale Variable zu deklarieren. Diese ist nur im statement-Teil der Deklaration sichtbar und veränderbar.

Auflistung 3.32: Local Variable in Xtext

```
1  Variable:
2      "var" varDecl=VariableDecl "in" statement=Statement "end" "var"?;
3
4  VariableDecl:
5      varList=VariableList ":" (type=EsterelID | type=BaseType)
6      ({VariableDecl.left=current}"," varList=VariableList ":" (type=EsterelID | type=BaseType
           ))*;
7
8  VariableList:
9      variable=EsterelID (":=" expression=DataExpr)?
10     ({VariableList.left=current} "," variable=EsterelID (":=" expression=DataExpr)?)*;
```

3.3.20 Assignment

Unter *Assignment* (Auflistung 3.33) versteht man in Esterel die Belegung einer Variablen mit einem neuen Wert. Dabei kann es sich um einen einzelnen Wert oder einen Ausdruck, der erst ausgewertet werden muss, handeln.

Auflistung 3.33: Assignment in Xtext

```
1 Assignment:
2     variable=EsterelID ":=" expr=DataExpr;
```

3.3.21 Weak abort

Weak Abort ist dem normalen *Abort* in der Funktion sehr ähnlich. Wird die Abbruchbedingung erfüllt, so wird der statement-Teil nicht weiter ausgeführt. Anders als *Abort* erlaubt *Weak Abort* allerdings, dass in der Instanz, in der die Bedingung erfüllt wird, der statement-Teil noch ausgeführt wird.

Auflistung 3.34: Weak abort in Xtext

```
1  WeakAbort:
2      "weak" "abort" statement=Statement "when" weakAbortBody=WeakAbortBody;
3
4  WeakAbortBody:
5      WeakAbortInstance | WeakAbortCase;
6
7  WeakAbortInstance:
8      delay=DelayExpr ("do" statement=Statement "end" ("weak"? "abort")?)?;
9
10 WeakAbortCase:
11     cases+=AbortCaseSingle (cases+=AbortCaseSingle)* "end" ("weak"? "abort")?;
```

Das aus diesen Deklarationen resultierende Modell für Esterel ist in Abbildung 3.3 zu sehen. Diese Abbildung soll lediglich einen Eindruck über den Umfang des Metamodells vermitteln.

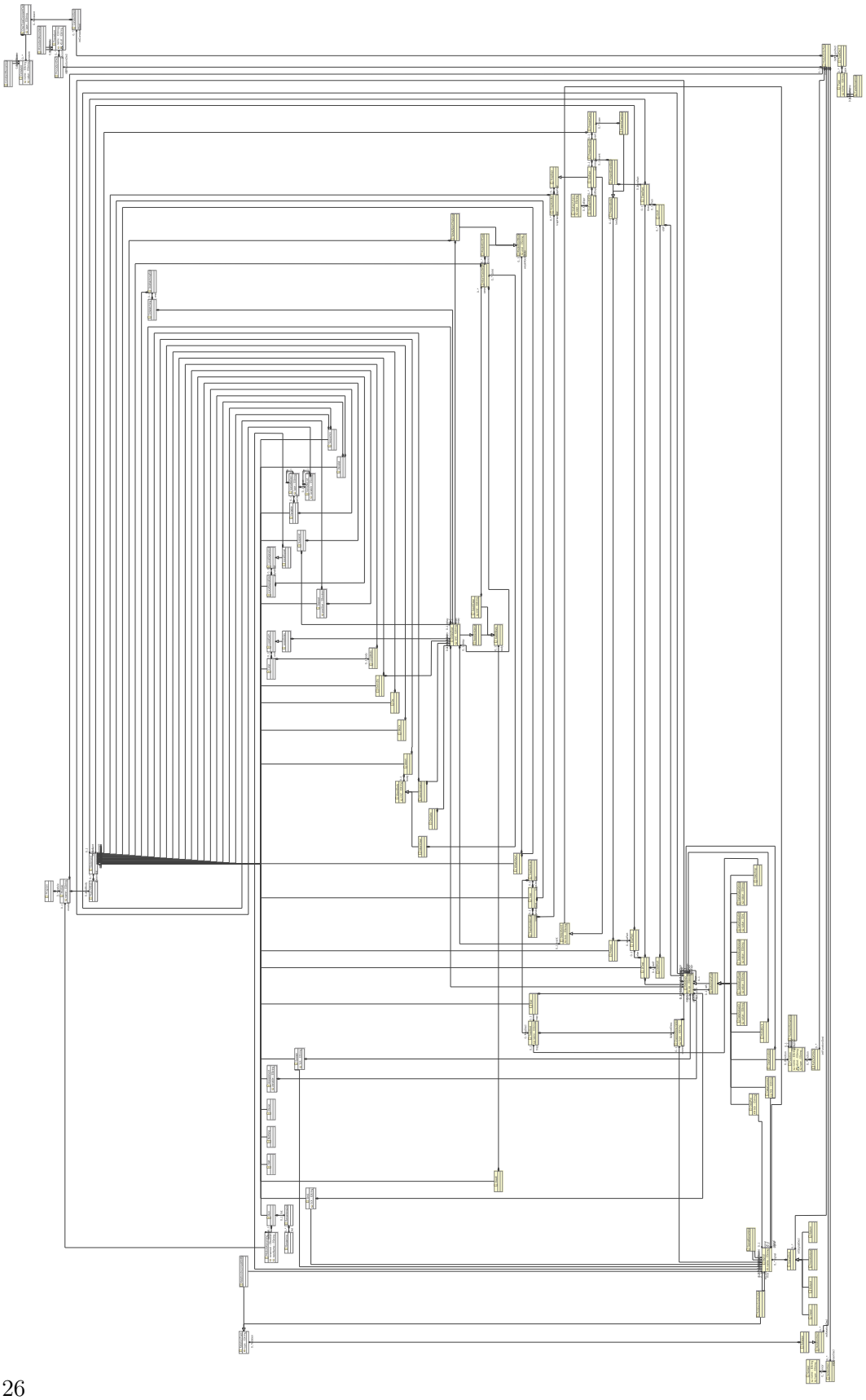


Abbildung 3.3: Eindruck über den Umfang des Modells von Esterel

4 Transformation von Esterel nach SyncCharts

Für die Transformation von Esterel nach SyncCharts wird die Sprache Xtend verwendet (Unterabschnitt 1.3.4). Als Ausgangspunkt der Transformationen wird das durch Xtext erzeugte Modell von Esterel genutzt (Kapitel 3). Das Zielmodell der Transformationen ist das in ThinKCharts genutzte Modell der SyncCharts (Abbildung 2.2).
Es werden die gleichen Transformationstheorien verwendet, die in KIEL genutzt und bewiesen wurden. Im Folgenden sollen diese noch einmal aufgeführt werden. Für Beweise verweise ich auf Prochnow et al. [7] und die Diplomarbeit von Lars Kühl [5].

