Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Synchronous Java

Mirko Heinold

30. September 2010

Institut für Informatik
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
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,

Zusammenfassung

Diese Arbeit beschreibt Aufbau und Implementierung eines Java-Frameworks zur synchronen Programmierung. Dieses Framework erlaubt auch eine einfache Darstellung der grafischen synchronen Sprache SyncCharts in Java.

Das hier entwickelte Synchronous Java (SJ) hat viele Gemeinsamkeiten mit Synchronous C (SC), einer makrobasierten C-Erweiterung zur synchronen Programmierung. Es wird der Entwurf sowie die Implementierung des Frameworks vorgestellt. Des Weiteren werden Probleme beim Umsetzen synchroner Konzepte in Java erörtert. Auch Probleme, die beim Realisieren von C-Programmen in Java auftreten, werden diskutiert.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Aufgabenstellung 1
1.2. Aufbau des Dokuments 2

2. Verwandte Arbeiten **5**
2.1. Synchrone Sprachen 5
2.2. Esterel 6
2.3. SyncCharts 9
2.4. Synchronous C 10

3. Verwendete Technologien **13**
3.1. Java 13
3.1.1. Java-Bytecode 13
3.1.2. Java-Reflections 15
3.1.3. JUnit Test-Framework 15
3.2. JavaScript Object Notation 16

4. Designentscheidungen **17**
4.1. Nebenläufigkeit 18
4.2. Zeitliche Abfolge 19
4.3. Synchrone Befehle 20
4.4. Realisierung von Signalen 21
4.5. Steuerung des Programmflusses 22
4.5.1. Sprungmarken für break und continue 22
4.5.2. Sprünge im Java-Bytecode 23
4.5.3. Nutzen von Methoden 25
4.5.4. Nutzen einer Switch-Anweisung 28
4.5.5. Vergleich der Lösungsmöglichkeiten 29

5. Implementierung **33**
5.1. Signale 33
5.1.1. Abfrage vorheriger Signalzustände 33
5.1.2. Deklaration von Signalen 34
5.1.3. Valued Signals 36
5.1.4. Kausalitätsprüfung 37
5.2. Programmfluss 38
5.3. Scheduling 40

5.4. Initiales Warten 41
5.5. Preemptionen 41
5.6. Debugging 42
5.7. Fehlerbehandlung 43
5.8. Testen und Validieren 45

6. Bewertung und Ausblick **47**
6.1. Anwendungsszenarien 47
6.2. Mögliche Erweiterungen und Ergänzungen 48
6.3. Fazit 49

Literaturverzeichnis **51**

A. Verwendung des Frameworks **53**

Abkürzungsverzeichnis

API Application programming interface

BASIC Beginner's All-purpose Symbolic Instruction Code

JSON JavaScript Object Notation

Java SE Java Platform, Standard Edition

Java ME Java Platform, Micro Edition

JDK Java Development Kit

JVM Java Virtual Machine

leJos Java for LEGO Mindstorms

SC Synchronous C

SJ Synchronous Java

SSM Safe State Machine

UML Unified Modeling Language

Abbildungsverzeichnis

1.1. Anwendungsszenarien 2

2.1. Zeitstrahl zum abgebildeten Esterel-Programm 7
2.2. ABRO als SyncChart 9

3.1. Java-Bytecode 14

4.1. Bestandteile des SJ-Frameworks 17
4.2. Kooperatives Scheduling in SJ 19
4.3. Zeitliche Abfolge von Befehlen in SJ 20
4.4. Programmzustände in SJ 22
4.5. Programme mit identischem Bytecode 25
4.6. Vergleich Switch-Case und Methodenansatz 31

5.1. Klassendiagramm von „Signal“ 34
5.2. Klassenhierarchie von SJ-Programmen 36
5.3. Zustände eines SJ-Programms 38
5.4. Zustände eines SC-Programms 38
5.5. Programmablauf von doTick() 39
5.6. Zustände eines SJThreads 40
5.7. Beeinflussung der Zustände eines SJThreads 41
5.8. SJ Fehlermeldung 44
5.9. Automatisiertes Testen von SJ 45

A.1. Erstellen eines SJ-Programms 56

Listings

2.1. Verdeutlichung eines Ticks . . . 7
2.2. Verdeutlichung eines Ticks (fortges.) . . . 7
2.3. ABRO als Esterel-Programm . . . 9
2.4. SC-Programm ABRO . . . 11

3.1. Java-Programm EvenNumbers . . . 14
3.2. Bytecode zu EvenNumbers . . . 14
3.3. Bytecode Assembler Darstellung von EvenNumbers . . . 14
3.4. Beispielhafte Nutzung von Java-Reflections . . . 15
3.5. JSON-Beispiel . . . 16

4.1. Sprungmarkenbenutzung mit break . . . 23
4.2. Java-Bytecode mit goto . . . 24
4.3. Hello World mit Label . . . 25
4.4. Hello World ohne Label . . . 25
4.5. Tick-Funktion der Oberklasse des Methodenansatzes . . . 26
4.6. Struktur eines methodenbasierten SJ-Programms . . . 27
4.7. Struktur eines Switch-Case basierten SJ-Programms . . . 28
4.8. ABRO mit Methodenansatz . . . 31
4.9. ABRO mit Switch-Case Ansatz . . . 31

5.1. Signaldeklaration in ABRO . . . 35
5.2. Alternative Signaldeklaration in ABRO . . . 35
5.3. Kausalitätsprobem in Esterel . . . 37
5.4. Kausalitätsprobem in SJ . . . 38
5.5. Protokollierung im JSON-Fromat . . . 43

A.1. main-Methode zum Steuern von ABRO . . . 53
A.2. Komplettes ABRO in SJ . . . 54

Tabellenverzeichnis

2.1. Befehle in Esterel 8

4.1. Umgang mit Signalen in SJ 21
4.2. Pros und Kontras des Methodenansatzes 30

5.1. Fehlermeldungen in SJ 44

1. Einleitung

Im Bereich der Echtzeitsysteme und eingebetteten Systeme laufen häufig viele Prozesse in der betrachteten Umwelt gleichzeitig ab. Es sollen mehrere anliegende Signale gleichzeitig betrachtet und auf ihren Wert reagiert werden.

Trotz vieler, gleichzeitig ablaufender Prozesse sollen die Ausgaben dieser reaktiven Systeme vorhersagbar sein. Dies ist besonders wichtig, wenn diese Systeme in sicherheitskritischen Bereichen eingesetzt werden [2]. Ein Realisieren der Nebenläufigkeit mit Threads, so wie sie in Java oder C verwendet werden, ist jedoch schwer, fehleranfällig und größtenteils nicht deterministisch [13].

Um Nebenläufigkeit, vor allem für reaktive Systeme, zu realisieren, wurde das Konzept der synchronen Sprachen entworfen [6]. Dieses Konzept erlaubt das Realisieren von Nebenläufigkeit bei gleichzeitigem deterministischen Verhalten [4].

1.1. Aufgabenstellung

Diese Arbeit beschäftigt sich mit der Bereitstellung eines Java-Frameworks zur synchronen Programmierung, im folgenden Synchronous Java (SJ)-Framework genannt. Ziel ist es, Konstrukte zum Nutzen von synchronen Programmierkonzepten in Java bereitzustellen. Die mit diesen Konstrukten arbeitenden Programme werden in dieser Arbeit als SJ-Programme bezeichnet. Zusätzlich zum Ermöglichen einer synchronen Programmierung in Java soll mit Hilfe dieses Frameworks die automatische Umwandlung von synchronen Sprachen nach Java vereinfacht werden. Hierbei liegt ein besonderes Augenmerk auf der grafischen, synchronen Programmiersprache SyncCharts [2].

Da SJ auch für eingebettete Systeme genutzt werden soll, müssen bei der Entwicklung des Frameworks auch die Bedürfnisse und Besonderheiten dieser Systeme berücksichtigt werden.

Die Hauptanwendungsszenarien für SJ sind in Abbildung 1.1 als ein UML-Anwendungsfalldiagramm dargestellt. Der Anwendungsfall, SyncCharts nach SJ zu übersetzen, ist nicht Teil dieser Arbeit. Dennoch wird beim Entwurf des Frameworks darauf geachtet, dass dieses eine automatische Übersetzung von SyncCharts nach Java vereinfacht.

Eine zentrale Motivation hinter der Entwicklung von SJ ist, dass synchrone Sprachen wie Esterel [6], SyncCharts oder Lustre [8] weniger weit verbreitet sind als beispielsweise C oder Java [9]. Ziel ist es, die vergleichsweise hohe Zahl an Java-Programmierern auch für synchrone Programmierung, beispielsweise von reaktiven Systemen, einsetzen zu können. So sollen Vorteile synchroner Sprachen, wie

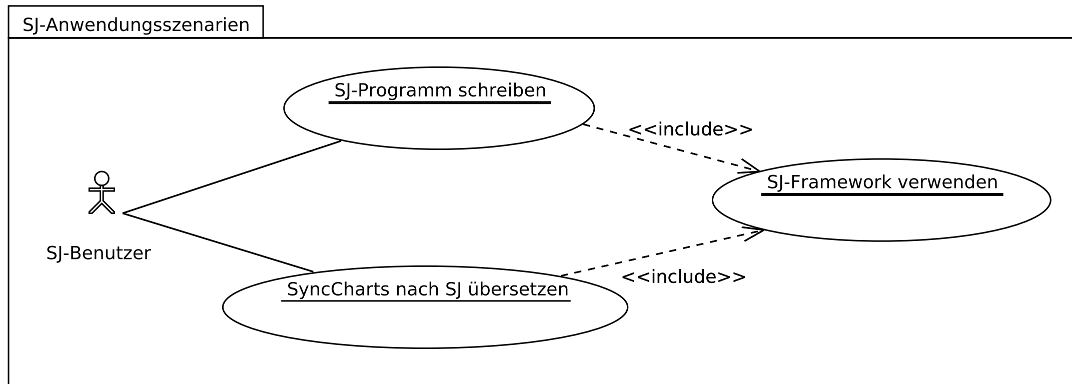

Abbildung 1.1.: Anwendungsszenarien für in SJ

beispielsweise deterministische Nebenläufigkeit, mit Vorteilen von Java, wie hohe Verfügbarkeit, kombiniert werden.

