Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Synthese von SC-Code aus SyncCharts

Ein Compiler für SyncCharts

Torsten Amende

22. Mai 2010

Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:

Dr. Claus Traulsen
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,

“

Studieren lehrt uns Regeln, das Leben die Ausnahmen.

”

Peter Niemann

Zusammenfassung

SyncCharts sind synchrone Statecharts zum Modellieren reaktiver Systeme. Sie zeichnen sich durch eine präzise und deterministische Semantik aus. Für Codeerzeugung und Simulation ist der Standard, dass SyncCharts zunächst nach Esterel übersetzt werden und ein Esterel–Compiler daraus C–Code erzeugt. Der erzeugte Code ist sehr effizient, hat jedoch zwei entscheidende Nachteile: Zum einen lässt sich der Kontrollfluss, der in SyncCharts durch Transitionen ausgedrückt werden kann, nicht direkt auf Esterel abbilden. Zum anderen ist für den generierten C–Code der Bezug zum SyncChart kaum herzustellen.

In dieser Arbeit wird ein alternativer Ansatz für die Synthese von C–Code aus SyncCharts vorgestellt. Dabei werden SyncCharts ohne Umweg direkt nach Synchronous C (SC) übersetzt. SC ist eine Erweiterung der Programmiersprache C und ergänzt sie um deterministische Nebenläufigkeit und Preemption. Der synthetisierte SC–Code bewahrt die Struktur des SyncCharts, wodurch er sich für Validierung und Zertifizierung gut eignet.

Für die Codegenerierung werden Thread–Prioritäten benötigt, die unter Berücksichtigung von Datenabhängigkeiten in optimierter Form zugewiesen werden. Dadurch wird der generierte SC–Code klein gehalten und kann sich durch eine gute Performance auszeichnen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Verwandte Arbeiten **5**

3 SyncCharts und SC **9**

3.1 Synchrone Sprachen . . . 9
3.2 SyncCharts . . . 10
3.2.1 Vom Automaten zum SyncChart . . . 10
3.2.2 SyncCharts im Detail . . . 11
3.3 Synchronous C (SC) . . . 15

4 Der Compiler **21**

4.1 Elemente in SyncCharts und deren Übersetzung . . . 21
4.1.1 Regionen . . . 22
4.1.2 Zustände . . . 22
4.1.3 Transitionen . . . 27
4.2 Prioritätenzuweisung . . . 28
4.2.1 Abhängigkeiten in SyncCharts . . . 29
4.2.2 Abhängigkeitsbäume . . . 33
4.2.3 Topologische Sortierung . . . 34
4.2.4 Zuweisung von Prioritäten . . . 37
4.2.5 Prioritäten für das Wechseln zwischen aktiven Threads . . . 38
4.3 Optimierungen . . . 47
4.3.1 Vermeidung von unerreichbaren oder redundanten Code . . . 47
4.3.2 Makros zur Minimierung der Codegröße . . . 49
4.3.3 Optimierung von Sprunganweisungen . . . 49
4.3.4 Prioritäten und PRIO–Anweisungen . . . 50
4.3.5 Sonstige Optimierungen . . . 52
4.4 Nicht unterstützte SyncCharts–Elemente . . . 53
4.4.1 OnExit actions . . . 53
4.4.2 Valued Signals . . . 54
4.4.3 Host Code . . . 54
4.4.4 Reference States . . . 54

5 Implementierung und verwendete Techniken **55**

5.1 Die Eclipse Plattform . . . 55

5.2 Das Eclipse Modeling Framework 56
5.3 Xpand, Xtend und Check 57
5.4 Sonstiges 62

6 Integration in Kiel Integrated Environment for Layout Eclipse Rich-Client (KIELER) 65
6.1 Kiel Integrated Environment for Layout Eclipse Rich-Client (KIELER) 65
6.1.1 Thin Kieler SyncCharts Editor (ThinKCharts) 66
6.1.2 Automatiches Layout 66
6.1.3 Strukturbasiertes Editieren 67
6.1.4 Der Execution Manager 67
6.2 Visualisierung 68
6.2.1 Anbindung des SC–Code an KIELER 70
6.3 Validierung 71

7 Versuchsergebnisse 73

8 Zusammenfassung und Ausblick 81

Literaturverzeichnis 83

Abbildungsverzeichnis

1.1 Reaktives System mit Eingabe– und Ausgabe–Signalen 1
1.2 Die Programmiersprache Esterel mit dem ABRO–Beispiel 2
1.3 Verschiedene Wege der Generierung von C–Code aus SyncCharts . . 3

2.1 Ausschnitt vom SyncCharts Compilers Collection (SCC)–Code für ABRO 8

3.1 ABRO, das „Hello World“ der synchronen Sprachen 11
3.2 Das Beispiel Shifter3 14
3.3 Verwendete Thread–Operatoren aus SC [41] 16
3.4 Verwendete Signal–Operatoren aus SC [41] 17
3.5 Einfaches nicht konstruktives SyncChart und der äquivalente Esterel– (b) bzw. C–Code (c) 19
3.6 Trace für die Ausführung des Codes für das nicht konstruktive SyncChart aus Abbildung 3.5 19

4.1 SyncChart mit einer Region und drei inneren Zuständen 22
4.2 Struktur des generierten Codes für Zustände 23
4.3 SyncChart mit on entry– und on inside actions 24
4.4 SyncChart mit Start– und Endzustand in einer Region 25
4.5 SyncChart mit hierarchischem inneren Zustand und zwei darin liegenden parallelen Regionen 26
4.6 Hierarchischer Zustand und allen möglichen ausgehenden Typen von Transitionen 27
4.7 SyncChart mit einer einfachen und einer komplexen Transition und ein Ausschnitt des dazu generierten SC–Codes 29
4.8 Modifiziertes ABRO mit Signalabhängigkeit 30
4.9 Einfacher Abhängigkeitsbaum und das modifizierten ABRO mit eingezeichneten Signal– und Kontrollflussabhängigkeiten 34
4.10 Pseudocode für topologische Sortierung 36
4.11 Topologisch sortierter Abhängigkeitsbaum von 4.9 36
4.12 Codeausschnitt für die Thread–Initialisierung von 4.8 38
4.13 Topologisch sortierter Abhängigkeitsbaum von 4.6 40
4.14 SyncChart mit wechselnden Signalabhängigkeiten, der dazu gehörige Abhängigkeitsbaum und ein Teil des generierten SC–Codes 41
4.15 Topologisch sortierter Abhängigkeitsbaum von 4.14 42
4.16 SyncChart 43

4.17 Einfacher und erweiterter Abhängigkeitsbaum für SyncChart 4.16 . . 44
4.18 Topologisch sortierte Abhängigkeitsbäume für 4.16 sowohl mit als auch ohne Transitionsabhängigkeiten 45
4.19 Generierter Code für das Beispiel aus 4.16 46
4.20 SyncChart für das zwei Threads generiert werden (inkl. Abhängigkeitsbaum und Codeausschnitt) 51

5.1 Eclipse Plattform im Überblick [12] 56
5.2 Hierarchie der Metamodellierung 57
5.3 Codebeispiel für einen Ausdruck in Check 58
5.4 Codebeispiel für eine Funktion in Xtend, zur Erweiterung der Funktionalität im Metamodell 59
5.5 Codebeispiel für eine Funktion in Xtend, die Berechnungen nach Java auslagert 60
5.6 Xpand–Templates für die Erzeugung von SC–Code aus SyncCharts . 61
5.7 Teil der *tick*–Funktion für ABRO vor und nach der Formatierung mit dem Beautifier 63

6.1 KIELER–System mit einer geöffneten SyncCharts–Modellinstanz . . . 66
6.2 Simulation für das SyncChart aus Abbildung 4.20(a) 69
6.3 70
6.4 Ausschnitt der *eso*–Datei für ABRO aus Abbildung 3.1 72
6.5 Verschiedene Wege zur Validierung des Compilers 72

7.1 Verschiedene Pfade für die Erzeugung von Code für die Versuchsergebnisse 74
7.2 Zeitmessung für kleine SyncCharts in Taktzyklen 75
7.3 Größe der ausführbaren Datei für kleine SyncCharts in Byte 75
7.4 Größe in Lines Of Code (LOC) für kleine SyncCharts 76
7.5 Zeitmessung für mittelgroße SyncCharts in Taktzyklen 76
7.6 Größe der ausführbaren Datei für mittelgroße SyncCharts in Byte . . 77
7.7 Größe in LOC für mittelgroße SyncCharts 77
7.8 Zeitmessung für große SyncCharts in Taktzyklen 78
7.9 Größe der ausführbaren Datei für große SyncCharts in Byte 78
7.10 Größe in LOC für große SyncCharts 78
7.11 Zeitmessung im Vergleich zu Columbia Esterel Compiler (CEC) in einer Virtuelle Maschine (VM) in Taktzyklen 79
7.12 Größe der ausführbaren Datei im Vergleich zu CEC in einer VM in Byte 79
7.13 Größe in LOC im Vergleich zu CEC in einer VM 79

Abkürzungsverzeichnis

API	Application Programming Interface
BLIF	Berkeley Logic Interchange Format
CEC	Columbia Esterel Compiler
EMF	Eclipse Modeling Framework
E-Studio	Synfora's Esterel Studio
FSM	Endlicher Automat (Finite State Machine)
GCC	GNU Compiler Collection
HDL	Hardware Description Language
IDE	Integrated Development Environment
JDT	Java Developement Tooling
JNI	Java Native Interface
JET	Java Emitter Templates
JSON	JavaScript Object Notation
KEP	Kiel Esterel Processor
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich-Client
KIEM	KIELER Execution Manager
KlePto	KIELER leveraging Ptolemy semantics
KSBasE	KIELER Structure-Based Editing
LOC	Lines Of Code
MOF	Meta Object Facility
oAW	OpenArchitectureWare
OOP	Objektorientierte Programmiersprache
OSGi	Open Services Gateway initiative
PDE	Plugin Developer Environment
RCA	Rich Client Application
RSML	Reservoir Simulation Markup Language

SCC SyncCharts Compilers Collection

SC Synchronous C

SJ Synchronous Java

SSM Safe State Machine

ThinKCharts Thin Kieler SyncCharts Editor

UML Unified Modeling Language

VHDL Very High Speed Integrated Circuit Hardware Description Language

VM Virtuelle Maschine

XML Extensible Markup Language

XMI XML Metadata Interchange

1 Einleitung

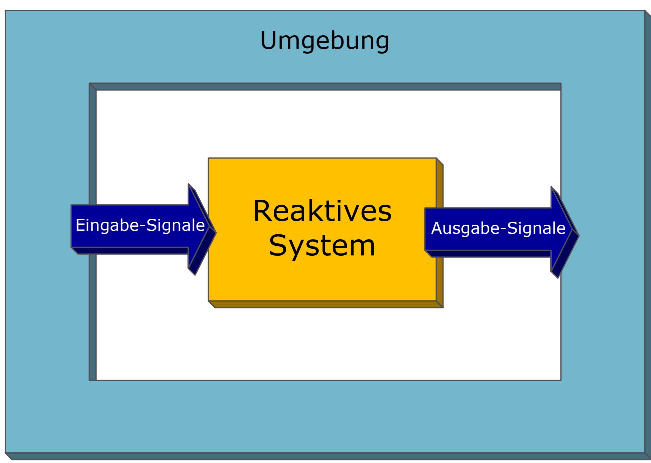


Abbildung 1.1: Reaktives System mit Eingabe– und Ausgabe–Signalen

Systeme, die kontinuierlich mit ihrer Umgebung interagieren, nennt man reaktive Systeme. Solche Systeme bekommen Eingaben von außen, führen Berechnungen durch und erzeugen daraus Ausgaben für die Umgebung. Im Unterschied zu interaktiven Systemen wird der Ablauf reaktiver Systeme nicht durch das System sondern durch die Umgebung bestimmt. Die Ausführung von reaktiven Systemen definiert sich demnach durch die Umwelt und den internen Zustand selbst. Um solche Systeme zu modellieren, benötigt man sowohl Nebenläufigkeit als auch Preemption. Oftmals ist dabei eine deterministische Ausführung vorausgesetzt. Abbildung 1.1 zeigt schematisch ein reaktives System in einer Umgebung und die Kommunikation über Eingabe– und Ausgabe–Signale.

Synchrone Sprachen wurden für die Implementierung reaktiver Systeme entwickelt. In Abbildung 1.2 ist ein Beispiel zu sehen, in dem die Verwendung von deterministischer Nebenläufigkeit und Preemption durch die synchrone Sprache Esterel beschrieben wird. Das System erwartet jeweils die Eingabe–Signale `A` und `B`. Wenn beide Signale registriert wurden, dann wird sofort das Ausgabe–Signal `O` geliefert. Das System wird unmittelbar in den Startzustand zurück gesetzt, wenn `R` anliegt, unabhängig davon, ob die Signale `A` und `B` anliegen.

Besonders für sicherheitskritische Systeme, wie sie beispielsweise im Automobil–Bereich verwendet werden, ist es nicht nur wichtig, dass der Entwickler der Software

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

Abbildung 1.2: Die Programmiersprache Esterel mit dem ABRO–Beispiel

das Verhalten nachvollziehen kann, sondern auch andere Experten, wie Ingenieure oder externe Gutachter verstehen können, wie sich das System verhält. Darüber hinaus ist es oftmals notwendig zertifizierenden Stellen einen hinreichenden Einblick in das System zu verschaffen. Um dies zu gewährleisten, wurden bereits in den 80er Jahren graphische Modellierungssprachen wie Statecharts [20] entwickelt. Die Statechart–Notation erweitert klassische endliche Automaten um zusätzliche Eigenschaften wie Hierarchie, Orthogonalität oder Broadcast–Mechanismen, um zwischen nebenläufigen Komponenten kommunizieren zu können. Zu dem eignen sich Statecharts nicht nur zur Entwicklung reaktiver Systeme, sondern bieten die Möglichkeit, das Verhalten des entwickelten Systems visuell darzustellen und zu simulieren.

Als weitgehend allgemeingültige Form der Statecharts hat sich die Statechart–Notation von Harel [21] etabliert. Darüber hinaus sind etliche Dialekte, wie zum Beispiel Argos, RSML oder Timed Statecharts, entwickelt worden [4]. Statecharts sind ebenfalls in Unified Modeling Language (UML) eingebunden [14, 29]. Neben etlichen frei verfügbaren UML–Werkzeugen wie ArgoUML oder Fujaba[1] werden Statecharts von vielen kommerziellen Tools wie Matlab/Simulink/Stateflow von The MathWorks, IBM Rational Statemate [21] oder IBM Rational Rose unterstützt.

In dieser Arbeit werden SyncCharts betrachtet, ein Statechart–Dialekt, der von André in den 90er Jahren entwickelt wurde [3]. SyncCharts — in der kommerziellen Verwendung auch Safe State Machine (SSM) genannt — besitzen eine formale Semantik, die auf dem synchronen Berechnungsmodell basiert. Deshalb sind sie besonders für sicherheitskritische Anwendungen geeignet [5]. Eines der kommerziellen Tools, das SyncCharts als Modellierungssprache verwendet, ist Synfora's Esterel Studio (E-Studio). E-Studio stellt einige Möglichkeiten zur Verfügung, Code aus SyncCharts zu synthetisieren. Dazu gehört die Generierung von Hardwarebeschreibungssprachen wie Verilog HDL oder VHDL, Programmiersprachen wie C oder Modellierungssprachen wie SystemC. Die Synthese in E-Studio beinhaltet als Zwischensprache Esterel [7] (siehe Beispiel 1.2), wofür es bereits eine Menge an Übersetzungs–Ansätzen gibt [33].

[1] http://www.oose.de/service/uml-werkzeuge.html

Die Übersetzung von SyncCharts nach Esterel ist jedoch recht aufwendig und weist eine weitere Problematik auf. In SyncCharts wird der Kontrollfluss durch Transitionen zwischen Zuständen dargestellt. Esterel hingegen verwendet Schleifen und aufgrund der mangelnden `GOTO`–Anweisungen ist der Bezug zum SyncChart schwer nachzuvollziehen. Der weitere Weg von Esterel nach C beinhaltet zudem eine komplizierte Übersetzung. Das bedeutet zwar auf der einen Seite, dass die Übersetzung von Esterel nach C effizienten Code liefern kann, auf der anderen Seite ist die Verbindung zum resultierenden SyncChart kaum noch vorhanden. Das hat wiederum zum Resultat, dass der entstandene Code schwer zu validieren oder zu verifizieren ist.

In dieser Arbeit wird die Code–Synthese von Synchronous C (SC) [42, 41] aus SyncCharts beschrieben. SC ist eine Erweiterung der Programmiersprache C und erlaubt es, synchronen und reaktiven Kontrollfluss direkt in C einzubetten. Durch SC werden dem Entwickler Operatoren zur Verfügung gestellt, die es ermöglichen, deterministisches Multi–Threading auf Basis von Prioritäten zu realisieren. Darüber hinaus existieren Mechanismen für Signal–basiertes Broadcasting. SC stellt diese Operatoren in Form von C–Makros zur Verfügung[2]. Der Vorteil von SC ist, dass es direkt mit jedem üblichen C–Compiler übersetzt werden kann und somit auf fast allen Plattformen ausführbar ist. Ein weiterer wichtiger Aspekt ist der direkte Bezug zum SyncChart. Dadurch ist der Code besser lesbar und lässt sich, im Gegensatz zur Übersetzung via Esterel, einfacher validieren oder mit dem Ausgangsdiagramm abgleichen. Versuchsergebnisse zeigen, dass sowohl Größe als auch Ausführungsgeschwindigkeit des erzeugten SC–Codes mit existierenden Ansätzen vergleichbar sind. Vor allem die Codegröße ist im Vergleich zu anderen Methoden kompakter. Der in dieser Arbeit vorgestellte Compiler generiert SC Code direkt und ohne Zwischenschritt aus SyncCharts. Da die Prioritäten für die zu erstellenden Threads automatisch berechnet werden, lässt sich SC–Code für größere SyncCharts erstellen, für die der manuelle Weg äußerst aufwendig wäre.

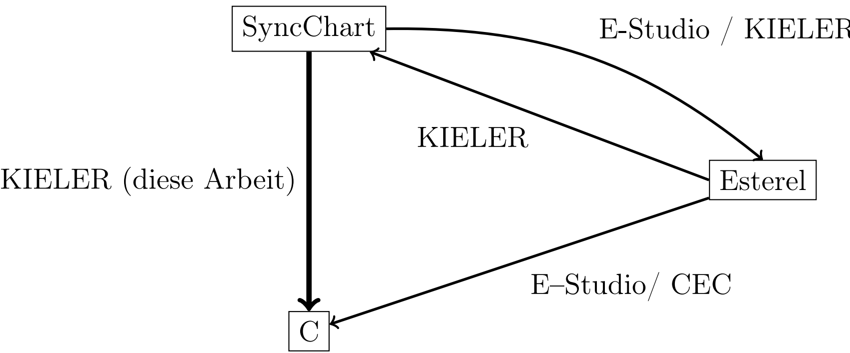


Abbildung 1.3: Verschiedene Wege der Generierung von C–Code aus SyncCharts

Abbildung 1.3 zeigt zwei verschiedene Wege der Generierung von C–Code aus SyncCharts. Der vorgestellte Compiler geht den direkten Weg und nutzt als Platt-

[2] `www.informatik.uni-kiel.de/rtsys/sc/`

form Kiel Integrated Environment for Layout Eclipse Rich-Client (KIELER)[3], eine Modellierungsplattform zur Evaluierung von Techniken zur Verbesserung von Modellierungspragmatiken. Im Vergleich dazu steht der Compiler aus E-Studio, der die Zwischensprache Esterel verwendet. KIELER bietet ebenfalls die Möglichkeit einer Transformation von SyncCharts nach Esterel, wodurch die Programme des SC–Compilers mit den generierten Programmen des CEC[4] bzw. des Standard Esterel Compiler von Inria[5] verglichen werden kann.

Im nächsten Kapitel werden Arbeiten vorgestellt, die einen ähnlichen Kontext aufweisen. Es wird ein grober Überblick über bestehende Ausarbeitungen und deren Unterschiede zur vorliegenden Arbeit gegeben. Im darauf folgenden Kapitel 3 werden einige Grundlagen geschaffen. Insbesondere werden synchrone Sprachen und SC erläutert. Kapitel 4 ist der Hauptteil dieser Arbeit und beschreibt den Compiler im Detail. Es werden Optimierungen und mögliche Erweiterungen vorgestellt. Das 5. Kapitel beschreibt die Techniken, die für die Entwicklung des Compilers verwendet wurden und gibt einen Einblick in die Implementierung. Das darauf folgende Kapitel 6 stellt die Plattform vor, in welcher der Compiler eingebettet ist und wie diese Plattform genutzt wird, um den erzeugten Code zu visualisieren und zu validieren. Die letzten beiden Kapitel 7 und 8 beschreiben die Ergebnisse die der Compiler im Vergleich zu anderen Methoden der Codegenerierung liefert, gefolgt von einer kurzen Zusammenfassung der Arbeit und einem Blick auf weiterführende, zukünftige Arbeiten.

[3] www.informatik.uni-kiel.de/rtsys/kieler
[4] http://www1.cs.columbia.edu/~sedwards/cec/
[5] www-sop.inria.fr/esterel.org/files/

2 Verwandte Arbeiten

Für die Beschreibung von reaktiven Systemen sind Statecharts gut geeignet; effizienten Code daraus zu generieren ist jedoch nicht so einfach zu realisieren. Es gibt prinzipiell drei verschiedene Ansätze für die Erzeugung von Code aus Statecharts, die im Folgenden genannt werden. Diese Ansätze können jedoch nicht direkt für SyncCharts übernommen werden, da dieser spezielle Statechart–Dialekt sich von den anderen Dialekten semantisch unterscheidet. Eine Beschreibung der Semantik von SyncCharts folgt im nächsten Kapitel.

1. Pattern–Übersetzung in eine Objektorientierte Programmiersprache (OOP)

   In der Arbeit von Ali und Tanaka [1] wird dieser Weg beschrieben, in dem Zustände direkt in Klassen, zum Beispiel für Java, übersetzt werden.

2. Dynamische Simulation

   Die Arbeit von Wasowski [43] beschreibt die Übersetzung von Statecharts in Hierarchie–Bäume, aus denen die Ausführungsreihenfolge berechnet wird.

3. Umwandlung in einen endlichen Automaten FSM

   Dabei werden Statecharts so umgeformt, dass keine Hierarchien und parallele Komponenten mehr enthalten sind [44]. Die Übersetzung des entstandenen FSM ist dann einfacher.

Der einfachste Weg ist die Übersetzung in einen endlichen Automaten, für den der Code direkt abgeleitet werden kann. Der Nachteil dieser Methode ist allerdings eine Zustandsexplosion, die bei der Auflösung von Hierarchien und Orthogonalität entstehen kann. Deshalb wird oft ein Mittelweg zwischen der Transformation in FSMs und dynamische Simulation verwendet.

Köhler [23] stellt drei Ansätze zur Codegenerierung aus UML–Statechart–Diagrammen vor. Ein Ansatz ist die Implementierung durch Fallunterscheidung. Hier werden für alle möglichen Ereignisse, die in einem Statechart auftreten können, Methoden (z.B. für Java) erzeugt, welche dann mit Hilfe von Fallunterscheidung ausgeführt werden. Es gibt jedoch keine Angabe darüber was passiert, wenn Ereignisse simultan ausgeführt werden sollen. Ein anderer Ansatz ist die Implementierung durch so genannte Vererbungshierarchien. Dieser Ansatz ist eng an der Arbeit von Ali und Tanaka angelehnt und verwendet auch *State–Pattern*. In einem dritten Ansatz werden objektorientierte Zustandstabellen benutzt und Zustände direkt abgebildet. Hier wird jedoch das Problem geschildert, dass die Abarbeitung von Ereignissen nicht synchron geschehen kann. Des Weiteren wird in dieser Arbeit im Gegensatz zu

SyncCharts ein UML–Statechart–Dialekt verwendet, der keine echte Nebenläufigkeit bereit stellt.

Im Gegensatz zu anderen Statechart–Dialekten basieren SyncCharts auf einem synchronen Framework und weisen sich durch deterministisches Verhalten aus. Die Erzeugung von Code für SyncCharts muss demnach diese wichtigen Eigenschaften berücksichtigen. Die Ausführung von SyncCharts mit SC kann in die Kategorie der simulationsbasierten Ansätze klassifiziert werden. Der Simulator wird dabei durch SC bereit gestellt. Hierbei ist es nicht entscheidend wie SC implementiert ist. Es ist ebenfalls denkbar, dass der Code in einer virtuellen SC–Maschine ausgeführt wird oder SC als Prozessorerweiterung implementiert wird, um den generierten Code dort auszuführen.

Eine Übersetzung von SyncCharts nach Esterel wurde mit der Definition der SyncCharts und deren Semantik von André [3] vorgestellt. Diese Übersetzung ist einfach und strukturell zu vollziehen. In E-Studio wird sie mit zusätzlichen nicht dokumentierten Optimierungen angewendet. Dennoch ist es schwierig den erzeugten Code zum resultierenden SyncChart abzubilden. SC ist mit der Erweiterung von Esterel mit GOTOs verwandt, welche von Tardieu und Edwards [39] vorgestellt wurde. In dieser Esterel–Erweiterung werden ebenfalls verschiedene GOTO–Anweisungen benutzt, um beispielsweise von einem in einen anderen Thread zu springen.

Der hier entwickelte Compiler weist Ähnlichkeiten mit dem Compiler von SyncCharts nach Kiel Esterel Processor (KEP) Assembler [8, 37] auf. Dieser Compiler erzeugt aus Esterel–Programmen, die aus SyncCharts synthetisiert werden können, Assembler–Code für die Ausführung auf dem synchronen reaktiven Prozessor KEP [25, 24]. Dazu wird ein Esterel–Programm in ein dazu äquivalentes zweites Esterel–Programm überführt, das eine Teilmenge der Esterel–Ausdrücke benutzt. Anschließend wird dieses Programm in einen Graphen überführt, der das Kontrollflussverhalten des KEP–Assembler–Programms repräsentiert. Die Knoten des Graphen enthalten die resultierenden KEP–Assembler–Instruktionen und die Kanten repräsentieren Kontrollfluss und Preemptionen. Im Weiteren werden ausgehend von Signalabhängigkeiten, die im Esterel–Programm existieren, Prioritäten für die zu erzeugenden Threads im Assembler–Code berechnet. Diese Prioritäten werden dem Graphen in Form von Knoten hinzugefügt. Aus dem Graph mit den berechneten Thread–Prioritäten und den abgebildeten Thread–IDs kann dann der Assembler–Code direkt erzeugt werden.

Die Struktur des KEP–Assembler Programms ähnelt der Struktur der generierten Programme für den hier vorgestellten Compiler. Das liegt daran, dass die SC–Makros eng am Instruktionssatz des KEP angelehnt sind. In dieser Arbeit wird ebenfalls ein Graph mit Abhängigkeiten — bzw. ein Baum, da dieser azyklisch sein muss — erzeugt, der für die Zuweisung von Thread–Prioritäten verwendet wird. Dennoch ist die Berechnung der Thread–Prioritäten in einer anderen Weise realisiert, da der hier erzeugte Baum eine größere Menge an Abhängigkeiten in Form von Kanten besitzt und die Prioritäten nicht als Knoten im Baum existieren, sondern mit Hilfe des Baum

berechnet werden. Ein weiterer Unterschied ist die direkte Synthese von Code aus SyncCharts, ohne den Zwischenschritt über Esterel zu benutzen. Außerdem werden für SC ausschließlich Prioritäten für Threads verwendet. Der generierte SC–Code ist nicht an einen speziellen Prozessor gebunden und ist daher weitestgehend Plattformunabhängig. Des Weiteren müssen alle *complex expressions* für die Übersetzung nach KEP–Assembler extrahiert werden. Dieser Schritt ist für SC nicht notwendig, da durch die Verwendung von C als Ziel–Sprache auch komplizierte Signalausdrücke direkt übersetzt werden können.

Ein Compiler, der ebenfalls SyncCharts direkt in C–Code übersetzt, wird in der Arbeit von Boucaron [9] vorgestellt. Die dort entwickelte SyncCharts Compilers Collection (SCC)[1] erzeugt aus SyncCharts sowohl C als auch BLIF. Der hier verwendete Ansatz besteht darin, durch strukturelle Transformationen aus SyncCharts boolesche Gleichungen zu erzeugen. Durch die Verwendung binärer Entscheidungsdiagramme wird sichergestellt, dass die Gleichungen frei von Zyklen sind. Das Hauptaugenmerk dieses Compilers liegt in der effizienten Übersetzung von SyncCharts in Hardware ähnlich zur Übersetzung von Esterel in Schaltkreise. Der erzeugte Code repräsentiert eine Menge von booleschen Gleichungen, wobei die Performance weniger wichtig war als die Anzahl der verwendeten Register. Dadurch ist der Code auch schwer lesbar und im Vergleich zu SC ist ein Rückschluss zum SyncChart nahezu unmöglich. Abbildung 2.1 zeigt einen kleinen Ausschnitt des vom SCC generierten C–Code für ABRO (siehe Abb. 3.1). Hier ist der fehlende Bezug zum SyncChart deutlich erkennbar.