4.1 Module

Das *Module* enthält sowohl Deklarationen, als auch den gesammten Programmablauf. Bei der Transformation in ein SyncChart werden die Signaldeklarationen in das SyncChart übernommen. Weiterhin wird ein Makrozustand erzeugt, der gleichzeitig der initiale Zustand des SyncChart ist und den Programmcode enthält.
Auflistung 4.1 zeigt den verwendeten Transformationscode beispielhaft an der Transformation von *Module*. Abbildung 4.1 zeigt die Transformationsregel.

Auflistung 4.1: Transformationscode von module

```
1  //@r: rootregion of the syncChart
2  //@m: Esterel−Module, source of transformation
3  transModule(Region r, Module m):
4      let s = new State:
5      let ri = new Region:
6      let i = new State:
7      //module name −> SnycChart name
8      s.setLabel(m.name)−>
9      //transform interface if there is one
10     (if m.modInt == null
11     then s
12     else s.setSignals(transIntSignalDecl(m.modInt.intSignalDecl))) −>
13     //configure inner macrostate and start transformation of module body
14     i.setIsInitial(true) −>
15     i.setIsFinal(true) −>
16     transStatement(i,m.modBody.statement.first()) −>
17     //link regions and states
```

```
18    ri.innerStates.add(i) ->
19    s.regions.add(ri) ->
20    r.innerStates.add(s);
```

module *<Identifier>*:
<ModuleInterface>
p
end module

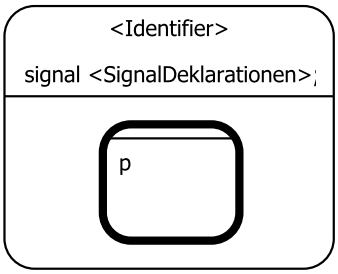


Abbildung 4.1: Transformation von Module

4.2 Sequence

Die einzelnen *Statements* innerhalb von *Sequence* werden in der angegebenen Reihenfolge nacheinander ausgeführt. In SyncCharts wird dies durch eine Reihe von Makrozuständen repräsentiert, die durch *normal terminations* miteinander verbunden sind. Abbildung 4.2 zeigt die Transformationsregel.

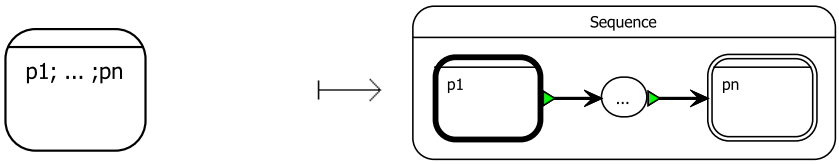


Abbildung 4.2: Transformation von Sequence

4.3 Parallel

Die einzelnen *Statements* innerhalb von *Parallel* werden gleichzeitig ausgeführt. In SyncCharts wird dies erreicht, indem diese *Statements* in parallelen Makrozuständen ausgeführt werden. Die Transformationsregel ist in Abbildung 4.3 zu sehen.

4.4 Abort

Durch *Abort* wird die Ausführung des umschlossenen Programms unterbrochen, sobald eine der Abbruchbedingungen erfüllt ist. Dabei kann „abort p when d“ in „abort

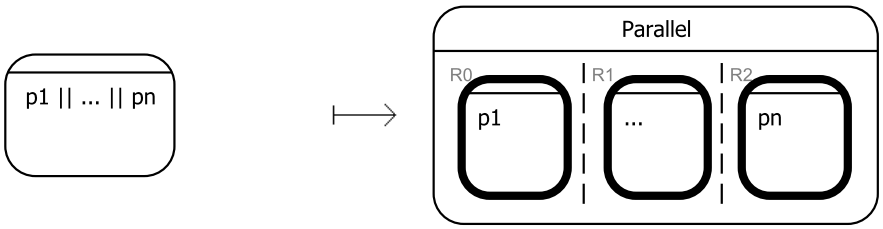


Abbildung 4.3: Transformation von Parallel

p when case d do nothing“ übersetzt werden. Im SyncChart wird der Abbruch durch *strong abortions* realisiert. Jede dieser Transitionen entspricht einem case-Fall. Die Abbruchbedingung wird zum Transitionslabel. Ziel der Transition ist ein Makrozustand, in dem der do-Teil des entsprechenden case-Falls ausgeführt wird. Die Transitionen bekommen Prioritäten anhand der textuellen Reihenfolge. In Abbildung 4.4 ist die Transformationsregel zu sehen.

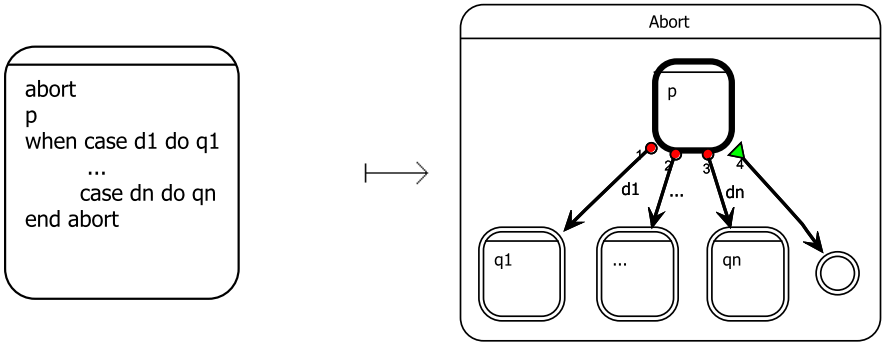


Abbildung 4.4: Transformation von Abort

4.5 Await

Das *Await-Statement* hält die Ausführung eines Threads an, bis eine bestimmte Bedingung eintritt. Dies wird im SyncChart durch einen einfachen Zustand ausgedrückt, der nur über eine *strong abortion* verlassen werden kann. Jeder case-Fall wird durch eine Transition ausgedrückt, deren Priorität anhand der textuellen Reihenfolge bestimmt wird. Das Transitionslabel entspricht dabei der Bedingung, auf die der entsprechende case-Fall wartet. Abbildung 4.5 zeigt die Transformationsregel.