1.2. Aufbau des Dokuments

Im Folgenden werden zunächst die Grundlagen der synchronen Programmierung näher erläutert und eingeführt. Dieses wird erreicht, indem verwandte Themen und Arbeiten zu synchronen Sprachen eingeführt werden. Es werden hierzu die Konzepte synchroner Sprachen, Esterel, SyncCharts und die C-Erweiterung Synchronous C (SC) näher betrachtet

Im Kapitel 3 folgt ein Überblick über die verwendeten Technologien. Es handelt sich hierbei vor allem um Technologien im Zusammenhang mit der Programmiersprache Java. Auch das Datenaustauschformat JSON wird betrachtet, um später das Loggen von Befehlen in SJ besser erläutern zu können. Ziel ist es, das zum Verständnis der folgen Kapitel notwendige technische Wissen zu schaffen.

Im Kapitel 4 zu Designentscheidungen wird dann auf mögliche Wege einer Realisierung von SJ eingegangen. Es wird betrachtet, auf welche Weise Nebenläufigkeit, zeitliche Reihenfolge, ein synchroner Befehlssatz oder Signale realisiert werden können. Auch die Steuerung des Programmflusses ist Inhalt des Kapitels. Bieten sich unterschiedliche Möglichkeiten eines Designs für einen Teilbereich des Frameworks an, so werden diese verglichen.

Neben dem Kapitel Designentscheidungen ist das Kapitel 5, Implementierung von SJ, ein Schwerpunkt der Arbeit. Hier wird unter anderen die konkrete Implementierung eines Signals in Java ausführlich betrachtet. Außerdem werden die Implementierungen des Programmflusses, des Schedulings, des initialen Wartens und von Preemptionen diskutiert. In den letzten drei Abschnitten des Kapitels geht es um das Erkennen und den Umgang mit Fehlern.

Zum Abschluss findet eine Bewertung von SJ statt. Es wird ein Ausblick auf mögliche Anwendungsfelder gegeben und es werden noch zu erledigende Verbesserungen und Erweiterungen angesprochen. Danach wird im Anhang das Benutzen von SJ anhand eines Beispiels näher erläutert.

2. Verwandte Arbeiten

Um die Aufgabenstellung und die Eigenschaften des zu entwickelnden Frameworks näher bestimmen zu können, ist zunächst eine Betrachtung verwandter Arbeiten sinnvoll. Da es sich bei SJ um ein Framework zur synchronen Programmierung handelt, bedeutet ein Betrachten verwandter Arbeiten ein Auseinandersetzen mit dieser Programmierform.

Hierzu wird zunächst das Konzept hinter synchronen Sprachen betrachtet. Anschließend werden die Programmiersprachen Esterel, SyncCharts und SC eingeführt.

2.1. Synchrone Sprachen

Eine zentrale und wichtige Eigenschaft synchroner Programme ist der Determinismus [6, Seite 89]. Das bedeutet, dass das Verhalten eines Programms immer exakt vorhersagbar ist. Diese Eigenschaft sorgt insbesondere dafür, dass sich ein Programm bei gleichen Werten immer gleich verhält. Nach Berry et.al. sind deterministische Programme einfacher zu debuggen, zu analysieren und zu spezifizieren als nicht-deterministische [6, Seite 89]. Dies ist einer der Gründe warum deterministische Programme für die Programmierung reaktiver Systeme geeignet sind.

Ein trivialer Ansatz, Determinismus sicherzustellen, ist das Erlauben eines ausschließlich sequentiellen Programmablaufes. Ein klassischer, sequentieller Programmablauf ist allerdings nicht immer zum Modellieren der Realität geeignet, da in der Realität viele Aktivitäten parallel ablaufen. Synchrone Programme lassen deshalb Nebenläufigkeit zu, stellen aber aufgrund der im weiteren Verlauf erläuterten Eigenschaften trotzdem eine deterministische Programmausführung sicher.

Parallel ausgeführte Programmteile laufen gleichzeitig und unabhängig voneinander ab [15]. Nebenläufig ausgeführte Programmteile hingegen können auch nacheinander oder sich immer wieder abwechselnd ausgeführt werden. Nebenläufigkeit eignet sich als Abstraktion von Parallelität [10].

Um diesen Determinismus bei gleichzeitig erlaubter Nebenläufigkeit von Programmteilen sicherzustellen, erfüllen synchrone Sprachen die Synchronitätshypothese [6, Seite 91]. Sie trifft folgenden Annahmen [1]:

Perfect Synchrony bedeutet, dass alle Prozesse eines Systems gleichzeitig ablaufen und keine Zeit verbrauchen. Diese Gleichzeitigkeit bezieht sich auch auf einander folgende Anweisungen oder auf das Broadcast-Verhalten von Signalen. Aus diesem Konzept folgt auch, dass Ausgaben zur selben Zeit erzeugt werden, zu der auch die Eingaben gelesen werden. Dies geschieht auch, wenn Ausgaben von Eingaben abhängen.

Zero delay sagt aus, dass für den Übergang von einem Zustand des Programms in seinen Folgezustand keine Zeit verbraucht wird. Soll eine zeitliche Darstellung des Übergangs von einem Zustand in seinen Folgezustand erfolgen, so muss hierfür ein zusätzlicher Zustand eingeführt werden.

Multiform notion of time bedeutet, dass Zeit durch die Reihenfolge von Ereignissen bestimmt wird. Zeit wird ausdrücklich nicht durch physikalische Zeit bestimmt. Um physikalische Zeit zu modellieren, kann diese beispielsweise durch das Auftreten eines Eingabesignals modelliert werden. Meist werden zur Modellierung der physikalischen Zeit deshalb Signale benutzt, die als zu regelmäßigen Zeitpunkten, beispielsweise minütlich, auftretend definiert sind.

Um diese Annahmen sicherzustellen, wird bei synchronen Sprachen üblicherweise eine Modellierung anhand von Ticks vorgenommen.

Definition. *Alle während eines* ***Ticks*** *emittierten Signale werden als gleichzeitig auftretend angesehen. Mit einer Anweisung an die auszuführende Umgebung oder nach einer bestimmten Zeit gelangt man von einem Tick in den nächsten. Während dieses Ticks auftretende Signale werden als zeitlich nach den Signalen des vorigen Ticks auftretend angesehen.*

Um die abstrakte Darstellung von Zeit in synchroner Sprachen besser verstehen zu können, bietet sich eine Betrachtung der synchronen Programmiersprache Esterel an. Dieses ist ebenfalls sinnvoll da SyncCharts als grafische Darstellung von Esterel angesehen werden können [3].

2.2. Esterel

Die Konzepte der synchronen Programmierung lassen sich gut an der Programmiersprache Esterel [6] erläutern. Bei Esterel handelt es sich um eine klassische synchrone Programmiersprache. Sie wurde speziell für eine Ausführung auf reaktiven Systemen entworfen [6]. Esterel lässt sich mittels Compiler nach C übersetzen. Aufgrund der hohen Verbreitung von C lässt sich Esterel so auf vielen unterschiedlichen Systemen ausführen.

Esterel arbeitet mit Signalen, um Zustände zu repräsentieren. Bei Signalen kann es sich um Eingabesignale, Ausgabesignale oder lokale Signale handeln. Für lokaler Signale und Ausgabesignale gilt das *Logical Coherence Law*:

Definition. *Das* ***Logical Coherence Law*** *besagt, dass ein Signal vom Zustand* present *ist, genau dann, wenn es während eines Ticks emittiert wurde [5].*

Um dies zu erreichen, haben lokale Signale und Ausgabesignale neben den Zuständen *present* und *absent* auch den Zustand $\perp$ (*unknown*). Ist nicht eindeutig, ob ein Signal während eines Tick emittiert werden muss, so hat es den Zustand $\perp$.

Esterel bietet sich für eine nähere Untersuchung des synchrone Ausführungsmodell von Ticks an.

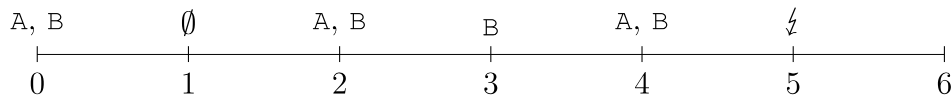

Abbildung 2.1.: Zeitstrahl zum abgebildeten Esterel-Programm

Listing 2.1: Verdeutlichung eines Ticks

```
1  module tickfunc:
2  %Demonstrate the function of logical
       ticks
3    signal A, B in
4
5      emit A;
6      emit B;
7      pause;
8      present B then emit A end;
9      pause;
10     emit B;
11     present B then emit A end;
```

Listing 2.2: Verdeutlichung eines Ticks (fortges.)

```
12     pause;
13     present pre(A) then emit B end;
14     pause;
15     [present A then emit B end
16     ||
17     emit A];
18     pause;
19     present B then emit A end;
20     emit B
21
22   end signal
23 end module
```

Abgebildet ist ein Esterel-Programm, welches in Listing 2.1 und Listing 2.2 unterteilt ist, sowie ein dazugehöriger Zeitstrahl in Abbildung 2.1. Im Esterel-Programm sind zwei lokale Signale `A` und `B` definiert, sowie eine Abfolge von Befehlen festgelegt. Die im Listing verwendeten Befehle sind in Tabelle 2.1 aufgeführt.

Die Synchronitätshypothese besagt unter Anderem, dass die Ausführung von Anweisungen keine Zeit verbraucht. Um diese Voraussetzung zu erfüllen, arbeiten synchrone Sprachen mit den im Unterkapitel 2.1 bereits eingeführten logischen Ticks. Auf Esterel bezogen bedeutet das Arbeiten mit Ticks, dass ein Befehl standardmäßig keine Zeit verbraucht. Für bestimmte Befehle gilt dies nicht, und man gelangt bei ihrer Ausführung von einem Tick in den nächsten. Welche Befehle im abgebildeten Beispiel Zeit verbrauchen, ist in Spalte 3 der Tabelle angegeben.

Im gezeigten Beispiel werden während des Ticks 0 die Signale `A` und `B` emittiert. Zu sehen ist dieses in Zeile 5 und 6. Der Zeitpunkt des Auftretens von `A` und `B` ist als gleichzeitig anzusehen, auch wenn die Anweisungen zum Emittieren dieser Signale im Programmcode nacheinander stehen. Die in Zeile 7 stehende **`pause`**-Instruktion sorgt für einen Wechsel in den nächsten Tick. Nach dessen Ausführung befinden wir uns im Tick 1. Mit Betreten eines neuen Ticks werden alle lokalen Signale wieder zurückgesetzt.