[1] julien.boucaron.free.fr/wordpress/?page_id=6

```
1  //Level2
2  __MS__M_0__STG__A__STATE__S_0A4__REINCARNATION = __UP__REINCARNATION || __MS__M_0__STG__A__STATE__S_0A4__REINCARNATION_SA ||
       __MS__M_0__STG__A__STATE__S_0A4__REINCARNATION_WA ;
3  __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__ACTIVE =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__A__ACTIVE &&
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__B__ACTIVE ;
4  __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__SUSPEND = __MS__M_0__STG__A__STATE__S_0A4__SUSPEND ;
5  __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__RESET_SA = __MS__M_0__STG__A__STATE__S_0A4__RESET_SA ;
6  __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A5__SUSPEND = __MS__M_0__STG__A__STATE__S_0A4__SUSPEND ;
7  __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A5__RESET_SA = __MS__M_0__STG__A__STATE__S_0A4__RESET_SA ;
8  __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A5__RESET_WA = __MS__M_0__STG__A__STATE__S_0A4__RESET_WA ;
9  //Level3
10 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__REINCARNATION =
       __MS__M_0__STG__A__STATE__S_0A4__REINCARNATION ||
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__REINCARNATION_SA ||
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__REINCARNATION_WA ;
11 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__ACTIVE =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__ACTIVE ;
12 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__A__STATE__S_0_4_4A5__SUSPEND =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__SUSPEND ;
13 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__A__STATE__S_0_4_4A5__RESET_SA =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__RESET_SA ;
14 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__A__STATE__S_0_4_4A7__SUSPEND =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__SUSPEND ;
15 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__A__STATE__S_0_4_4A7__RESET_SA =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__RESET_SA ;
16 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__B__STATE__S_0_4_4B4__SUSPEND =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__SUSPEND ;
17 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__B__STATE__S_0_4_4B4__RESET_SA =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__RESET_SA ;
18 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__B__STATE__S_0_4_4B6__SUSPEND =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__SUSPEND ;
19 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__MS__M_0_4_4__STG__B__STATE__S_0_4_4B6__RESET_SA =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__RESET_SA ;
20 //Level4
21 __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__ACTIVE =
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A4__ACTIVE ||
       __MS__M_0__STG__A__STATE__S_0A4__MS__M_0_4__STG__A__STATE__S_0_4A5__ACTIVE ;
```

Abbildung 2.1: Ausschnitt vom SCC–Code für ABRO

3 SyncCharts und SC

Dieses Kapitel beschreibt die Eingabe– und Zielsprache des Compilers. Als Input verwendet er SyncCharts, ein graphischer Formalismus zum Modellieren von reaktiven Systemen. Als Produkt des Compilers steht dem SC gegenüber, eine Erweiterung von C zur Beschreibung synchroner Nebenläufigkeit. Das Fundament für SyncCharts bilden synchrone Sprachen wie Argos [26] oder Esterel [7]. Im Folgenden werden wichtigen Grundlagen eingeführt und mit Hilfe einiger Beispiele beschrieben.

3.1 Synchrone Sprachen

Synchrone Programmiersprachen sind Sprachen, die für die Entwicklung von reaktiven Systemen optimiert wurden. In reaktiven Systemen findet eine permanente Interaktion mit der Umgebung statt. Unter den Anforderungen, die an reaktive Systeme gestellt werden, zählen Nebenläufigkeit und Determinismus zu den wichtigsten. Außerdem müssen Eingaben von außen, die zum Beispiel durch Sensoren gelesen werden, innerhalb zeitlich fest definierter Grenzen verarbeitet werden [18].

Die Grundlage aller synchronen Sprachen ist die Synchronitätshypothese. Dabei werden folgende Annahmen getroffen [32]:

- Perfekte Synchronität

  Alle Prozesse eines Systems laufen gleichzeitig ab und es wird keine Zeit für die Reaktion des Systems benötigt. Deshalb werden Ausgaben zur selben Zeit erzeugt, wie Eingaben gelesen werden.

- *Multiform notion of time*

  Zeit wird nicht in physikalischer Zeit gemessen, sondern durch die Reihenfolge von Ereignissen bestimmt. Die Zeitdauer kann zum Beispiel durch das mehrmalige Auftreten einer Eingabe definiert werden. Dies führt zu einem deterministischen Verhalten, das durch „echte“ Zeit nicht sichergestellt werden kann.

- *Zero–delay*

  Für den Übergang von einem Zustand in einen anderen wird keine Zeit benötigt. Möchte man diesen Übergang zeitlich darstellen, müssen für ihn zusätzliche Zustände eingeführt werden.

Aufgrund der oben genannten Eigenschaften werden synchrone Sprachen insbesondere für sicherheitskritische Anwendungen verwendet. Einige der etabliertesten Vertreter sind Esterel, Lustre oder Argos. Besonders das deterministische Verhalten führt dazu, dass synchrone Sprachen für Probleme geeignet sind, die durch endliche Automaten ausgedrückt werden können. Daraus motiviert sind graphische Darstellungen synchroner Sprachen entstanden.

3.2 SyncCharts

SyncCharts ist eine graphische Modellierungssprache, die 1996 von André vorgestellt wurde [2]. Aufgrund der Nähe zu den synchronen Sprachen sind SyncCharts zum Modellieren reaktiver Systeme besonders gut geeignet. Grundsätzlich sind SyncCharts in die Klasse der Zustandsdiagramme einzuordnen. Darüber hinaus besitzen sie jedoch Eigenschaften wie Determinismus, welche sie besonders Attraktiv für die Modellierung sicherheitskritischer Systeme machen.

3.2.1 Vom Automaten zum SyncChart

Automaten werden in der Informatik vielfältig eingesetzt und finden vor allem in der theoretischen Informatik Anwendung. Sie eignen sich aber auch hervorragend zur Beschreibung diskreter Prozesse und Abläufe, in denen Übergänge von einem Zustand in einen anderen statt finden. SyncCharts sind Erweiterungen von endlichen Mealy Automaten [15].

Definition 1 (Mealy Automat)**.** *Ein endlicher Mealy Automat ist ein 6–Tupel* $(S, \Sigma, \Gamma, \delta, \lambda, s)$*, mit*

S	*Endliche Menge von Zuständen,*
Σ	*Eingabealphabet,*
Γ	*Ausgabealphabet,*
$\delta : S \times \Sigma \to Q$	*Transitionsfunktion,*
$\lambda : S \times \Sigma \to \Gamma$	*Ausgabefunktion und*
$s \in S$	*Startzustand.*

StateCharts sind Mealy Automaten mit zusätzlichen Eigenschaften wie Hierarchie, Nebenläufigkeit, Preemption und Broadcast. Sie sind jedoch im allgemeinen per Definition nicht deterministisch. Der Statechart–Dialekt SyncCharts verfügt zusätzlich über deterministisches Verhalten und basiert auf der Synchronitätshypothese. Im Unterschied zu den StateCharts existieren in SyncCharts keine *inter–level*–Transitionen, also Transitionen, die hierarchieübergreifend verlaufen.

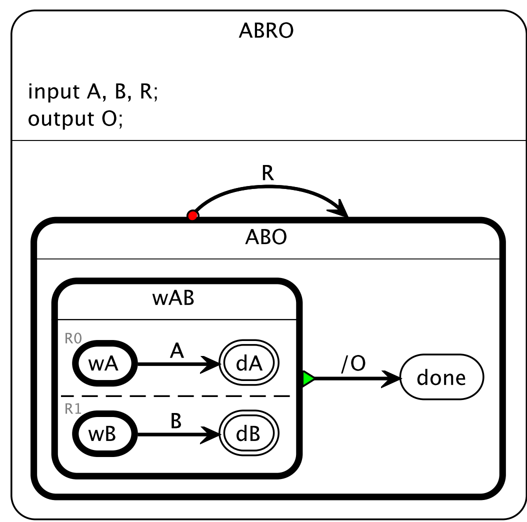


Abbildung 3.1: ABRO, das „Hello World“ der synchronen Sprachen

3.2.2 SyncCharts im Detail

Das Basisobject in SyncCharts ist ein Zustand mit seinen ausgehenden Transitionen. Zustände können eine oder mehrere Regionen beinhalten in denen wiederum Zustände existieren. Transitionen verbinden genau zwei Zustände, die sich in einer Region befinden. Wie in jeder Programmiersprache gibt es auch für SyncCharts ein „Hello World“, das hier ABRO genannt wird. Abbildung 3.1 zeigt dieses SyncChart und soll für die weitere Einführung als Beispiel dienen. Es ist zu beachten, dass dieses ABRO äquivalent zum Esterel–Code aus dem vorherigen Kapitel (Abbildung 1.2) ist.

In ABRO sind die Konzepte der SyncCharts wie deterministisches Verhalten, Nebenläufigkeit, Preemption und Kommunikation über Signale in einfacher Art und Weise veranschaulicht. Nach dem Start sind die Zustände wA in Region R0 und wB aus Region R1 aktiv. In wA wird auf die Eingabe von A und in wB auf die Eingabe von B gewartet. Wenn eine Eingabe von A erfolgt, dann wird die Transition zu Zustand dA ausgeführt und dieser wird aktiv. Analog gilt dieses Verhalten für die parallele Region R1. Wenn beide Zustände dA und dB betreten wurden, wird der Makrozustand wAB direkt (instantan) verlassen, es wird O ausgegeben und in den Zustand done gesprungen. Die Eingabe von R setzt ABRO wieder zum Startzustand zurück und in diesem Berechnungsschritt, der als *Tick* bezeichnet wird, kann auch kein O ausgegeben werden. Insbesondere wird O auch dann nicht ausgegeben (emittiert), wenn A, B **und** R als Eingabe existieren.

ABRO enthält natürlich nicht alle Bestandteile, die für SyncCharts zur Verfügung stehen. Die wichtigsten werden im Folgenden kurz zusammengefasst. Für eine umfassende Beschreibung mit einer Reihe von vielen guten Beispielen ist *Semantics of SyncCharts* von André [3] die empfohlene Literatur.

Zustände

Zustände können in einfache und Makrozustände unterteilt werden. Makrozustände enthalten wiederum Elemente von SyncCharts in einer oder mehreren inneren, parallelen Regionen. Alle Zustände können interne *actions* enthalten. *On entry actions* werden beim Betreten eines Zustands ausgeführt, *on inside actions* beim Ausführen und *on exit actions* beim Verlassen von Zuständen. Darüber hinaus kann ein Zustand initial oder final sein. Jede Region beinhaltet genau einen initialen Zustand, wobei es mehrere Endzustände sein können. Ein Endzustand signalisiert die Beendigung der Ausführung einer Region. Kein „echter Zustand“ ist der *conditional state*. Dieser *Pseudozustand* repräsentiert die `if ... else if ... else`–Statements aus den klassischen Programmiersprachen. Bei der Auswertung von *conditional states* vergeht keine Zeit, weswegen er als transienter Zustand bezeichnet werden kann. Möchte man andere Zustände importieren, so wie es in vielen Programmiersprachen üblich ist, kann der *reference state* verwendet werden. Dieser Zustand hat einen Verweis auf einen anderen Zustand anstatt diesen *in–line* einzufügen.

Signale

SyncCharts erben das Konzept der Signale (*pure signals*) und wertebehafteten Signale (*valued signals*) von Esterel. Es gibt sowohl Eingabe– und Ausgabe–Signale für die Interaktion mit der Umgebung, als auch interne lokale Signale von Zuständen, deren Wertebereich beliebig tief geschachtelt werden kann. Standardmäßig sind Signale *absent*, was dem *false* von üblichen Programmiersprachen entspricht. Das Pendant dazu ist ein Signal das *present* (*true*) ist. Man sagt auch, analog zu Schaltkreisen in der Elektrotechnik, dass ein Signal anliegt oder nicht anliegt.

Damit ein Signal *present* ist, muss es entweder ein Eingabesignal sein, welches von der Umgebung auf *present* gesetzt wurde, oder es ist ein lokales bzw. Ausgabesignal, das im aktuellen Tick des SyncChart emittiert wurde. Ein *valued signal* kann zusätzlich — unabhängig davon ob es *present* oder *absent* ist — einen Wert enthalten. Dieser Wert kann einmal initial oder beim Ausführen einer Aktion gesetzt werden. Für *valued signals* sind die Typen *Integer*, *Bool* und *Float* vorgesehen. Zusätzlich können *valued signals* mit einer Funktion kombiniert werden. Für *Integer*–Signale könnte das „+“ oder „$max()$“ sein, für *Bools* „$\vee$“ oder „$\wedge$“. Diese so genannte *combine Funktion* kombiniert mehrfaches Emittieren von *valued signals* in einem Tick. Dadurch kann ein Signal mehrmals in einem Tick mit unterschiedlichen Werten emittiert werden und das SyncChart bleibt dennoch deterministisch. Es gibt es keine Reihenfolge für mehrfache Emissionen in einem Tick und semantisch gesehen passieren diese auch gleichzeitig. Deshalb muss die *combine Funktion* kommutativ und assiziativ sein, um den Determinismus zu erhalten. Gibt es keine *combine Funktion* für *valued signals*, so wird angenommen, dass es in einem Tick höchstens einmal emittiert wird. Der Wert des Signals bleibt so lange erhalten, bis er in einem der späteren Ticks neu gesetzt wird. Dabei ist der Wert unabhängig davon, ob das Signal *present* oder *absent* ist.

Es ist zu beachten, dass mehrfaches Emittieren von *pure signals* unproblematisch ist

Des Weiteren ist es möglich, mit dem Schlüsselwort `pre` den Status oder den Wert von Signalen des vorherigen Ticks zu ermitteln. Ein einfaches, aber sehr schönes Beispiel, das die Funktionsweise von *valued signals* und `pre` gut beschreibt, ist *Shifter3* (vergleiche [3]) aus Abbildung 3.2(a). Hier wird der Wert von Eingabesignal `I`, wenn `I` *present* ist, im nächsten Tick nach `S0` geschrieben, dieser wieder einen Tick später nach `S1` weiter geleitet und schlussendlich im darauf folgenden Tick in `O` ausgegeben. In Abbildung 3.2(b) ist ein verkürzter Trace für eine Ausführung von *Shifter3* gezeigt. Die Zahl 1 aus Input `I` wird mit jedem Tick an das nächste Signal weiter geleitet und mit `O` ausgegeben. Das Beispiel veranschaulicht ebenfalls die Kommunikation zwischen parallelen Regionen über Signale.

Transitionen

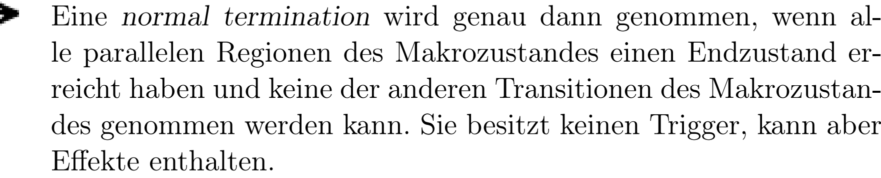
Transitionen können Trigger Effekte und/oder Host–Code enthalten. Mit Triggern kann überprüft werden, ob ein Signal anliegt oder nicht anliegt ist, oder welchen Wert ein Signal hat. Außerdem sind boolesche Kombinationen und die Verwendung von Host–Code in Triggern möglich. Wenn ein Trigger als *true* ausgewertet werden kann, dann wird die Transition genommen und — falls enthalten — seine Effekte ausgeführt. In Effekten können Signale emittiert werden, Signal–Werte gesetzt oder geändert oder *Host Code* ausgeführt werden.

Charakteristisch für SyncCharts sind die verschiedenen Arten der Preemptionen. Diese werden durch unterschiedliche Typen von Transitionen ausgedrückt:

- *weak abortion*

  Wird ein Makrozustand durch ein *weak abort* verlassen, dann wird alles ausgeführt, was im inneren dieses Zustands in diesem Tick noch ausgeführt werden kann. Erst danach wird der Zustand verlassen („*letzter Wille*“).
- *strong abortion*

  Wird ein Makrozustand über ein *strong abort* verlassen, so gilt der „*letzte Wille*“ nicht mehr. Es wird keine Aktion aus dem Inneren des Zustands ausgeführt.
- *normal termination*

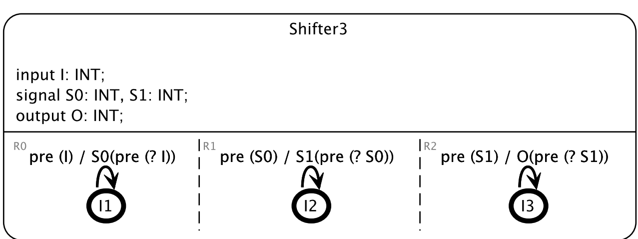
  

  Eine *normal termination* wird genau dann genommen, wenn alle parallelen Regionen des Makrozustandes einen Endzustand erreicht haben und keine der anderen Transitionen des Makrozustandes genommen werden kann. Sie besitzt keinen Trigger, kann aber Effekte enthalten.

- *suspension*

Die *suspension* ist eigentlich eher ein *connector* als eine Transition. Sie besitzt keine Effekte und suspendiert (pausiert) den Makrozustand wenn der Trigger zu *true* ausgewertet werden kann.

Analog zu Esterel können Transitionen entweder *immediate* oder verzögert (*delayed*) sein. Um eine *delayed* Transition ausführen zu können, muss der Quellzustand zu Beginn des Ticks aktiv gewesen sein. Eine *immediate* Transition kann genommen werden, sobald der Zustand aktiv wird. Dadurch können mehrere Zustände in einem Tick betreten werden. Besondere *delayed* Transitionen sind welche mit einem *count delay*. Für diese muss der Trigger mehrere Ticks zu *true* ausgewertet werden können, bevor die Transition ausgeführt wird.



(a) SyncChart

```
1  {"I":{"present":true,"value":1}}
2  ...
3  VAL:    0/_L_TICKEND determines value of sig_O/1 as 0
4  VAL:    0/_L_TICKEND determines value of sig_S0/2 as 0
5  VAL:    0/_L_TICKEND determines value of sig_S1/3 as 0
6  ...
7  {}
8  ...
9  VAL:    0/_L_TICKEND determines value of sig_O/1 as 0
10 VAL:    0/_L_TICKEND determines value of sig_S0/2 as 1
11 VAL:    0/_L_TICKEND determines value of sig_S1/3 as 0
12 ...
13 {}
14 VAL:    0/_L_TICKEND determines value of sig_O/1 as 0
15 VAL:    0/_L_TICKEND determines value of sig_S0/2 as 1
16 VAL:    0/_L_TICKEND determines value of sig_S1/3 as 1
17 ...
18 {}
19 ...
20 VAL:    0/_L_TICKEND determines value of sig_O/1 as 1
21 VAL:    0/_L_TICKEND determines value of sig_S0/2 as 1
22 VAL:    0/_L_TICKEND determines value of sig_S1/3 as 1
23 ...
```

(b) Trace einer Ausführung

Abbildung 3.2: Das Beispiel Shifter3

Damit SyncCharts trotz Zuständen mit mehreren ausgehenden Transitionen immer deterministisch bleiben, besitzt jede Transition eine Priorität. Somit wird die erste mögliche Transition mit der höchsten Transitionspriorität genommen. Transitionsprioritäten müssen demnach pro Zustand eindeutig sein. Außerdem ist zu beachten, dass *strong aborts* eine höhere Priorität besitzen als *weak aborts*, eine *normal termination* hat stets die kleinste Priorität und wird als letztes in einem Tick überprüft.

3.3 Synchronous C (SC)

Synchronous C (SC) [42, 41] ist eine Makro–basierte Erweiterung der Programmiersprache C. Durch die Erweiterung von zusätzlichen Funktionen wird C um Eigenschaften wie Nebenläufigkeit oder Preemption bereichert. Der Unterschied zur bereits existierenden Nebenläufigkeit und Preemption ist, dass SC dies deterministisch sicher stellt. Damit erfüllt SC die Synchronitätshypothese. Die Instruktionen von SC sind durch C–Makros realisiert und können deshalb mit jedem herkömmlichen C–Compiler übersetzt werden. Dadurch ist SC plattformunabhängig und vielseitig einsetzbar. Es ist aber auch möglich SC als virtuelle Maschine oder Hardware–Erweiterung für bestehende Prozessoren bereit zu stellen. Die SC–Instruktionen wurden durch den KEP motiviert, der sich wiederum auf Esterel zurück führen lässt [25].

SC wurde entwickelt, um das Verhalten von SyncCharts in C abzubilden. Eine der wichtigsten Entscheidungen bei der Designfindung war der hohe Bezug vom Code zum SyncChart und die damit verbundene gute Lesbarkeit. In den Abbildungen 3.3 und 3.4 werden die SC–Instruktionen mit Erklärung aufgelistet, die der Compiler verwendet. Tabelle 3.3 listet Thread–Operatoren auf, 3.4 Signal–Operatoren. Operatoren, die mit einem „*“ markiert sind, sind Dispatcher–Instruktionen. Das bedeutet, dass durch sie ein Threadwechsel erwirkt werden kann. Die Operatoren, die mit einem „$^{+}$“ gekennzeichnet sind, beinhalten automatisch generierte implizite Label, die nach der Expansion der Makros im Code sichtbar sind.

Durch SC wird C um ein prioritätenbasiertes Thread–Modell erweitert. In diesem Modell hat jeder Thread einen Programmzähler und eine ID. Die ID wird als Priorität für den jeweiligen Thread gehandhabt und deshalb auch so genannt. Die ersten SC–Versionen unterschieden IDs und Prioritäten von Threads. Dies wurde im Verlauf der Entwicklung zur ausschließlichen Verwendung von Prioritäten geändert. Der daraus entstandene Vorteil ist ein einfacheres Modell, das für den SC–Entwickler transparenter zu implementieren ist. Für den Compiler ist diese Änderung unerheblich, da für voneinander unabhängigen Threads (wie in ABRO) eine Priorität statt eine ID für jeden Thread berechnet werden muss.

Im laufenden Programm wählt der Dispatcher immer den Thread mit der höchsten aller Prioritäten. Der Thread wird dann solange ausgeführt, bis ein anderer Thread eine höhere Priorität hat oder er (für diesen Tick) beendet wird. Ein Thread kann entweder aktiv oder inaktiv sein. Aktive Threads werden im aktuellen Tick entweder

Operatoren	**Erklärung**
TICKSTART* (p)	Starttick (initial), weist dem Main-Thread die Priorität p zu.
TICKEND	Abschließender Tick, gibt 1 zurück, wenn es noch einen aktiven Thread gibt.
PAUSE*+	Deaktiviert den aktuellen Thread für diesen Tick.
HALT*+	Kurz für l: PAUSE; GOTO(l).
TERM*	Beendet den aktuellen Thread.
ABORT	Abortiert alle *Descendant*-Threads.
SUSPEND* (*cond*)	Suspendiert (Pause) den Thread und seine *Descendant*-Threads, wenn *cond* zu wahr ausgewertet wird.
SUSPENDG* (l)	Suspendiert (Pause) den Thread und seine *Descendant*-Threads und fährt mit l fort.
FORK(l, p)	Erzeugt einen neuen Thread mit Startadresse l and Priorität p.
FORKE* (l)	Beende das FORK und fahre mit l fort.
JOINELSE*+ (l_{else})	Wenn alle *Descendant*-Threads normal terminiert wurden, fahre fort; ansonsten pausiere und springe zu l_{else}
JOIN*+	Kurz für l_{else}: JOINELSE(l_{else}).
PRIO*+ (p)	Setzt die Priorität des aktiven Threads auf p.
PPAUSE*+ (p)	Kurz für PRIO(p); PAUSE.
GOTO(l)	Springt zum Label l.

Abbildung 3.3: Verwendete Thread–Operatoren aus SC [41]

ausgeführt oder können deaktiviert werden. Für inaktive Threads muss unterschieden werden, ob sie im aktuellen Tick bereits ausgeführt wurden oder noch ausgeführt werden müssen. Aufgrund dieser Eigenschaften und den aktuell gegebenen Prioritäten entscheidet dann der Dispatcher, welcher Thread ausgeführt wird. Der Dispatcher wird immer dann ausgeführt, wenn eine SC–Instruktion den Status des Threads ändert bzw. die Priorität des aktiven Threads nach unten setzt.

PAUSE– und HALT–Anweisungen sind Operatoren, die einen Tick beeinflussen. Durch sie wird die Ausführung eines aktiven Threads im aktuellen Tick beendet, indem er deaktiviert wird. Durch ein FORK–Statement wird ein neuer Thread initialisiert. Er bekommt ein Label und eine initiale Priorität. Nach mehreren FORK–Anweisungen folgt ein FORKE, das zum einen alle Threads aktiviert, die durch vorausgegangene FORK's initialisiert wurden und zum anderen das Label im Code beinhaltet, in dem der aktuelle Thread fortgeführt wird. Im Beispiel von ABRO aus Abbildung 3.1 wird im Thread für Zustand ABO durch zwei FORK–Anweisungen jeweils ein Thread für Region R0 und R1 erzeugt. FORKE aktiviert diese beiden Threads und verweist auf den Code von wAB mit dem der Thread für ABO fortgeführt wird.

Es ist wichtig, dass zwei Threads niemals die gleiche Priorität erhalten, auch wenn es — wie in ABRO — irrelevant ist, welcher der beiden Threads zuerst ausgeführt werden muss. Auch beim Ändern der Priorität durch eine PRIO–Anweisung darf keine Priorität verwendet werden, die von einem aktiven Thread genutzt wird.

Operatoren	**Erklärung**
SIGNAL(S)	Initialisiert ein lokales Signal S.
EMIT(S)	Emittiert Signal S.
SUSTAIN^{*+}(S)	Kurz für l: EMIT(S); PAUSE; GOTO(l).
PRESENT(S)	Wahr, wenn S *present* ist.
AWAIT^{*+}(S)	Kurz für l_{else}: PAUSE; PRESENT(s, l_{else}).
AWAITI^{*+}(S)	Kurz für GOTO(l); l_{else}: PAUSE; l: PRESENT(s, l_{else}).
EMITINT(S, val)	Emittiert *valued signal* S vom Typ Integer/Bool mit dem Wert val.
EMITINTMUL(S, val) / EMITINTADD(S, val) / EMITINTMAX(S, val) / EMITINTMIN(S, val)	Emittiert *valued signal* S vom Typ Integer, kombiniert mit Miltiplikation / Addition / Maximumfunktion / Minimumfunktion mit dem Wert val.
EMITBOOL(S, val)	Emittiert *valued signal* S vom Typ Bool mit dem Wert val.
EMITBOOLOR(S, val) / EMITBOOLAND(S, val)	Emittiert *valued signal* S vom Typ Bool, kombiniert mit Oder- / Und-Operator mit dem Wert val.
VAL(S)	Ermittelt den Wert von Signal S.
PRESENTPRE(S, l_{else})	Ist Wahr, wenn S im vorherigen Tick *present* war. Wenn S ein lokales Signal im Thread t ist, geht es um den letzten Tick in dem t aktiv — also nicht suspendiert — war.
VALPRE(S)	Ermittelt den Wert von Signal S des vorherigen Ticks.

Abbildung 3.4: Verwendete Signal–Operatoren aus SC [41]

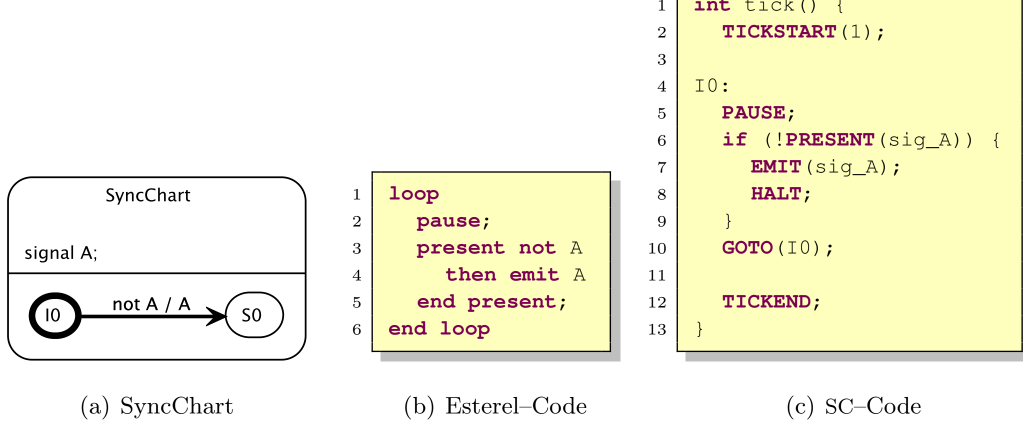
Um den reaktiven Kontrollfluss zu gewährleisten, der essentiell für SyncCharts ist, enthält SC einen Signal–basierten Kommunikationsmechanismus. Dadurch können Nachrichten threadübergreifend gesendet und empfangen werden (*Broadcasting*). Das Verhalten von Signalen in SC leitet sich dem von SyncCharts bzw. Esterel ab. Ein Signal kann entweder anliegen oder nicht anliegen. Standardmäßig sind sie *absent* und sind genau dann *present*, wenn sie im aktuellen Tick emittiert wurden. Für

SC gibt es keinen Unterschied zwischen Eingabe– und Ausgabe–Signalen. Ein `EMIT`–Statement emittiert ein Signal, durch eine `SIGNAL`–Anweisung wird ein lokales Signal (neu)initialisiert. Um *valued signals* zu emittieren, existieren spezielle Anweisungen, um den Typ des Signals zu spezifizieren. Mit einer `EMITINT`–Anweisung wird beispielsweise ein Integer–Signal mit einem gegebenen Wert emittiert, `EMITBOOL` emittiert ein Bool–Signal mit *true* oder *false* als Wert. Um *valued Signals* mit *combine Funktionen* zu emittieren, existieren ebenfalls spezielle Anweisungen, die in Abbildung 3.4 aufgelistet sind.

Da C sequentiell ist und SC in C eingebettet ist, ist auch SC sequentiell und stellt eine deterministische Reihenfolge für die Ausführung aller Systeme in einem Tick sicher. Damit unterscheidet sich SC von SyncCharts oder Esterel, wo per Definition alle Statements eines Ticks simultan ausgeführt werden. Das ist jedoch keine Einschränkung von SC, denn dadurch lassen sich Signale im Gegensatz zu klassischen synchronen Sprachen flexibler behandeln. Führt man beispielsweise in einem Tick mehrfach `EMITINT(`S`, `val`)` mit unterschiedlichen Werten als zweites Argument aus, so gäbe es in Sprachen wie Esterel keine sichere Aussage über den Wert von S nach dem Tick. Anders ist es in SC, wo der Wert stets wohldefiniert ist, denn S hat immer den zuletzt zugewiesenen Wert aller `EMITINT`–Statements in einem Tick. Alternativ könnte man solche Szenarien generell verbieten und so den klassischen synchronen Sprachen entsprechen. Das wäre mit einer statischen Analyse des Codes möglich, so wie es in Esterel ebenfalls gemacht wird, oder dynamisch bei der Ausführung. Eine Abweichung zur Synchronitätshypothese wäre auch dahingehend möglich, dass man erlaubt, ein Signal in einem Tick als *absent* zu erkennen und es danach zu emittieren. Damit könnten Signale innerhalb eines Ticks sowohl *present* als auch *absent* sein. Da die Ausführungsreihenfolge der Statements immer die gleiche ist, wäre auch dies ein deterministisches Verhalten. SC hält sich dennoch an die klassischen synchronen Sprachen und erkennt einen Fehler, wenn ein Signal als *absent* erkannt wurde und im selben Tick emittiert wird. Diese Fehlererkennung kann allerdings bei bedarf deaktiviert werden.