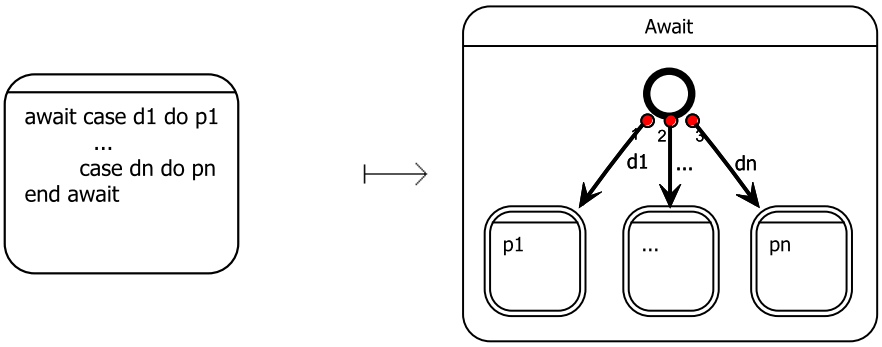


Abbildung 4.5: Transformation von Await

4.6 Do-Upto

Do-Upto führt das *Body-Statement* aus und wartet anschließend. Diese Ausführung kann jederzeit unterbrochen werden, sobald die Abbruchbedingung eintritt. Im SyncChart entspricht dies einem Makrozustand, der das *Statement* enthält und nur über eine *strong abortion* mit der Abbruchbedingung als Transitionslabel verlassen werden kann. In Abbildung 4.6 ist die Transformationsregel zu sehen.

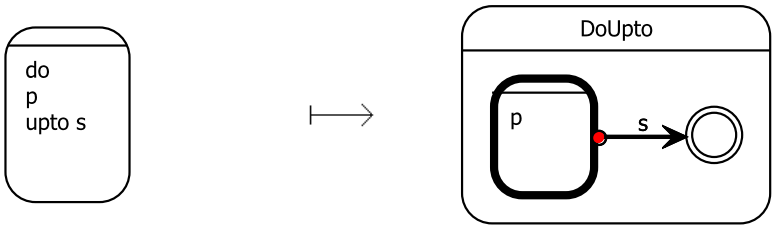


Abbildung 4.6: Transformation von Do-Upto

4.7 Do-Watching

Do-Watching führt das *Body-Statement* aus und terminiert, wenn auch das *Body-Statement* terminiert. Die Ausführung kann jederzeit abgebrochen werden, sobald die Abbruchbedingung eintritt. *Do-Watching* ist semantisch äquivalent zu *Abort* mit nur einer Abbruchbedingung. Die Transformation ins SyncChart findet daher analog statt. Abbildung 4.7 zeigt die Transformationsregel.

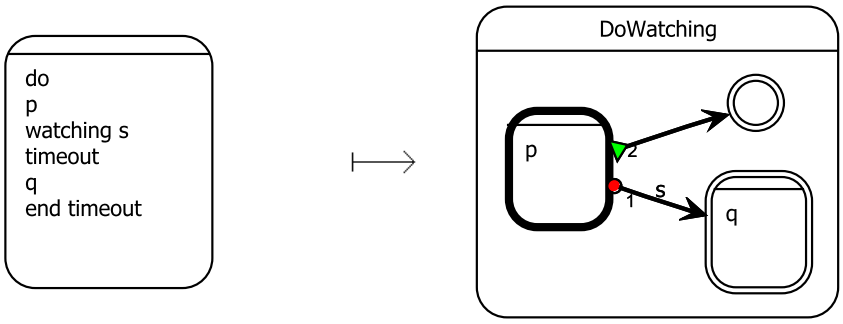


Abbildung 4.7: Transformation von Do-Watching

4.8 Emit

Durch *emit* wird ein Signal ausgesendet, das optional einen Wert enthält. *Emit* ist instantan. Im SyncChart wird dies durch einen Makrozustand realisiert, der beim Betreten das Signal aussendet und sofort durch die Transition in einen finalen Zustand terminiert. Die Transformationsregel ist in Abbildung 4.8 zu sehen.

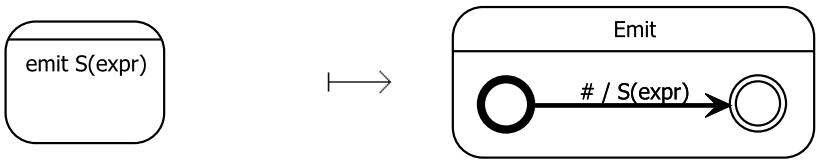


Abbildung 4.8: Transformation von Emit

4.9 Every

Every wartet zunächst auf das Eintreffen der Bedingung. Ist diese erfüllt, so wird das *Body-Statement* ausgeführt. Terminiert das *Statement*, so wird auf das Eintreffen der Bedingung gewartet. Trifft die Bedingung ein, so wird die Ausführung des *Statements* sofort abgebrochen und neu gestartet. Im SyncChart bedeutet dies, dass der initiale Zustand nur durch eine *strong abortion* verlassen werden kann, wenn die Bedingung zutrifft. Dabei wird der Zustand betreten, der den *Statement-Body* ausführt. Dieser wird bei Eintreffen der Bedingung durch eine *strong abortion* verlassen und neu betreten. Die Transformationsregel ist in Abbildung 4.9 zu sehen.