In abgebildeten Beispiel ist das Signal `B` also nicht mehr *present* und es wird somit in diesem Tick kein Signal emittiert. Im unter dem Programm abgebildeten Zeitstrahl ist das Verhalten der einzelnen Signale im jeweiligen Tick grafisch dargestellt. Unter der Linie steht jeweils die Zahl des an der Stelle beginnenden Ticks, und über der Linie die Menge der Signale, die in diesen Tick emittiert werden.

In Tick 3, 4 und 5 lassen sich weitere Eigenschaften von Esterel beobachten. In

Esterel-Befehl	Bedeutung	Verbraucht Zeit?

`signal <signal>`
`<statements>`
`end signal`
	Deklaration eines lokalen Signals	Nein

`emit <signal>`	Emittieren des angegebenen Signals	Nein
`pause`	Pausieren des Programmflusses bis zum nächsten Tick	Ja

`present <signal> then`
`<statements>`
`else <statements>`
`end present`
	Prüfen ob ein Signal *present* ist und wenn ja `<statements>` ausführen, andernfalls `<statements>`	Abhängig von `<statements>`

`pre(<signal>)`	Abfrage des Signalzustandes des vorausgegangenen Ticks	Nein
`<statements> ;` `<statements>`	Sequenzielle Ausführung von Programmteilen	Abhängig von `<statements>`
`<statements> ||` `<statements>`	Nebenläufige Ausführung von Programmteilen	Abhängig von `<statements>`

Tabelle 2.1.: Befehle in Esterel

Tick 3 sorgt der Befehl **pre** in Zeile 13 dafür, dass auf den Zustand des Signals aus dem vorherigen Tick zugegriffen werden kann. Aufgrund der Tatsache, dass A im vorausgegangenen Tick *present* war, wird B emittiert. In Tick 4 werden zwei Programmteile parallel ausgeführt. Zu der parallelen Ausführung kommt noch, dass der erste Programmteil vom zweiten abhängt. Es handelt sich aber nicht um eine gegenseitige Abhängigkeit. Deshalb kann Esterel aufgrund der zugrunde liegenden Logik damit umgehen und es werden sowohl A als auch B emittiert. In Tick 5 wird zuerst B geprüft und anschließend B emittiert. Aufgrund des Semikolons passieren diese Ausführungen nacheinander, auch wenn die Anweisungen keine Zeit verbrauchen. Da aber der Zustand eines Signals innerhalb eines Ticks eindeutig sein muss, liegt hier ein fehlerhaftes Programm vor. Esterel erlaubt solche Programme nicht und erkennt sie beim Kompilieren [7].

Ein weiteres Esterel-Programm wird im folgenden Abschnitt zu den mit Esterel verwandten SyncCharts erläutert.

2.3. SyncCharts

Bei SyncCharts [2], im kommerziellen Umfeld auch Safe State Machine (SSM) genannt, handelt es sich um eine grafische, synchrone Programmiersprache. SyncCharts wurden 1996 als eine grafische Darstellung von Esterel entworfen. SyncCharts sind deshalb sehr eng mit der Programmiersprache Esterel verwandt. Es ist möglich, beliebigen Esterel-Code in SyncCharts inline zu verwenden, oder auch aus SynchCharts automatisch Esterel-Code zu erzeugen [3].

SyncCharts erweitern in ihrer Darstellung Statecharts [11] und bieten zusätzlich die Darstellung eines synchronen Programmablaufes an. Bei Statecharts handelt es sich um ein hierarchisches Modell. Statecharts sind Mealy-Automaten, sind orthogonal, haben Broadcast-Kommunikation und unterstützen nebenläufige Modellierung [11]. SyncCharts erfüllen im Unterschied zu Statecharts die Synchronitätshypothese. Daraus folgt unter anderem ein deterministisches Verhalten und eine in jedem Zustand des Diagramms eindeutige Signalbelegung [2, 3]. Um die Möglichkeiten und Eigenschaften von SyncCharts besser verstehen zu können, betrachten wir Abbildung 2.2.

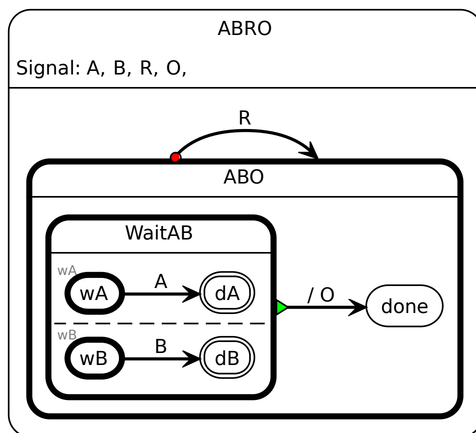

Abbildung 2.2.: ABRO als SyncChart

Listing 2.3: ABRO als Esterel-Programm

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

Die Abbildung zeigt das SyncChart ABRO. ABRO ist so etwas wie das *Hello world* der synchronen Sprachen [17]. In Listing 2.3 ist ABRO als Esterel-Programm abgebildet. Die Reaktion auf unterschiedliche Signalbelegungen ist bei beiden Programme identisch. Das Verhalten von ABRO lässt sich wie folgt, anhand seiner einzelnen Bestandteile, beschreiben:

- Das Signal R sorgt für einen Neustart (Reset) des Programmablaufes. Wann immer in einem Tick R anliegt, wird der bisherige Ausführungsstatus verlassen. Im SyncChart geschieht dies durch die Transition mit dem Bezeichner R. Der

rote Punkt symbolisiert dabei eine starke Abbruchbedingung, die dafür sorgt, dass sobald das Signal R anliegt der innere Programmablauf abgebrochen wird. Die dazu äquivalente Anweisung im Esterel-Programm ist das **each** R.

- Im abgebildeten SyncChart werden die in WaitAB zusammengefassten Zustände wA und wB parallel ausgeführt. Dargestellt wird dies durch die gestrichelte Linie zwischen den Zuständen. Auch die Befehle **await** A und **await** B im Esterel-Programm werden parallel ausgeführt. Das durch diese Abschnitte bestimmte Verhalten von ABRO lässt sich so beschreiben: Beim ersten Erreichen der Zustände (wA, wB) bzw. der Befehle (**await** A, **await** B) geschieht ein initiales Warten. In jedem Folge-Tick wird dann in den parallelen Teilen jeweils auf Signal A oder B gewartet.

- Die Signale A und B sind Eingabesignale. Sie werden also nicht vom Programm, sondern von der zugehörigen Umgebung emittiert. Das Auftreten der Signale ist in den parallel ausgeführten Programmteilen von Bedeutung. Bei ihrem Auftreten wird der entsprechende Programmteil, im SyncChart R0 oder R1, ausgeführt. Ein erneutes Auftreten eines Signals hat, nach Ausführen des entsprechenden Programmteils, erst wieder nach einem Auftreten von R eine erneute Wirkung.

- Das Signal O wird emittiert, sobald die beiden parallel auszuführenden Programmteile terminieren. Damit ein paralleler Zustand oder ein paralleles Statement terminiert, müssen alle parallel ausgeführten Programmteile terminieren. Danach befindet sich das Programm in einem Zustand, in dem erst ein Auftreten von R eine Veränderung bewirkt. Im abgebildeten SyncChart ist das der Zustand done.

  Vereinfacht ausgedrückt wird O emittiert, wenn die Signale A und B gleichzeitig oder nacheinander auftreten. Im initialen Tick werden A und B nicht weiter beachtet. Ist das Signal R *present*, so hat der von R verursachte Neustart des Programms Priorität gegenüber dem Ausgeben von O.

Dieses Beispiel gibt einen Eindruck von der Funktionsweise und der Syntax von SyncCharts. Außerdem wurden ein Esterel-Programm und ein SyncChart gegenübergestellt, und ihr gemeinsames Verhalten erläutert, um das Verständnis beider Sprachen zu vertiefen.

2.4. Synchronous C

Bei Synchronous C (SC)[1] handelt es sich um eine auf C-Makros basierende Erweiterung der Programmiersprache C. Ziel ist es, die formale Korrektheit und deterministische Nebenläufigkeit von Sprachen wie Esterel oder SyncCharts mit der Effizienz und weiten Verbreitung von C zu kombinieren [9]. SJ verwendet viele Ideen und

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

Konzepte von SC. Dieses bietet sich an, da SC wie SJ ein Realisieren synchroner Programmierung in einer nichtsynchronen Sprache erlaubt.

Ein großer Unterschied zu typischen synchronen Sprachen ist der Umgang mit Nebenläufigkeit. In Esterel oder auch SyncCharts wird die Ausführungsreihenfolge der im Programm als nebenläufig gekennzeichneten Teile automatisch bestimmt. In SC ist dieses aufgrund der von C bereitgestellten Konstrukte nicht möglich. Zwar unterstützt C mit Hilfe von POSIX und den darin definierten PThreads Nebenläufigkeit [12], diese ist aber nicht deterministisch.

Listing 2.4: Tickfunktion von ABRO in SC

```
1  int tick()
2  {
3    TICKSTART(4);
4
5   ABO:
6    FORK(AB, 1);
7    FORKE(ABOMain);
8
9   AB:
10   FORK(WaitA, 2);
11   FORK(WaitB, 3);
12   FORKE(ABMain);
13
14  WaitA:
15   AWAIT(A);
16   TERM;
17
18  WaitB:
19   AWAIT(B);
20   TERM;
21
22  ABMain:
23   JOIN;
24   EMIT(O);
25   TERM;
26
27  ABOMain:
28   AWAIT(R);
29   TRANS(ABO);
30
31   TICKEND;
32 }
```

Um Nebenläufigkeit in SC zu realisieren wird deshalb mit im Programmcode festgelegten Prioritäten gearbeitet [9]. Die nebenläufig ausgeführten Programmteile werden von uns im folgenden als Threads bezeichnet. Diese Threads werden beim erstellen durch die Befehle **FORK** und **FORKE** mit einer Priorität versehen. Die Priorität lässt sich während der Ausführung eines Threads mit der Anweisung **PRIO** verändern. Die Ausführungsreihenfolge der Threads hängt von ihren Prioritäten ab. Nach dem Ändern der Priorität, dem Erstellen eines Threads, oder nach dem Pausieren und Beenden eines Threads wird die Ausführungsreihenfolge neu bestimmt.