Es ist sowohl für SyncCharts als auch für Esterel möglich widersprüchliche bzw. nicht konstruktive Programme zu schreiben. Diese Programme sollten von einem Compiler erkannt und abgewiesen werden. Ein Beispiel dafür ist in Abbildung 3.5 zu sehen. Das SyncChart auf der linken Seite entspricht dem Esterel Code in der Mitte. Auf der rechten Seite befindet sich der dazugehörige SC–Code. Wenn Signal `A` nicht anliegt, dann wird es im selben Tick emittiert und zu Zustand `S0` transferiert. Da nun aber `A` anliegt, hätte die Transition nicht genommen werden dürfen.

Für SC ist die Situation nicht eindeutig. Auf der einen Seite ist der Ausführungsreihenfolge durch das sequentielle Verhalten von C gegeben, auf der anderen Seite ist es einfach Programme zu schreiben, welche die Synchronitätshypothese verletzen. Um dem Entwickler im Umgang mit SC zu unterstützen, sind Laufzeit–Checks standardmäßig aktiviert. Diese Checks erkennen beispielsweise in den oben genannte Situation, dass ein Signal, was gelesen wurde und als *absent* erkannt wurde, im gleichen Tick nicht emittiert werden darf. Abbildung 3.6 zeigt den Trace für den Ablauf



(a) SyncChart (b) Esterel–Code (c) SC–Code

Abbildung 3.5: Einfaches nicht konstruktives SyncChart und der äquivalente Esterel– (b) bzw. C–Code (c)

des SC–Codes aus 3.5(c). In Zeile 8 des Traces, welche dem zweiten Tick entspricht, wird für den Benutzer angezeigt, dass es ein Kausalitätsproblem gibt und somit der Code nicht konstruktiv ist.

```
RESET:    0/(null) reset automaton
RESET:    0/(null) reset automaton

PAUSE:    1/_L_INIT pauses, active = 03

PRESENT: 1/_L35 determines sig_A/0 absent
EMIT:     1/_L35 emits sig_A/0
SC ERROR (Causality): Signal sig_A/0 emitted after test for presence!
```

Abbildung 3.6: Trace für die Ausführung des Codes für das nicht konstruktive SyncChart aus Abbildung 3.5

Zusätzlich gibt es noch weitere Checks auf Konsistenz. Es wird beispielsweise überprüft, ob eine Priorität die für einen Thread gesetzt wird, nicht schon von einem anderen Thread verwendet wird. Das ist sehr wichtig, da durch Setzen einer bereits verwendeten Priorität der Thread überschrieben und damit das Verhalten fehlerhaft wird. Der Compiler stellt sicher, ob für ein SyncChart ein statisches Scheduling existiert und weist dadurch nicht konstruktive Programme ab. Außerdem wird für die Berechnung der Prioritäten sicher gestellt, dass keine Priorität zur Laufzeit mehrfach vorkommen kann.

4 Der Compiler

In diesem Kapitel wird beschrieben, wie aus einem SyncChart SC–Code generiert wird. Es werden Muster vorgestellt, welche für die Code–Synthese der einzelnen SyncChart–Elemente verwendet werden. Im Anschluss wird die Berechnung der Prioritäten erläutert und gezeigt, wie daraus die Nebenläufigkeit von SyncCharts im sequentiellen Code simuliert wird. Anhand einiger Beispiele wird diese Vorgehensweise im Detail besprochen. Ein weiterer Abschnitt des Kapitels beschreibt die Optimierungen des Compilers und Möglichkeiten, wo zusätzliches Potential für weitere Optimierungen vorhanden ist. Zum Schluss werden die syntaktischen Elemente der SyncCharts besprochen, die für den Compiler noch umgesetzt werden müssen und wie dies getan werden kann.

Da in diesem Kapitel häufig die gleichen Namen für unterschiedliche Repräsentationen verwendet werden, soll für das Kapitel folgende Konvention gelten:

- Elemente von SyncCharts — Zustände, Transitionen oder Regionen — werden wie folgt dargestellt:

  S1 , R1

- Knoten in Abhängigkeitsbäumen werden mit dem *Formel*–Style dargestellt:

  $S1$, H^s, $(S1, T1)^w$

- Für Elemente im generierten Code wird der `codefont`–Style verwendet:

  `S1`, `GOTO`

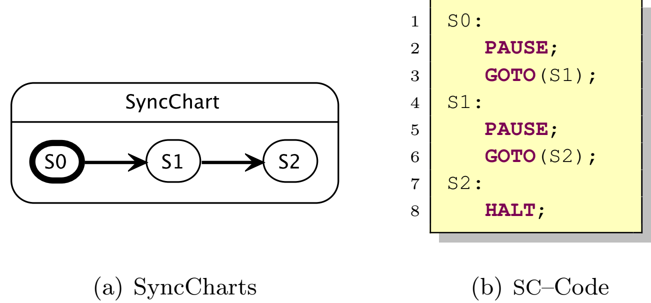
- Signale, Inputs und Outputs werden ebenfalls durch den `codefont`–Style hervorgehoben. Außerdem werden Inputs und Outputs groß und Signale klein geschrieben.

4.1 Elemente in SyncCharts und deren Übersetzung

In SyncCharts wird zwischen drei Elementen unterschieden, die für die Übersetzung nach SC wichtig sind. Diese Elemente sind *Zustände*, *Transitionen* und *Regionen*. Alle anderen Bestandteile von SyncCharts sind in diesen Elementen enthalten oder werden durch diese abgedeckt.

4.1.1 Regionen

Als erstes werden *Regionen* betrachtet. Eine Region besitzt immer einen Makro–Zustand, in welchem sie sich befindet. Die einzige Ausnahme bildet die *Root–Region*. Diese ist sozusagen der oberste Container, welcher das SyncChart beinhaltet. Jede Region beinhaltet mindestens einen Zustand, wobei genau ein Zustand je Region initial sein muss. Für jede Region wird jeder darin liegende Zustand betrachtet und für diesen Code synthetisiert. Dabei ist es erst einmal unerheblich, was das für ein Zustand ist. Um den Bezug vom SyncChart zum SC–Code zu erhalten, werden die Zustände einer Region anhand des *Kontrollflusses* im Diagramm sortiert und für diese in dieser Reihenfolge Code generiert. Für die Sortierung wird eine einfache Breitensuche verwendet. Abbildung 4.1 zeigt eine Region mit drei einfachen Zuständen und schematisch den dazu generierten Code. Der Kontrollfluss geht von Zustand S0 über S1 nach Zustand S2. Für den C–Compiler ist es unerheblich in welcher Reihenfolge der Code für die Zustände steht, da diese durch Sprunganweisungen erreicht werden. Jedoch ist der Code einfacher zu lesen, wenn sich dort die Reihenfolge im Diagramm wider spiegelt. Außerdem können `GOTO`–Anweisungen eingespart werden, falls durch die Reihenfolge vorgegeben ist, dass ein Zustand genau nach einem anderen ausgeführt werden muss. In Abschnitt 4.3 wird dies ausführlicher besprochen.



(a) SyncCharts (b) SC–Code

Abbildung 4.1: SyncChart mit einer Region und drei inneren Zuständen

4.1.2 Zustände

Zustände können in SyncCharts verschiedene Komplexitäten aufweisen. Die Komplexität eines Zustands spiegelt sich in der Struktur des generierten Codes wider. Die zwei wesentlichen Kriterien für den resultierenden Code sind ausgehenden Tansitionen und die Frage, ob ein Zustand hierarchisch ist. Abbildung 4.2 beschreibt den generellen Aufbau aller Zustände im SC–Code. Zustände werden in einen initialen und internen Teil unterteilt. Während der initale Teil alle Bestandteile beinhaltet, die *instantan* erreicht werden, enthält der interne Teil die Elemente, die im nachfolgenden Tick beginnen. Das Label `Label_afterJoin` befindet sich im internen Teil,

jedoch vor der `PAUSE`–Anweisung. Dieses Label wird benötigt um *normal terminations* zu behandeln. Es steht deshalb vor `PAUSE`, da *normal terminations* instantan geprüft werden. Je nach Art des Zustands kann die Struktur vereinfacht und um entsprechende Anweisungen minimiert werden. Dies wird in Abschnitt 4.3 beschrieben.

```
1  Label:
2     # strong transitions
3     onEntry actions
4     PRIO(weak)
5     # weak transitions
6  Label_intern:
7     PRIO(strong)
8     normal termination
9  Label_afterJoin:
10    PAUSE
11    host code
12    all strong transitions
13    onInside actions
14    PRIO(weak)
15    all weak transitions
16    GOTO(Label_intern)
```

Abbildung 4.2: Struktur des generierten Codes für Zustände

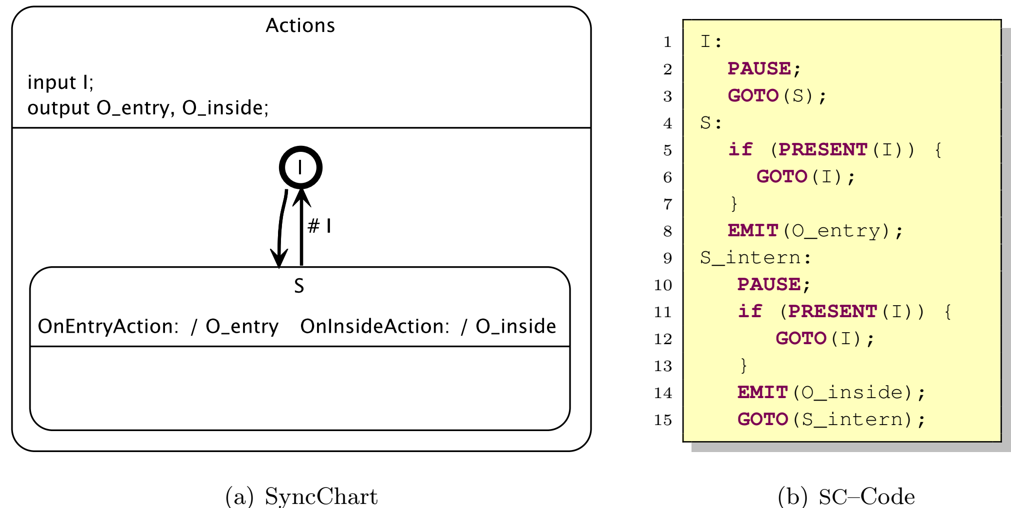
actions von Zuständen

Unabhängig davon, ob ein Zustand hierarchisch ist oder nicht, kann er *on entry*–, *on inside*– und *on exit actions* beinhalten. *On exit actions* werden vom Compiler aktuell nicht unterstützt. Die Semantik der *on entry*– und *on inside actions* [3] bestimmt, dass der Code zu jedem Zustand in ein initiales und ein internes Teilstück aufgeteilt werden muss. Im initialen Abschnitt werden alle *on entry actions* ausgeführt und im internen Abschnitt die *on inside actions*. Diese Unterteilung ist notwendig, da ein Zustand mehrere Ticks aktiv sein kann, jedoch nicht jedes Mal die *on entry actions* ausgeführt werden dürfen. Außerdem soll keine *on inside action* ausgeführt werden, wenn der Zustand durch eine *immediate*–Transition verlassen wird.

In Abbildung 4.3 emittiert Zustand `S` als *on entry action* `O_entry` und als *on inside action* `O_inside`. Liegt das Signal `I` beim Betreten von `S` an, so wird auch im generierten Code `O_inside` nicht emittiert, da `S_intern` nie ausgeführt wird. Liegt `I` an, nachdem `S` betreten wurde, so wird `O_inside` ebenfalls nicht emittiert. Außerdem wird `O_entry` kein zweites mal emittiert.

Arten von Zuständen

Es gibt verschiedene Arten von Zuständen mit unterschiedlichen Eigenschaften. Für diese ist der synthetisierte Code dementsprechend auch unterschiedlich. Es wurde



(a) SyncChart (b) SC–Code

Abbildung 4.3: SyncChart mit on entry– und on inside actions

bereits erwähnt, dass Zustände entweder hierarchisch oder einfach sein können. Diese Unterscheidung ist bei der Codesynthese wichtig, wobei man noch unterscheiden muss, ob hierarchische Zustände mehrere parallele Regionen besitzen oder nicht.

Die Codegenerierung für einfache Zustände ist strukturell und kann nahezu direkt übersetzt werden. Man muss allerdings noch unterscheiden, ob es sich um einen *Startzustand* oder einen *Endzustand* handelt. Abbildung 4.4(a) zeigt ein einfaches SyncChart mit einem hierarchischem Zustand H, welcher den Startzustand I und den Endzustand F beinhaltet. Da I ein Startzustand ist, wird dieser genutzt, um das Label (Zeile 1 im Code von Abbildung 4.4(b)) für den zu startenden Thread zu definieren. Der Endzustand F wird in eine `TERM`–Anweisungen übersetzt (Zeile 8). Diese sorgt dafür, dass der aktuelle Thread (hier `I`) terminiert wird. Ein Zustand wie H ohne ausgehende Transitionen, egal ob einfach oder hierarchisch, wird in ein `HALT` übersetzt.

Ein spezieller Zustand ist der *Conditional–Pseudo–Zustand*, der wie der Name sagt kein echter Zustand ist. Er ist das Pendent der `if-then-else`-Anweisungen von Programmiersprachen. Die Übersetzung nach SC ähnelt jedoch der von den allgemeinen Zuständen mit dem Unterschied, dass aufgrund der Semantik [3] alle ausgehenden Transitionen *immediate* sind und deshalb kein interner Teil und keine `PAUSE`–Anweisung für den Zustand existiert. *Conditional–Pseudo–Zustände* bekommen ebenfalls ein Label und werden durch eine `GOTO`–Anweisung erreicht.

Hierarchische Zustände können zunächst einmal darin unterschieden werden, ob sie nur eine Region oder mehrere parallele Regionen besitzen. Für jede Region eines hierarchischen Zustands muss ein neuer Thread gestartet werden. Das Label des Threads wird durch den Startzustand einer Region bestimmt. Dies dient einmal der besseren Lesbarkeit und ist auch von praktischem Nutzen, da jede Region ge-

H

I → F

(a) SyncChart

```
1  H:
2      FORK(I, 3);
3      FORKE(SyncChart_main);
4  I:
5      PAUSE;
6      GOTO(F);
7  F:
8      TERM;
9  H_main:
10     HALT;
```

(b) SC–Code

Abbildung 4.4: SyncChart mit Start– und Endzustand in einer Region

nau einen Startzustand besitzen muss. Der Thread für jede Region wird mit einer FORK–Anweisung gestartet, die als erstes Argument das Label und als zweites die Priorität des Threads besitzt. Die Berechnung der Prioritäten wird in Abschnitt 4.2.4 detailliert beschrieben und soll hier nicht weiter beachtet werden.

Nach den FORK–Anweisungen für jede Region eines hierarchischen Zustand folgt eine FORKE–Anweisung, die kennzeichnet, wo der Code des letzten Threads dieser Region beendet ist. Als Label für den Code wird die ID des Zustands verwendet und durch den Suffix _main ergänzt. Nachdem rekursiv für alle inneren Elemente jeder Region Code generiert wurde, wird schlussendlich der hierarchische Zustand selbst übersetzt. Wie aus der FORKE–Anweisung ersichtlich wurde, bekommt dieser als Label seine ID gefolgt von einem _main. Abbildung 4.5 zeigt ein SyncChart mit einem hierarchischen Zustand und zwei inneren Regionen und den dazu generierten SC–Code. Für die Regionen R0 und R1 werden in Zeile 5 und 6 die dazugehören Threads erstellt. Die FORKE–Anweisung aus Zeile 7 signalisiert, bis wohin der Code für diese beiden Threads gilt und wo mit dem Code für den Zustand H fortgefahren wird.

Ausgehende Transitionen

Ein Zustand kann mehrere ausgehende Transitionen besitzen und diese können sich wiederum in ihrer Art unterscheiden. Die Codegenerierung einer einzelnen Transition wird im folgenden Abschnitt beschrieben, jedoch gehört sie teilweise in die Beschreibung der Zustände. Jeder Zustand besitzt, abhängig von den ausgehenden Transitionen, eine andere Repräsentation im SC–Code. Aus der Struktur von Abbildung 4.2 ist ersichtlich, dass die Reihenfolge im Code von der Art der Transition abhängt. Alle *immediate* Transitionen werden im initialen Teilbereich des Zustands behandelt, *immediate–strong* Transitionen vor den *on entry actions* und *immediate–weak* Transitionen danach. Daraus ergibt sich, dass der initiale Teil nicht immer existiert und wegoptimiert werden kann. Im internen Teil ist die Anordnung der Transitionen

```
SyncChart:
   FORK(H, 2);
   FORKE(SyncChart_main);
H:
   FORK(I0, 4);
   FORK(I1, 6);
   FORKE(H_main);
I0:
   PAUSE;
   GOTO(S0);
S0:
   HALT;
I1:
   PAUSE;
   GOTO(S1);
S1:
   HALT;
H_main:
   HALT;
SyncChart_main:
   HALT;
```

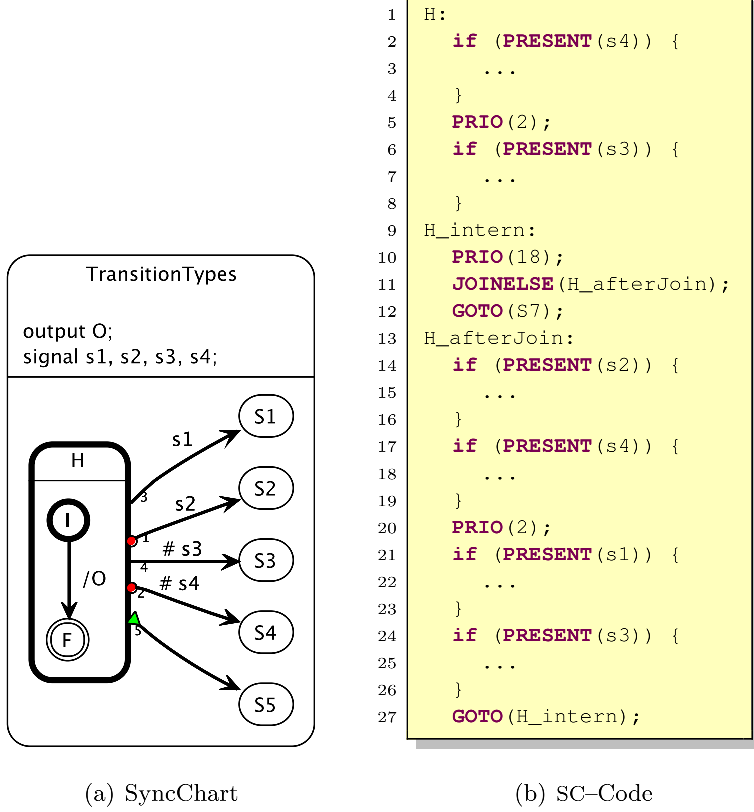
(a) SyncChart (b) SC–Code

Abbildung 4.5: SyncChart mit hierarchischem inneren Zustand und zwei darin liegenden parallelen Regionen

ähnlich. Wichtig ist, dass hier alle Transitionen, insbesondere auch die, für die bereits Code im initialen Teil generiert wurde, behandelt werden. Wenn mehrere Transitionen der gleichen Art existieren, werden diese anhand der Transitionsprioritäten im SyncChart sortiert. Die *normal termination* hat die kleinste Transitionspriorität und darf pro Zustand nur einmal auftreten. Sie steht im im internen Teil des generierten Codes vor allen anderen Transitionen. Wenn alle inneren Threads des Zustands terminiert wurden, dann wird die *normal termination* genommen und der Zustand verlassen.

Abbildung 4.6(a) zeigt einen hierarchischen Zustand (H), der alle möglichen Typen von ausgehenden Transitionen enthält. In 4.6(b) wird der dazugehörige Code für diesen Zustand verkürzt dargestellt. In diesem Beispiel spiegelt sich die Struktur aus Abbildung 4.2 gut wieder. Dort wird beschrieben, dass im initialen Teil eine `PAUSE`–Anweisung stehen muss. Dieses `PAUSE` ist hier nicht notwendig, da das `JOINELSE`–Makro so definiert ist, dass intern eine `PAUSE`–Anweisung nach dem Sprung, der durch `JOINELSE` resultiert, ausgeführt wird. Diese `PAUSE`–Anweisung steht im Beispiel implizit vor dem Label `S0_afterJoin` in Zeile 11.

Zu beachten ist außerdem noch, dass die Werte für die `PRIO`–Anweisungen einen großen Wertebereich aufweisen. Auch hier sind einige Optimierungen möglich, die später erklärt werden.



(a) SyncChart (b) SC–Code

Abbildung 4.6: Hierarchischer Zustand und allen möglichen ausgehenden Typen von Transitionen

4.1.3 Transitionen

Zustände müssen immer mit den dazugehörigen Transitionen betrachtet werden, da diese die Komplexität des resultierenden Codes bestimmen. Wie eine Transition allerdings übersetzt wird, hängt von der Art der Transition und vom Transitionslabel ab. Man kann Transitionen zunächst einmal in einfach und komplex unterscheiden. Einfache Transitionen besitzen als *Trigger* oder *Effekt* lediglich einfache Signale. Das heißt Trigger, die auf ein einziges *pure signal* warten und Effekte, die *pure signals* emittieren. Komplexe Trigger und Effekte können *valued signals* enthalten, aus booleschen Ausdrücken bestehen oder *Host–Code* beinhalten. Außerdem können komplexe Transitionen Operatoren und PRE–Anweisungen umfassen.

Einfache Transitionen

Einfache Transitionen besitzen entweder einen Trigger mit einem *pure signal* oder keinen Trigger. Sie emittieren ein oder mehrere Signale oder besitzen keine Effekte.

Die Übersetzung für Transitionen mit einfachen Triggern und Effekten funktioniert nach dem folgenden Schema. Die hochgestellten Werte in den Klammern geben die minimale und die maximale Anzahl der Statements an. So Bedeutet STATEMENT$^{(1,*)}$, dass dieses Statement mindestens einmal, aber ohne Begrenzung, ausgeführt werden muss.

```
if (PRESENT(signal))^(0,1) {
   EMIT(signal)^(0,*)
   GOTO(Label)
}
```

In Abbildung 4.7(a) wird in Region R0 Ausgabe O emittiert, wenn Input I anliegt. Im generierten Code (Abbildung 4.7(b): Zeile 4 - 7) ist das weiter oben erwähnte Schema abgebildet. Die Übersetzung für einfache Transitionen ist demnach strukturell und kann direkt dem zu Grunde liegenden SyncChart entnommen werden.

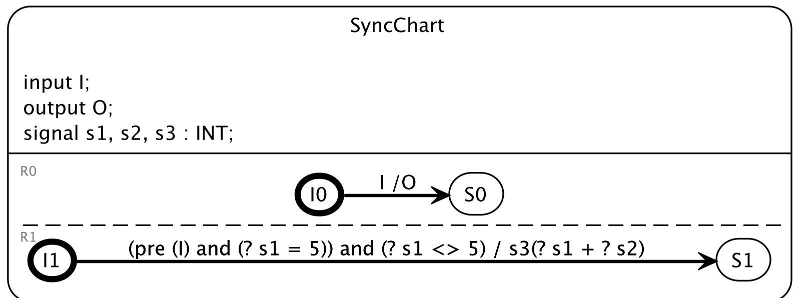
Komplexe Transitionen

Für komplexe Transitionen genügt das oben beschriebene Schema nicht mehr. Es wird zwar auch auf einen Trigger reagiert und es können auch Signale emittiert werden, diese sind jedoch nicht mehr direkt aus der Transition abzulesen. Wenn ein Trigger oder ein Effekt als komplex erkannt wird, muss dieser aus dem Modell geparsed werden. Da Transitionslabel intern in einer Baumstruktur gespeichert sind, kann der Baum rekursiv auslesen werden. An den Blättern befinden sich dann wieder Signale und der Weg durch den Baum ergibt die Struktur im erzeugten Code. Für das Emittieren der verschiedenen *valued signals* existieren in SC auch verschiedene Makros, um den richtigen Typ zu beschreiben. Variablen können mit der VAL– bzw. VALPRE–Anweisung ausgelesen werden.

Abbildung 4.7(b), Zeile 12 bis 15 zeigt den generierten Code für die komplexe Transition in Region R1 aus Abbildung 4.7(a). Es ist zu beachten, dass Signale bei der Übersetzung in Code mit dem Präfix sig_ versehen werden, um Kollisionen mit Schlüsselwörtern aus C zu vermeiden.

4.2 Prioritätenzuweisung

Während die Generierung von Code für die einzelnen Elemente in SyncCharts weitest gehend strukturell ist, ist die Berechnung und die Zuweisung der Prioritäten eine komplexe Aufgabe. SyncCharts zeichnen sich durch ihre synchrone Nebenläufigkeit aus. Es muss sichergestellt werden, dass jedes Signal, das in einem *Tick* geschrieben wird auch gelesen werden kann. Da SC eine sequentielle Sprache ist, ist es wichtig, dass sowohl hierarchische als auch parallele Regionen in eine richtige Reihenfolge gebracht werden. In vielen Fällen reicht diese Reihenfolge jedoch nicht aus, da in einem Tick mehrere Signale geschrieben und gelesen werden können. Deshalb ist es nicht nur



(a) SyncChart

```
1  ...
2  L_SyncChart_R0_I0:
3    PAUSE;
4    if (PRESENT(sig_I)) {
5      EMIT(sig_O);
6      GOTO(L_SyncChart_R0_S0);
7    }
8    GOTO(L_SyncChart_R0_I0);
9  ...
10 L_SyncChart_R1_I1:
11   PAUSE;
12   if (((PRESENTPRE(sig_I)) && ((VAL(sig_s1)) == 5)) && ((VAL(sig_s1)) != 5)) {
13     EMITINT(sig_s3, ((VAL(sig_s1))+(VAL(sig_s2))));
14     GOTO(L_SyncChart_R1_S1);
15   }
16   GOTO(L_SyncChart_R1_I1);
17 ...
```

(b) SC–Code

Abbildung 4.7: SyncChart mit einer einfachen und einer komplexen Transition und ein Ausschnitt des dazu generierten SC–Codes

notwendig eine Reihenfolge von Threads festzulegen, sondern zwischen diesen zu wechseln, wenn es erforderlich ist. Im folgenden Abschnitt wird beschrieben, wie mit Hilfe von *Abhängigkeitsbäumen*, die man aus einem SyncChart erzeugt, Prioritäten berechnet werden, die für die Thread–Wechsel notwendig sind.

4.2.1 Abhängigkeiten in SyncCharts

In SyncCharts gibt es verschiedene Beziehungen zwischen Elementen. Einige dieser Beziehungen sind für die Berechnung von Prioritäten wichtig. Aus diesen Beziehungen, die durch kleiner– oder kleiner–gleich–Relationen dargestellt werden können, werden Abhängigkeiten ermittelt, die später eine Reihenfolge für die Ausführung definieren.

Definition 2. *Sei $p : Z \longmapsto \mathbb{N}$ eine Funktion, die jeder Repräsentation Z von Zuständen eine natürliche Zahl zuordnet. Für zwei Zustände A und B gilt:*

$$A \leq_\tau B \Longrightarrow p(A) \leq p(B) \text{ und } A <_\tau B \Longrightarrow p(A) < p(B)$$

wobei

$$\tau \in \{c, h, s, t\} \text{ mit}$$

c: Kontrollflussabhängigkeit,

h: Hierarchieabhängigkeit,

s: Signalabhängigkeit und

t: Transitionssabhängigkeit.

Es zu beachten, dass hierarchische Zustände (H) sowohl als *weak* (H^w) als auch als *strong* (H^s) repräsentiert werden und die Funktion p auch auf diese Repräsentationen angewendet wird. Aus den Prioritäten p der einzelnen Repräsentationen von Zuständen entstehen im nächsten Schritt die Prioritäten der Threads im SC–Code. Eine Beschreibung der Vorgehensweise dafür folgt in den nächsten Absätzen.

Abbildung 4.8 zeigt eine modifizierte Version von ABRO, die im weiteren Verlauf als Beispiel dienen soll. Diese Version unterscheidet sich vom originalen ABRO lediglich darin, dass wenn Signal A anliegt, direkt Signal B emittiert wird und somit — falls R nicht anliegt — beide Endzustände aus wAB erreicht werden und O ausgegeben wird.

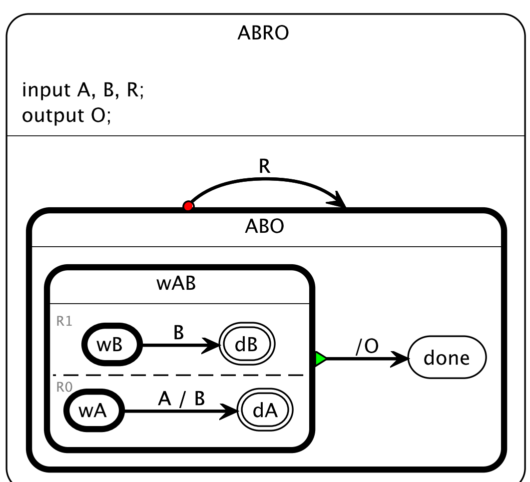


Abbildung 4.8: Modifiziertes ABRO mit Signalabhängigkeit

Kontrollflussabhängigkeiten

Kontrollflussabhängigkeiten bestehen jeweils zwischen zwei Zuständen, wenn diese durch eine Transition miteinander verbunden sind. Deshalb können sie dem SyncChart direkt entnommen werden. Jede Transition hat einen Startzustand und einen Zielzustand. Da der Code vom Startzustand vor dem Code des Zielzustands ausgeführt werden muss, folgt:

Definition 3 (Kontrollflussabhängigkeit)**.** *Für jede Transition mit Startzustand S und Zielzustand $T, S \neq T$ gilt:*

$$T \leq_c S \Rightarrow p(T) \leq p(S).$$

Kontrollflussabhängigkeiten sind die einzigen Abhängigkeiten mit einer kleiner–gleich–Relation. Das liegt daran, dass Prioritäten von Zuständen unter Umständen gleich sein können.