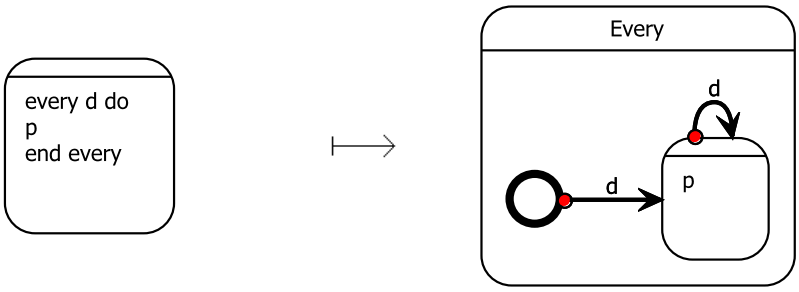


Abbildung 4.9: Transformation von Every

4.10 Halt

Halt terminiert nicht, sondern wartet eine unbegrenzte Zeit. Im SyncChart bedeutet dies, dass ein einfacher, nicht finaler Zustand ohne ausgehende Transitionen betreten wird. Die Transformationsregel ist in Abbildung 4.10 zu sehen.

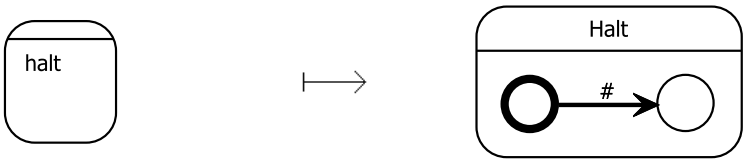


Abbildung 4.10: Transformation von Halt

4.11 If

Das *If-Statement* wertet die Bedingungen nacheinander aus. Das *Statement*, das zur textuell ersten Bedingung gehört, die erfüllt wird, wird ausgeführt. Im SyncChart bedeutet dies, dass jede Möglichkeit als Transition ausgehend vom initialen Zustand realisiert wird. Die Bedingung bildet das Transitionslabel. Ziel der Transition ist ein Zustand, in dem das zur Bedingung gehörende *Statement* ausgeführt wird. Die Transitionen bekommen eine Priorität gemäß der textuellen Reihenfolge. Abbildung 4.11 zeigt die Transformationsregel.

4.12 Loop

Loop wiederholt undendlich oft das angegebene *Statement*. Im SyncChart entspricht dies einem Zustand, in dem das *Statement* ausgeführt wird. Dieser wird über eine *normal termination* verlassen und sofort wieder betreten. Die Transformationsregel ist in Abbildung 4.12 abgebildet.

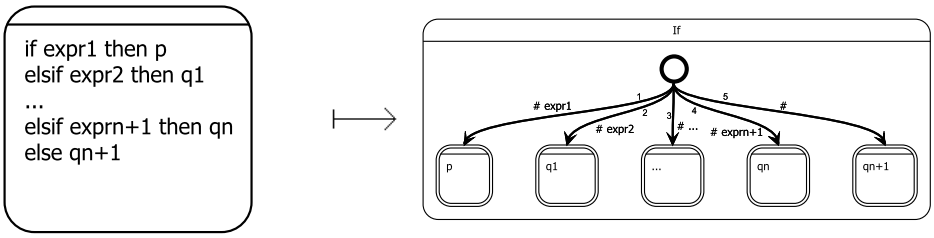


Abbildung 4.11: Transformation von If

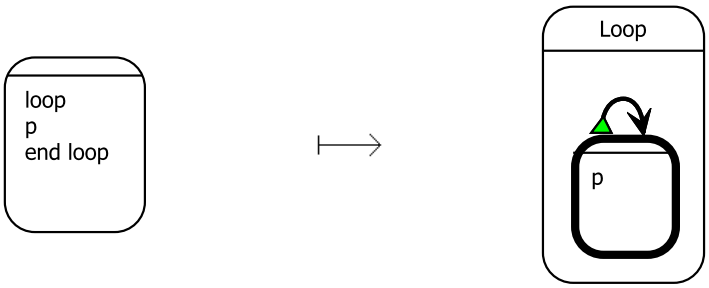


Abbildung 4.12: Transformation von Loop

4.13 Loop-Each

Loop-Each führt das *Body-Statement* aus. Terminiert dieses, so pausiert das *Statement* bis zum Eintreffen der Bedingung. Die Bedingung kann jederzeit eintreffen. Wenn sie eintrifft, so wird die Ausführung des *Statements* sofort abgebrochen und neu gestartet. Im SyncChart entspricht das einem Zustand, der das *Statement* enthält. Dieser Zustand wird durch eine *strong abortion* verlassen und sofort betreten, wenn die Bedingung eintrifft. Die Transformationsregel ist in Abbildung 4.13 zu sehen.

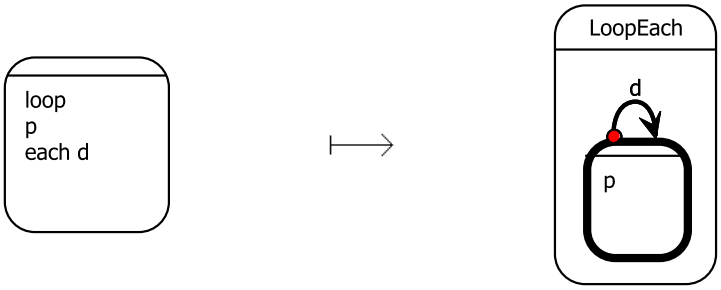


Abbildung 4.13: Transformation von Loop-Each

4.14 Nothing

Nothing terminiert sofort. Im SyncChart entspricht dies einem sofortigen Übergang in einen finalen Zustand nach Betreten des Makrozustands. Abbildung 4.14 zeigt die Transformationsregel.
Der ThinKCharts-Editor erlaubt, dass ein Zustand sowohl initial, als auch final ist. Es wäre daher möglich, die Transition in den finalen Zustand zu entfernen und den initialen Zustand gleichzeitig final zu machen. Jedoch bietet der ThinKCharts-Editor derzeit keine besondere Darstellung für initiale Zustände, die auch final sind, sondern zeigt diese lediglich als initial an. Durch das Weglassen der Transition in *Nothing* und analog dazu auch in *Halt*, wären *Nothing* und *Halt* im SyncChart nicht mehr unterscheidbar. Der besseren Lesbarkeit wegen wird daher an der Stelle auf eine Vereinfachung verzichtet.