Um einen besseren Einblick in die Funktionsweise von SC zu bekommen, ist auf Abbildung 2.4 die Tickfunktion eines SC-Programms zu sehen. Es handelt sich hierbei

um das Programm ABRO, welches wir schon im Unterkapitel 2.3 betrachtet haben.

Die Funktionsweise von dieser ABRO-Implementierung in SC ist identisch mit der Esterel- und SyncChart-Variante. Aus diesen Grund geht es hier vor allem um die Implementierung von ABRO in SC und die konkreten Unterschiede gegenüber SyncCharts und Esterel.

Bei der Anweisung **`TICKSTART`** in Zeile 3, und der Anweisung **`FORK`** in Zeile 6, 10 und 11, lässt sich das Setzen von Prioritäten beobachten. Die Anweisung **`TICKSTART`** weist hierbei dem Programm die Startpriorität 4 zu. Mit **`FORKE`** werden dann die parallel auszuführenden Programmteile gescheduled. Danach wird das Programm entsprechend der beim Scheduling ermittelten Reihenfolge weiter ausgeführt. Die Anweisung **`TRANS`** in Zeile 29 sorgt bei Ausführung für eine sofortige Unterbrechung aller aus diesem Programmteil erzeugten nebenläufigen Ausführungen. Anschließend sorgt **`TRANS`** dafür, dass an dem als Parameter übergeben Label das Programm fortgesetzt wird.

SC macht viel Gebrauch von in C möglichen Konstrukten, wie `goto`-Anweisungen, berechneten Sprunganweisungen (*gcc's computed goto extension*) [9] und Makros. In Java gibt es aber einige C-Techniken, wie den Präprozessor, nicht. Auch eine einfache `goto`-Anweisung existiert in Java nicht [18]. Deshalb ist ein einfaches Übertragen der Implementierung von SC nach Java nicht möglich. Zusätzlich bietet Java aber Objektorientierung mit den dazugehörigen Konzepten an. Für eine Java-Implementierung zur synchronen Programmierung, so wie SC dies für C ermöglicht, muss deshalb ein eigener Java-spezifischer Weg gefunden werden.

3. Verwendete Technologien

Nachdem die mit SJ verwandten Arbeiten betrachtet wurden, betrachten wir nun zur Implementierung von SJ verwendete Technologien. Da SJ in der Programmiersprache Java realisiert ist, handelt es sich bei den betrachteten Technologien überwiegend um Java-Techniken.

Zunächst wird deshalb die Programmiersprache Java kurz eingeführt. Danach werden Java-Bytecode, Java-Reflections sowie das Java-Test-Framework JUnit eingeführt. Anschließend geht es in diesem Kapitel um das Datenaustauschformat JSON.

3.1. Java

Bei Java[1] handelt es sich um eine objektorientierte und weit verbreitete Programmiersprache. Während die Grundkonzepte und die Basisfunktionsweisen unter Informatikern allgemein bekannt sind, bietet Java auch Spezialgebiete. Hinzu kommt eine hohe Anzahl an Frameworks und Bibliotheken [18].

Eine Besonderheit von Java ist, dass Java-Code nach dem Übersetzen in einer virtuellen Maschine ausgeführt wird. Diese Java Virtual Machine (JVM) genannte virtuelle Maschine organisiert die Ausführung eines Java-Programms.

Die für SJ betrachteten und verwendeten Konzepte von Java werden im Folgenden erläutert.

3.1.1. Java-Bytecode

Bei Java-Bytecode [14] handelt es sich um Instruktionen für die JVM. Java-Bytecode stellt eine komprimierte und für die JVM besser geeignete Form eines Java-Programms dar. Zur Verdeutlichung was Bytecode ist und wie er aussieht dient Abbildung 3.1.

Listing 3.1 zeigt das Java-Programm *EvenNumbers*, welches alle geraden Zahlen zwischen 0 und 99 ausgibt. Jedes Java-Programm wird mittels Java-Compiler in eine *Class*-Datei, den sogenannten Bytecode, übersetzt. Die Bytecode Übersetzung für EvenNumbers ist in Listing 3.2 zu sehen. Die nicht darstellbaren Zeichen wurden hierbei durch Punkte ersetzt.

In jeder *Class*-Datei befinden sich in binärer Form Java Virtual Machine Instruktionen. Für diesen Bytecode gibt es einen eigenen Instruktionssatz [14], ähnlich Assembler-Instruktionssätzen für gängige Prozessorarchitekturen. Die einzelnen Instruktionen sind im Bytecode codiert vorhanden.

[1] `http://www.java.com/de/`

Listing 3.1: Java-Programm EvenNumbers

```
1 class EvenNumbers{
2
3   public static void main(String[]
      args){
4     for(int i = 0; i < 100; i+=2){
5       System.out.println(i);
6     }
7   }
8
9 }
```

Listing 3.2: Bytecode zu EvenNumbers

```
.......2.........................
....()V...Code...LineNumberTable.
..main...([Ljava/lang/String;)V..
.StackMapTable...SourceFile...Eve
nNumbers.java....................
....EvenNumbers...java/lang/Objec
t...java/lang/System...out...Ljav
a/io/PrintStream;...java/io/Print
Stream...println...(I)V..........
......................*.........
........................I........
.<..d...........................
................................
.........
```

Listing 3.3: Bytecode Assembler Darstellung von EvenNumbers

```
1  Compiled from "EvenNumbers.java"
2  class EvenNumbers extends java.lang.
       Object{
3  EvenNumbers();
4    Code:
5     0: aload_0
6     1: invokespecial #1; //Method java/lang
         /Object."<init>":()V
7     4: return
8
9  public static void main(java.lang.String
       []);
10   Code:
11    0: iconst_0
12    1: istore_1
13    2: iload_1
14    3: bipush 100
15    5: if_icmpge 21
16    8: getstatic #2; //Field java/lang/
         System.out:Ljava/io/PrintStream;
17    11: iload_1
18    12: invokevirtual #3; //Method java/io/
         PrintStream.println:(I)V
19    15: iinc 1, 2
20    18: goto 2
21    21: return
22
23 }
```

Abbildung 3.1.: Java-Bytecode

Der Befehl **aload_0** ist zum Beispiel durch ein * (Hexadezimal 2a) im Bytecode codiert [14]. In Listing 3.3 ist die zum abgebildeten Java-Programm gehörende Bytecode Assembler Darstellung zu sehen. Dieser Code wurde mit Hilfe des Tools javap[2] direkt aus der erzeugten *Class*-Datei generiert.

Es lässt sich festhalten, dass es möglich ist, Programme direkt als Java Bytecode-Datei zu schreiben, ohne je eine Java-Datei dazu erstellt zu haben. Auch lässt sich der aus Java-Dateien erstellte Bytecode manipulieren. Es gibt sogar Compiler, die Code anderer Programmiersprachen in Java-Bytecode umwandeln. Der Compiler NestedVM[3] zum Beispiel übersetzt unter anderem C-Code in Java-Bytecode. Die Programmiersprache Scala[4] lässt sich ebenfalls in der JVM ausführen.

[2] http://download.oracle.com/javase/1.5.0/docs/tooldocs/solaris/javap.html
[3] http://nestedvm.ibex.org/
[4] http://www.scala-lang.org/

3.1.2. Java-Reflections

Bei Java-Reflections handelt es sich um eine Technik, mit der man zur Laufzeit Klassen und Objekte der JVM untersuchen kann [18]. Ein Beispiel, wozu diese Technik verwendet werden kann, ist in Listing 3.4 abgebildet.

Listing 3.4: Beispielhafte Nutzung von Java-Reflections[a]

```java
public String getStringProperty(Object object, String methodname) {
  String value = null;
  try {
    Method getter = object.getClass().getMethod(methodname, new Class[0]);
    value = (String) getter.invoke(object, new Object[0]);
  } catch (Exception e) {}

  return value;
}
```

[a]Beispiel von http://de.wikipedia.org/w/index.php?title=Reflexion_(Programmierung)&oldid=69865549

Zu sehen ist die Java-Methode `getStringProperty(...)`. Diese ruft eine als String übergebene Methode eines ebenfalls übergebenen Objektes auf. Anschließend wird der Rückgabewert dieser Methode als String zurückgegeben.

Hierzu wird zunächst in Zeile 4 mit Hilfe der Java-Reflection-API auf die als String übergebene Methode des übergebenen Objektes zugegriffen. Der Einfachheit halber besitzt die Methode keine Parameter. Anschließend wird in Zeile 5 diese Methode ausgeführt. Die Fehlerbehandlung, beispielsweise eine Fehlermeldung falls es eine Methode mit den übergebenen Namen nicht gibt, wurde für eine bessere Übersichtlichkeit im Listing weggelassen.

Mit Reflection-Techniken ist es beispielsweise möglich, in einer abstrakten Klasse Methoden der diese Klasse implementierenden Unterklasse aufzurufen. Die Java-Reflection-API ist seit dem JDK 1.1 Bestandteil von Java SE [16]. Implementierungen für Java ME[5] oder andere Java Implementierungen für eingebettete Systeme wie leJos[6] unterstützen Reflections allerdings häufig nicht. Ein Einsatz dieser Technik auf eingebetteten Systemen ist deshalb problematisch.

3.1.3. JUnit Test-Framework

Zum Testen von Java-Programmen gibt es das Java-Framework JUnit[7]. Es bietet Mechanismen zum Testen von Java-Klassen und Methoden an. Mit diesen Mechanismen lassen sich Testfälle für Programme oder Programmteile in Java schreiben.

Durch einen gute Integration in die Programmierumgebung Eclipse[8] ist ein hoher Automatisierungsgrad beim Testen mit JUnit möglich. Auch ein Verwenden von

[5]http://www.oracle.com/technetwork/java/javame/
[6]http://lejos.sourceforge.net/
[7]http://www.junit.org/
[8]http://eclipse.org/

JUnit mit den Tool Ant[9] ist leicht möglich und erlaubt das automatische Testen als Teil des Build-Prozesses eines Java-Programms [18]. JUnit eignet sich deshalb gut zum automatischen Testen von Programmen oder sogar ganzen Frameworks.

3.2. JavaScript Object Notation

Bei JavaScript Object Notation (JSON)[10] handelt es sich um ein einfaches Datenaustauschformat, dessen Ziel es ist, sowohl für den Menschen als auch für Maschinen leicht zu lesen zu sein ist. Mit JSON lassen sich komplexe Datenstrukturen einfach darstellen. Bei JSON handelt es sich um ein reines Textformat. In Listing 3.5 ist beispielhaft die JSON-Darstellung eines Bildes angegeben.