In Abbildung 4.8 existieren Kontrollflussabhängigkeiten zwischen Knoten wB und dB oder zwischen wAB und done . Wie man sieht macht es für diese Abhängigkeiten keinen Unterschied, um welche Art von Knoten es sich handelt. Keine Kontrollflussabhängigkeit wird durch die Transition gebildet, die R emittiert, da Start– und Zielzustand identisch sind.

Hierarchieabhängigkeiten

Auch Hierarchieabhängigkeiten existieren jeweils zwischen zwei Zuständen. Dabei muss jedoch einer der beiden innerer Zustand des Anderen sein. Es muss unterschieden werden, ob der hierarchische Zustand ein *schwacher* oder ein *starker Zustand* ist. Ein hierarchischer Zustand wird als schwach definiert, wenn keine der ausgehenden Transitionen eine *strong abortion* ist, anderenfalls als stark.

Definition 4 (Hierarchieabhängigkeit)**.** *Für jeden hierarchischen Zustand H und jeden darin liegenden direkten Unterzustand S gilt:*

$$H^w <_h S \Rightarrow p(H^w) < p(S)$$

bzw.

$$S <_h H^s \Rightarrow p(S) < p(H^s)$$

mit H^w schwacher Zustand (weak) und H^s starker Zustand (strong).

Im Beispiel 4.8 ist ABO ein starker Zustand und es folgt $p(done) <_h p(ABO^s)$. wAB ist ein schwacher Zustand und demzufolge erhält man $wAB^w <_h wA$. Sollten innere Zustände ebenfalls Hierarchie besitzen, werden diese zwar auch als schwach bzw. stark indiziert, maßgeblich für die Abhängigkeits–Relation ist jedoch der Vater–Zustand. Hierarchieabhängigkeiten existieren nicht über mehr als eine Hierarchieebenen hinweg. Diese Abhängigkeiten werden implizit über die direkte Relation behandelt.

Signalabhängigkeiten

Die wichtigsten Abhängigkeiten für die Berechnung von Thread–Prioritäten sind Signalabhängigkeiten. Signalabhängigkeiten existieren ebenfalls zwischen Zuständen. Da die Betrachtung reiner Zustände nicht immer ausreicht, muss ein Zustand für jede ausgehende Transition neu betrachtet werden und es werden 2–Tupel (Zustand, Transition) gebildet. Weshalb das notwendig ist, wird in Abschnitt 4.2.5 beschrieben. Zunächst genügt es jedoch von reinen Zuständen als Knoten auszugehen. Signalabhängigkeiten existieren nur dann, wenn sich die Zustände in unterschiedlichen Regionen befinden, die benachbart sind. Dies gilt auch hierarchieübergreifend.

Definition 5 (Signalabhängigkeit)**.** *Sei H ein hierarchischer Zustand mit zwei inneren Regionen `R0` und `R1`. Für alle Zustände — und rekursiv die dazugehörigen Kind–Zustände — U aus `R0` und V aus `R1` gilt:*

$$U <_s V \Rightarrow p(U) < p(V),$$

falls V in einer action *ein Signal emmitert oder von V aus — in einem Tick — eine ausgehende Transition erreichbar ist, wo ein Signal emittiert wird, welches in einer ausgehenden Transition oder in einer* action *von U gelesen werden kann.*

Die Signalabhängigkeiten sind für die Reihenfolge von Zuständen deshalb wichtig, da durch diese sicher gestellt wird, dass ein Signal nicht geschrieben wird, bevor es woanders gelesen wurde. Dieser Sachverhalt wird als *Readers–Writers–Problem* bezeichnet [11]. Im modifizierten `ABRO` aus Abbildung 4.8 exisitert eine Signalabhängigkeit zwischen Zustand [wA] und [wB], die besagt, dass [wA] vor [wB] ausgeführt werden muss und somit Signal `B` erst geschrieben werden muss, bevor versucht wird es zu lesen.

Transitionsabhängigkeiten

Zwischen je zwei Transitionen existieren Abhängigkeiten aufgrund ihrer Transitionsprioritäten. Da die Knoten des Abhängigkeitsbaums zur Berechnung von Thread–Prioritäten durch Zustände repräsentiert werden, muss die Abhängigkeit zwischen Transitionen ebenfalls auf Zustände projiziert werden. Aus diesem Grund wird für jede ausgehende Transition eines Zustands ein 2–Tupel (Zustand, Transition) gebildet. Diese Tupel bilden die Knoten im folgenden Abhängigkeitsbaum. Transitionsabhängigkeiten sind dem SyncChart einfach zu entnehmen, weil sie lediglich die Transitionsprioritäten eines Zustands in eine Reihenfolge abbilden.

Definition 6 (Transitionsabhängigkeit)**.** *Seien t_1 und t_2 ausgehende Transition von Zustand S mit Transitionsprioritäten ρ für t_1 und σ für t_2. Seien (S, t_1) und (S, t_2) die dazugehörigen 2–Tupel aus Zustand und Transition. Dann gilt:*

$$\rho < \sigma \Rightarrow (S, t_1) <_t (S, t_2) \Rightarrow p((S, t_1)) < p((S, t_2)).$$

4.2.2 Abhängigkeitsbäume

Nachdem alle Abhängigkeiten eines SyncCharts ermittelt wurden, lässt sich daraus ein Abhängkeitsgraph erstellen. Dieser Graph enthält vier verschiedene Kantenarten, je nach Abhängigkeit im SyncChart:

- Kontrollflusskanten
- Hierarchiekanten
- Signalkanten
- Transitionskanten

Die Knoten des Graphen werden durch die Zustände im Modell gebildet. Im Abschnitt 4.2.5 wird der Graph dann dahingehend erweitert, dass die Knoten aus einem 2–Tupel (Zustand, Transition) bestehen. Der Einfachheit halber genügt es zunächst die Knoten jeweils durch einen Zustand zu repräsentieren. Um den erstellten Graph für die Berechnung von Prioritäten zu nutzen, ist es erforderlich, dass er azyklisch ist, was im Folgenden erläutert wird.

Definition 7 (Einfacher Abhängigkeitsbaum)**.** *Ein einfacher Abhängigkeitsbaum ist ein gerichteter azyklischer Graph $G(V, E)$ mit:*

$$V = \{s_1, \ldots s_n, h_1^w, h_1^s \ldots h_m^w, h_m^s\},$$

$$E = D_c \cup D_h \cup D_s,$$

s_i*: einfache Zustände,*

h_i^w*: hierarchische Zustände, repräsentiert als schwache Zustände,*

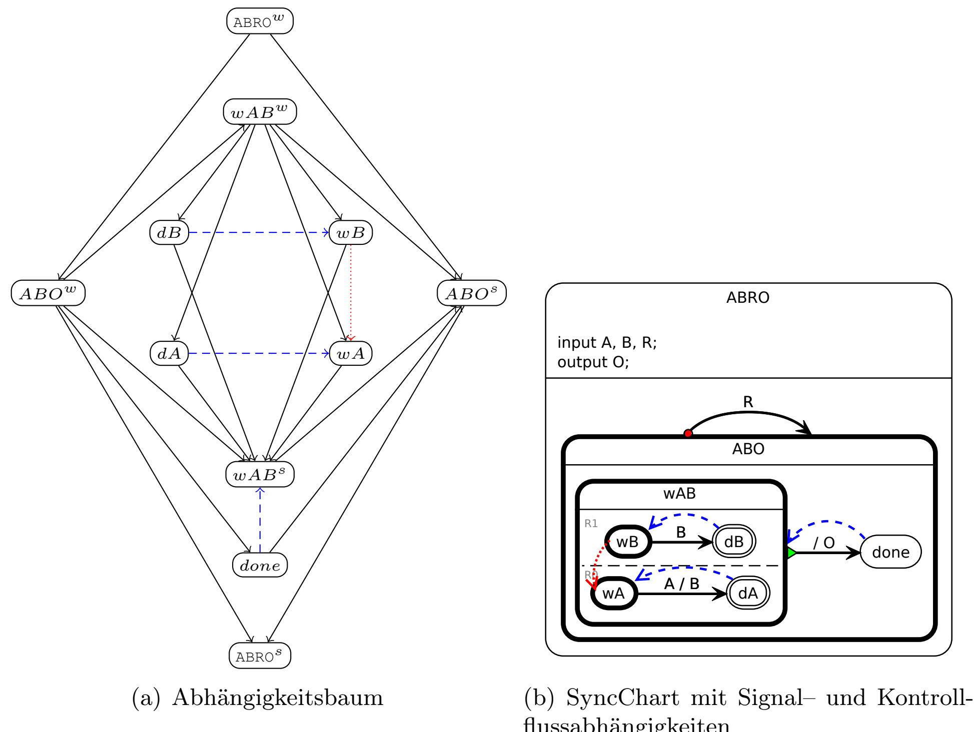
h_i^s*: hierarchische Zustände, repräsentiert als starke Zustände,*

D_c*: Menge der Kontrollflussabhängigkeiten,*

D_h*: Menge der Hierarchieabhängigkeiten und*

D_s*: Menge der Signalabhängigkeiten.*

Abbildung 4.9(a) zeigt den einfachen Abhängigkeitsbaum für das modifizierte ABRO aus Abbildung 4.8. In Abbildung 4.9(b) werden die Kanten aus Abbildung 4.9(a) schematisch im SyncChart dargestellt. Die schwarzen durchgezogenen Kanten (nur im Baum) stellen die hierarchischen Abhängigkeiten dar. Mit den blau, eng–gestrichelten Kanten werden Kontrollflussabhängigkeiten und mit den rot, gepunkteten Kanten Signalabhängigkeiten abgebildet. Nicht im Beispiel sind Transitionsabhängigkeiten. Diese werden durch grüne weit–gestrichelte Kanten illustriert. Auffällig ist, dass der Baum eine relativ symmetrische Struktur besitzt, da jeder hierarchische Zustand im SyncChart als schwach und als stark repräsentiert wird.



(a) Abhängigkeitsbaum

(b) SyncChart mit Signal– und Kontrollflussabhängigkeiten

Abbildung 4.9: Einfacher Abhängigkeitsbaum und das modifizierten ABRO mit eingezeichneten Signal– und Kontrollflussabhängigkeiten

4.2.3 Topologische Sortierung

Um Prioritäten für Threads im SC–Code zu berechnen, benötigt man eine Reihenfolge für die Zustände im SyncChart. Da für jede Region im SyncChart ein Thread in SC erzeugt wird und jeder Thread durch den initialen Zustand der Region repräsentiert wird, erhält man eine Reihenfolge für die Prioritäten der Threads. Mithilfe des Abhängigkeitsbaums, der gerichtet und azyklisch ist, lässt sich eine Reihenfolge ermitteln indem man die Knoten topologisch sortiert. Um den Begriff der topologischen Sortierung zu definieren, ist es zunächst notwendig einige Dinge einzuführen. Eine detaillierte Einführung in die Graphentheorie und der topologischen Sortierung, insbesondere für die Verwendung in Algorithmen, wird von Ottmann und Widemayer [31] beschrieben.

Definition 8 (Halbordnung)**.** *Eine binäre Relation*

$$R \subseteq V \times V$$

heißt ***Halbordnung*** *auf V genau dann wenn folgendes gilt:*

1. $\forall v \in V : (v, v) \in R$
2. $\forall v, w \in V : (v, w) \in R \wedge (w, v) \in R \Rightarrow v = w$

3. $\forall u, v, w \in V : (u, v) \in R \wedge (v, w) \in R \Rightarrow (u, w) \in R$

Lemma 1. *Sei $G = (V, E)$ ein gerichteter azyklischer Graph.*
Die Relation $R \subseteq V \times V$ mit

$$(v, w) \in R \Leftrightarrow \exists \text{ gerichteter Weg von } v \text{ nach } w$$

definiert eine Halbordnung auf V.

Definition 9 (Topologische Ordnung)**.** *Sei $G = (V, E)$ ein gerichteter azyklischer Graph mit $V = \{v_1, \ldots, v_n\}$. Eine Knotenreihenfolge $L = (v_{i_1}, \ldots, v_{i_n})$ aller Knoten aus V heißt **topologische Ordnung** von G genau dann, wenn*

$$\left(v_{i_j}, v_{i_k}\right) \in E \Rightarrow i_j < i_k.$$

Mit Hilfe der o.g. Definitionen wird ein Algorithmus zur Berechnung der topologischen Sortierung entwickelt werden. Abbildung 4.10 zeigt den verwendeten Algorithmus in Pseudocode. In Zweile 2 - 5 werden die Liste der Vorgänger–Knoten und die Ausgabeliste initialisiert. In der `for`-Schleife ab Zeile 6 wird jeder Knoten durchlaufen, bis einer gefunden wird, der keine Vorgänger besitzt. Dieser wird der Liste der sortierten Knoten hinzugefügt (Zeile 17) und für alle anderen Knoten, die diesen als Vorgänger haben, wird die Zahl der Vorgänger dekrementiert (Zeile 14 - 16), was zur Folge hat, dass neue Knoten ohne Vorgänger entstehen können. Wird kein Knoten ohne Vorgänger gefunden, existiert ein Zyklus im Graphen und es wird in Zeile 9 ein Fehler ausgegeben.

Wenn der Algorithmus abbricht, weil ein Zyklus im Graphen gefunden wurde, dann existiert ebenfalls ein Zyklus im SyncChart und es wird kein Code für dieses generiert. Es ist schnell ersichtlich, dass durch diese Restriktion sehr viele SyncCharts als zyklisch erkannt werden, obwohl diese Zyklen gutartig sind. Wenn Zyklen im Graph durch Signalflusskanten entstehen, dann ist das ein Zeichen dafür, dass auch das SyncChart Datenzyklen enthält und nicht konstruktiv ist [3]. Kontrollflusskanten bilden häufig gutartige Zyklen im Abhängigkeitsgraphen. Aus diesem Grund ist es erforderlich, dass nur die „richtigen“ Kontrollflussabhängigkeiten in den Abhängigkeitsgraphen aufgenommen werden, so dass der Graph ein Baum bleibt.

Die Identifizierung dieser „richtigen“ Kontrollflusskanten ist eine nicht triviale Aufgabe. Grundsätzlich gilt die Prämisse nur die nötigen Kontrollflusskanten in den Graphen aufzunehmen, ohne dabei Zyklen zu erzeugen. Jede überflüssige Kante bedeutet eine höhere Laufzeit für die Berechnung der Prioritäten. Da Zyklen aufgrund von Kontrollflusskanten, die aus *delayed* Transitionen resultieren, gutartig sind (diese Zyklen produzieren keine instantane Schleife), werden diese Kontrollflusskanten nicht in den Abhängigkeitsbaum aufgenommen, um nicht zu viele gutartige SyncCharts abzuweisen. Für *immediate* Transitionen werden nur dann Kontrollflusskanten erzeugt, wenn sie entweder Signale emittieren oder Signale lesen, mit denen Signalabhängigkeiten existieren. Auch von diesen Kontrollflussabhängigkeiten werden nicht

```
topologicalSort() {
  initialize resultList;
  /* count predecessors of each node */
  compute predecessorList;
  /* one loop for each node */
  for (all nodes) {
    find a source node(); /* node without predecessor */
    if(no source found) {
      error(cycle in graph!);
      break;
    } else {
      pedecessorList[source] = empty;
      /* remove all predecessor nodes of source */
      for (each node j with edge to source) {
          predecessorList[j]--;
        }
      resultList.add(source);
    }
  }
  return resultList;
}
```

Abbildung 4.10: Pseudocode für topologische Sortierung

immer alle für die Prioritätenberechnung benötigt, eine feinere Unterscheidung und Optimierung könnte die Anzahl der Kanten im Baum minimieren.

In Abbildung 4.11 ist der Graph aus 4.9 topologisch sortiert und zeigt die Reihenfolge, in welcher die Zustände im SyncChart ausgeführt werden können. Es ist zu beachten, dass die topologische Sortierung nicht eindeutig sein muss. Das liegt daran, dass nicht vorhersehbar ist, welcher Knoten als nächstes in die sortierte Menge aufgenommen wird, wenn mehrere zur Auswahl stehen. Für die Berechnung der Reihenfolge der Knoten ist dies allerdings unbedenklich, da aufgrund der Existenz aller notwendigen Abhängigkeiten alle möglichen erzeugten Abhängigkeitsbäume eine gültige Lösung liefern.

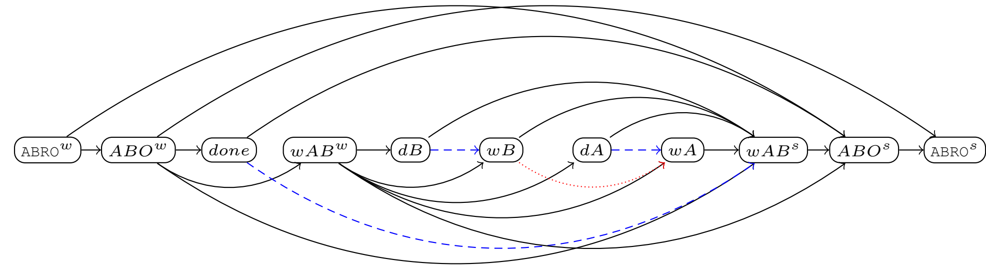


Abbildung 4.11: Topologisch sortierter Abhängigkeitsbaum von 4.9

4.2.4 Zuweisung von Prioritäten

Aus dem entstandenen Abhängigkeitsbaum und der resultierenden topologischen Sortierung können nun Prioritäten für Threads im SC–Code ermittelt werden. Jeder Knoten des Abhängigkeitsbaums bekommt eine eigene Priorität abhängig davon, an welcher Stelle er im sortierten Baum steht. Damit es keine Namenskonflikte zwischen Prioritäten von Threads und Prioritäten von Knoten gibt, wird der Begriff der *Zustandspriorität* eingeführt.

Definition 10 (Zustandspriorität). *Sei $G_S = (V, E)$ der Abhängigkeitsgraph von SyncChart S, E die Kantenmenge und V die Menge der Knoten, definiert durch die Zustände aus S. Sei $L = v_{i_1}, \ldots, v_{i_n}$ eine topologisch sortierte Folge aller Knoten aus V, $|V| = n$. Die Abbildung*

$$f : L \longmapsto \mathbb{N}$$

mit

$$f(v_{i_j}) = j$$

weist jedem Zustand aus S *den Wert seiner Position in der topologischen Ordnung zu. Dieser Wert wird als* ***Zustandspriorität*** *bezeichnet.*

Die Prioritäten für die Initialisierung der Threads können nun direkt den Zustandsprioritäten entnommen werden. Betrachtet man wieder das modifizierte ABRO und die sortierten Zustände aus Abbildung 4.11, so erhält man für die Threads folgende Werte:

- Der Main–Thread bekommt die Priorität 10 (von Knoten ABO^s).
- Der Thread für die Region in ABO bekommt die Priorität 4.
- Der Thread für die Region RO in wAB bekommt die Priorität 8.
- Und der Thread für Region R1 in wAB bekommt die Priorität 6.

Nach Minimierung der Werte für die Prioritäten, die in Abschnitt 4.3 beschrieben wird, ergeben sich für den Main–Thread Priorität 4, für den Thread in ABO Priorität 1 und für die beiden parallelen Threads in wAB die Prioritäten 2 und 3. Hier ist nun auch zu erkennen, dass der Thread für R0 eine höhere Priorität bekommt als der für R1. Somit wird in R0 Signal B geschrieben, bevor ein Lesezugriff aus R1 auftreten kann. Ein Ausschnitt des erzeugten Codes für die Thread–Initialisierung des modifizierten ABRO ist in Abbildung 4.12 zu finden.

Wenn keine Abhängigkeiten zwischen Startzuständen von parallelen Regionen bestehen, dann ist die Reihenfolge für die Vergabe der Prioritäten dennoch von dem Ergebnis der topologischen Sortierung abhängig. Für den Code vom originalen ABRO aus Abbildung 3.1 wäre es für die Threads für die beiden Regionen in wAB möglich, dass sowohl R0 eine höhere Priorität als R1 bekommt und umgekehrt, da keine keine

```
1  int tick() {
2    TICKSTART(4);
3
4  L_ABO:
5    FORK(L_wAB, 1);
6    FORKE(L_ABO_main);
7
8  L_wAB:
9    FORK(L_wA, 3);
10   FORK(L_wB, 2);
11   FORKE(L_wAB_main);
```

Abbildung 4.12: Codeausschnitt für die Thread–Initialisierung von 4.8

Signalabhängigkeit zwischen wA und wB existieren. Somit ist nicht vorhersagbar, welcher der beiden Knoten im Abhängigkeitsbaum als erstes in die sortierte Liste eingefügt wird. Das stellt allerdings auch kein Problem dar, da ohne Signalabhängigkeiten die Reihenfolge der Ausführung keine Rolle spielt. Problematisch wird es erst, wenn eine Reihenfolge existiert, diese aber zu einem späteren Zeitpunkt fehlerhaft wäre. Sprich, wenn Thread `T1` eine höhere Priorität bekommt als Thread `T2`, dann aber zu einem späteren Zeitpunkt `T2` ein Signal emittiert, welches `T1` im selben Tick lesen könnte. Die Lösung dieser Problematik wird im folgenden Abschnitt besprochen.

4.2.5 Prioritäten für das Wechseln zwischen aktiven Threads

Es gibt einige SyncCharts, für die es nicht ausreicht, die Thread–Prioritäten zu berechnen und beim Initialisieren der Threads zu vergeben. Häufig ist es der Fall, dass Prioritäten während der Ausführung wechseln müssen, um eine korrekte und deterministische Ausführung des Codes zu gewährleisten. Die folgenden zwei Abschnitte beschreiben die Szenarien, in denen ein solcher Prioritäten–Wechsel durchgeführt werden muss und wie man diese berechnet.

`PRIO`–Anweisung in hierarchischen SyncCharts

Im Abschnitt 4.2.2 wurde beschrieben, dass hierarchische Zustände sowohl als schwache als auch als starke Zustände im Abhängigkeitsbaum repräsentiert werden. Diese Unterscheidung ist notwendig, um mit Zuständen umgehen zu können, die verschiedene ausgehende Transitionen haben. Genau genommen benötigt man die Repräsentation von Zuständen als stark bzw. schwach genau dann, wenn:

- der Zustand hierarchisch ist

  und

- er mindestens eine ausgehende Transition vom Typ *strong abortion* hat

und

- mindestens eine ausgehende Transition besitzt, die keine *strong abortion* ist.

Das SyncChart aus Abbildung 4.6 beinhaltet einen hierarchischen Zustand [H] mit allen möglichen Typen von ausgehenden Transitionen. Betrachtet man den Ablauf einer möglichen Ausführung, dann können die folgenden zwei Szenarien auftreten:

1. Zustand [H] wurde betreten und im nächsten Tick liegt Signal `s2` an.
2. Zustand [H] wurde betreten und im darauf folgenden Tick liegt Signal `s1` an.

Zu Szenario 1

Wenn man davon ausgeht, dass alle Threads initialisiert wurden, dann hat der Main–Thread Priorität 4 bekommen und der Thread für den hierarchischen Zustand [H] Priorität 2. Liegt nun `s2` an, so wird der Thread mit der höchsten Priorität zuerst ausgeführt. Es wird folglich festgestellt, dass `s2` anliegt und in den Zustand [S2] gesprungen. Gleichzeitig wird [H] abortiert und Output `O` nicht emittiert. Das ist auch das erwartete Verhalten und mit dieser Prioritätenvergabe folgen richtige Ergebnisse.

Zu Szenario 2

Auch hier sind die Threads initialisiert und für die Prioritäten gilt das gleiche wie im oberen Szenario. Wenn nun `s1` anliegt, ist die richtige Ausführung, dass zuerst die innere Region von [H] ausgeführt wird und somit `O` emittiert wird. Dann kommt der äußere Thread zum Zuge und die Transition nach [S1] wird genommen. Da die Prioritäten aber so zugeteilt wurden, dass der äußere Thread eine höhere hat, wird dieser auch zuerst ausgeführt. Es wird zu [S1] gesprungen und durch das Abortieren des inneren Threads wird `O` auch nicht ausgegeben. Diese Ausführung ist fehlerhaft.

Lösung des Problems

Die beiden obigen Szenarien machen deutlich, dass es hier nicht reicht, die Prioritäten einmal festzulegen, denn es ist nicht vorhersagbar, welche Signalbelegungen bei der Ausführung auftreten. Die Lösung dieses Problem besteht darin, erst alle *strong aborts* zu überprüfen und wenn keine dieser Transitionen genommen wird, die eigene Thread–Priorität zu verringern, um inneren Threads die Möglichkeit zu geben, ausgeführt zu werden. Dafür kann man nun die Knoten für schwache und starke Zustände im Abhängigkeitsbaum verwenden. Der topologisch sortierte Baum für das oben genannte Beispiel ist in Abbildung 4.13 zu sehen. Hier sind alle inneren Zustände von [H] zwischen H^w und H^s einsortiert. Wenn man nun die Priorität

herabsetzen will, nachdem alle *strong abortions* geprüft wurden, wird der Main–Thread auf die Zustandspriorität von H^w gesetzt. Diese Priorität ist kleiner als die des inneren Threads für H .

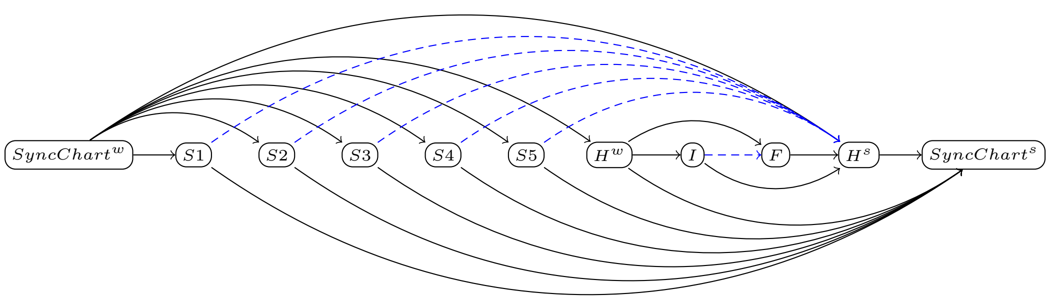


Abbildung 4.13: Topologisch sortierter Abhängigkeitsbaum von 4.6

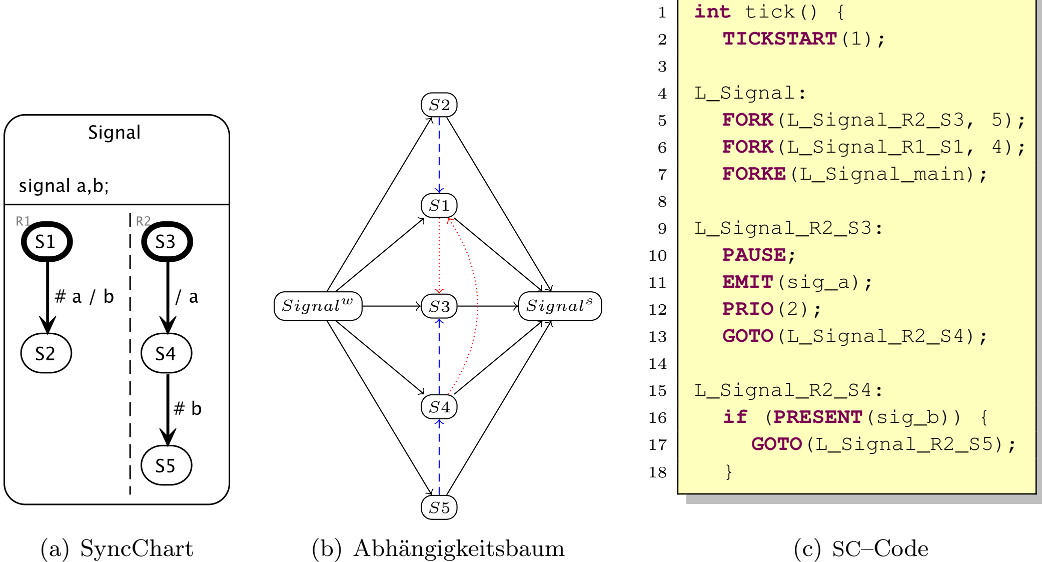
PRIO–Anweisung aufgrund von Signalabhängigkeiten

Wenn Signalabhängigkeiten in parallelen Regionen eines SyncChart existieren, dann ist es manchmal notwendig Prioritäten während der Ausführung des dazu generierten Codes zu ändern. Dies verdeutlicht das Beispiel aus Abbildung 4.14(a). In den beiden parallelen Regionen existieren zwei Signalabhängigkeiten. Es wird in R2 das Signal `a` emittiert, welches in R1 im selben Tick gelesen wird. Da nun `a` anliegt, wird `b` emittiert, was wiederum in R2 gelesen wird. Auch dies geschieht immer noch in diesem Tick.

Abbildung 4.14(b) zeigt den erzeugten Abhängigkeitsbaum für SyncChart 4.14(a). In 4.15 sieht man den selben Baum topologisch sortiert. Im Folgenden wird der Thread, der durch Region R1 gebildet wird `T1` und der, der durch R2 gebildet wird, `T2` genannt. Betrachtet man die Reihenfolge nach dem Sortieren, so muss `T2` vor `T1` ausgeführt werden. In den Zeilen 5 bis 7 von Abbildung 4.14(c) werden diese beiden Threads mit den Prioritäten der errechneten Reihenfolge erzeugt. Ohne einen Wechsel der Threads im ersten Tick würde zuerst `T2` ausgeführt und `a` emittiert, und danach in der Ausführung von `T1` `b` emittiert werden. Am Ende des Ticks wären die Zustände S2 und S4 aktiv und S5 würde nicht betreten werden.

Da dies nicht das erwartete Verhalten ist, muss nach dem Schreiben von Signal `b` in `T1` wieder zu `T2` gesprungen werden, um dort `b` zu lesen und in Zustand `S5` zu springen. Um dies zu erreichen, muss, nachdem in `T2` `a` geschrieben wurde, eine `PRIO`–Anweisung erfolgen, um die eigene Priorität zu verringern. Damit ist die Priorität von `T1` größer als die eigene. Das Selbe muss geschehen, wenn in `T1` — nach dem Lesen von `a` — `b` emittiert wird, um wieder zu `T2` zu springen.

Für die Generierung von Code aus solchen SyncCharts sind zwei Dinge wichtig.