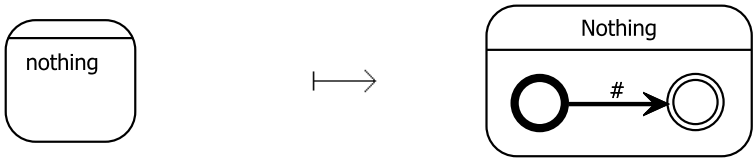


Abbildung 4.14: Transformation von Nothing

4.15 Pause

Pause unterbricht die Ausführung in der aktuellen Zeitinstanz und terminiert in der darauffolgenden. Im SyncChart wird das durch eine Transition vom initialen in den finalen Zustand realisiert, die erst beim nächsten tick genommen wird. Die Transformationsregel ist in Abbildung 4.15 abgebildet.

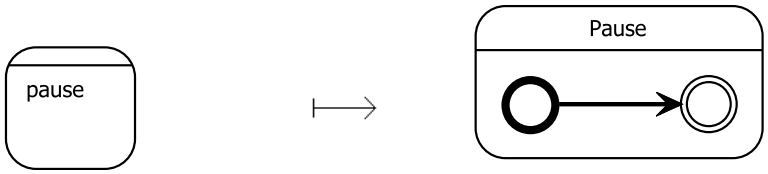


Abbildung 4.15: Transformation von Pause

4.16 Present

Es werden bei *Present* die Bedingungen abgeprüft und die textuell erste, die zutrifft, ausgeführt. Damit ist *Present* in seinem Verhalten dem *If* ähnlich. Aus diesem Grund findet die Transformation von *Present* analog zur *If*-Transformation statt. Die Transformationsregel ist in Abbildung 4.16 dargestellt.

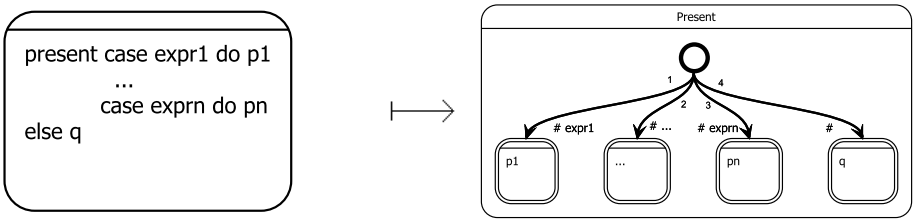


Abbildung 4.16: Transformation von Present

4.17 Local Signal

Signale, die durch *Local Signal* deklariert sind, stehen dem *Body-Statement* zur Verfügung. Im SyncChart wird bei der Transformation der Makrozustand um die deklarierten Signale erweitert. Abbildung 4.17 zeigt die Transformation.

4.18 Suspend

Die Ausführung des *Statements* innerhalb von *Suspend* wird angehalten, wann immer die Bedingung erfüllt ist. Trifft die Bedingung nicht zu, so wird das *Statement* weiter ausgeführt. Dies entspricht im SyncChart einem Zustand, der die Bedingung als *suspension trigger* nutzt. In Abbildung 4.18 ist die Transformation abgebildet.

4.19 Sustain

Sustain sendet in jeder Zeitinstanz ein Signal aus. Das entsprechende SyncChart muss beim Betreten des Makrozustands das Signal aussenden. Der initiale Zustand wird sofort durch eine Transition verlassen und sendet dabei das Signal aus. Der Zielzustand hat eine Transition auf den initialen Zustand. Dadurch werden der initiale Zustand und die aussendende Transition in jeder Zeitinstanz ein mal aktiv. Abbildung 4.19 zeigt die Transformationsregel.

4.20 Trap

Durch *Trap* wird die Ausführung des *Statements* unterbrochen, sofern eine *exception* auftritt. Jede *exception* hat optional einen *handler*, der ausgeführt wird, sofern

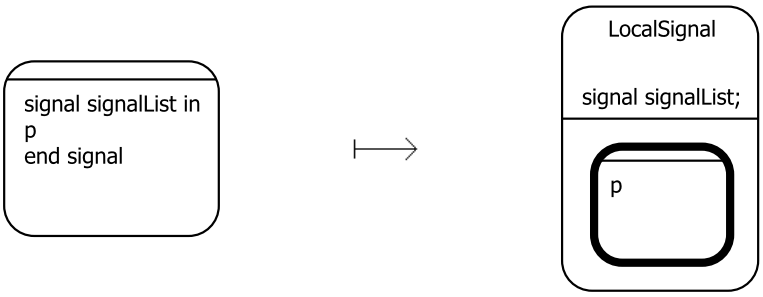


Abbildung 4.17: Transformation von Local Signal

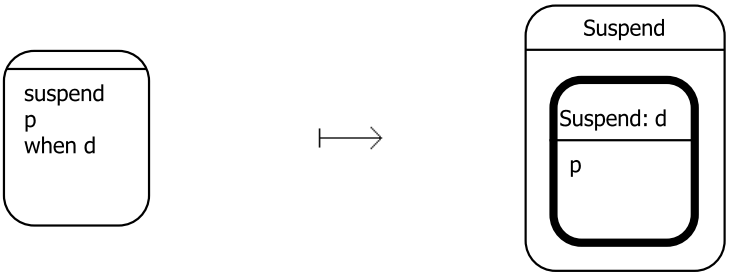


Abbildung 4.18: Transformation von Suspend

die entsprechende *exception* aufgetreten ist. Sind mehrere *exceptions* aufgetreten, so werden die entsprechenden *handler* parallel ausgeführt. Bei den Ausführungsbedingungen der *handler* handelt es sich um boolsche Ausdrücke auf den *Trap*-Signalen. Für die Realisierung im SyncChart müssen zusätzliche Signale eingeführt werden. Eine *Trap* hat die lokalen Signale traphalt und t1 bis tn. Dabei wird traphalt genutzt, um die Ausführung der *Trap* durch eine *weak abortion* anzuhalten, sofern eine hierarchisch höhere *Trap* ausgelöst wurde. Die Signale t1 bis tn entsprechen den *Trap*-Deklarationen und werden genutzt, um das Auslösen der entsprechenden *Trap* anzuzeigen. Die *handler* werden parallel in einem finalen Makrozustand ausgeführt.