Listing 3.5: JSON Beispiel[a]

```
1  {
2    "Image": {
3      "Width":  800,
4      "Height": 600,
5      "Title":  "View from 15th Floor",
6      "Thumbnail": {
7        "Url":    "http://www.example.com/image/481989943",
8        "Height": 125,
9        "Width":  100
10     },
11     "IDs": [116, 943, 234, 38793]
12   }
13 }
```

[a]Beispiel von http://www.ietf.org/rfc/rfc4627.txt

[9]http://ant.apache.org/
[10]http://json.org/

4. Designentscheidungen

Wie in Kapitel 1 angesprochen, ist es Ziel dieser Arbeit, ein Framework zu entwickeln, das eine synchrone Programmierung in Java ermöglicht. Außerdem soll dieses Framework eine automatische Übersetzung von SyncCharts nach Java vereinfachen. Hierzu sollen typische Eigenschaften von SyncCharts, wie beispielsweise Nebenläufigkeit, möglichst einfach mit diesem Framework realisierbar sein.

Das zu entwickelnde Framework soll also Befehle, Signale und andere zur synchronen Programmierung benötigte Konzepte bereitstellten. Dazu müssen eine oder mehrere Java-Klassen geschrieben werden. Die Klassen werden, wie in Java üblich, in Pakete aufgeteilt. Einige der Bestandteile eines solchen SJ-Frameworks sind in Abbildung 4.1 dargestellt.

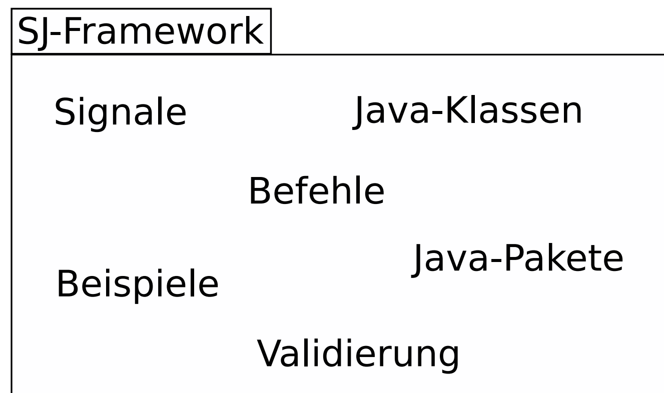

Abbildung 4.1.: Bestandteile des SJ-Frameworks (Basisbestandteile)

Dieses Framework soll vom Framework-Nutzer für seinen SJ-Programmcode genutzt werden. Die Nutzung des Frameworks soll mit üblichen Java Techniken wie Vererbung, Erstellen eines Objektes der Klasse oder direktem Zugriff auf statische Methoden und Variablen möglich sein. Inhalt dieses Kapitels ist es, ein Design für ein Framework zu finden, so dass dieses synchrone Programmierung in Java ermöglicht.

Wichtige Punkte bei der Bewertung unterschiedlicher Designansätze sind:

- Wie groß ist die Ähnlichkeit zu SC?
- Wie Java-typisch ist eine Herangehensweise und wie gut kommen Java-Programmierer mit diesem Ansatz zurecht?
- Wie einfach ist es mit diesem Ansatz SJ-Programme zu schreiben?

- Ist eine Nutzung von SJ auf unterschiedlichen Plattformen mit diesem Ansatz möglich?

Im folgenden Kapitel geht es darum, für unterschiedliche Problemstellungen im Zusammenhang mit der Entwicklung von SJ Lösungen zu finden. Zunächst werden dazu mögliche Realisierungen der deterministischen Nebenläufigkeit betrachtet. Danach geht es um die zeitliche Abfolge und die Realisierung eines synchronen Befehlssatzes. Im Anschluss wird ein Design für Signale in SJ diskutiert. Zum Abschluss des Kapitels findet eine intensive Auseinandersetzung mit der Steuerung des Programmflusses in SJ statt. Hierzu werden unterschiedlichste Ansätze wie Java-Sprungmarken, Sprünge im Java-Bytecode oder Switch-Anweisungen betrachtet.

Einige der zu treffenden Designentscheidungen wurden bereits bei der Entwicklung von SC [9] getroffen, andere sind Java-spezifisch. So gibt es zum Beispiel für die Realisierung der Nebenläufigkeit bereits eine brauchbare Lösung in SC.

4.1. Nebenläufigkeit

Betrachten wir nun zunächst mögliche Realisierungen der Nebenläufigkeit in SJ. Diese soll unter Sicherstellung des Determinismus in SJ realisiert werden.

Hierzu muss die Voraussetzung gelten, dass beim Bestimmen des als nächstes auszuführenden Threads der Zufall keine Rolle spielt. Es darf also nicht von Race Conditons, der Architektur, oder vom Betriebssystem abhängen, welcher Thread als nächstes ausgeführt wird. somit können für SJ keine Java-Threads oder andere nebenläufige Java-Konstrukte benutzt werden.

Da aber eine Nebenläufigkeit möglich sein soll, bietet sich hierfür ein Arbeiten mit dynamischen, im SJ-Programm zu setzenden Prioritäten an. Diese Prioritäten bestimmen die Ausführungsreihenfolge und das Umschalten zwischen den auszuführenden Programmteilen. Es findet also eine kooperatives Scheduling, zum Bestimmen des als nächstes auszuführenden Programmteils statt.

Durch Setzen und Ändern von Prioritäten wird zwischen mehreren sich in der Ausführung befindenden Programmteilen hin- und hergeschaltet. Diese Programmteile bezeichnen wir als SJ-Threads.

Das SJ-Programm gibt die Ausführungsreihenfolge anhand der im Code gesetzten Prioritäten vor. Es findet zu keiner Zeit eine physische gleichzeitige Ausführung von SJ-Threads statt.

Zur Verdeutlichung dieses Modells der Ausführung nebenläufiger Programmteile ist in Abbildung 4.2 die Ausführungsreihenfolge von drei SJ-Threads schematisch dargestellt. Die Ausführungsreihenfolge von SJ-Threads wird, wie bereits erwähnt, von ihren Prioritäten bestimmt. In der im Diagramm abgebildeten Ausführungsreihenfolge wird jeweils der Threads mit der höchsten Priorität ausgeführt. Ändert sich die Priorität, findet ein erneutes Scheduling zum Bestimmen des Threads mit der höchsten Priorität statt. Terminiert ein Thread, wird er beim Scheduling nicht mehr beachtet und es wird der Thread mit der nächsthöheren Priorität ausgeführt. Diese Festlegung

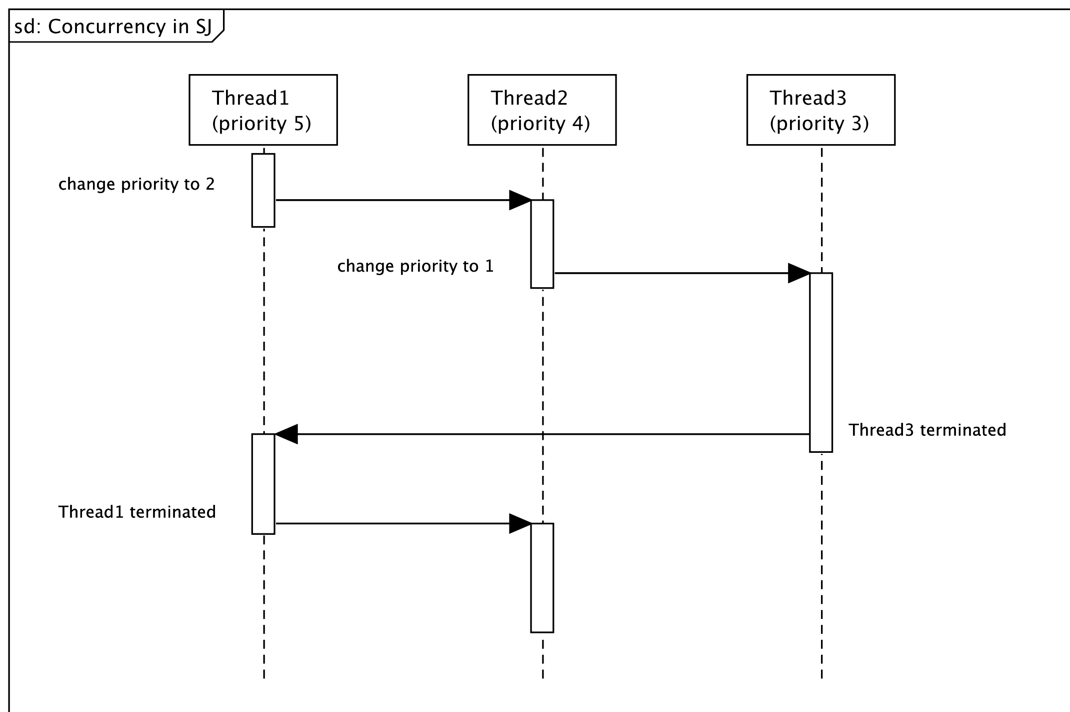

Abbildung 4.2.: Kooperatives Scheduling in SJ

der Ausführungsreihenfolge im Programm garantiert Determinismus und eine genau festgelegte Ausführungsreihenfolge aller Programmteile.

Diese Art der Realisierung von Nebenläufigkeit bietet sich zudem an, da auf diese Weise auch Nebenläufigkeit in SC gehandhabt wird [9]. Ein einheitliches Konzept beider Sprachen erleichtert das Lernen der jeweils anderen Sprache, bei Kenntnis einer. Auch führt ein einheitliches Verfahren zum Umgang mit Nebenläufigkeit in SJ und SC, bei Nutzern beider Sprachen zu weniger Verwirrung.

Ebenfalls für den in SC verwendeten Ansatz spricht, dass es eine automatische Übersetzung von SyncCharts nach SC [1] gibt. In diesem Übersetzungsverfahren wurde das richtige Ermitteln von dynamischen Prioritäten aus einem SyncChart bereits implementiert. Mit leichten Anpassungen kann dieser Mechanismus übernommen werden, um die Prioritäten bei einer Übersetzung von SyncChars nach SJ zu ermitteln. Dies vereinfacht das Erstellen einer automatischen Übersetzung nach SJ deutlich.

4.2. Zeitliche Abfolge

Als nächstes wird die Modellierung der zeitlichen Reihenfolge betrachtet. Auch hierfür muss eine Designentscheidung getroffen werden. Es soll, wie in synchronen Sprachen

üblich, mit logischen Ticks (siehe Unterkapitel 2.2) gearbeitet werden.