(a) SyncChart (b) Abhängigkeitsbaum (c) SC–Code

Abbildung 4.14: SyncChart mit wechselnden Signalabhängigkeiten, der dazu gehörige Abhängigkeitsbaum und ein Teil des generierten SC–Codes

Zum einen stellt sich die Frage, wann es notwendig, ist eine `PRIO`–Anweisung zu erzeugen, um dort den aktiven Thread zu wechseln. Zum anderen muss die richtige Priorität errechnet werden, damit diese kleiner ist als die des parallelen Nachbar–Threads. Zusätzlich müssen Prioritäten so gewählt werden, dass n Threadwechsel in einem Tick zwischen m verschiedenen Threads möglich sind.

Wann muss eine `PRIO`–Anweisung erfolgen?

Ob eine `PRIO`–Anweisung für einen Thread–Wechsel notwendig ist, kann aus dem Abhängigkeitsbaum ermittelt werden. Wenn dort ein Knoten existiert, der eine ausgehende Kante besitzt, die eine Signalabhängigkeit repräsentiert (rot gepunktete Linie), dann muss im generierten Code eine `PRIO`–Anweisung folgen, nachdem die Signale emittiert wurden und bevor durch ein `GOTO` zu einem anderen Zustand gesprungen wird. Damit wird sichergestellt, dass ein anderer lesender Thread ausgeführt wird, nachdem das Signal geschrieben wurde. Wenn alle Threads mit einer höheren Priorität beendet wurden, wird der Ursprungs–Thread weiter ausgeführt und mit dem Code nach der `GOTO`–Anweisung fortgefahren.

Wie berechnet sich die Priorität für diese `PRIO`–Anweisung?

Wenn im generierten Code einer Transition eine `PRIO`–Anweisung erfolgt, befindet

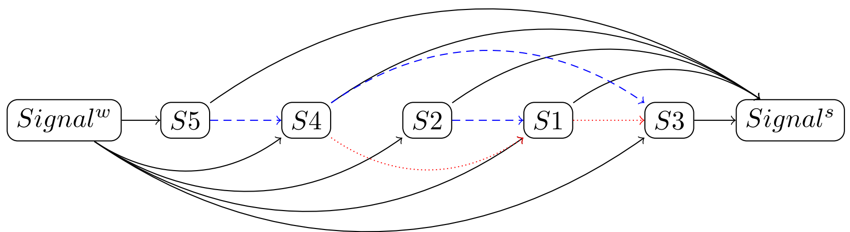


Abbildung 4.15: Topologisch sortierter Abhängigkeitsbaum von 4.14

man sich bildlich gesehen im SyncChart auf den Weg von einem Zustand in einen anderen. Der Knoten im Abhängigkeitsbaum, zu dem die Signalabhängigkeit besteht, die für den Prioritätenwechsel verantwortlich ist, hat nach der topologischen Sortierung eine kleinere Zustandspriorität als der Ausgangsknoten. Im Beispiel *Signal* aus Abbildung 4.14 ist der Ausgangsknoten $S3$ und die Signalabhängigkeit besteht zu Knoten $S1$. Die Zustände S1 und S3 definieren im generierten Code zwei Threads mit den Prioritäten 4 für `T1` und 5 für `T2`. Um den Thread zu wechseln, muss in `T2` die Priorität so gesetzt werden, dass sie kleiner ist, als die von `T1`. Wenn man sich wieder auf den topologisch sortierten Abhängigkeitsbaum bezieht, ist zu erkennen, dass $S4$ eine kleinere Zustandspriorität besitzt als $S1$, folglich kann die Priorität von `T1` auf die Zustandspriorität von $S4$ gesetzt werden. Die Auswahl von $S4$ ist jedoch nicht willkürlich. Dieser Knoten wird gewählt, weil Zustand S4 im SyncChart genau der Zustand ist, zu welchem nach dem Emittieren von `a` gesprungen wird. Da $S4$ durch die Abhängigkeit zu $S1$ im Abhängigkeitsbaum auf jeden Fall eine kleinere Zustandspriorität als $S1$ haben muss, wird durch eine `PRIO` Anweisung im Code auch der Thread gewechselt. Das gilt auch im Allgemeinen. Wenn eine `PRIO`–Anweisung aufgrund einer Signalabhängigkeit erforderlich ist um den Thread zu wechseln, dann wird diese mit der Zustandspriorität des Ziel–Zustands der Transition, die für die Abhängigkeit verantwortlich ist, ausgeführt.

Mehrere Signalabhängigkeiten eines Zustands

Da ein Zustand über mehrere Transitionen verlassen werden kann, können auch mehrere Signalabhängigkeiten mit diesem Zustand bestehen. Dies führt mit den bisher beschriebenen Abhängigkeitsbäumen zu Problemen. Betrachtet man das Beispiel aus Abbildung 4.16 so würde der einfache Abhängigkeitsbaum (siehe Abb. 4.17(a)), wie er in Abschnitt 4.2.2 definiert wurde einen Zyklus aufweisen. Der Zyklus kommt zustande, weil eine ausgehende Transition von Zustand I0 Signal `s1` emittiert, das im selben Tick an einer ausgehenden Transition von I1 gelesen werden kann. Außerdem wird durch eine ausgehende Transition von I1 Signal `s2` emittiert, welches wiederum im selben Tick von einer ausgehenden Transition von I0 gelesen werden kann. Der Zyklus besteht demnach zwischen den Zuständen I0 und I1. Mit dem

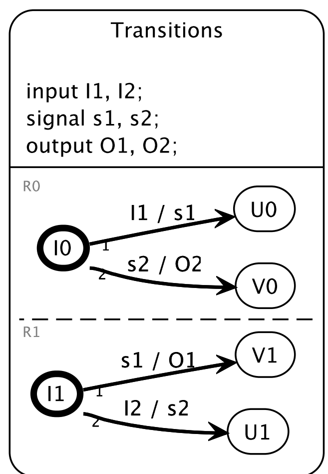


Abbildung 4.16: SyncChart

einfachen Abhängigkeitsbaum wäre folglich keine topologische Sortierung möglich, da er zyklisch ist, obwohl das SyncChart konstruktiv und logisch korrekt ist. Damit nicht zu viele SyncCharts aufgrund von gutartigen Zyklen im Abhängigkeitsbaum vom Compiler abgelehnt werden, muss dieser erweitert werden.

Definition 11 (Erweiterter Abhängigkeitsbaum). *Ein erweiterter Abhängigkeitsbaum ist ein gerichteter azyklischer Graph $G(V, E)$ mit:*

$$V \subseteq S \times T, S = \{s_1, \dots s_n, h_1^w, h_1^s \dots h_m^w, h_m^s\}, T = Transitionen,$$

$$E = D_c \cup D_h \cup D_s \cup D_t,$$

s_i*: einfache Zustände,*

h_i^w*: hierarchische Zustände, repräsentiert als schwache Zustände,*

h_i^s*: hierarchische Zustände, repräsentiert als starke Zustände,*

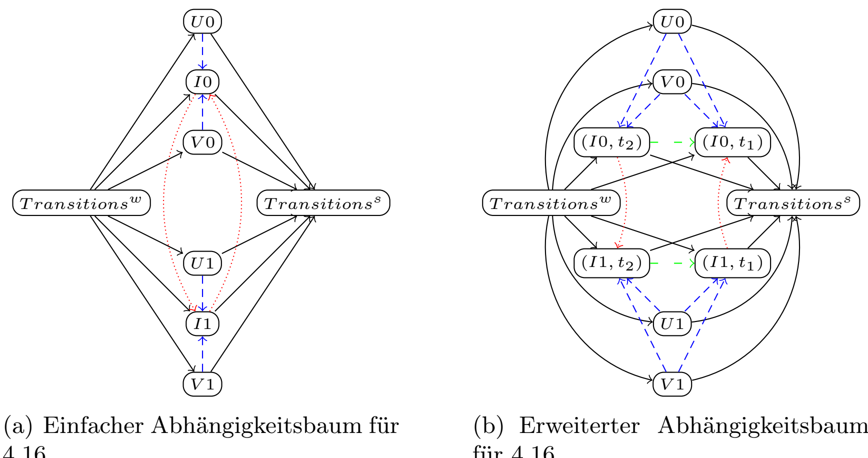
D_c*: Menge der Kontrollflussabhängigkeiten,*

D_h*: Menge der Hierarchieabhängigkeiten,*

D_s*: Menge der Signalabhängigkeiten und*

D_t*: Menge der Transitionssabhängigkeiten.*

Abbildung 4.17(b) zeigt den erweiterten Abhängigkeitsbaum für das Beispiel aus Abbildung 4.16. Da hier Knoten nicht nur durch Zustände, sondern durch 2–Tupel von Zuständen und seinen ausgehenden Transitionen repräsentiert werden, wird der Zyklus des einfachen Abhängigkeitsbaums aufgebrochen. Die zweite Komponente des Tupels definiert die Transition des Zustands abhängig von der Priorität. Hat ein Zustand (S) fünf ausgehende Transitionen, dann werden diese anhand der Prioritäten



(a) Einfacher Abhängigkeitsbaum für 4.16

(b) Erweiterter Abhängigkeitsbaum für 4.16

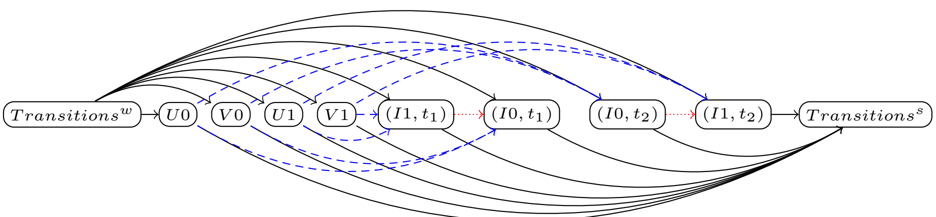
Abbildung 4.17: Einfacher und erweiterter Abhängigkeitsbaum für SyncChart 4.16

von t_1 bis t_5 durchnummeriert, so dass der Abhängigkeitsbaum fünf Knoten (S, t_1) bis (S, t_5) enthält. Der neue Baum ist azyklisch und kann deshalb topologisch sortiert werden. Somit muss das dazugehörige SyncChart nicht vom Compiler abgelehnt werden. Der Nachteil des erweiterten Abhängigkeitsbaum liegt in der Komplexität. Im schlechtesten Fall hat er im Vergleich zum einfachen Baum mit n Knoten nun $n^2 - n$ Knoten. Das schlägt sich negativ auf die Berechnungsdauer der Prioritäten nieder, wodurch sich die Laufzeit des Compilers erhöht. Dennoch ist diese Erweiterung notwendig um gutartige Zyklen aufzubrechen.

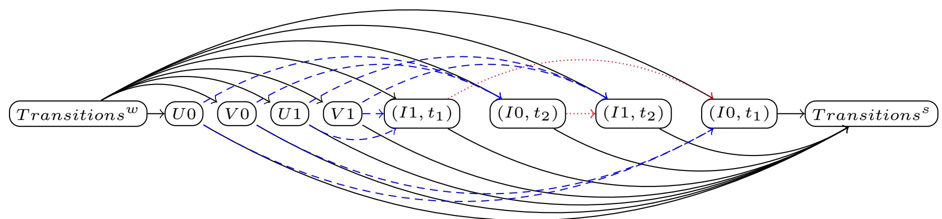
Für den erweiterten Abhängigkeitsbaum werden außerdem neue Kanten benötigt. Diese grün weit–gestrichelten Kanten repräsentieren Transitionsabhängigkeiten für ausgehende Transitionen von gleichen Zuständen. Sie sind notwendig, um den generierten Code deterministisch zu halten. Für SyncCharts ist der Determinismus unter Anderem durch Prioritäten für Transitionen gegeben. Dadurch wird sichergestellt, dass wenn im Beispiel aus Abbildung 4.16 die Inputs `I1` und `I2` anliegen, in die Zustände U0 und V1 gesprungen wird. Würden die Transitionsprioritäten nicht existieren und würden `I1` und `I2` anliegen, so könnte nicht entschieden werden, welche Transitionen ausgeführt werden müssen. Für den generierten Code muss diese Eindeutigkeit auch gelten. Das wird einmal durch `PRIO`–Anweisungen und zum anderen durch die neuen Kanten für Transitionsabhängigkeiten gewährleistet.

Zur Übersichtlichkeit soll im Folgenden erneut der Thread für R0 `T0` und der für R1 `T1` heißen. Wenn im Abhängigkeitsbaum keine Transitionsabhängigkeiten enthalten wären, könnte der Algorithmus zur Berechnung der topologischen Sortierung die Ergebnisse aus Abbildung 4.18(a) und 4.18(b) liefern. Im ersten Fall würden die Threads `T0` und `T1` mit den Prioritäten 6 und 7 erzeugt werden. Demnach würde im darauf folgenden Tick überprüft werden, ob `s1` anliegt. Da dieses Signal aber in `T0` hätte emittiert werden können, wurden die Prioritäten falsch vergeben. Die

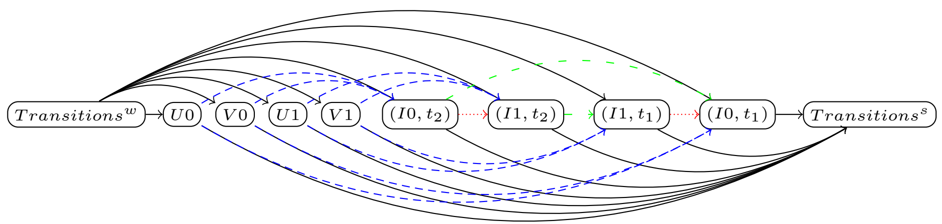
zweite Variante erzeugt die beiden Threads mit Priorität 6 für `T1` und 7 für `T0`. Angenommen Input `I1` liegt an, dann wird `s1` emittiert. Durch eine `GOTO`–Anweisung nach Überprüfung der Transitionsbedingung wird `T0` pausiert und mit `T1` fortgefahren. Dieses mal liegt `s1` an, es wird `O` ausgegeben und der Thread beendet. Zum Schluss wird der Thread `T0` weiter ausgeführt und beendet. Somit ist das gewünschte Verhalten des SyncCharts im Code umgesetzt.



(a) Variante 1: ohne Transitionsabhängigkeiten



(b) Variante 2: ohne Transitionsabhängigkeiten



(c) Variante 3: mit Transitionsabhängigkeiten

Abbildung 4.18: Topologisch sortierte Abhängigkeitsbäume für 4.16 sowohl mit als auch ohne Transitionsabhängigkeiten

Man kann sich nicht darauf verlassen, dass der richtig topologisch sortierte Baum als Ergebnis erzeugt wird. Eine zusätzliche Berechnung zur Prioritätenvergabe würde das vorgestellte Konzept unnötig komplizierter werden lassen, da man nun verschiedene Fälle betrachten müsste. Deshalb werden die zusätzlichen Kanten im erweiterten Abhängigkeitsbaum benötigt. Wenn man die Knoten $I0$ und $I1$ für die Startzustän-

de I0 und I1 im sortierten Baum aus Abbildung 4.18(c) betrachtet, dann wird deutlich, dass durch diese neu eingefügten Kanten nur eine einzige Sortierung dieser Knoten möglich ist. Diese Sortierung liefert das richtige Ergebnis für die Erzeugung von Threads, und damit ist der Determinismus auch im Code hergestellt.

```
1  int tick() {
2    TICKSTART(1);
3
4  L_Transitions:
5    FORK(L_Transitions_R1_I1, 5);
6    FORK(L_Transitions_R0_I0, 6);
7    FORKE(L_Transitions_main);
8
9  L_Transitions_R1_I1:
10   PAUSE;
11   if (PRESENT(sig_s1)) {
12     EMIT(sig_O1);
13     GOTO(L_Transitions_R1_V1);
14   }
15   if (PRESENT(sig_I2)) {
16     EMIT(sig_s2);
17     PRIO(2);
18     GOTO(L_Transitions_R1_U1);
19   }
20   GOTO(L_Transitions_R1_I1);
21
22 L_Transitions_R1_V1:
23   HALT;
24
25 L_Transitions_R1_U1:
26   HALT;
```

```
27
28 L_Transitions_R0_I0:
29   PAUSE;
30   if (PRESENT(sig_I1)) {
31     EMIT(sig_s1);
32     PRIO(3);
33     GOTO(L_Transitions_R0_U0);
34   }
35   PRIO(4);
36   if (PRESENT(sig_s2)) {
37     EMIT(sig_O2);
38     GOTO(L_Transitions_R0_V0);
39   }
40   GOTO(L_Transitions_R0_I0);
41
42 L_Transitions_R0_U0:
43   HALT;
44
45 L_Transitions_R0_V0:
46   HALT;
47
48 L_Transitions_main:
49   HALT;
50
51   TICKEND;
52 }
```

Abbildung 4.19: Generierter Code für das Beispiel aus 4.16

Es ist nun noch wichtig zu erkennen, wann eine `PRIO`–Anweisung ausgeführt werden muss, wenn ein Zustand mehrere Signalabhängigkeiten aufweist. Da es notwendig sein kann vor der Überprüfung einer Transitionsbedingung den Thread zu wechseln, muss auch die `PRIO`–Anweisung vor dem generierten Code für diese Transition stehen. Dies ist nur dann notwendig, wenn ein Zustand mehr als eine ausgehende Transition besitzt, zu welchen Signalabhängigkeiten bestehen. Der Thread soll aber nicht nach jeder überprüften Transition gewechselt werden, sondern nur dann, wenn dies notwendig ist. Deshalb muss untersucht werden, ob in der zu überprüfenden Transition ein Signal gelesen wird, das in einem parallelen Thread geschrieben werden kann. Nur dann erfolgt der Thread–Wechsel über eine `PRIO`–Anweisung.

Die Priorität, die der Thread bekommt, um einen anderen den Vortritt zu lassen, berechnet sich aus der Zustandspriorität für den aktuellen Knoten und die Transition, die für den Wechsel verantwortlich ist. Diese Priorität muss kleiner sein als die aktuelle, da eine Signalabhängigkeit zu einem Knoten in einem anderen Thread besteht, und die Zustandspriorität dieses anderen Knoten per Definition größer ist. Im

generierten Code für Transitions aus Abbildung 4.19 werden in den Zeilen 5 und 6 die Threads für die Regionen mit den Prioritäten erzeugt, wie sie im erweiterten Abhängigkeitsbaum berechnet wurden. Die `PRIO`–Anweisung aus Zeile 35 erzwingt den Wechsel zum Nachbar–Thread aufgrund der Signal– und Transitionsabhängigkeiten. Auch hier ist zu beachten, dass die Prioritäten nur relativ mit den Zustandsprioritäten übereinstimmen, da bereits Optimierungen eingeflossen sind.

Es ist nicht einfach zu definieren, welche Klasse von SyncCharts vom Compiler übersetzt werden kann. Die für die Arbeit betrachteten und verwendeten SyncCharts weisen alle eine korrekte Ausführung auf. Dennoch kann nicht sicher von der Hand gewiesen werden, dass für nicht konstruktive Programme Code generiert wird. Für den Compiler existiert ein Validierungsmechanismus, der in Abschnitt 6.3 beschrieben wird. Dieser Mechanismus kann als Anhaltspunkt für die Menge von korrekten Übersetzungen verwendet werden.

4.3 Optimierungen

Die Optimierung des generierten Codes ist ein weites, und wichtiges Thema. Es ist schwierig den optimalen Code zu erzeugen, abgesehen davon, dass nicht klar ist was optimaler SC–Code ist (Laufzeit, Größe, Speicherbedarf). Die durchgeführten Optimierung sind eine Ansammlung von Prinzipien für bestimmte Probleme. Dieser Abschnitt soll hauptsächlich zeigen, dass in diesem Bereich viel möglich ist und mit einfachen Regeln signifikante Verbesserungen erzielt werden können.

4.3.1 Vermeidung von unerreichbaren oder redundanten Code

Einer der Gründe für die Verwendung von SC ist seine Kompaktheit. Gerade für eingebettete Systeme ist Speicherplatz oft sehr begrenzt. Aus diesem Grund ist es auch wichtig, dass der Compiler möglichst kompakten Code erzeugt. Sicherlich wird man nicht — oder nur sehr schwer — kompakteren Code generieren, als es durch die Erzeugung von SC–Code per Hand möglich ist. Dennoch kann man mit einigen einfachen Regeln unerreichbaren oder doppelten Code einsparen.

Unerreichbarer Code

Es kommt häufig vor, dass Code erzeugt wird, der niemals ausgeführt wird. Wenn man sich das Schema für die Codegenerierung von Zuständen aus Abbildung 4.2 verdeutlicht, wird klar, dass dieses allgemeine Layout für viele Zustände mehr Informationen beinhaltet, als benötigt. So ist es beispielsweise möglich, dass SyncCharts modelliert werden, für die Code erzeugt wird, der niemals ausgeführt werden kann. Existiert beispielsweise eine Transition ohne Trigger, dann muss für alle anderen Transitionen mit niedrigerer Transitionspriorität kein Code erzeugt werden.

```
1  Label:
2    EMIT(<signal>);
3    GOTO(LabelOne);
4    if (PRESENT(<signal>) {
5      <code>
6    }
7    GOTO(Label_intern);
```

⟹

```
1  Label:
2    EMIT(<signal>);
3    GOTO(LabelOne);
```

Ähnliches gilt für Zustände ohne ausgehende Transitionen. Hier benötigt man keinen initialen und internen Teil. Außerdem ist auch hier ein Rücksprung nicht notwendig, da dieser durch eine HALT–Anweisung ersetzt wird.

```
1  Label:
2  Label_intern:
3    HALT;
4    GOTO(Label_intern);
```

⟹

```
1  Label:
2    HALT;
```

Redundanter Code

Durch die generelle Struktur von Zuständen kommt es immer wieder vor, dass Code dupliziert wird. Dieser doppelt erzeugte Code führt zwar nicht dazu, dass die Ausführung fehlerhaft ist, erhöht aber die Codegröße unnötig. Ein häufig auftretendes Szenario, für welches die Optimierung einfach ist, ist ein Makrozustand mit mehreren ausgehenden Transitionen, welche alle *immediate* sind. Wie in Abschnitt 4.1.2 beschrieben, wird für *immediate* Transitionen sowohl im initialen als auch im internen Teil Code generiert. Das ist auch notwendig, damit diese Transitionen sowohl beim Betreten des Zustandes als auch zu einem späteren Zeitpunkt ausgeführt werden kann. Wenn jedoch ein Zustand ausschließlich *immediate* Transitionen besitzt oder alle *immediate* Transitionen eine höhere Priorität besitzen als die restlichen Transitionen, dann ist ein interner Teil nicht notwendig. Es ist lediglich die richtige Anordnung der PAUSE-Anweisung zu beachten. Die folgenden Codefragmente veranschaulichen dies.

```
1  Label:
2    <# transition 1>;
3    ...
4    <# transition n>;
5  Label_intern:
6    PAUSE;
7      <# transition 1>;
8      ...
9      <# transition n>;
10 GOTO(Label_intern);
```

⟹

```
1  Label:
2    <# transition 1>;
3    ...
4    <# transition n>;
5    PAUSE;
6  GOTO(Label);
```

4.3.2 Makros zur Minimierung der Codegröße

Um lauffähigen Code zu erzeugen genügt es, sich auf eine Teilmenge aller SC–Makros zu beschränken. Dies vereinfacht auch die Regeln für den Compiler. Dennoch kann man durch Ersetzung einiger Makros die Codegröße minimieren. Ein Beispiel hierfür soll ein Zustand liefern, der lediglich eine ausgehende Transition besitzt, welche einen einfachen Trigger hat. Hier kann man den `PAUSE - IF - GOTO`–Block zu einer `AWAIT`–Anweisung verkürzen. Es ist zu beachten, dass der verkürzte Code nach der Makro–Expansion nicht kleiner ist. Einen Vorteil bringt diese Regel jedoch für Lesbarkeit und für eine eventuelle Nutzung des Codes in einer virtuellen SC-Maschine.

```
Label:
  PAUSE;
  if (PRESENT(<signal>)) {
    GOTO(TargetLabel);
  }
  GOTO(Label);
```

⟹

```
Label:
  AWAIT(<signal>);
  GOTO(TargetLabel);
```

4.3.3 Optimierung von Sprunganweisungen

Häufig kommt es vor, dass Sprunganweisungen getätigt werden, ohne dass sie notwendig wären, weil zum Beispiel die Sprungadresse ohne Sprung erreichbar ist. Es gibt auch Situationen, in denen ein `GOTO` vermieden werden kann, indem man den Code entsprechend umstrukturiert. Zwei dieser einfachen Optimierungsmöglichkeiten werden im Folgenden beschrieben.

Wenn man eine Kette von Zuständen hat, in der jeder Zustand jeweils durch eine Transition mit einem Folgezustand verbunden ist und die Transition keinen Trigger besitzt, dann kann man darauf verzichten mit einer `GOTO`–Anweisung zum nächsten Zustand zu springen, da dieser Codeabschnitt ohnehin direkt danach ausgeführt werden würde. Hier kann nicht benötigter Code eingespart werden.

```
Label_One:
  <effect>
  GOTO(Label_Two);
GOTO(Label_One);

Label_Two:
  <code>
  GOTO(Label_Two);
```

⟹

```
Label_One:
  <effect>

Label_Two:
  <code>
  GOTO(Label_Two);
```

Unter gewissen Umständen ist es möglich, auf eine Sprunganweisung zu verzichten, wenn man den auszuführenden Code vom Ziel in den Code der Quelle des Sprungs verschiebt. Dabei ist es jedoch wichtig, dass dies nur passiert, wenn der Code des Ziels nicht durch einen anderen Sprung erreichbar sein muss. Das bedeutet, dass der

Knoten, für den der Ziel–Code erzeugt wird nur eine eingehende Transition, nämlich genau die vom Quell–Knoten, besitzen darf. Diese Optimierung kann zwar die Größe vom generierten Code minimieren, wirkt sich allerdings negativ auf die Lesbarkeit aus. Daher wird die Optimierung nur dann durchgeführt, wenn der Zielcode einfach genug ist. Einfach bedeutet, dass der Ziel–Zustand nur eine ausgehende Transition ohne Trigger, oder eine *entryAction* und keine Transition besitzt.

```
1  Label_Source:
2    PAUSE;
3    if (<trigger>) {
4      <effects>
5      GOTO(Label_Target);
6    }
7    GOTO(Label_Source);
8
9  Label_Target:
10   <code target>
```

⟹

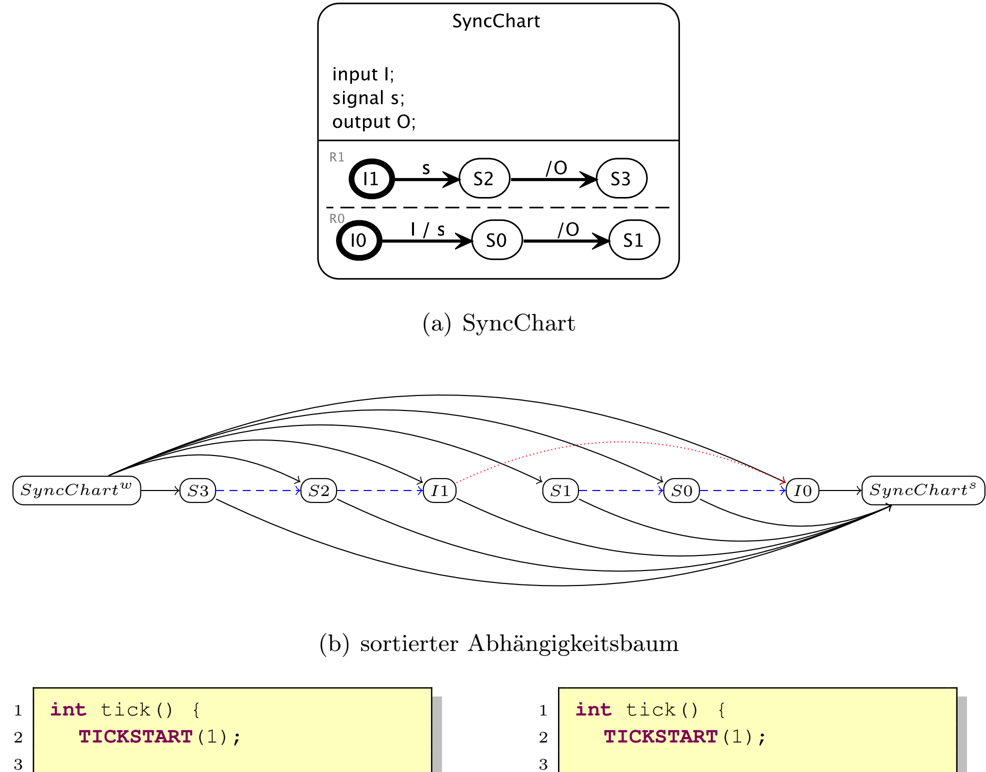
```
1  Label_Source:
2    PAUSE;
3    if (<trigger>) {
4      <effects>
5      <code target>
6    }
7    GOTO(Label_Source);
```

4.3.4 Prioritäten und PRIO–Anweisungen

Eines der wichtigsten Optimierungsfelder für SC ist die Optimierung von Prioritäten und PRIO–Anweisungen. Der Grund dafür liegt in der Beschaffenheit von SC. Zum einen kostet die Ausführung von PRIO–Anweisungen viel Zeit und unnötige PRIOs wirken sich negativ auf auf die Ausführungszeiten des generierten Codes aus. Zum anderen soll das erzeugte Programm so kompakt wie möglich sein. Um dies zu gewährleisten, werden beispielsweise die Prioritäten für Threads in einem Bitvektor gespeichert, wenn das möglich ist. So kann ein 4–Byte Integer Prioritäten von 0 bis 31 darstellen. Wenn ein Thread mit Priorität 32 erzeugt werden muss, dann reicht der 4–Byte Integer nicht mehr aus und die Prioritäten müssen in Arrays gespeichert werden. Dies beeinflusst sowohl die Laufzeit als auch die Größe des Programms negativ.

Minimierung von Prioritäten für Threads

In Abschnitt 4.2.4 wurde gezeigt, wie man von Zustandsprioritäten zu Thread–Prioritäten im generierten SC–Code kommt. Dabei wurde schnell klar, dass nicht alle Zustandsprioritäten für den Code benutzt werden müssen, da nicht jeder Zustand einen Thread erzeugt. Das spiegelt sich darin wieder, dass ein Thread zum Beispiel mit Priorität 10 erzeugt wird und diese während seiner Ausführung auf 3 reduziert wird. Angenommen es würde nur noch einen zweiten Thread mit Priorität 5 geben, dann hätte man Prioritäten bis 10 verbraucht, wovon aber nur drei benötigt werden. Um dies zu verhindern, werden die Prioritäten so gestaucht, dass keine Priorität ungenutzt bleibt und somit versucht werden kann, ein kleineres Format, wie der oben beschriebene 4–Byte Integer, zu verwenden.