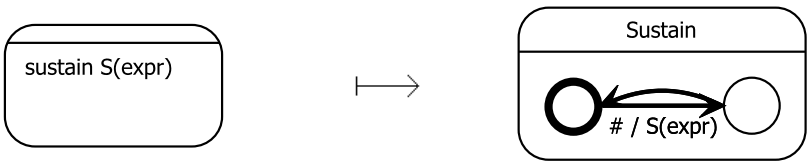


Abbildung 4.19: Transformation von Sustain

Das Programm, welches ein *handler* ausführt, wenn er ausgelöst wird, wird in einem finalen Makrozustand ausgeführt. Wenn eine *exception* ausgelöst wird, hat dies zur Folge, dass die *Trap* nur terminiert, wenn die ausgelösten *handler* terminieren. Insbesondere bedeutet dies, dass der Thread nach der *Trap* nicht weiter ausgeführt wird, sollte einer der ausgelösten *handler* nicht terminieren, was z.B. durch ein *Halt* eintreten könnte. Abbildung 4.20 zeigt die Transformationsregel.

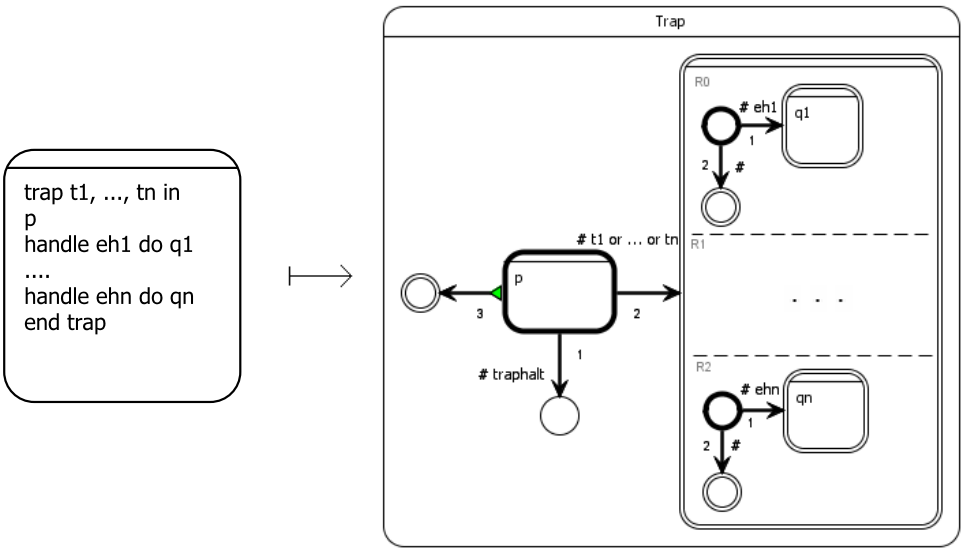


Abbildung 4.20: Transformation von Trap

4.21 Exit

Durch *Exit* wird eine *Trap* ausgelöst. Es wird ähnlich dem *Emit* das entsprechende Signal ausgesendet. *Exit* terminiert danach nicht, weshalb der enstprechende Zustand im SyncChart kein finaler Zustand ist. Es werden außerdem die traphalt-Signale jener *Traps* ausgendet, die von der ausgelösten *Trap* umschlossen werden. Die Transformationsregel ist in Abbildung 4.21 zu sehen.

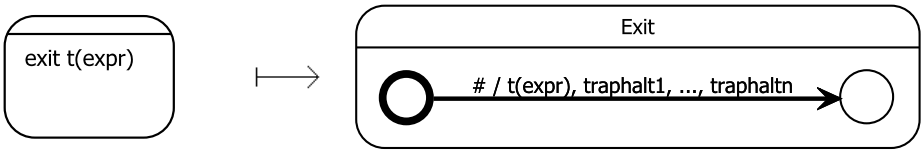


Abbildung 4.21: Transformation von Exit

4.22 Local Variable

Durch *Local Variable* werden Variablen deklariert, die dem *Body-Statement* zur Verfügung stehen. Lokale Variabeln werden im Interface eines Zustands deklariert. Das *Body-Statement* wird innerhalb dieses Zustands ausgeführt. In Abbildung 4.22 ist die Transformationsregel abgebildet.

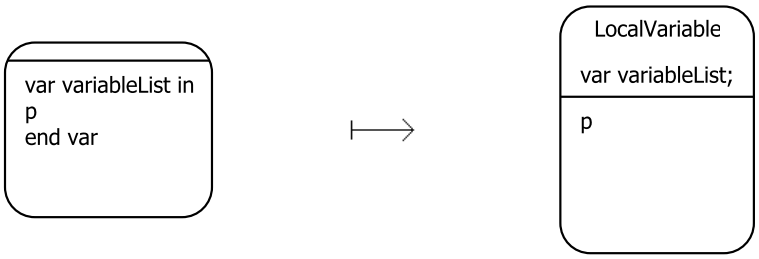


Abbildung 4.22: Transformation von Local Variable

4.23 Assignment

Durch *Assignment* wird einer lokalen Variablen ein Wert zugewiesen. Im SyncChart wird beim Betreten eines Makrozustands die Zuweisung durchgeführt und der Makrozustand sofort wieder verlassen. In Abbildung 4.23 ist die Transformationsregel zu sehen.