In Abbildung 4.3 wird dieses Konzept des zeitlichen Ablaufs dargestellt. Während eines Ticks werden jeweils nacheinander mehrere zu diesem Tick gehörende Befehle abgearbeitet. Nach Beenden aller Befehle eines Ticks kann der nächste Tick ausgeführt werden.

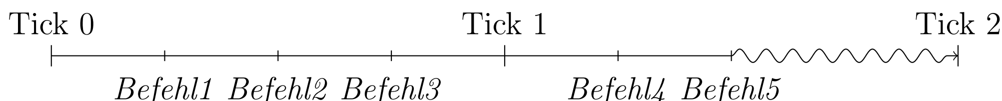

Abbildung 4.3.: Zeitliche Abfolge von Befehlen in SJ

Hierfür bietet sich eine Funktion an, die den jeweils nächsten Tick ausführt. Innerhalb dieser Funktion werden jeweils die zum aktuellen Tick gehörenden Befehle ausgeführt. Dazu muss eine Möglichkeit geschaffen werden, je nach Tick die dazugehörigen Programmteile auszuführen.

Dieses geschieht, indem an bestimmten Stellen im Code angegeben wird, dass der folgende Code erst im nächsten Tick auszuführen ist. Mit dem Befehl zu Ausführung eines Ticks wird dann jeweils bis zu dieser Stelle das Programm ausgeführt. Im nächsten Tick wird das Programm dann dort fortgesetzt.

Auch das hier erläuterte Konzept des zeitlichen Ablaufes wird in der hier beschriebenen Weise in SC realisiert [9].

4.3. Synchrone Befehle

Um synchrones Verhalten mit SJ modellieren zu können, benötigen wir einen Befehlssatz. Dieser ist unter anderem zur Modellierung von Zeit anhand von logischen Ticks nötig. Beim Arbeiten mit logischen Ticks ist zum Beispiel eine *pause*-Instruktion, die das Programm unterbricht und genau an dieser Stelle im nächsten Tick fortsetzt, unverzichtbar.

Hinzu kommen unter anderem noch Befehle zum Ändern der Priorität und zum Erstellen einer parallelen Ausführung von Programmteilen. Auch das Emittieren von Signalen muss in geeigneter Weise realisiert werden.

In Esterel ist dies Teil der Sprache [6]. In SC wurde ein solcher Befehlssatz mit Hilfe von C-Makros realisiert [9]. Beides ist in Java nicht möglich, da weder Makros, noch eine Möglichkeit eigene Befehle der Sprache hinzuzufügen, existieren.

Als Lösung dieses Problems bietet sich ein Arbeiten mit Methoden an. Die Idee ist, dass ein Arbeiten mit Methoden folgendermaßen ablaufen soll:

- Bestimmte Methoden sind SJ-Befehle.
- Durch Vererbung oder eine Instanz einer bestimmten Klasse greift man auf diese Methoden zu.

- Diese Methoden können wie Befehle aufgerufen werden, und die entsprechende Klasse sorgt für ein korrektes Verarbeiten.

Eine Klasse, die diese Methoden, welche den SJ-Befehlssatz bilden, zur Verfügung stellt, ist ein wesentlicher Bestandteil des SJ-Frameworks.

4.4. Realisierung von Signalen

Auch ein Design für Signale als Teil des SJ-Frameworks muss gefunden werden. Signale sind ein wesentlicher Bestandteil der synchronen Programmierung und müssen in SJ verwendet werden können.

In SC werden Signale mit Hilfe von Enumerations realisiert. Ein Weg, der Java-typischer ist, ist hier das Arbeiten mit Objekten. Diese erlaubt zudem, dass ein Signalobjekt mehrere Eigenschaften haben kann. So können die aktuelle Signalbelegung, ein Wert, vorherige Signalbelegungen, und ein Zugriff auf diese Eigenschaften alles Teil eines Signalobjektes sein.

Ebenfalls für die Realisierung von Signalen als Objekt spricht, dass das Arbeiten mit Objekten die für einen Java Programmierer vertraute Weise ist, mit solchen Problemstellungen umzugehen. Das Erstellen einer Klasse *Signal* als Teil des SJ-Framework für das Arbeiten mit Signalen stellt eine Java-typische Lösung dar.

Eine ebenfalls in Verbindung mit Signalen zu treffende Designentscheidung ist, wie Signale emittiert werden sollen und wie die Signalbelegung abgefragt werden soll. Auch ein Abfragen vorheriger Signalbelegungen soll möglich sein.

In der Tabelle 4.1 werden zwei Möglichkeiten eines Arbeitens mit Signalen in SJ vorgestellt. Außerdem beinhaltet die Tabelle jeweils die in Esterel und SC verwendeten

Esterel-Befehl	SC-Befehl	SJ-Befehl	Alternativer SJ-Befehl
`emit s`	`EMIT(s);`	`s.emit();`	`emit(s);`
`present s then`	`PRESENT(s);`	`s.present();`	`present(s);`
`pre(s)`	≈ `PRESENTPRE(s);`	`s.pre();`	`pre(s);`

Tabelle 4.1.: Umgang mit Signalen in SJ

Möglichkeiten. Der Buchstabe `s` steht dabei jeweils für ein beliebiges Signal.

Die beiden in SJ möglichen Varianten zum Arbeiten mit Signalen unterscheiden sich dadurch, dass in dem einen Fall die genutzten Methoden Teil eines Signalobjektes sind, und im anderem Fall Teil der Klasse mit SJ-Befehlen. Der Vorteil einer Realisierung von Signalen als Teil der Klasse mit SJ-Befehlen, dargestellt in Spalte 4 der Tabelle, ist die höhere Ähnlichkeit zu SC. Die andere Variante ist in Spalte 3 der Tabelle dargestellt. Sie hat den Vorteil, dass die Realisierung der Befehle zum Arbeiten mit Signalen als Teil eines Signalobjektes, eine Java-typischere Herangehensweise ist. Deshalb fällt die Entscheidung, wie die Befehle zur Signalabfrage und Manipulation realisiert werden, auf dieses Design.

4.5. Steuerung des Programmflusses

Zum Abschluss des Kapitels wird aufgeführt, wie eine Steuerung des Programmflusses in SJ realisiert werden kann. In SC wird dieses mittels im entsprechenden Code gesetzten Sprungmarken und vielfachen Nutzens des *goto*-Befehls gelöst. Das Ganze geschieht in den Makros der dazugehörigen Header-Datei.

Wie im Unterkapitel 2.4 schon erwähnt, ist der *goto*-Befehl, wie er aus Sprachen wie C oder BASIC bekannt ist, in Java nicht nutzbar [18]. Es muss also ein anderer Weg gefunden werden, den Programmfluss mehrerer Threads in SJ steuern zu können.

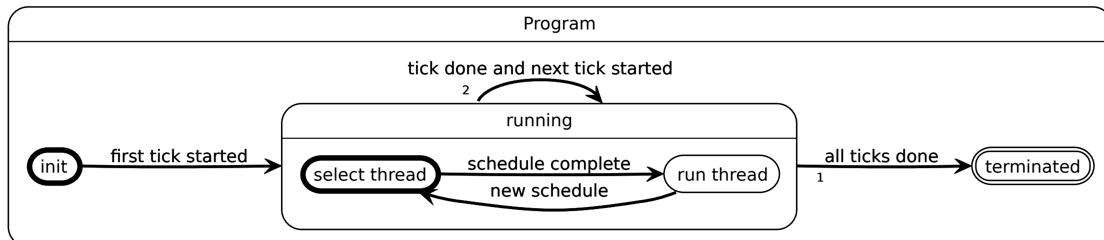

Abbildung 4.4.: Programmzustände in SJ

Einen Überblick über den zu steuernden Programmfluss und die Programmzustände bietet Abbildung 4.4. Um nebenläufigen Programmfluss zu realisieren, muss es möglich sein, nach Änderung einer Priorität zum aktuell auszuführenden Thread springen zu können. Es muss also, nach einem Ermitteln des auszuführenden Threads in select thread, der ermittelte Thread in run thread ausführbar sein.

Dafür müssen in irgendeiner Weise Sprungmöglichkeiten und Sprungmarken geschaffen werden. Sprungmarken werden im Programm häufig auch nach Anweisungen die ein Scheduling (new schedule) erfordern benötigt. Für eine Fortsetzung eines Threads nach einer Unterbrechung muss es einen Einstiegspunkt im Code geben.

Auch für die Modellierung anhand von Ticks muss der Programmfluss steuerbar sein. So müssen, je nach Tick und anliegenden Signalen, unterschiedliche Programmteile ausgeführt werden. Hierfür muss die Ausführungsreihenfolge der Programmteile jederzeit vom Programm frei wählbar sein.

4.5.1. Sprungmarken für `break` und `continue`

Zunächst betrachten wir eine Verwendung von Sprungmarken als Möglichkeit zur Steuerung des Programmflusses. Diese Sprungmarken, auch Label genannt, können prinzipiell an jeder Stelle im Quellcode stehen [18].

Neben der Nutzung in Switch-Case Konstrukten können diese Sprungmarken auch im Zusammenspiel mit `break` und `continue` verwendet werden. Hier ist es möglich, einen solchen Befehl mit einer Sprungmarke zu versehen. Ziel dieser Sprungmarken ist es, in einer verschachtelten Schleifenstruktur aus einer beliebigen Schleife, und nicht ausschließlich aus der innersten, springen zu können.

Damit diese Konzept des Springen innerhalb verschachtelter Schleifen besser verstanden werden kann, ist in Listing 4.1 ein Beispiel abgebildet.

Listing 4.1: Sprungmarkenbenutzung mit `break`

```
1   label1:
2   while(true) {
3
4     label2:
5     while(true) {
6       break label1;
7     }
8
9   }
10  System.out.println("Program terminated");
```

Befindet sich das Programm beim abgebildeten Quellcode in der inneren While-Schleife, so wird eine Sprungmarke benötigt, wenn wir mittels **`break`** die äußere While-Schleife verlassen wollen. Im abgebildeten Beispiel sorgt das **`break`** `label1` in Zeile 6 dafür, dass die äußere While-Schleife beendet wird und *Program terminated* auf der Konsole ausgegeben wird.