(a) SyncChart

(b) sortierter Abhängigkeitsbaum

```
1  int tick() {
2    TICKSTART(1);
3
4  L_SyncChart:
5    FORK(L_SyncChart_R0_I0, 7);
6    FORK(L_SyncChart_R1_I1, 4);
7    FORKE(L_SyncChart_main);
```

⟹

```
1  int tick() {
2    TICKSTART(1);
3
4  L_SyncChart:
5    FORK(L_SyncChart_R0_I0, 3);
6    FORK(L_SyncChart_R1_I1, 2);
7    FORKE(L_SyncChart_main);
```

(c) Code für Thread–Initialisierung unoptimiert (links) und optmiert (rechts)

Abbildung 4.20: SyncChart für das zwei Threads generiert werden (inkl. Abhängigkeitsbaum und Codeausschnitt)

Dazu wird zuerst der Code mit Prioritäten, resultierend aus den Abhängigkeitsbäumen, erzeugt. Wenn die SC–Datei generiert wurde, wird der Code einmal komplett eingelesen und die Prioritäten von Thread–Initialisierung und PRIO–Anweisung ermittelt. Diese Prioritäten werden sortiert in eine Liste eingefügt, so dass nach dem kompletten Parsen eine sortierte Liste aller verwendeten Prioritäten existiert. Diese Liste fungiert nun als Muster für Paare von alten und neuen Prioritäten. Die alte Priorität ist das Element der Liste und die neue der Index. Nun kann der generierte Code ein zweites mal durchlaufen werden und alle alten Prioritäten werden durch neue ersetzt. Abbildung 4.20(a) und 4.20(b) zeigen ein einfaches SyncChart mit zwei parallelen Regionen und den generierten Abhängigkeitsbaum in topologischer Sortierung. In Abbildung 4.20(c) ist der dazu generierte Code mit unoptimierten Prioritäten dem optimierten gegenüber gestellt.

Vermeidung unnötiger PRIO–Anweisungen

Auch bei der Generierung von PRIO–Anweisungen kann sowohl bei der Codegröße als auch bei der Performance gespart werden. Jedes PRIO bedeutet zeitliche Einbußen und lässt den Code unnötig anwachsen. Außerdem kann durch nicht benötigte PRIO–Anweisungen die Gesamtzahl der Prioritäten ansteigen, was zu den oben genannten negativen Eigenschaften führt. Solche überflüssigen Anweisungen zu identifizieren ist jedoch nicht immer einfach. Ein eher selten auftretender Fall, der einfach zu optimieren ist, sind zwei aufeinander folgende PRIO–Anweisungen mit der selben Priorität.

Ein häufig auftretender Fall ist die Codegenerierung für einen hierarchischen Zustand, der nur ausgehende *strong abortions* hat. In Abschnitt 4.2.5 wird beschrieben, wie für hierarchische Zustände mit *strong aborts* nach der Überprüfung dieser Transitionen die Priorität herab gesetzt wird, um den Thread für den inneren Teil des Zustands auszuführen. Gibt es keine *normal terminations* oder *weak aborts*, dann muss auch die Priorität nicht runter gesetzt werden, weil der innere Thread nach der Überprüfung der Transitionen zum Zuge kommt, wenn die Transition nicht zu *true* ausgewertet wird. Das folgende Beispiel verdeutlicht das.

```
1 Label:
2   <# strong aborts>
3   PRIO(<weak>);
4 Label_intern:
5   PRIO(<strong>);
6   PAUSE;
7   <all strong aborts>
8   GOTO(Label_intern);
```

⟹

```
1 Label:
2   <# strong aborts>
3 Label_intern:
4   PAUSE;
5   <all strong aborts>
6   GOTO(Label_intern);
```

4.3.5 Sonstige Optimierungen

Zusätzlich zu den oben genannten Optimierungen sind weitere Verbesserungen möglich. Es lässt sich beispielsweise Platz sparen, indem man die Arrays für *valued signals* nur so groß initialisiert, wie sie auch wirklich verwendet werden. Der erste naive Weg ist die Initialisierung mit der Anzahl aller Signale. Das funktioniert zwar, hat aber den Nachteil, dass für ein SyncChart mit n Signalen auch ein Integer–Array mit n Indizes erstellt werden muss. Das gleiche gilt für das Array von pre–Signalen mit Wert. Auch hier muss das Array nur so groß initialisiert werden, wie benötigt. Zudem kann es komplett weggelassen werden, wenn das SyncChart keine pre–Zugriffe enthält.

Einen weiteren Spezialfall stellt der oberste Zustand eines SyncCharts dar. Dieser Container enthält keine ausgehenden Transitionen. Deshalb benötigt man auch keinen generierten Code für diesen Zustand. Das gilt natürlich nicht für die inneren Regionen und Zustände. Außerdem muss für die innere Region des Root–Zustands

kein neuer Thread erzeugt werden, wenn er nur eine Region enthält. In diesem Fall ist die einzige innere Region bereits durch den Main–Thread abgedeckt. Einen etwas aufwendigeren Optimierungsschritt zieht ein Root–Zustand mit $n > 1$ inneren Regionen nach sich. Hier reicht es, wenn man einen der Threads für die Regionen in den Main–Thread verlagert und nur für die restlichen $n - 1$ Regionen einen Thread mittels `FORK` erzeugt. Für Beispiel 4.20(a) würde sich dann folgende Änderung ergeben. Diese Optimierung ist aktuell noch nicht enthalten, zeigt aber welche Möglichkeiten noch vorhanden sind.

```
1  int tick() {
2    TICKSTART(1);
3
4  L_SyncChart:
5    FORK(L_SyncChart_R0_I0, 3);
6    FORK(L_SyncChart_R1_I1, 2);
7    FORKE(L_SyncChart_main);
```

$\Longrightarrow$

```
1  int tick() {
2    TICKSTART(2);
3
4    FORK(L_SyncChart_R1_I1, 1);
5    FORKE(L_SyncChart_main);
```

4.4 Nicht unterstützte SyncCharts–Elemente

SyncCharts beinhalten viele Features, wodurch es möglich ist, komplexe Sachverhalte zu modellieren. Der Compiler unterstützt nicht alle dieser speziellen Features. Im Folgenden werden diese wenigen Elemente aufgeführt und beschrieben, wie eine Umsetzung aussehen kann.

4.4.1 OnExit actions

Der Compiler beinhaltet Übersetzungsregeln für *on entry*– und *on inside actions*. *On exit actions* werden aktuell nicht unterstützt. Diese Entscheidung hat mehrere Gründe. Das semantische Verhalten von *on exit actions* [3] ist sehr komplex. Für *on exit actions* auf oberster Ebene wäre es einfach möglich den Code beim Verlassen des Zustands über eine Transition auszuführen. Damit wäre aber nur ein Teil des möglichen Verhaltens abgedeckt. Problematisch wird es, wenn eine *on exit action* in einer tieferen Hierarchie Ebene existiert. Hier muss diese auch dann ausgeführt werden, wenn ein äußerer Zustand über eine Transition verlassen wird. Dabei werden jedoch alle inneren Threads terminiert und somit würde auch die Information der *on exit action* verloren gehen. Demnach muss für eine hierarchisch beliebig tief liegende *on exit action* zu jeder ausgehenden Transition von umschließenden Makrozuständen zusätzlich Code generiert werden. Da es möglich ist die Verwendung von *on exit actions* zu umgehen, indem man beim Modellieren diese *action* dort zu den Transitionen hinzufügt, wo sie auftreten kann, soll dies als Ansatz für eine Implementierung dienen. SC realisiert die Verwendung von *on exit actions*, indem für jede auftretende *on exit action* eine Funktion implementiert wird, die dann an den Stellen, wo die *action* auftreten kann, aufgerufen werden muss.

4.4.2 Valued Signals

In SyncCharts gibt es *valued signals* für die Typen *Integer*, *Bool* und *Float*. Sowohl SC als auch der Compiler unterstützen momentan Integer und Bool. Dabei wurde die Funktionalität von Bool und den verschiedenen Kombinationen wie Multiplikation, Division, *and* und *or* für Integer und Bool im Zuge der Implementierung des Compilers zu SC hinzugefügt. Boolesche Signale werden intern durch die Verwendung von Integer abgedeckt. Dies kann natürlich durch Verwendung vom Datentyp *Char* weiter optimiert werden. Für den *Proof of Concept* genügt jedoch die aktuelle Implementierung. Zudem ist es vergleichsweise einfach SC und den Compiler mit weiteren Datentypen und Kombinationen beliebig zu erweitern. Damit ist es möglich den SC über den Umfang von SyncCharts hinaus auszubauen.

4.4.3 Host Code

Ein weiteres großes Feld ist *Host Code*. Um die komplette Bandbreite an Möglichkeiten abzudecken, welche die Verwendung von Host Code bietet, muss ein erheblicher zeitlicher Aufwand investiert werden. Host Code kann teilweise komplex sein und auf verschiedene Art und Weise verwendet werden. Außerdem kann *Host Code* für unterschiedliche Programmiersprachen verwendet werden, was zusätzliche Compiler erfordern würde. Aktuell akzeptiert der SC–Compiler C–Code für Zustände indem der Code direkt an diese Stelle eingebunden wird. Solange dort keine Signale oder `PRIO`–Anweisungen verwendet werden, gibt es keine Konflikte zu bestehenden Abhängigkeiten. Um den vollen Umfang von *Host Code* zu unterstützen, ist es erforderlich den Code zu analysieren und eventuelle Abhängigkeiten aufzulösen.

4.4.4 Reference States

Referenz–Zustände können verwendet werden, um gleiche Sachverhalte nicht mehrfach modellieren zu müssen oder große Makrozustände auszulagern, um das SyncChart kompakter und besser lesbar zu halten. Da eine der Hauptaufgaben für den generierten Code die Anbindung und Simulation im KIELER System (siehe Kapitel 6) ist, werden *reference states* aktuell nicht berücksichtigt. Der Grund dafür liegt darin, dass eine Simulation dann schwer durchführbar ist, da der referenzierte Zustand nicht im selben Diagramm liegen muss. Ein weiterer Grund für das Fehlen dieses Features ist die Tatsache, dass für SyncCharts mit referenzierten Zuständen eine Transformation möglich ist, welche diese Referenzen auflöst. Nach dieser Modell–Transformation existieren keine *reference states* mehr und der Compiler kann das SyncChart auf übliche Weise verarbeiten. Dieser Mechanismus befindet sich aktuell in der Entwicklung und kann vom Compiler verwendet werden ohne Änderungen vorzunehmen. Es ist auch möglich, *reference states* beim Übersetzen selbst aufzulösen und den dafür generierten Code direkt in die SC–Datei einzufügen. Um eine Kollision für die Simulation zu vermeiden, muss dieses Feature dazu abgeschaltet werden.

5 Implementierung und verwendete Techniken

Der SC–Compiler verarbeitet SyncCharts, die im Kiel Integrated Environment for Layout Eclipse Rich-Client (KIELER) (siehe 6.1) modelliert sind. Diese Modelle sind Instanzen des SyncCharts–Metamodells, welches wiederum ein EMF–Ecore–Modell ist. In diesem Kapitel wird beschrieben, welche Techniken für die Entwicklung des Compilers verwendet wurden und wo diese einzuordnen sind.

5.1 Die Eclipse Plattform

Einst als Entwicklungsumgebung für die Programmiersprache Java entwickelt, ist Eclipse heute ein mächtiges Werkzeug zur Programmierung für eine große Anzahl von Programmiersprachen. Eclipse basiert seit der Version 3.0 auf dem OSGi–Framework. Damit lassen sich unabhängige Anwendungen — basierend auf dem Eclipse Framework — schreiben, ohne an die komplette Eclipse–IDE gekoppelt zu sein. Bestandteil dieser Anwendungen sind verschiedene Plugins, die modular hinzugefügt oder entfernt werden können. Abbildung 5.1 zeigt eine Übersicht der Eclipse Plattform. Das wichtigste Plugin einer Java–basierten Rich Client Application (RCA) ist das Plugin Developer Environment (PDE). PDE setzt die Spezifikationen der Open Services Gateway initiative (OSGi) um, so dass das bestehende System durch beliebige Plugins erweitert werden kann [13].

Auch das KIELER–System, in das der Compiler vollständig integriert ist, basiert auf der *Eclipse Rich Client Platform*. KIELER wurde während der Arbeit um zwei Plugins erweitert. Zum einen ist das der Compiler, der für die reine Codegenerierung verantwortlich ist. Zum anderen wurde ein Simulator entwickelt, der den generierten Code visualisiert (siehe Abschnitt 6.2). Die Simulation ist hauptsächlich in Java realisiert und teilt sich in zwei Teile auf. Einmal die Anbindung an KIELER, speziell an KIELER Execution Manager (KIEM) und zum Zweiten die Kommunikation zwischen dem generierten Code und KIEM. In Abschnitt 6.1 wird die Funktionsweise dieser Plugins beschrieben und mit Beispielen illustriert. Das größere Plugin ist das zur Erzeugung von Code. Ein Teil dieses Plugins ist ebenfalls in Java implementiert, der andere Teil verwendet die Sprachen *Xpand* und *Xtend*. In den nächsten Abschnitten wird darauf näher eingegangen.

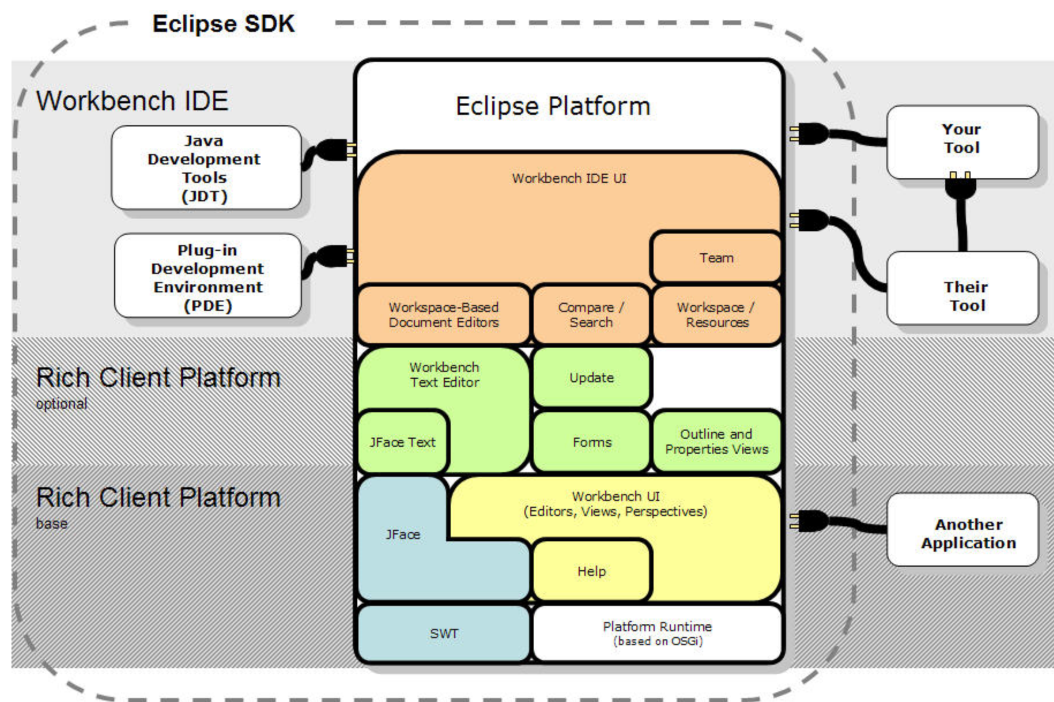


Abbildung 5.1: Eclipse Plattform im Überblick [12]

5.2 Das Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) ist ein Java–Framework zum Erstellen von Anwendungen auf Basis strukturierter Modelle. Ausgehend von diesen Modellen kann Java Code erzeugt werden, mit dessen Hilfe Modellinstanzen erstellt werden können [38]. Die Bestandteile von EMF sind unter anderem Java Emitter Templates (JET), ein Framework für die Generierung von Code, und das Ecore–Metamodell, eine Implementierung der Spezifikationen der Meta Object Facility (MOF). Abbildung 5.2 zeigt — mit Bezug zu KIELER— vier Ebenen die MOF definiert, um Daten abzustufen. An oberster Stelle steht EMF, das die Meta–Meta–Modell–Ebene beschreibt. Eine Abstraktionsebene tiefer befindet sich das SyncCharts–Metamodell in vereinfachter Form. Es definiert die Struktur der konkreten Modell–Instanzen, welche in der M3–Ebene anzufinden sind. Diese Instanzen sind die SyncCharts, die der Compiler verarbeitet. Die unterste Ebene ist eher schematisch zu sehen und beschreibt die reaktiven Systeme die mit SyncCharts modelliert werden können. Dies kann dann beispielsweise in Form des generierten Codes der Fall sein.

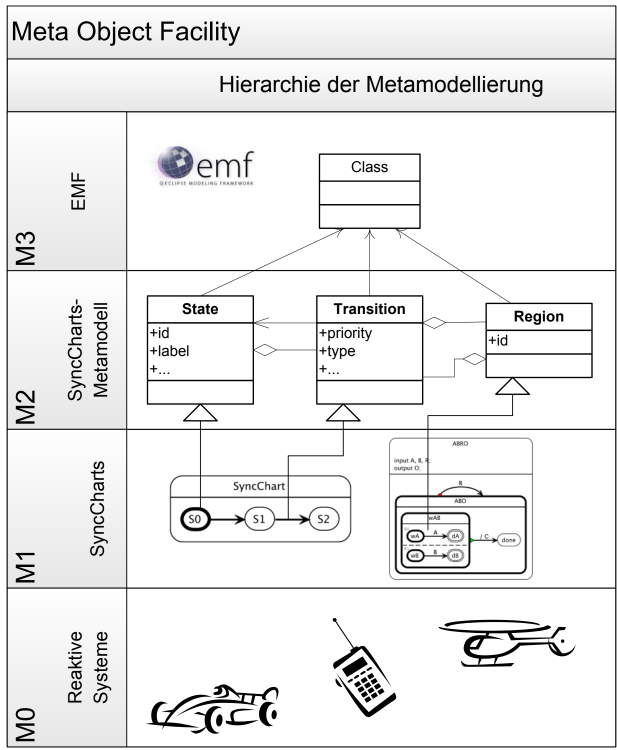


Abbildung 5.2: Hierarchie der Metamodellierung

5.3 Xpand, Xtend und Check

Xpand, Xtend und Check [22, 40] gehören zur OpenArchitectureWare (oAW)–Sprachfamilie[1], eine Plattform für modellgetriebene Softwareentwicklung. Xpand ermöglicht es, Templates zu definieren, mit denen aus graphischen Modellen Text generiert werden kann. Xtend kann genutzt werden, um bestehende Metamodelle mit zusätzlicher Logik und Funktionalität zu erweitern. Das erlaubt es, ein Metamodell klein und übersichtlich zu halten und dennoch für beliebige Anwendungsfälle flexibel nutzbar zu machen. Check wird benutzt, um Validierungsregeln auf Modellen zu definieren. Damit lassen sich Modelle statisch validieren, um sie beispielsweise für die Generierung von Code vorzubereiten. Alle drei Sprachen bestimmen einen großen Anteil für die Entwicklung des Compilers.

Check

Das SyncCharts–Metamodell ist verhältnismäßig allgemein gehalten, um es nicht zu überladen und um ein hohes Maß an Flexibilität zu bieten. Dadurch ist es jedoch

[1] `http://www.openarchitectureware.org/`

leicht möglich, SyncCharts zu erstellen, die semantisch oder syntaktisch nicht korrekt sind. Für den Compiler ist es schwierig zu erkennen, ob ein SyncChart fehlerhaft ist. Im generierten Code kann es zu Laufzeitfehlern kommen, welche dann schwer nachvollziehbar sein können. Besser ist es vor der Codegenerierung zu überprüfen, ob das SyncChart fehlerfrei ist. Dabei ist Check von großem Nutzen, denn mit Hilfe dieser domänenspezifischen Sprache lassen sich Bedingungen auf Modelle definieren, die vor jedem Übersetzungsvorgang geprüft werden können. Zusätzlich lassen sich daraus Fehlermeldungen, Warnungen oder Hinweise ableiten, die dem Benutzer als Annotation im Modell angezeigt werden können. Ausdrücke in Check sind wie folgt aufgebaut:

(1) Für welche Elemente gilt der Ausdruck?

(2) Welche Eigenschaft muss das Element haben, um für den Check in Frage zu kommen?

(3) Definition von Fehlermeldungen / Warnungen / Informationen für den Benutzer.

(4) Unter welchen Umständen wird die Meldung aus (3) ausgegeben?

Abbildung 5.3 zeigt ein Beispiel für einen Ausdruck in Check. Hier wird in Zeile 1 definiert, dass der Check nur für Regionen Anwendung findet. In den Zeilen 2 - 4 wird sicher gestellt, dass es sich nicht um die Root–Region handelt und dass der Zustand um die Region eine ausgehende *normal termination* besitzt. Die Fehlermeldung aus Zeile 6 und 7 wird dann ausgegeben, wenn die betrachtete Region keinen finalen Zustand enthält (Zeile 9).

```
1 context Region if
2   ((parentState.parentRegion.parentState != null) &&
3   (parentState.outgoingTransitions.exists(t|
4     t.type == TransitionType::NORMALTERMINATION)))
5
6   ERROR "A hierarchical state with an outgoing normal termination \n"
7     + "has to contain at least one final state in every parallel region." :
8
9   (innerStates.select(s|s.isFinal).size > 0);
```

Abbildung 5.3: Codebeispiel für einen Ausdruck in Check

Xtend

Für die Synthese von Code aus SyncCharts sind die Informationen, die durch das SyncCharts–Metamodell definiert sind, nicht immer ausreichend. Gerade wenn es um die Durchführung größere Berechnungen geht, ist es häufig notwendig, die Funktionalität mit Xtend zu erweitern. Für die Berechnung der Prioritäten, die in Abschnitt 4.2.4 beschrieben wird, werden viele Methoden über Xtend nach Java ausgelagert.

Xtend bietet auch die Möglichkeit, Modelltransformationen durchzuführen. Diese Funktionalität findet allerdings für den Compiler keine Anwendung.

In Abbildung 5.4 wird eine typische Xtend–Funktion aufgeführt, die das Metamodell um eine zusätzliche Funktion erweitert. Diese Funktion gibt den obersten Zustand eines SyncCharts zurück. Da dieser Root–Zustand eindeutig ist, wird mit `cached` signalisiert, dass diese Berechnung nur einmal ausgeführt werden muss. Damit wird das Ergebnis zwischengespeichert und beim erneuten Aufruf der Funktion auf dieses Ergebnis zurück gegriffen, ohne eine erneute Berechnung zu veranlassen. Dieses Beispiel zeigt auch anschaulich die Verwendung von Rekursionen in Xtend. Gerade wenn man auf hierarchischen Modellen wie SyncCharts arbeitet, wird viel über Rekursionen realisiert. Im Beispiel wird überprüft, ob ein Zustand in einem anderen Zustand eingebettet ist. Ist das der Fall, wird die Funktion mit dem *parent state* rekursiv aufgerufen, anderenfalls wird der Zustand zurück gegeben.

```
1 cached State getRootState(State state):
2   state.parentRegion.parentState == null ? state :
3   getRootState(state.parentRegion.parentState)
4 ;
```

Abbildung 5.4: Codebeispiel für eine Funktion in Xtend, zur Erweiterung der Funktionalität im Metamodell

Abbildung 5.5 zeigt ein häufig verwendetes Szenario in dem aus einer Xtend–Funktion eine Java–Methode aufgerufen wird. Das ist oftmals dann sinnvoll, wenn die Berechnung umfangreicher ist und die Mittel, die Xtend zur Verfügung stellt, nicht ausreichen. Es ist in manchen Situationen auch komfortabler in Java zu implementieren, gerade wenn viele Schleifen und Datentypen, die in Java einfacher zu verarbeiten sind, notwendig sind. In dem angeführten Beispiel (Abb. 5.5) wird die Liste aller Signale für einen Zustand berechnet. Auch hier kann mit `cached` gearbeitet werden, da sich diese Signale nicht dynamisch ändern können. Die Java–Methode muss den gleichen Rückgabewert wie die Xtend–Funktion besitzen.

Xpand

Der größte Teil des Codes für den Compiler ist in Xpand implementiert. Mit Xpand–Templates lässt sich beliebiger Text aus graphischen Modellen generieren. Alles was nicht mit Escapezeichen (`«...»`) umschlossen ist, wird direkt in den generierten Text übernommen. Der Aufbau dieser Templates ist wie folgt definiert:

- `IMPORT`

  Das Importieren eines Namensraums ermöglicht es, die Typen und Definitionen von Metamodellen zu nutzen, ohne diese bei jeder Verwendung mit vollen Pfad zu identifizieren. `IMPORT` ist äquivalent zu den `import`–Statements von Java.

```
1 cached List getStateSignals(State state):
2   JAVA template.Helper.getStateSignals(de.cau.cs.kieler.synccharts.State)
3 ;
```

(a) Xtend–Code

```
1 public static List<StateAndSignals> getStateSignals(final State state) {
2   stateSignalDependencies.clear();
3   fillStateSignalList(state);
4   return stateSignalDependencies;
5 }
```

(b) Java–Code

Abbildung 5.5: Codebeispiel für eine Funktion in Xtend, die Berechnungen nach Java auslagert

- `EXTENSION`

  Um die oben beschriebenen Xtend–Dateien zu verwenden, müssen sie importiert werden. Dies geschieht mit dem `EXTENSION`–Statement.

- `DEFINE`

  Die eigentliche Funktionalität von Xpand steckt in den `DEFINE`–Blöcken. In diesen Templates wird der Text für die Metaklasse generiert, für die das `DEFINE` definiert ist. `DEFINE`s können Parameter enthalten, haben jedoch keinen Rückgabewert. Einige wichtige Statements in `DEFINE`–Blöcken sind:

  – `FILE`

    Die Datei, in welche der zu generierende Text geschrieben wird

  – `EXPAND`

    Expandiert einen anderen `DEFINE`–Block

  – `ERROR`

    Bricht die Ausführung ab und wirft eine *XpandException*

  – `IF`, `FOR`, `FOREACH`, ...

Abbildung 5.6 zeigt den Xpand–Code für die Generierung von SC–Code aus einem SyncChart. In der ersten Zeile wird das Metamodell und in Zeile 3 die Xtend–Hilfsdatei, die das Metamodell um zusätzliche Funktionalität erweitert, importiert. Das `main`–Template in Zeile 5 - 19 legt die C–Datei an, in welche der Code synthetisiert wird. Der Name der Datei richtet sich danach, ob die Datei zum Simulieren (siehe Abschnitt 6.2) erzeugt wird oder nicht (Zeile 10). Hier wird ebenfalls die Verwendung von globalen Variablen gezeigt, die von außen — vor dem Generierungsprozess — gesetzt werden können.

```
1  « IMPORT syncharts »
2
3  « EXTENSION template::Helper »
4
5  « DEFINE main FOR Region -»
6    « IF this.innerStates.isEmpty -»
7      « ERROR("There is no SyncChart to generate code for!") »
8    « ELSE -»
9      « computeThreadPriorities(this.innerStates.first()) »
10     « FILE ((String)(GLOBALVAR name)) + ".c" -»
11     /* generated SC-code */
12     « EXPAND init -»
13     « EXPAND start -»
14     « EXPAND finish -»
15     « EXPAND step -»
16     « ENDFILE -»
17     « EXPAND CodegenData::main »
18   « ENDIF -»
19 « ENDDEFINE »
20
21 « DEFINE init FOR Region -»
22   #include "sc.h"
23   « EXPAND CodegenMisc::generateSigType FOR allSignals(this.innerStates.first()) -»
24   « EXPAND CodegenMisc::generateSigArray FOR allSignals(this.innerStates.first()) -»
25   #define _SC_valSigInt_SIZE « allSignals(this.innerStates.first()).size »
26
27   int valSigInt[_SC_valSigInt_SIZE];
28   int valSigIntPre[_SC_valSigInt_SIZE];
29   « EXPAND CodegenMisc::generateInitialize FOR this -»
30
31   « FOREACH allSignals(this.innerStates.first()) AS signal -»
32     « IF ((Signal)signal).isInput || !(((Signal)signal).isInput
33       || ((Signal)signal).isOutput) -»
34       « EXPAND CodegenMisc::generateInputSignal FOR ((Signal)signal) »
35     « ENDIF -»
36   « ENDFOREACH -»
37   « EXPAND CodegenMisc::generateCallOutputs FOR allSignals(this.innerStates.first()) »
38   « EXPAND CodegenMisc::generateReset -»
39 #ifndef EXTERNTICK
40
41   int tick(){
42   TICKSTART(« EXPAND CodegenPriority::mainThreadPrio FOR this.innerStates.first() -»);
43 « ENDDEFINE »
44
45 « DEFINE start FOR Region -»
46   « EXPAND CodegenRegion::region FOR this -»
47 « ENDDEFINE »
48
49 « DEFINE finish FOR Region -»
50   TICKEND;} #endif
51 « ENDDEFINE »
52
53 « DEFINE step FOR Region »
54   int « innerStates.first().id »(){
55     return tick();
56   }
57 « ENDDEFINE »
```

Abbildung 5.6: Xpand–Templates für die Erzeugung von SC–Code aus SyncCharts

Als nächstes wird die Struktur der C–Datei geschaffen, indem im `DEFINE`–Block für `init` aus Zeile 21 - 43 einige Initialisierungen vorgenommen und für `finish` aus Zeile 49 - 51 die `tick`–Funktion abgeschlossen werden. Der Code in der `tick`–Funktion wird mit der Root–Region in `start` (Zeile 46) eingeleitet und dann rekursiv für alle Elemente des SyncCharts generiert. Durch die doppelte Doppelpunkt–Notation ist es möglich, Templates aus anderen Xpand–Dateien zu expandieren, wodurch sich der Xpand–Code modular gestalten lässt. Deshalb gibt es für Elemente von SyncCharts, wie Regionen oder Zustände, jeweils eine eigene Template–Datei.