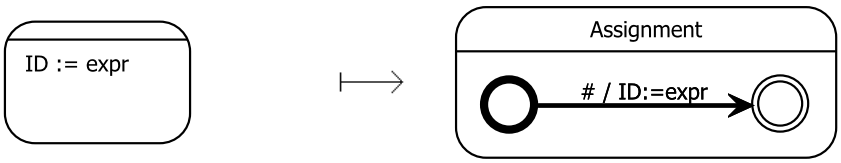


Abbildung 4.23: Transformation von Assignment

4.24 Weak abort

Analog zum *Abort* bricht *Weak Abort* die Ausführung des *Statements* ab, wenn die Bedingung erfüllt ist. Bei *Weak Abort* geschieht dies allerdings erst am Ende der Zeitinstanz, in der die Bedingung erfüllt wurde. Entsprechend wird das SyncChart analog zum *Abort* konstruiert, wobei die *strong abortions* durch *weak abortions* ersetzt werden. Weiterhin muss sichergestellt werden, dass *Weak Abort* die Ausführung abbricht und nicht die eigenen Abbruchbedingungen noch ausführt, falls eine übergeordnete *Trap* ausgelöst wurde. Dies wird erreicht, indem eine *weak abortion* zu

einem nicht finalen Zustand ohne ausgehende Transitionen hinzugefügt wird, sofern *Weak Abort* durch eine *Trap* umschlossen ist. Die Transition wird durch immediate TE ausgelöst, wobei TE die oder-Verknüpfung der hierarchisch übergeordneten trap-halt-Signale und der exceptionEvent-Signale der hierarchisch nächst höheren *Trap* ist. In Abbildung 4.24 ist die Transformationsregel zu sehen.
Inwiefern übergeordnete *Traps* eine Rolle für das *Weak Abort* spielen, will ich anhand eines Beispiels verdeutlichen.
Auflistung 4.2 zeigt das Beispielprogramm, in dem ein *Weak Abort* durch eine *Trap* umschlossen wird. Das Programm enthält zwei parallele Threads, die im ersten tick beide durch ein pause angehalten werden. Im zweiten tick emitiert einer der Threads das Signal W, welches die Abbruchbedingung des *Weak Abort* auslösen würde. Das *Weak Abort* führt in diesem tick sein Programm noch aus, ehe es die Abbruchbedingung ausführen muß. Dies hat zur Folge, dass A emitiert und die *Trap* T ausgelöst wird. Da die umschließende *Trap* ausgelöst wurde, führt das *Weak Abort* den when-Teil nicht mehr aus, obwohl die Abruchbedingung erfüllt ist. Das bedeutet insbesondere, dass C nicht emitiert wird. Genau dieses Verhalten wird im SyncChart dadurch erreicht, dass die Transition mit der höchsten Priorität durch immediate TE ausgelöst wird und in einem normalen Zustand endet.

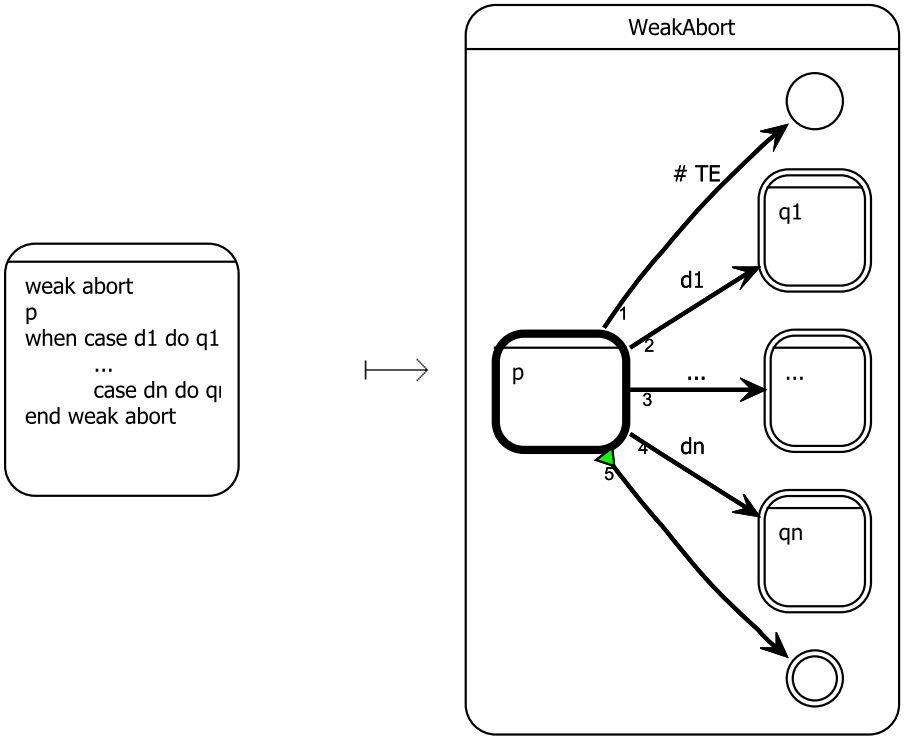


Abbildung 4.24: Transformation von Weak Abort

Auflistung 4.2: Beispiel für Weak Abort mit umschließender Trap

```
1  trap T in
2    weak abort
3     pause;
4     emit A;
5     exit  T;
6     emit B;
7    when W
8      emit C
9  end trap;
10 ||
11 pause;
12 emit W;
```

Durch die Anwendung dieser Transformationsregeln auf das Programm ABRO (Auflistung 4.3), erhält man das SyncChart in Abbildung 4.25.

Auflistung 4.3: ABRO in Esterel

```
1  module ABRO:
2  input A, B, R;
3  output O;
4  loop
5  [
6    await A
7    ||
8    await B
9  ];
10 emit O;
11 each R
12 end module
```