Die Sprungmarke gibt also an, dass sich das **`break`** auf die in Zeile 2 stehende, und mit der Sprungmarke `label1` versehende, Schleife bezieht. Ein **`break`** ohne Sprungmarke ist in diesen Beispiel äquivalent zu **`break`** `label2`.

Der Befehl **`continue`** verhält sich analog zu **`break`**, mit dem Unterschied, dass nicht zum Ende, sondern zum Anfang einer Schleife gesprungen wird.

Zum Springen an beliebige Zeilen im Programmcode ist der hier vorgestellte Mechanismus nicht nutzbar. Der Grund ist, dass der zu einer Schleife gehörende Bezeichner jeweils direkt vor dieser stehen muss. Würden in unserem Beispiel zwischen `label1` in Zeile 1 und **`while`**`(`**`true`**`)` in Zeile 2 weitere Anweisungen, wie beispielsweise `System.out.println()` stehen, würde das Programm nicht kompiliert werden.

Diese Vorschrift verhindert das Springen an beliebige Zeilen im Quellcode. Aus diesem Grund sind **`break`** und **`continue`** in Verbindung mit Sprungmarken für unsere Zwecke ungeeignet. Sie ersetzen keinesfalls ein `goto`, wie man dieses aus anderen Programmiersprachen kennt.

Wir können auf diese Weise nicht den sequentiellen Programmfluss gemäß unseren Vorstellungen zum Wechseln zwischen Programmteilen manipulieren.

4.5.2. Sprünge im Java-Bytecode

Wie im Unterkapitel 3.1.1 beschrieben steht mit den Java-Bytecode ein Assembler-Instruktionssatz für die JVM zur Verfügung. Wie außerdem beschrieben, lässt sich Java-Bytecode nicht nur aus Java-Code generieren sondern auch aus Code anderer Programmiersprachen.

Für SJ bedeutet das, dass theoretisch das gesamte Framework in Java-Bytecode-Assembler, oder einer nach Java-Bytecode übersetzbaren Sprache, geschrieben werden

kann. Der in einer anderen Sprache als Java geschriebenen Code könnte dann in der JVM ausgeführt werden.

Um diesen generierten Bytecode aus Java-Dateien heraus mit Methoden und Klassen aufrufen und nutzen zu können, muss der Code eine bestimmte Struktur haben [14]. Diese Struktur ist eng an die Vorschriften und Regeln eines Java-Programms angelehnt. Eine Möglichkeit zum Manipulieren dieser Struktur, bei Erhaltung dieser als gültige Java-Struktur, ist essentiell zum Steuern des Programmflusses. Lässt sich die Struktur nicht ausreichend anpassen, so ergeben sich durch Nutzung von Java-Bytecode keine zusätzlichen Möglichkeiten zum Steuern des Programmflusses.

Listing 4.2: Java-Bytecode mit `goto`

```
1  Compiled from "CountToTen.java"
2  public class CountToTen extends java.lang.Object{
3  public CountToTen();
4   Code:
5    0: aload_0
6    1: invokespecial #1; //Method java/lang/Object."<init>":()V
7    4: return
8
9  public static void main(java.lang.String[]);
10  Code:
11   0: iconst_0
12   1: istore_1
13   2: iload_1
14   3: bipush 10
15   5: if_icmpgt 21
16   8: iinc 1, 1
17   11: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
18   14: iload_1
19   15: invokevirtual #3; //Method java/io/PrintStream.println:(I)V
20   18: goto 2
21   21: return
22
23 }
```

Im Java-Bytecode Assembler-Instruktionssatz finden sich mehrere Sprungbefehle, wie beispielsweise eine `goto`-Instruktion [14]. Mit diesem Sprungbefehl lässt sich im Code zu beliebigen Labels springen. Allerdings sorgt der Bytecode-Verifier der JVM dafür, dass Sprünge innerhalb der Grenzen des Rumpfes einer Java Methode bleiben müssen [14].

Anhand von Listing 4.2 lässt sich diese Prinzip verdeutlichen. Das in Zeile 20 stehende **`goto`** kann zu beliebigen Instruktionen in `main` springen, nicht aber beispielsweise zu **`aload_0`**.

Nutzt man Java-Methoden so, dass diese von anderen Klassen als solche erkannt werden, sind Sprünge aus einer Methode in eine andere verboten. Es lässt sich also nicht der Kontrollfluss eines Java-Codes durch das Nutzen eines Framework beliebig ändern.

Es gibt ein weiteres Problem, das dafür sorgt, dass eine Nutzung der zusätzlichen Möglichkeiten von Java-Bytecode zum Steuern des Programmflusses von SJ keinen

Sinn macht. Will man mit einem in Bytecode geschriebenen Framework den Kontrollfluss eines in Java geschriebenen SJ-Programms manipulieren, muss man dazu in diesem die unterschiedlichen Programmteile identifizieren können.

Es muss also möglich sein, die nacheinander abhängig von ihren Prioritäten auszuführenden Programmteile einzeln anzusprechen. Hierzu kann man beispielsweise, wie in Abschnitt 4.5.3 erläutert, die einzelnen Programmteile in Methoden gliedern. Dieses ist allerdings auch ohne direktes Nutzen des Bytecodes mit Hilfe von Java-Reflections möglich.

Ein anderer Ansatz ist das Nutzen von Sprungmarken. Diese können wie bereits erwähnt, an beliebigen Stellen im Java-Code stehen. Stehen diese allerdings nicht wie im Abschnitt 4.5.1 beschrieben direkt vor einer Schleife, werden sie vom Compiler entfernt. Anschaulich dargestellt ist diese Verhalten in Listing 4.3 und Listing 4.4.

Listing 4.3: Hello World mit Label

```
1 class LabelTest {
2
3   public static void main(
        String[] args) {
4     myLabel:
5     System.out.println("Hello
          world!");
6   }
7 }
```

Listing 4.4: Hello World ohne Label

```
1 class LabelTest {
2
3   public static void main(
        String[] args) {
4
5     System.out.println("Hello
          world!");
6   }
7 }
```

Abbildung 4.5.: Programme mit identischem Bytecode

Wie zu erkennen, unterscheiden sich Listing 4.3 von Listing 4.4 durch das Label `myLabel`. Kompiliert man aber nun beide Java-Dateien, erhält man einen identischen Bytecode.

Ein Nutzen von Labels, um die einzelnen Programmteile zu kennzeichnen, ist nicht möglich. Anders sieht es mit einer Benutzung von Labels aus, wenn sie im Zusammenhang mit einem Switch Statement benutzt werden. Aber auch dafür ist kein Programmieren in Bytecode nötig.

4.5.3. Nutzen von Methoden

Eine weitere Möglichkeit zur Steuerung des Programmflusses ist das Nutzen von Methoden in Verbindung mit Java-Reflections. Die damit realisierbare Struktur eines SJ-Programms wäre dann eine Methode für jeden Programmteil, zu dem während der Ausführung eventuell gesprungen werden muss. Es handelt sich dabei zum Beispiel um Stellen im Programmcode, an denen nach einer Pause ein Programm im nächsten Tick fortgesetzt werden soll, oder Stellen an denen sich die Priorität ändert.

Das Aufrufen der einzelnen Methoden und das Bestimmen der Reihenfolge könnte bei diesem Konzept beispielsweise die Tick-Funktion der Oberklasse übernehmen.

Diese Oberklasse wäre bei diesem Konzept Teil des SJ-Frameworks. Dort würde man die jeweils nächste Methode ermitteln, und diese dann mittels Java-Reflections ausführen.

Eine beispielhafte Implementierung der Tick-Funktion einer solchen Oberklasse ist in Listing 4.5 zu sehen. In Zeile 2 wird dafür gesorgt, dass, solange der aktuelle

Listing 4.5: Tick-Funktion der Oberklasse des Methodenansatzes

```
1  public void tick() {
2    while (!isTickDone()) {
3      String nextState = state();
4      try {
5        Method m = this.getClass().getDeclaredMethod(nextState, new Class[0]);
6        m.invoke(this, new Object[0]);
7      } catch(InvocationTargetException ex) {
8        throw (RuntimeException)ex.getTargetException();
9      } catch(Exception e) {
10       e.printStackTrace();
11     }
12   }
13 }
```

Tick nicht abgeschlossen ist, der jeweils nächste Programmteil ausgeführt wird. In Zeile 3 wird der Methodenname des nächsten Programmteils ermittelt, und in Zeile 5 und 6 wird diese Methode aufgerufen. In den Zeilen 7 bis 11 geschieht dafür eine Fehlerbehandlung. Die abgebildete Methode führt einen kompletten Tick eines den Methodenansatz nutzenden SJ-Programms aus.

Für diesen Ansatz müssen die Methoden zum Steuern des Programmflusses die jeweiligen Methodennamen der auszuführenden Programmteile als Parameter enthalten. Beispielsweise müssen Befehle, die die parallele Ausführung von Programmteilen organisieren, neben der Priorität der Programmteile auch den Methodennamen, an dem der jeweilige Programmteil beginnt, erhalten.

Am besten lässt sich die bei einem solchen Ansatz verwendete Struktur eines SJ-Programms anhand eines Beispiels näher erläutern. In Listing 4.6 ist so eine Struktur abgebildet.

Dargestellt ist ein Java-Programm, welches von der Klasse *SJProgram* erbt. Diese Klasse definiert alle benötigten Methoden, sowie die Tick-Funktion zum Ausführen des Programms. Im Konstruktor des Programms wird in Zeile 6 der Konstruktor der Oberklasse aufgerufen und ihm mitgeteilt, dass das Programm mit der Methode `init` und der Priorität 4 beginnt. Beim ersten Aufruf der Tick-Funktion werden die Methode `init` und die darin stehenden Befehle ausgeführt.

Die Aufrufe von `fork` und `forke` sorgen dafür, dass `label1` und `label2` parallel ausgeführt werden. Bei `tickend` soll das Programm nach Aufrufen der parallelen Teile fortgesetzt werden.

Zu beachten ist, dass bei diesem Konzept, immer wenn sich der Programmfluss ändert, die aktuelle Methode beendet werden muss. Dieses ist beispielsweise nach einem `pause`, `goto` oder `forke` der Fall. An diesen Stellen muss nämlich der nun auszufüh-

Listing 4.6: Struktur eines methodenbasierten SJ-Programms

```
import sj.SJProgram;

public class StructureDemo  extends SJProgram {

    public StructureDemo() {
      super("init", 4);
      ...
    }

    public void init() {
      fork("label1", 2);
      fork("label2", 3);
      forkeB("tickend");
    }

    public void label1() {
      ...
      gotoB("label3");
    }


    public void label2() {
      ...
    }

    public void label3(){
      ...
    }

    public void tickend() {
      ...
    }
}
```

rende Programmteil bestimmt werden. Dieses Springen zum nächsten Programmteil ist nicht innerhalb der Methoden möglich.