5.4 Sonstiges

Zusätzlich zur eigentlichen Implementierung des Compiler waren einige weitere Implementierungen notwendig, die im Folgenden kurz beschrieben werden.

SC–Beautifier

Xpand ist sehr mächtig und bietet viele Möglichkeiten, die bei der Generierung von Code von Vorteil sind. Ein entscheidender Nachteil ist jedoch, das Xpand selbst keine Formatierung vornimmt. Jegliche Zeilenumbrüche, Leerzeichen und Tabulatoren, die im Xpand–Code vorkommen, erscheinen auch im generierten Code. Da es für SC jedoch wichtig ist, dass der erzeugte Code einen hohen Bezug zum SyncChart hat, muss dieser auch gut lesbar sein. Es gibt zwei Lösungsansätze, das zu realisieren.

Der erste Ansatz ist, den Xpand–Code so zu strukturieren, dass die SC–Datei „schön“ wird. Das funktioniert und war auch der Ansatz für die ersten Versuche. Der Nachteil liegt auf der Hand; der Xpand–Code ist mit wachsender Komplexität nicht mehr lesbar und vor Allem nicht wartbar. Aus diesem Grund ist ein zweiter Ansatz in Form eines Beautifiers entstanden. Der Xpand–Code bleibt wohl strukturiert und der SC–Code wird, nachdem er erstellt wurde, noch einmal durchlaufen und anhand einiger Regeln in ein „schönes“, lesbares Format gebracht. Abbildung 5.7 zeigt exemplarisch einen Teil vom Code der `tick`–Funktion für ABRO (Abbildung 3.1) vor und nach der Formatierung mit dem Beautifier.

Erweiterung der SC–Header–Datei

Eine andere Aufgabe, die sich bei der Implementierung des Compilers ergab, war die Erweiterung der Zielsprache um zusätzliche Funktionalität. So enthielt die ursprünglichen Version von SC weder boolesche *valued signals*, noch alle *combine*–Funktionen für Signale (siehe dazu 3.2.2). Für den Compiler, der diese Dinge unterstützen sollte, musste deshalb die SC–Header–Datei dahingehend angepasst werden. Eine genaue Beschreibung von SC und der dazugehörigen Header–Datei ist dem Technischen Report für SC [42] zu entnehmen.

```
1   int tick(){
2   TICKSTART(
3   4
4
5   );
6        L_ABRO_ABO:
7   FORK(L_ABRO_ABO_wAB,  1
8   );
9   FORKE(L_ABRO_ABO_main);
10       L_ABRO_ABO_wAB:
11    FORK(L_ABRO_ABO_wAB_R0_wA,  3
12  );
13    FORK(L_ABRO_ABO_wAB_R1_wB,  2
14  );
15
16  FORKE(L_ABRO_ABO_wAB_main);
17       L_ABRO_ABO_wAB_R0_wA:
18
19
20
21    PAUSE;
22  if (PRESENT(sig_A)){
23    TERM;
24  }
25
26
27    GOTO(L_ABRO_ABO_wAB_R0_wA);
28
29  /* rest of code */
30
31  TICKEND;}
```

(a) SC–Code ohne Beautifier

```
1   int tick() {
2     TICKSTART(4);
3
4   L_ABRO_ABO:
5     FORK(L_ABRO_ABO_wAB, 1);
6     FORKE(L_ABRO_ABO_main);
7
8   L_ABRO_ABO_wAB:
9     FORK(L_ABRO_ABO_wAB_R0_wA, 3);
10    FORK(L_ABRO_ABO_wAB_R1_wB, 2);
11    FORKE(L_ABRO_ABO_wAB_main);
12
13  L_ABRO_ABO_wAB_R0_wA:
14    PAUSE;
15    if (PRESENT(sig_A)) {
16      EMIT(sig_B);
17      TERM;
18    }
19    GOTO(L_ABRO_ABO_wAB_R0_wA);
20
21  /* rest of code */
22
23    TICKEND;
24  }
```

(b) SC–Code mit Beautifier

Abbildung 5.7: Teil der *tick*–Funktion für ABRO vor und nach der Formatierung mit dem Beautifier

Anbindung an KIELER

Für die Anbindung an KIELER mussten einige zusätzliche Implementierungen vorgenommen werden. Besonders Fehlerbehandlung und Benutzerfreundlichkeit standen hierbei im Vordergrund. Dazu gehört auch die Unterstützung des Compilers für die drei großen Betriebssysteme; Windows, Linux und Mac-OS, für die KIELER entwickelt wird. Des Weiteren wurde für die Visualisierung eine Kommunikation zwischen dem erzeugten SC-Programms und KIELER realisiert. Dieser Mechanismus wird in Abschnitt 6.2 beschrieben.

6 Integration in KIELER

Um aus graphischen Modellen wie SyncCharts textuellen Code zu erzeugen, benötigt man zunächst einmal eine Repräsentation der Modelle. Deshalb muss es eine Möglichkeit geben SyncCharts zu modellieren, um diese dann weiter verarbeiten zu können. Kiel Integrated Environment for Layout Eclipse Rich-Client (KIELER) bietet — unter Anderem — diese Möglichkeit und stellt damit eine hervorragende Plattform für den Compiler zur Verfügung. In Abschnitt 6.1 wird KIELER vorgestellt und kurz beschrieben welche Optionen es zusätzlich bietet, die auch für den Weg vom SyncChart nach SC ein hohes Maß an Komfort bieten. Kapitel 6.2 beschreibt, wie KIELER dafür genutzt wird, um den erzeugten Code visuell darzustellen. Ein wichtiges Thema bei der Entwicklung des Compilers ist die Überprüfung des Verhaltens vom generierten Code. Besonders nach größeren Änderungen ist eine manuelle Überprüfung auf Dauer nicht praktikabel. In Sektion 6.3 wird der Weg vom manuellen zum automatischen Validieren erläutert und gezeigt, welche zusätzlichen Möglichkeiten aus diesem Prozess entstanden sind.

6.1 Kiel Integrated Environment for Layout Eclipse Rich-Client (KIELER)

KIELER ist ein Forschungsprojekt zur Verbesserung der graphischen, modellbasierten Entwicklung von komplexen Systemen [16]. Das Hauptaugenmerk von KIELER besteht darin, neue Konzepte und Methoden für das Erstellen und Bearbeiten verschiedener Diagrammtypen zu entwickeln. Ziel ist es den Prozess der Entwicklung mit Modelliersprachen so zu optimieren, dass sowohl Designkriterien erfüllt, als auch die Zeiten für Implementierung und Modifizierung von Modellen minimiert werden. Grundlage und Motivation für die Entwicklung von KIELER ist das Vorgängerprojekt Kiel Integrated Environment for Layout (KIEL) [34]. Der Hauptunterschied von KIELER zu KIEL ist die Möglichkeit der Integration von anderen Modelliersprachen als StateCharts. Außerdem ist KIELER— wie der Name bereits vermuten lässt — in die *rich client* Plattform *Eclipse* integriert. Dadurch sind neue Technologien einer breiten Masse von Interessierten zugänglich. Für mehr Informationen und zum freien Download der Software (sowie des darin enthaltenen Compilers) sollte die Website des KIELER–Projekts besucht werden[1]. In den folgenden Abschnitten werden einige der Features von KIELER, die auch für die Entwicklung des Compilers von großem

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

Nutzen waren, kurz erläutert. Abbildung 6.1 zeigt das KIELER–System mit einem geöffneten SyncChart.

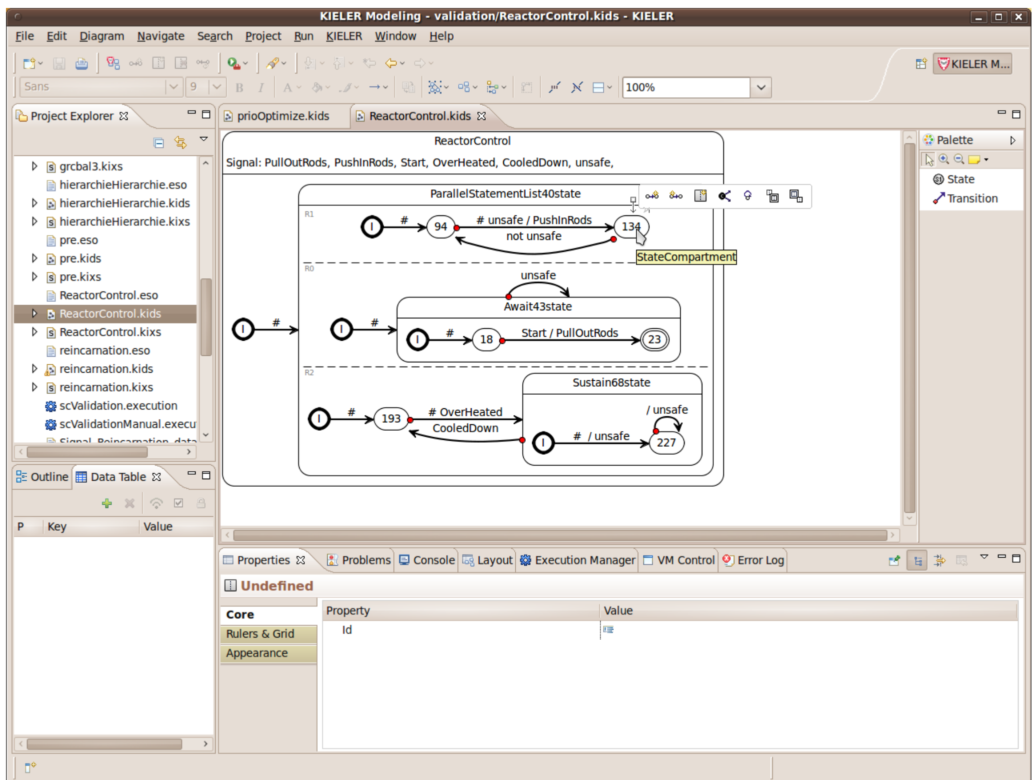


Abbildung 6.1: KIELER–System mit einer geöffneten SyncCharts–Modellinstanz

6.1.1 Thin Kieler SyncCharts Editor (ThinKCharts)

Der Thin Kieler SyncCharts Editor (ThinKCharts) [35] stellt die graphische Repräsentation der SyncCharts in KIELER zur Verfügung. Die Modelle, die der Compiler zur Codesynthese verwendet, werden mit ThinKCharts modelliert und sind intern in einer XML Metadata Interchange (XMI)-Struktur gespeichert. Mit Hilfe dieser Struktur lassen sich die SyncCharts parsen und nach SC expandieren.

6.1.2 Automatiches Layout

Das Layout von Modellen ist oft ein Indikator dafür, wie gut es zu lesen und wie schnell es nachzuvollziehen ist. Das beste (schönste) Layout für die einzelne Person ist zumeist nur durch manuelles modellieren möglich. Leider kann man mit dem händischen „verschönern“ eines mittelgroßen Diagramms viel Zeit verbringen, die dann nicht mehr im Verhältnis zur Modellierarbeit selbst steht. Die Frage nach Ästhetik

für Modelliersprachen und wie man das Layout so gut wie möglich daran anpasst stellt sich immer wieder und ist hoch interessant. Diese Thematik spielt auch für KIELER eine große Rolle. Die Plattform bietet viele verschiedene Layout–Algorithmen und kann diese für verschiedene Modelle kombinieren. Außerdem existieren spezielle Layout–Mechanismen für bestimmte Modelliersprachen. Dadurch kann auf die Eigenheit einzelner Sprachen Bezug genommen werden [36, 17]. Auch für die Verwendung des Compilers ist es wichtig, dass das SyncChart ein gut strukturiertes Layout besitzt. Da die Implementierung ein ständiges Testen und Verifizieren erfordert, ist es wichtig, dass das SyncChart gut geordnet und einfach zu verstehen ist. Der zeitliche Vorteil, den ein guter Layout–Mechanismus mit sich bringt, ist ebenfalls bei der Erstellung von Beispielen für Tests von großer Bedeutung. Bei den unzähligen SyncCharts die für die Entwicklung des Compilers modelliert wurden, wäre es zu zeitaufwendig gewesen, jedes einzelne per Hand zu layouten, besonders nicht für die ständigen Änderungen und Erweiterungen.

6.1.3 Strukturbasiertes Editieren

Wo *content assist* bzw. *auto completion* einem das Programmieren von textuellen Sprachen komfortabler gestalten, so wird dies in KIELER durch strukturbasiertes Editieren getan. Ein ständiges *Drag–&–Drop*–Editieren kann auf Dauer nervenaufreibend sein, besonders wenn man Diagramme erweitert, um neue Funktionalität hinzuzufügen, was bei der Entwicklung und Überprüfung des Compilers immer wieder auftritt. Das Feature *KIELER Structure-Based Editing (KSBasE)* [27] bietet die Möglichkeit vorgefertigte Transformationen zu benutzen, welche dann automatisch ausgeführt werden und nach denen ein automatisches Layout angestoßen wird. Solche Transformationen können zum Beispiel das Einfügen eines Folgezustands, die Aufwertung eines einfachen zu einem Makrozustand oder das Umdrehen einer Transition sein. Der Vorteil für den Entwickler ist zum einen, dass diese Transformationen sowohl mit der Maus als auch mit Tastatur–Shortcuts durchgeführt werden können, und zum anderen, dass beliebige individuelle Transformation selbst definiert werden können.

6.1.4 Der Execution Manager

Ein — für den Compiler besonders wichtiges — Feature, das KIELER bietet, ist der KIELER Execution Manager (KIEM) [28]. KIEM stellt eine Infrastruktur bereit, um unter Anderem graphische Modelle simulieren und ausführen zu können. Der Vorteil liegt darin, dass es mit KIEM möglich ist, Berechnungen für ein Modell auszuführen, welche visuell angezeigt werden können. Dadurch lässt sich auch der vom Compiler generierte Code nutzen, um das dazugehörige SyncChart zu simulieren. Im nächsten Abschnitt wird dies im Detail erörtert. Zusätzlich erlaubt KIELER die Ausführungen vom KIEM automatisiert zu nutzen und somit beispielsweise mehrere Simulationen nacheinander durchführen zu lassen. Dadurch ist es möglich einen Mechanismus zur

automatischen Validierung des Compilers zu entwickeln. Dies wird in Kapitel 6.3 näher beschrieben.

6.2 Visualisierung

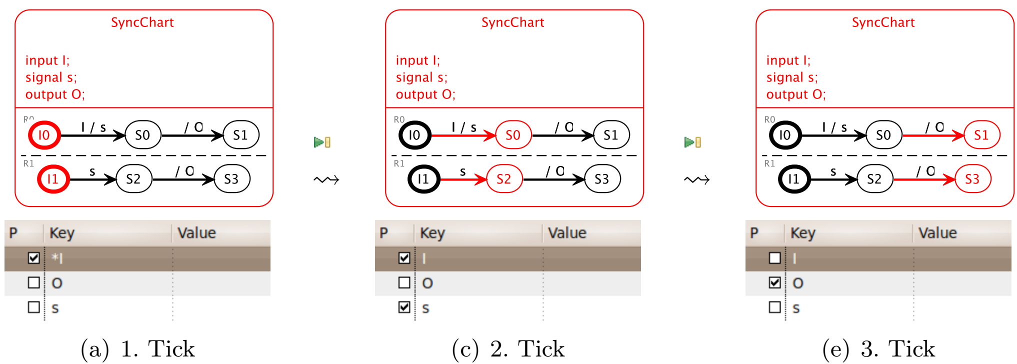
Bei der Implementierung des Compilers stellte sich die Frage, wie man komfortabel überprüfen kann, ob der erzeugte Code die richtigen Ergebnisse liefert. Eine Möglichkeit war es, die Tracing–Funktion zu nutzen, die SC zur Verfügung stellt. Damit ist es möglich auf der Konsole Eingaben zu schreiben und die resultierenden Ausgaben zu vergleichen. Diese Vergleiche sind auch automatisiert möglich, indem man sie direkt in die SC–Datei integriert. In Abbildung 3.6 wurde ein Beispiel der Tracing–Funktion gezeigt. Dadurch bieten sich zwar viele Möglichkeiten wie Fehlererkennung, doch komfortabel nutzbar ist das nicht. Wenn man mit graphischen Modellen „zum Anfassen“ arbeitet, dann möchte man nicht zurück zur Konsole, um diese dann zu überprüfen oder zu simulieren. Deshalb sollte es die Möglichkeit geben, die Ausführung des erzeugten Codes graphisch darstellen zu können. Daraus ergab sich auch direkt ein erster Anwendungsfall für den Compiler, der letztendlich auch ein Feature für KIELER ist.

Die Anforderungen für die graphische Darstellung leiten sich direkt aus den SyncCharts ab. Dazu ist es auch wichtig, dass ein Interface zur Verfügung steht um Eingabe–Signale zu setzen. Da Zeiten in Ticks dargestellt sind, möchte man den Status eines SyncCharts zu jedem Tick haben. Dazu gehören:

- Aktive Zustände
- Transiente Zustände (Zustände, die in einem Tick betreten und direkt wieder verlassen wurden)
- Genommene Transitionen
- Eventuell überprüfte Transitionen
- Emittierte Signale

Mit Hilfe des KIELER Execution Managers gibt es die Möglichkeit, die oben genannten Anforderungen zu erfüllen. KIEM stellt eine Schnittstelle zur Verfügung, um Aktionen direkt auf ein SyncChart auszuführen bzw. Daten auszulesen. In einer Tabelle sind alle Signale, die im SyncChart auftreten aufgelistet und können vom Benutzer gesetzt werden. Zusätzlich gibt es eine Benutzerschnittstelle mit der ein Simulationsablauf gestartet, gestoppt, pausiert oder schrittweise ausgeführt werden kann. Damit lassen sich die verschiedenen Ticks beim Ausführen eines SyncCharts darstellen. Abbildung 6.2 zeigt den Ablauf einer kurzen Simulation vom SyncChart 4.20(a). Initial sind die Zustände `I0` und `I1` aktiv. Dann wird Signal `I` auf aktiv gesetzt und durch ein Klick auf den *step*–Button der nächste Tick ausgeführt. In diesem zweiten Tick werden die Transitionen zu `S0` und `S2` genommen und diese Zustände auch betreten. Außerdem wird dadurch, dass `I` *present* ist, `s` emittiert.

Mit einem weiteren Ausführungsschritt werden die Zustände `S1` und `S3` betreten und `O` emittiert.



(a) 1. Tick (c) 2. Tick (e) 3. Tick

Abbildung 6.2: Simulation für das SyncChart aus Abbildung 4.20(a)

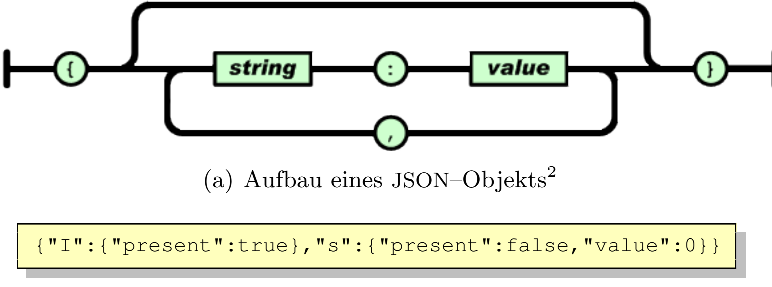
Ein Stern (*) vor einem Signal in der Datentabelle bedeutet, dass dieses Signal manuell neu gesetzt wurde und im nächsten Tick anliegt. Die Simulation muss nicht schrittweise durch Klicks ausgelöst werden, sondern kann auch automatisch ablaufen. Dafür lässt sich die Simulationsgeschwindigkeit regeln. Somit können SyncCharts, die zumeist reaktiv sind, einmal gestartet werden und das Verhalten durch einfaches Setzen von Inputs in der Datentabelle analysiert werden. Der Simulationsprozess hat dazu beigetragen, dass bei der Entwicklung des Compilers Änderungen im Code am SyncChart nachvollzogen werden konnten.

Es ist noch erwähnenswert, dass KIELER bereits eine Möglichkeit bietet, SyncCharts zu simulieren. Dies wird durch die Komponente KIELER leveraging Ptolemy semantics (KlePto) [28] realisiert, dass die Verbindung von Modellen in KIELER zu den dazu erzeugten Modellen in Ptolemy [10] herstellt und diese mit Hilfe von Ptolemy simuliert. Leider unterstützt diese Möglichkeit der Simulation nicht alle Features die SyncCharts abbilden kann. Der größte Nachteil ist die fehlende Unterstützung für *valued signals*, *immediate* Transitionen, *conditional states* und *actions*. Außerdem existiert der Umweg über die Ptolemy–Simulation. Ein Vorteil gegenüber der Simulation mit SC ist die Möglichkeit einer dynamischen *must*– und *cannot*–Analyse für Signale, die Ptolemy zur Verfügung stellt. Dadurch wird sicher gestellt, dass ein Signal nur dann *present* ist, wenn es emittiert werden musste und nur dann *absent* ist, wenn es nicht emittiert werden konnte. Damit gibt es keine spekulativen Aussagen über Signale, wodurch nur konstruktive SyncCharts simuliert werden [6, 30]. Ebenfalls von Vorteil ist, dass auch Ptolemy in Java implementiert ist, was die Integration von KlePto in KIELER vereinfacht.

6.2.1 Anbindung des SC–Code an KIELER

Da KIELER ein reines Java–Projekt und SC eine C–Makrosprache ist, musste ein Weg gefunden werden, beide Komponenten miteinander kommunizieren zu lassen. Ein einfacher, aber effektiver Weg ist die Kommunikation über Standard I/O. Führt man eine Simulation unter der Verwendung von SC aus, so wird zuerst der SC–Code für das SyncChart erzeugt und in ein temporäres Verzeichnis geschrieben. Verlief die Codesynthese erfolgreich, so wird der erzeugte Code mit dem *gcc* (aus der GNU Compiler Collection (GCC)) übersetzt und das Kompilat ausgeführt. Das ausgeführte Programm wartet dann auf Eingaben in Form von Signalen, führt einen Tick aus und gibt Informationen über Signale, Outputs und den zu highlightenden Elementen des SyncCharts zurück. Wird die Simulation durch Betätigung des *stop*–Buttons oder durch einen Fehler beendet, so wird auch das C–Programm beendet. Es wäre auch möglich gewesen die Kommunikation zwischen SC und KIELER über Java Native Interface (JNI) zu realisieren, jedoch ist der Weg über Standard I/O durchaus ausreichend und erlaubt eine eventuelle Anbindung des erzeugten Programms an andere Systeme. Außerdem kann es nicht zu Problemen beim dynamischen Nachladen von Bibliotheken, die für JNI benötigt werden, führen.

Die Kommunikation mit KIEM geschieht mittels JavaScript Object Notation (JSON). JSON ist ein leichtgewichtiges Format, um Daten zu übertragen bzw. auszutauschen. Der Vorteil von JSON liegt in der Größe der zu übertragenen Daten und im simplen Aufbau, der es erlaubt, JSON–Objekte einfach zu erzeugen bzw. zu parsen. Abbildung 6.3(a) zeigt den prinzipiellen Aufbau eines JSON–Objekts. In Abbilung 6.3(b) ist ein Beispiel für ein JSON-Objekt zu sehen, in dem das Eingabe–Signal `I` auf *true* gesetzt und dem Signal `s` der Wert 0 zugewiesen wird.



(a) Aufbau eines JSON–Objekts[2]

```
{"I":{"present":true},"s":{"present":false,"value":0}}
```

(b) JSON–Objekt für Eingabe `I` und Signal `s`

Abbildung 6.3

Für die Benutzung von JSON für SC wurde die *cJson*–Implementierung[3] verwendet. Nach einigen Tests mit verschiedenen anderen Implementierungen ist die Entscheidung schnell auf *cJson* gefallen. Ein wichtiger Grund dafür ist die geringe Komple-

[2] http://www.json.org/
[3] http://cjson.sourceforge.net/

xität und die übersichtliche Funktionalität, die allerdings für den Gebrauch in SC vollkommen genügt. Zudem liefert diese Implementierung, im Gegensatz zu vielen anderen, beim Parsen der Objekte wieder JSON–Objekte. Dadurch ist es nicht notwendig, dass Strings für die weitere Verwendung in die entsprechenden Datentypen gecastet werden müssen.

Vor der Ausführung eines jeden Ticks wird ein JSON–Output–Objekt angelegt. Immer wenn der Code für einen Zustand ausgeführt wird, wird ein entsprechendes Zustands–Objekt an das Output–Objekt akkumuliert. Das Gleiche geschieht für Transitionen bzw. dann, wenn ein Signal geschrieben wird. Wenn der Tick beendet ist, dann wird das Output–Objekt an KIEM übermittelt. Dort werden die entsprechenden Visualisierungen vorgenommen und die emittierten Signale in der Datentabelle angezeigt. Für die Inputs, die der Benutzer setzt, wird beim Klick des *step*–Buttons (oder automatisch, wenn die Simulation automatisiert läuft) auf KIELER Seite ein JSON–Objekt erstellt und an SC gesendet. Dort wird das Objekt ausgewertet und die Signale für die Ausführung des Ticks gesetzt. Die gesamte Kommunikation läuft blockierend, da es nicht vorkommen kann, dass SC mehrfach in einem Tick Daten an KIELER sendet.

6.3 Validierung

Ein sehr wichtiges Thema ist die Validierung des erzeugten Codes. Im Laufe der Arbeit haben sich eine Reihe von Beispiel–SyncCharts für bestimmte Szenarien angesammelt. Besonders die Optimierung des Codes bot immer die Gefahr, dass durch zu restriktive Vereinfachung einige dieser Beispiele ein falsches Verhalten aufweisen. Die Funktionalität des Compilers nach jeder kleinen Änderung per Hand zu überprüfen, war sowohl zeitlich als auch vom Arbeitsaufwand nicht vertretbar. Deshalb wurde nach einer Möglichkeit gesucht, dies automatisch zu verwirklichen. Mit der Simulation stand bereits ein Mechanismus zur Verfügung, um das Verhalten von SyncCharts abzubilden. Somit war KIEM der erste Anlaufpunkt für die Umsetzung.

Ein zusätzliches Feature, das auf KIEM aufsetzt, ist *KIEM automated* [19]. Damit ist es möglich, mehrere Simulationen im Hintergrund — also ohne Visualisierung — nacheinander ablaufen zu lassen. Mit Hilfe dieses neuen Features bietet sich die Möglichkeit Traces im *eso*–Format zu nutzen, um mehrere SyncCharts automatisch zu validieren. Eine *eso*–Datei beinhaltet für ein bestimmtes SyncChart Eingaben und die erwarteten Ausgaben für jeden Tick. Abbildung 6.4 zeigt einen Ausschnitt einer *eso*–Datei für `ABRO` aus Abbildung 3.1. Hier werden jeweils beginnend mit `! reset` zwei Ausführungen definiert. Die erste Ausführung hat zwei Ticks, wobei im ersten Tick keine Eingaben existieren und nichts emittiert wird. Im zweiten Tick werden `A` und `B` als Eingaben und `O` als Ausgabe angenommen. Analog ist der zweite Durchlauf mit drei Ticks zu lesen.

Es kann für jedes zu testende SyncChart eine *eso*–Datei erstellt werden, die für alle möglichen Eingaben die erwarteten Ausgaben beinhaltet. Mit dieser Datei und

```
1  ! reset;
2    % Output:;
3    A B
4    % Output: O;
5  ! reset;
6    A
7    % Output:;
8    B
9    % Output: O;
10   A B  R
11   % Output:;
```

Abbildung 6.4: Ausschnitt der *eso*–Datei für ABRO aus Abbildung 3.1

KIEM automated lässt sich das SyncChart simulieren, wobei nun als Eingaben keine Benutzereingaben, sondern die aus der *eso*–Datei übernommen werden. Die Ausgaben, welche die Simulation liefert, können mit den erwarteten Werten des *eso*–Files überprüft werden. Für jeden Durchlauf bekommt man ein Feedback über den Status, so dass eventuelle Fehler nachvollzogen werden können. Genauer gesagt, wird dargestellt welche Eingaben für erwartete Ausgaben nicht das richtige Ergebnis liefern. Mit diesem Mechanismus ist es nun möglich, mehrere SyncCharts, für die *eso*–Dateien existieren, automatisch im Hintergrund zu validieren.

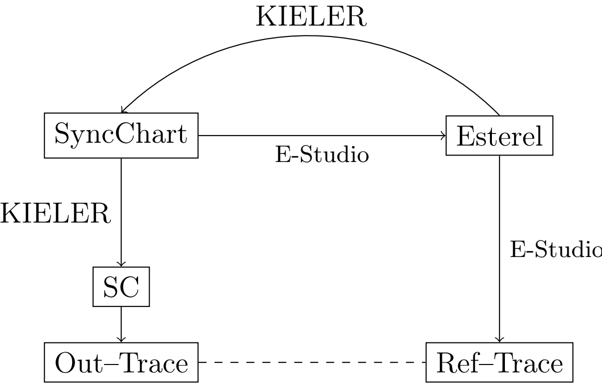


Abbildung 6.5: Verschiedene Wege zur Validierung des Compilers

Generell gibt es zwei Wege für die Validierung mit Trace–Dateien. Zum einen existieren einige SyncCharts sowohl in Esterel Studio und in KIELER, zum anderen können SyncCharts direkt in Esterel Programme synthetisiert werden. Für beide Wege können die automatisch generierten Trace–Dateien genutzt werden, die Esterel Studio zur Verfügung stellt. In Abbildung 6.5 sind diese Wege veranschaulicht. Außerdem ist es für kleine Beispiele einfach, eigene *eso*–Dateien per Hand zu schreiben, um den Compiler hinreichend zu überprüfen. Für eine feinere Validierung könnten zusätzlich die erwarteten aktiven bzw. transienten Zustände überprüft werden.