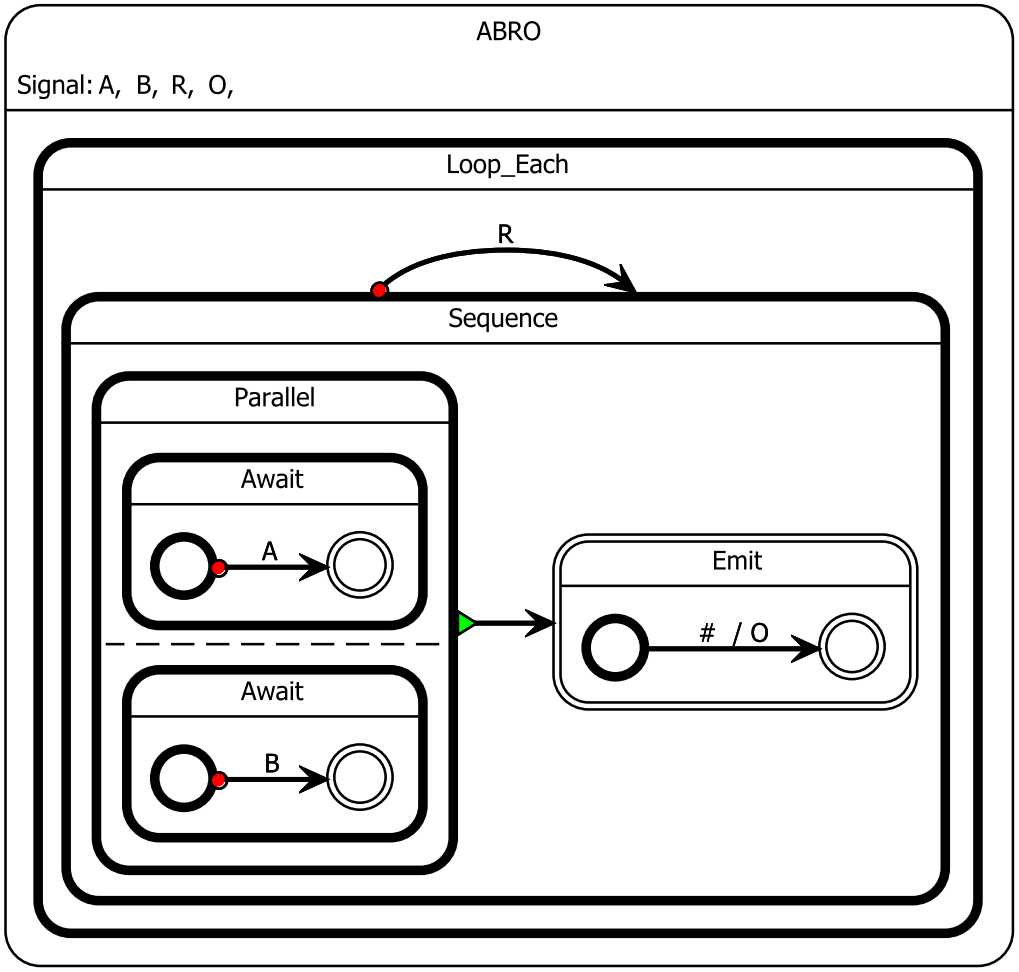


Abbildung 4.25: Transformationsergebnis von ABRO

5 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, die Vorteile von Esterel und SyncCharts bei der Modellierung von reaktiven Systemen zu vereinen.
Vorteile der textuellen Modellierung mit Esterel sind dabei vor Allem die leichte Erweiterbarkeit und das schnellere und leichtere Erstellen. Die grafische Darstellung in SyncCharts erlaubt hingegen ein leichteres Verständnis und eine bessere Fehleranalyse. Weiterhin bietet eine grafische Darstellung die bessere Grundlage für Simulationen.
Dieses Ziel wurde erreicht, indem zuerst ein textueller Editor und Parser für Esterel implementiert wurde. Für die Implementierung wurde Xtext verwendet. Xtext war für diesen Zweck besonders geeignet, da lediglich auf Basis der Grammatik alle benötigten Komponenten generiert werden konnten.
Als nächstes sollten die Transformationen von Esterel nach SyncCharts implementiert werden. Mit ThinKCharts hatte Eclipse bereits einen SyncChart-Editor, der für die Darstellung der resultierenden SyncCharts genutzt werden konnte. Daher sollte das durch ThinKCharts verwendete Metamodell als Ziel der Transformationen dienen.
Für die Transformationen wurde Xtend verwendet. Der Vorteil von Xtend war, dass die Transformationen als Regeln direkt auf den Modellen angegeben werden konnten. Dies war deutlich schneller realisiert, als den entsprechenden Java-Code zu verfassen. Sollten die Transformationen zukünftig erweitert werden sollen, so bietet Xtend auch eine leichtere Einarbeitung, als es bei einer reinen Java-Implementierung der Fall wäre.

Diese Arbeit bietet mehrere Ansatzpunkte für zukünftige Erweiterungen, die ich nun kurz erläutern will.

5.1 Optimierung von SyncCharts

Wie bereits in Kapitel 2 erwähnt, sind die aus der Transformation resultierenden SyncCharts sehr umfangreich. Sie lassen sich jedoch durch Optimierungen reduzieren, ohne dass das Programmverhalten verändert wird. Die dafür benötigten Transformationstheorien wurden bereits in KIEL genutzt [7].
Durch die Optimierung würden die SyncCharts in ihrem Umfang stark reduziert werden. Dies würde sie wiederum leichter lesbar machen und die Analyse des SyncCharts vereinfachen.

Für die Optimierungen könnte man, wie schon für die Transformationen, Xtend verwenden. Xtend hat sich als besonders einfach und schnell programmierbar bei Modell-zu-Modell Transformationen erwiesen. Die Optimierungen wären dabei Transformationen vom SyncChart-Modell ins SyncChart-Modell.

5.2 Erweiterung auf Esterel v7

Eine weitere Möglichkeit für zukünftige Erweiterungen bietet die Esterelversion. Esterel v7 unterstützt unter Anderem die Definition von eigenen Datentypen. Weiterhin stellt Esterel v7 Arrays zur Verfügung.
Derzeit sind der Parser und die Transformationen auf Esterel v5 ausgelegt. Dadurch, dass für den Esterel-Editor und Parser Xtext mit einer Grammatik verwendet wurde, wäre es lediglich nötig, die Grammatik um die neuen Funktionen zu erweitern. Dadurch würden Editor und Parser Esterel v7 erkennen.
Während die Transformationen rein technisch durch Xtend leicht erweiterbar sind, stellt die Theorie die größere Hürde dar. Es wäre zu prüfen, ob und in welchem Ausmaß die Esterel v7 Erweiterungen in SyncCharts transformierbar sind.

Literaturverzeichnis

[1] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille, France, July 1996. IEEE-SMC. `http://www.i3s.unice.fr/~andre/CAPublis/Cesa96/SyncCharts_Cesa96.pdf`.

[2] 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.

[3] 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.

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

[5] 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`.

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

[7] 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.

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

[9] Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. Visual comparison of graphical models. In *Proceedings of the Fourth IEEE International Workshop UML and AADL, held in conjunction with the 14th International International Conference on Engineering of Complex Computer Systems (ICECCS'09)*, Potsdam, Germany, 2 June 2009.

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

[11] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010.