7 Versuchsergebnisse

Nach der Fertigstellung des Compilers stellte sich die Frage, wie man im Vergleich zu anderen Ansätzen der Übersetzung steht. Bei genauer Betrachtung waren das drei Fragen. Wieso vergleicht man sich, womit vergleicht man sich und wie vergleicht man sich. Die Frage nach dem Wieso ist einfach zu beantworten. Nachdem man viel Zeit und Arbeit in ein solches Projekt gesteckt hat, will man wissen ob das Ergebnis nicht nur funktioniert, sondern auch sinnvoll funktioniert. Sinnvoll bedeutet für Synthese von Code, dass der Code kompakt und schnell sein soll. Womit man sich vergleicht hängt häufig davon ab, wer Ergebnisse liefert, die den eigenen ähneln. Für den entwickelten Compiler sind das vier verschiedene Methoden, die zum Vergleich verwendet wurden:

- Handgeschriebener SC–Code

  SC–Code, der per Hand geschrieben ist, kann für kleine Beispiele sehr effizient sein, da der Programmierer beliebig viel Energie in die Optimierung stecken kann. Deshalb ist das Ziel für den generierten Code sich so nah wie möglich (Geschwindigkeit und Größe) an den Hand–generierten Code anzunähern. Im technischen Bericht zu SC [42] sind eine Reihe kleiner Beispiele angegeben, von denen einige für den Vergleich genutzt werden. Für mittelgroße und große Beispiele ist dieser Vergleich nicht mehr möglich, da auf Grund der Komplexität dieser SyncCharts die manuelle Code–Erzeugung sehr aufwendig wird.

- Esterel CEC für virtuelle Maschinen (VM) und Esterel v5

  Der Columbia Esterel Compiler (CEC) und der Esterel v5–Compiler erzeugen jeweils C–Code aus Esterel. Die Codesynthese von SyncCharts nach Esterel wird dafür durch KIELER bereit gestellt. Durch den resultierenden Code wird ein äquivalenter Schaltkreis in C simuliert. Der Code des CEC wird für die Verwendung einer VM genutzt. Dabei gibt es Einschränkungen für SyncCharts mit `pre`–Operatoren, die vom CEC nicht unterstützt werden. Außerdem sind die SyncCharts in ihrer Größe beschränkt, da die VM auf 255 Signale limitiert ist und für jeden Zustand im SyncChart ein Signal generiert wird.

- E-Studio

  Der erzeugte Esterel–Code für die beiden obigen Methoden ist rein nach dem Ansatz von André [3] vollzogen und besitzt keine zusätzlichen Optimierungen. Im kommerziellen Tool E-Studio wird der gleiche Ansatz zur Erzeugung von Esterel–Code aus SyncCharts benutzt, dieser wird jedoch durch eine Reihe von zusätzlichen Optimierungen [33] deutlich verbessert.

Die letzte Frage, die es zu beantworten gilt, ist die Frage nach dem Wie. Ein Vergleich macht nur dann Sinn, wenn gleiche Voraussetzungen geschaffen werden und der Ablauf „fair“ von statten geht. Abbildung 7.1 zeigt die verschiedenen Wege aus denen die ausführbaren Dateien für den Vergleich entstehen. Hier wird ersichtlich, dass alle Wege als gemeinsamen Nenner die vom SC–Compiler generierte `x_data.c` haben. Diese C–Datei beinhaltet eine `main`–Funktion, welche Eingaben setzt und die Zeit misst. Jedes Programm wird dann mit randomisierten Eingaben eine Millione Mal ausgeführt und die Zeit in Taktzyklen gemessen. Dabei bekommen alle Programmvarianten die gleiche Eingabemenge und es wird als Konsistenzcheck verglichen, wie oft jedes Ausgabesignal emittiert wird.

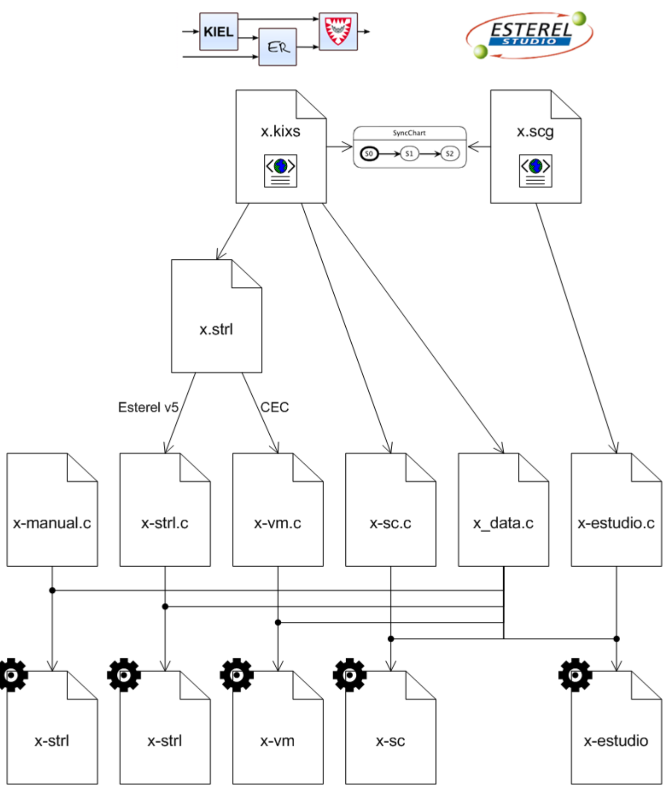


Abbildung 7.1: Verschiedene Pfade für die Erzeugung von Code für die Versuchsergebnisse

Für die Erfassung der Codegröße wird sowohl die Größe des generierten Codes in Lines Of Code (LOC), als auch die des fertigen ausführbaren Programms gemessen. Für den SC–Code (sowohl per Hand als auch vom Compiler generiert) gibt die LOC–Messung zusätzlich eine Abschätzung über die Codegröße für die Ausführung von SC in einer möglichen virtuellen Maschine oder auf Hardware mit geeigneten Instruktionssatz. Alle Programme werden mit GCC und der Standard Optimierung (O2) übersetzt und auf einem PC mit Intel Xeon Prozessor mit 3 GHz und 6 MB Cache ausgeführt.

Kleine SyncCharts

Für die Messung kleiner SyncCharts wurden einige Beispiele aus dem technischen Bericht zu SC [41] verwendet. Diese SyncCharts sind von der Größe so überschaubar, dass es möglich ist, SC–Code per Hand dafür zu schreiben. Deshalb sind das auch die einzigen SyncCharts für die ein Vergleich zum manuell geschriebenen Code existiert.

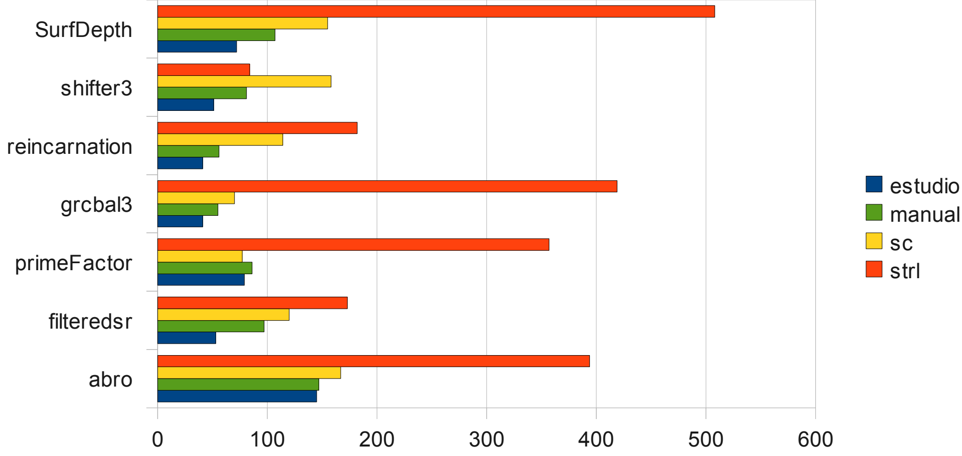


Abbildung 7.2: Zeitmessung für kleine SyncCharts in Taktzyklen

Abbildung 7.2 zeigt die Zeitmessungen für alle vier Methoden der Codeerzeugung. Hier wird deutlich, dass der vom Compiler generierte SC–Code nah an die Zeiten des per Hand generierten Codes heran kommt. Im Vergleich zum generierten Code des CEC–Compilers kann der SC–Compiler hier einen Vorteil aufzeigen.

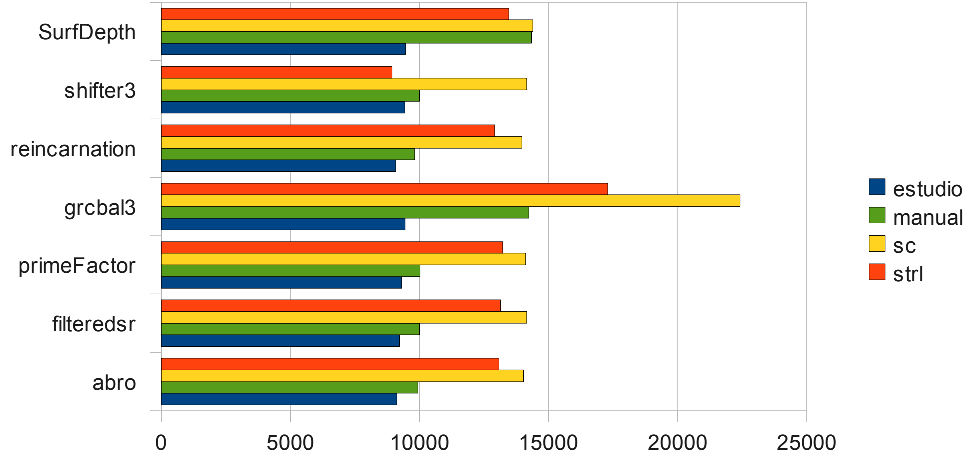


Abbildung 7.3: Größe der ausführbaren Datei für kleine SyncCharts in Byte

In den Abbildungen 7.3 und 7.4 werden die Größe des erzeugten Programms und die LOC–Größe illustriert. Hier zeigt sich, dass die Programmgrößen vergleichsweise ähnlich, die benötigten Codezeilen jedoch für den SC–Code — sowohl per Hand geschrieben als auch vom Compiler generiert — deutlich geringer sind.

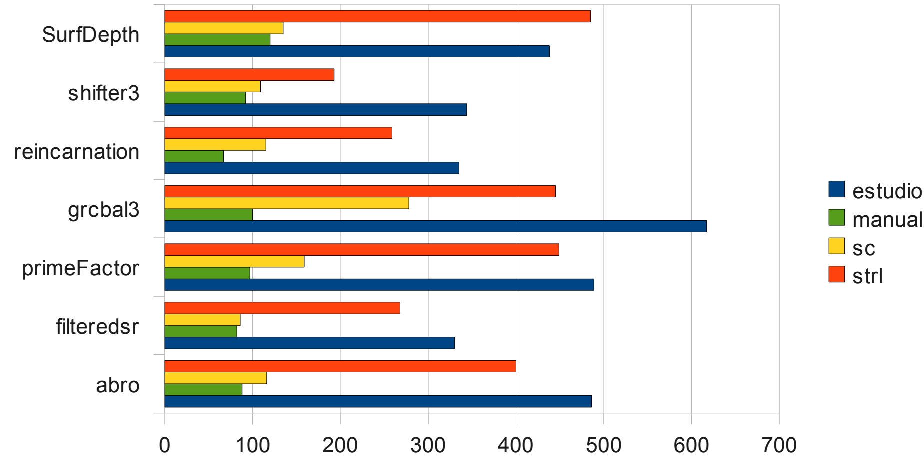


Abbildung 7.4: Größe in LOC für kleine SyncCharts

Mittelgroße SyncCharts

Die mittelgroßen SyncCharts entstammen einer Sammlung von Modellen, die in KIEL [34] modelliert und nach KIELER importiert wurden. Für den Compiler aus E-Studio wurden die SyncCharts zusätzlich in E-Studio modelliert. Die SyncCharts sind so groß, dass sie mit einem zeitlich verträglichen Aufwand modelliert werden können, die manuelle SC–Codeerzeugung jedoch unpraktikabel wäre. Für diese SyncCharts existieren deshalb auch keine Vergleiche zum handgeschriebenen Code. Für das Beispiel *tcint_debug32* existieren für den Columbia Esterel Compiler (CEC) keine Messungen, da der aus KIELER generierte Esterel–Code für dieses SyncChart vom CEC nicht übersetzt werden konnte.

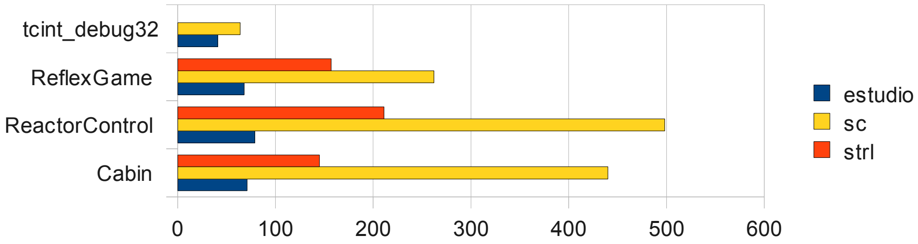


Abbildung 7.5: Zeitmessung für mittelgroße SyncCharts in Taktzyklen

Die Zeitmessungen für mittelgroße SyncCharts, die in Abbildung 7.5 dargestellt sind, zeigen, dass sich die Laufzeit der SC–Programme für größer werdende SyncCharts erhöht. Dies wird besonders für die reaktiven Beispiele *ReactorControl* und *Cabin* deutlich.

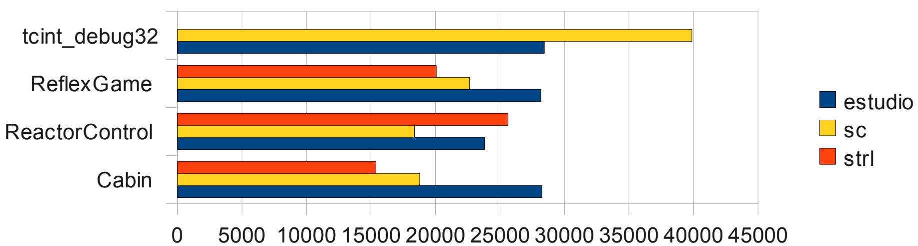


Abbildung 7.6: Größe der ausführbaren Datei für mittelgroße SyncCharts in Byte

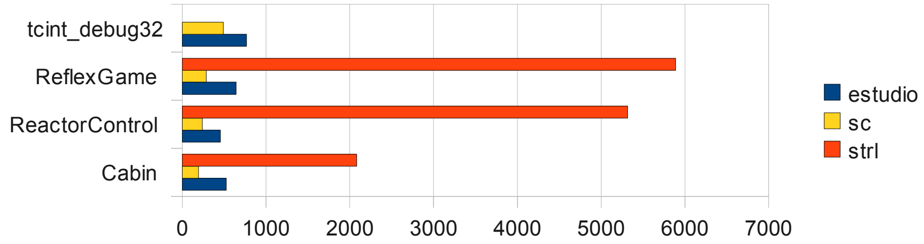


Abbildung 7.7: Größe in LOC für mittelgroße SyncCharts

Für die Größenmessungen in Abbildung 7.6 und 7.7 ergibt sich ein ähnliches Bild, wie für die kleinen SyncCharts. Der Vorteil der geringen Codegröße im vergleich zum CEC–Compiler wird hier jedoch deutlicher.

Große SyncCharts

Für die großen SyncCharts wurden ebenfalls zwei Modelle aus dem KIEL–Tool importiert. Um ein sehr großes SyncChart mit vielen parallelen Regionen und vielen Hierarchien zu testen, wurde ABRO mehrmals ineinander verschachtelt. Dieses SyncChart ist demnach ein rein synthetisiertes Modell, das in dieser Art in der Realität kaum Anwendungen finden wird. Für die Beispiele *multipleAbro* und *decos_monitor* konnte der CEC ebenfalls keinen Code erzeugen. Außerdem existiert kein Vergleich für *decos_monitor* zu E-Studio, da dieses SyncChart sehr groß ist und kein Modell dafür in E-Studio existiert. Dieses SyncChart ist jedoch von der Größenordnung den anderen beiden ähnlich.

Der Geschwindigkeitsvorteil des Compilers aus E-Studio wird in Abbildungen 7.8 sehr deutlich. Besonders für das synthetisch erzeugte *multipleAbro* zeigt sich, dass tiefe Hierarchien und viel Parallelität, die Geschwindigkeit der vom SC–Compiler

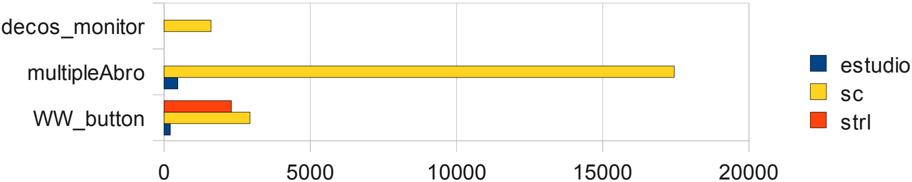


Abbildung 7.8: Zeitmessung für große SyncCharts in Taktzyklen

erzeugten Programme für große SyncCharts stark beeinflussen. Es zeigt sich aber auch, dass die Programme für ähnlich große SyncCharts wie *decos_monitor* mit der Geschwindigkeit mithalten können, wenn Hierarchien und Parallelitäten praktikabel verwendet werden.

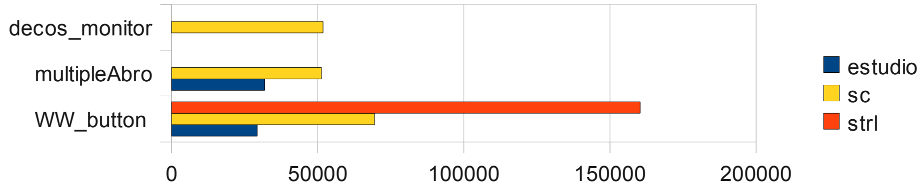


Abbildung 7.9: Größe der ausführbaren Datei für große SyncCharts in Byte

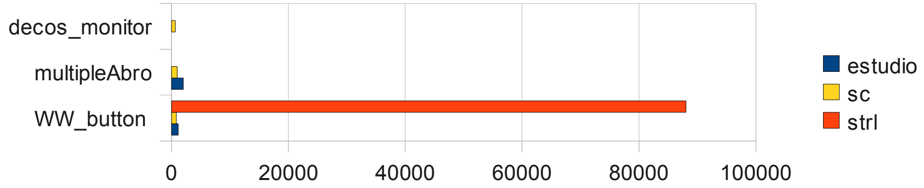


Abbildung 7.10: Größe in LOC für große SyncCharts

In den Abbildungen 7.9 und 7.10 zeigt sich widerum ein ähnliches Bild für die Größe der ausführbaren Datei und die LOC–Größe, zu den vorherigen kleinen und mittelgroßen SyncCharts.

Vergleich zum Esterel CEC–Compiler für virtuelle Maschinen (VM)

Für die Vergleich zum CEC–Compiler für VMs wurden die kleinen SyncCharts aus dem obigen Abschnitt verwendet. Aufgrund der beschriebenen Limitierungen, ist ein Vergleich zu mittelgroßen und großen SyncCharts nicht möglich. Abbildungen 7.11 macht deutlich, dass durch die Codegenerierung des CEC–Compilers für virtuelle Maschinen ein Verlust in der Performance zu verzeichnen ist. Die erzeugten Programme des SC–Compilers sind deutlich schneller. Ein ähnliches Bild zeigt sich auch

in der Messung der LOCs. Auch hier ist der SC–Code in den meisten Fällen deutlich kompakter.

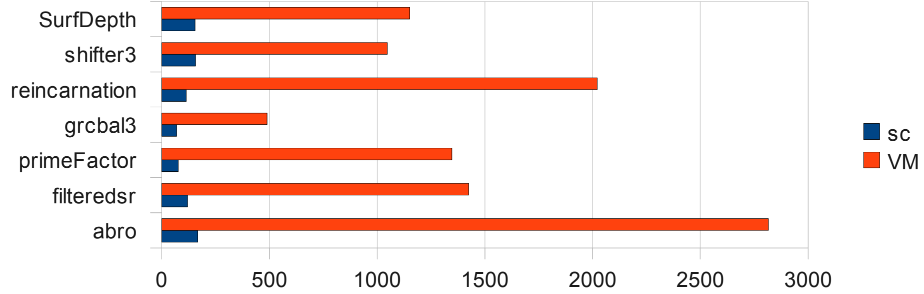


Abbildung 7.11: Zeitmessung im Vergleich zu CEC in einer VM in Taktzyklen

Einen Vorteil hat der CEC für Beispiel *grcbal3*, sowohl für die Größe des erzeugten Programms als auch für die Codezeilen (Abbildung 7.13). Ein Grund dafür könnten die vielen *conditional-pseude*–Zustände im SyncChart sein, für die im SC–Code jeweils ein Zustand erzeugt wird. Für die restlichen SyncCharts sind die LOC–Messungen für beide Methoden nahezu identisch.

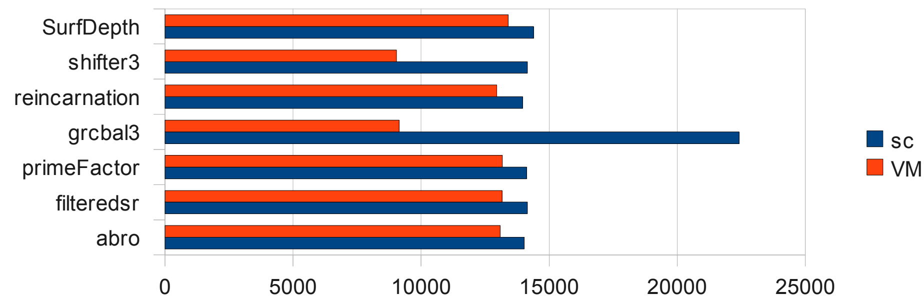


Abbildung 7.12: Größe der ausführbaren Datei im Vergleich zu CEC in einer VM in Byte

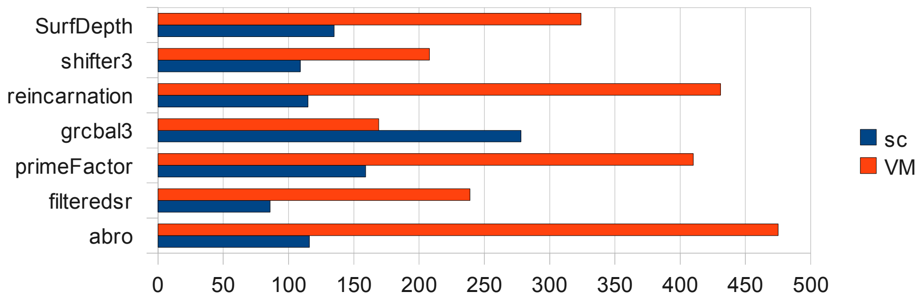


Abbildung 7.13: Größe in LOC im Vergleich zu CEC in einer VM

Zusammenfassung

Die Zeitmessungen zeigen, dass der vorgestellte Compiler für kleine SyncCharts nah an den handgeschriebenen Code heran reicht. Die vom CEC erzeugten Programme benötigen in der Regel mehr Taktzyklen als die vom Compiler erzeugten. Lediglich der E-Studio–Compiler ist von der Geschwindigkeit stets der schnellste. Das zeigt was durch weitere Optimierungen noch möglich ist. Für die großen und mittelgroßen SyncCharts wird dies noch deutlicher und insbesondere im Vergleich zu E-Studio sind zeitliche Unterschiede zu verzeichnen. Der Vergleich mit dem CEC–Compiler zeigt allerdings auch, dass die Ergebnisse dort in etwa der gleichen Größenordnung liegen.

Für die Messung der Größe ergibt sich für den entwickelten SC–Compiler ein anderes Bild. Wo die Ergebnisse für die Größe des generierten Programms vergleichbar sind, ist die LOC–Größe des erzeugten Codes sowohl im Vergleich zu dem Code vom CEC als auch zu dem für E-Studio geringer. Damit ergibt sich für den Entwickler eine Datei, die in ihrer Größe überschaubar bleibt und somit einfacher zu modifizieren ist. Dies wird zusätzlich durch den hohen Bezug zum SyncChart verstärkt.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Compiler vorgestellt, der aus SyncCharts direkt Synchronous C (SC) erzeugt, ohne dabei auf eine Zwischen-Sprache zurück zu greifen. Durch die direkte Übersetzung der SyncCharts und die Struktur der Sprache SC existiert ein hoher Bezug vom Code zum Modell. Das hat zur Folge, dass der Code auch für größere SyncCharts noch gut lesbar ist. Der Compiler ist als Plugin im KIELER-System integriert, wobei der Compiler neben der Codesynthese zur Simulation von SyncCharts verwendet werden kann.

Die Übersetzung der SyncCharts wird direkt vollzogen, indem alle existierenden Abhängigkeiten (Signale, Kontrollfluss, Hierarchie) ermittelt und in einen Abhängigkeitsbaum festgehalten werden. Ist dieser Abhängigkeitsbaum azyklisch, kann er topologisch sortiert werden und das Ergebnis liefert eine Ausführungsreihenfolge für den zu erzeugenden Code. Es werden Thread-Prioritäten berechnet, mit deren Hilfe die deterministische Natur der SyncCharts im generierten SC-Code erhalten bleibt. Der Vorteil der engen Beziehung zum SyncChart, der durch diese Methode resultiert, könnte geringe Einbußen in der Performance nach sich ziehen. Ein Zwischenschritt in eine Struktur, die den Kontrollfluss der SyncCharts besser widerspiegelt, könnte eine feingranularere Optimierung zur Folge haben.

Die Ergebnisse im Vergleich zu anderen, etablierten Methoden der Übersetzung von SyncCharts nach C zeigen, dass der Compiler für kleinere und mittelgroße SyncCharts ähnliche Ergebnisse für Performance und Programmgröße liefert. Die Größe der generierten SC–Dateien kann sich durch vergleichsweise Kompaktheit auszeichnen. Dessen ungeachtet steckt in der Entwicklung noch genügend Potential für weitere Optimierungen. Gerade die Vermeidung unnötiger `PRIO`-Anweisungen, welche die Laufzeit erhöhen, bietet in Sachen Geschwindigkeit noch Gelegenheit zur Verbesserung. Zusätzlich kann durch Verfeinerung und Hinzufügen von Optimierungsregeln der Anteil von unerreichbarem oder überflüssigen Code weiter minimiert werden.

Neben den Möglichkeiten der Verbesserung und Erweiterung des Compilers gibt es noch andere Dinge die im weiteren Umfeld getan werden können. Aktuell befindet sich eine Arbeit in der Entwicklung, in der versucht wird, SC-Code auf mehrere Kerne zu verteilen. Um den erhofften Geschwindigkeitszuwachs — gerade für sehr große SyncCharts— zu nutzen, muss der Compiler eventuell für die Verwendung dafür angepasst werden.

Ein weiterer Schritt, der den Compiler direkt betreffen würde, wäre eine genaue semantische Betrachtung von Synchronous C (SC). Dies könnte noch verfeinert und

an einer Verifizierung des Compilers gearbeitet werden. Aktuell unterliegen die Ergebnisse einer Validierung mit einer Reihe von sinnvollen Beispielen. Eine weitere Arbeit, die sich momentan in der Entwicklung befindet, ist Synchronous Java (SJ). SJ ist das Java–Pendant zu SC und der Compiler könnte durch geringe Modifikationen auch SJ-Code erzeugen. Damit wäre eine engere Anbindung an KIELER möglich, was zusätzliche Vorteile für Visualisierungen mit sich brächte.

Literaturverzeichnis

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

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

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

[4] Michael von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

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

[7] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] Marian Boldt. A compiler for the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-dt.pdf`.

[9] Julien Boucaron. Compilation de synccharts en automate implicite. Technical report, Université de Nice Sophia Antipolis, DEA d'Informatique, 06 2004.

[10] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. pages 527–543, 2002.

[11] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[12] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse, 2nd Edition*. Addison-Wesley Professional, 2 edition, 11 2004. `http://www.jdg2e.com/`.

[13] Berthold Daum. *Rich-Client-Entwicklung mit Eclipse 3.3*. Dpunkt.verlag Gmbh, 2007.

[14] Doron Drusinsky. *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*. Newnes, 1 edition, 4 2006.

[15] Katrin Erk and Lutz Priese. *Theoretische Informatik: Eine umfassende Einführung (Springer-Lehrbuch) (German Edition)*. Springer, 2., erw. aufl. edition, 10 2001.

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

[17] Emden R. Gansner. Drawing graphs with GraphViz. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 2004.

[18] Nicolas Halbswachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[19] Sören Hansen. Configurations and automated execution in the kieler execution manager. Bachelor project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, April 2010. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/soh-bt.pdf`.

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

[21] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[22] Volker Koster. Implementation and integration of a domain specific language with oaw and xtext. Technical report, MT-AG, 2007. http://www.mt-ag.com/web/download/experts_library/special_interest_artikel/Implementation%20and%20Integration%20of%20a%20DSL.pdf.

[23] Hans-Josef Köhler. Codegenerierung für uml-collaborations-, -sequenz- und -statechart-diagramme. Diploma thesis, Universität Paderborn, March 2000. http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Diplom/2000/DiplomHJKoehler.pdf.

[24] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006.

[25] Xin Li and Reinhard von Hanxleden. Multi-threaded reactive programming—the Kiel Esterel Processor. *IEEE Transactions on Computers*, accepted 2010.

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

[27] 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.

[28] 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.

[29] Object Management Group. Unified Modeling Lanugage—UML resource page, 2005. http://www.uml.org.

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

[31] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen (Sav Informatik) (German Edition)*. Spektrum Akademischer Verlag, 4. aufl. edition, 1 2002.

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

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

[34] 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.

[35] 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`.

[36] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. Automatic layout of data flow diagrams in KIELER and Ptolemy II. Technical Report 0914, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.

[37] Falk Starke, Claus Traulsen, and Reinhard von Hanxleden. Executing Safe State Machines on a reactive processor. Technical Report 0907, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, March 2009.

[38] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2 edition, 12 2008.

[39] Olivier Tardieu and Stephen A. Edwards. Instanteneous transitions in esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, March 2007.

[40] The Eclipse Foundation. Build your own textual dsl with tools from the eclipse modeling project. `http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html`.

[41] Reinhard von Hanxleden. SyncCharts in C. Technical Report 0910, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2009.

[42] 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.

[43] Andrzej Wasowski. On efficient program synthesis from Statecharts. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems (LCTES'03)*, volume 38, issue 7, June 2003. ACM SIGPLAN Notices.

[44] Andrzej Wasowski. Flattening statecharts without explosions. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 257–266, New York, NY, USA, 2004. ACM Press.