Diese Vorschrift stellt einen wesentlichen Unterschied dieses Ansatzes zu SC dar. In SC muss beispielsweise nicht nach jeder *pause*-Instruktion ein neues Label stehen, um an dieser Stelle das Programm im nächsten Tick fortsetzen zu können. Dies führt dazu, dass bei diesem Ansatz meist mehr Methoden benötigt werden als Label für ein SC-Programm mit identischer Funktion. Ein weiterer Unterschied dieses Ansatzes zu SC ist, dass er mit Methoden und nicht mit Labels arbeitet. Es muss auch keine Tick-Funktion selbst implementiert werden. Dies sorgt für ein sich deutlich von SC abgrenzendes Erscheinungsbild.

Im Gegensatz zu den bisher vorgestellten Verfahren handelt es sich bei diesem methodenbasierten Ansatz um eine für unsere Zwecke alle Vorgaben erfüllende Variante. Auf die hier vorgestellte Weise lässt sich der Kontrollfluss eines SJ-Programms steuern. Mit der erläuterten Struktur und einem drauf abgestimmten SJ-Framework lassen sich dann SJ-Programme schreiben.

4.5.4. Nutzen einer Switch-Anweisung

Eine der Struktur von SC wesentlich ähnlichere Variante zum Steuern des Kontrollflusses ist das Nutzen einer Switch-Anweisung. Die Idee hinter diesem Ansatz ist, dass jeder Programmteil, der im Kontrollfluss erreichbar sein soll, ein eigenes Label innerhalb einer Switch-Anweisung erhält.

Diese Switch-Anweisung wird entsprechend der auszuführenden Programmteile mittels Schleife mehrmals ausgeführt. Bei jedem Ausführen der Switch-Anweisung wird dann ermittelt, welcher Programmteil aktuell auszuführen ist.

Eine Funktion, um diesen Programmteil zu ermitteln, muss für diesen Ansatz vom SJ-Framework zur Verfügung gestellt werden. Ist der auszuführende Programmteil ermittelt, wird dieser im Switch-Statement ausgeführt.

Listing 4.7: Struktur eines Switch-Case basierten SJ-Programms

```
1  import sj.SJProgram;
2
3  public class StructureDemo extends SJProgram {
4
5    enum StateLabel {
6      init, label1, label2, label3, tickend
7    }
8
9    public StructureDemo() {
10     super(inti, 4);
11     ...
12   }
13
14   public void tick() {
15     while (!isTickDone()) {
16       switch (state()) {
17         case init:
18           fork(label1, 2);
19           fork(label2, 3);
20           forkeB(tickend);
21           break;
22
23         case label1:
24           ...
25           gotoB(label3);
26           break;
27
28         case label2:
29           ...
30           break;
31
32         case label3:
33           ...
34           break;
35
36         case tickend:
37           ...
38       }
39     }
40   }
41 }
```

Um diesen Ansatz besser verstehen zu können, lohnt sich eine nähere Betrachtung von Listing 4.7. Abgebildet ist die Struktur eines SJ-Programms welches den Switch-Case Ansatz verfolgt. Mit Hilfe des in Zeile 17 stehenden `state()` wird der jeweils nächste auszuführende Programmteil ermittelt.

Bei diesem Ansatz muss die Tick-Methode der Oberklasse überschrieben werden. Ziel ist es, dass in dieser Methode das gesamte Verhalten des Programms während des jeweils aktuellen Ticks abläuft. Damit die Methode `tick()` jeweils einen gesamten Tick ausführt, und nicht nur den jeweils nächsten Programmteil, befindet sich in Zeile 16 eine While-Schleife. Sie sorgt dafür, dass der jeweils nächste Programmteil solange ausgeführt wird, bis alle Programmteile beendet sind, oder auf den nächsten Tick warten.

Um die einzelnen Fälle einer Switch-Anweisung mit Bezeichnern (Labels) versehen zu können, bietet sich hier ein Verwenden des Java Konstrukts **`enum`** an. In unserem Beispiel sind `init`, `label1,2,3` und `tickend` vom Typ **`enum`**. Alternativ zu Enumerations unterstützen Switch-Anweisungen nur die Datentypen **`byte`**, **`char`**, **`short`** und **`int`** [18]. Mit diesen Datentypen wäre ein Angeben der einzelnen Label weniger elegant.

Wie der methodenbasierte Ansatz führt auch das Arbeiten mit einer Switch-Anweisung dazu, dass meist mehr Labels benötigt werden, als bei einem äquivalenten SC-Programm. Zudem ist der für diesen Ansatz verwendete Konstruktor identisch mit dem des Methodenansatzes.

4.5.5. Vergleich der Lösungsmöglichkeiten

Nach der bisherigen Untersuchung bleiben zwei Möglichkeiten übrig, die für die Steuerung des Programmflusses im SJ-Framework verwenden werden können. Zum einen das Nutzen von Methoden und Java-Reflections, zum anderen das Nutzen einer Switch-Anweisung.

Um eine der beiden Möglichkeiten auszuwählen, werden diese im Folgenden anhand ihrer Vor- und Nachteile verglichen. Eine zusammengefasste Form der wichtigsten Vor- und Nachteile des Methodenansatzes ist dafür in Tabelle 4.2 dargestellt. Die Bedeutung der einzelnen Punkte wird dabei anhand von ein oder zwei plus- bzw. minus-Zeichen verdeutlicht.

Um einige dieser Vor- und Nachteile anschaulicher beschreiben zu können, ist in Listing 4.8 und Listing 4.9 eine ABRO Implementierung für beide Lösungsansätze abgebildet. Es handelt sich jeweils um die Auszüge aus dem gesamten Programm, welche das Programmverhalten festlegen.

Einfachheit Ein Vorteil des methodenbasierten Ansatzes ist, dass es bei diesen keinen immer wieder identisch hinzuschreibenden, vorgeschriebenen Code gibt. Bei den Switch-Case Ansatz ist dies der Fall. So ist der in Listing 4.9 in Zeile 23 und 24 stehende Code bei jedem nach diesem Ansatz geschriebenem SJ-Programm immer nahezu genau so zu implementieren.

Pro	Bewertung	Kontra	Bewertung
Es gibt keinen immer wieder hinzuschreibenden, vorgeschriebenen Code	+	Weniger Ähnlichkeit zu SC als Switch-Variante	–
Braucht weniger Code als Switch-Variante	+	Schreibfehler in als Parameter übergebenen Methodennamen können leicht entstehen	–
		Keine Verwendung auf einigen eingebetteten Systemen möglich	– –

Tabelle 4.2.: Pros und Kontras des Methodenansatzes

SC Analogie Ein Vorteil des Switch-Case Ansatzes wiederum sind seine größeren Gemeinsamkeiten mit SC. So ist vor allem der Teil innerhalb des Switch-Blockes (Zeile 25-60) wesentlich ähnlicher zu SC-Programmen als die Methoden-Variante.

Fehleranfälligkeit Ein Nachteil des Methodenansatzes ist, dass Methodennamen als Strings übergeben werden und Schreibfehler darin erst zur Laufzeit auffallen. Dieses Problem tritt bei der Switch-Case Variante nicht auf, da es sich bei den Labels um Enums handelt. Dort werden unterschiedliche Schreibweisen als Syntaxfehler zur Compilezeit erkannt. Andererseits muss dafür bei dieser Variante jedes mal eine Enumeration mit den verwendeten Labels erstellt werden. Dieses führt zu zusätzlichem Java-Code.

Codeumfang Umfangreicherer Code ist ohnehin ein Nachteil der Switch-Case Variante. Es gibt eine zusätzlich zu deklarierende Enumeration und deren Import-Anweisungen. Dazu kommen die Methode `tick()` (Zeile 22) sowie die die Befehle **`while`** und **`switch`** und die jeweils zugehörigen schließenden Klammern. Dieser zusätzliche Code wächst allerdings *nicht* proportional zur Programmgröße. Dennoch lässt dies den Methodenansatz, zumindest in einfachen Beispielen, leichtgewichtiger und eleganter erscheinen.

Plattformunabhängigkeit Der wohl gravierendste Nachteil der Methodenvariante ist, dass sie nur in Verbindung mit Java-Reflections funktioniert. Wie in Abschnitt 3.1.2 bereits erwähnt, unterstützen viele Java-Implementierungen für eingebettete Systeme Reflections nicht, darunter mit Java ME eine Bedeutende. Gerade aber in diesem Bereich liegen auch Anwendungsszenarien für SJ. Für diese Anwendungsszenarien ist ein SJ, das den Methodenansatz nutzt, nicht verwendbar.

Listing 4.8: ABRO mit Methodenansatz

```
public ABRO() {
  super("ABO", 4);
  initSignals();
}

public void ABO() {
  fork("AB", 1);
  forkeB("ABOMain");
}

public void AB() {
  fork("WaitA", 2);
  fork("WaitB", 3);
  forkeB("ABMain");
}

public void WaitA() {
  if (awaitDoneCB(A)) {
    termB();
  }
}

public void WaitB() {
  if (awaitDoneCB(B)) {
    termB();
  }
}

public void ABMain() {
  if (joinDoneCB()) {
    O.emit();
    termB();
  }
}

public void ABOMain() {
  if (awaitDoneCB(R)) {
    transB("ABO");
  }
}
```

Listing 4.9: ABRO mit Switch-Case Ansatz

```
public ABRO() {
  super(ABO, 4);
  initSignals();
}

@Override public void tick() {
  while (!isTickDone()) {
    switch (state()) {
      case ABO:
        fork(AB, 1);
        forkeB(ABOMain);
        break;

      case AB:
        fork(WaitA, 2);
        fork(WaitB, 3);
        forkeB(ABMain);
        break;

      case WaitA:
        if (awaitDoneCB(A)) {
          termB();
        }
        break;

      case WaitB:
        if (awaitDoneCB(B)) {
          termB();
        }
        break;

      case ABMain:
        if (joinDoneCB()) {
          O.emit();
          termB();
        }
        break;

      case ABOMain:
        if (awaitDoneCB(R)) {
          transB(ABO);
        }
        break;

    }
  }
}
```

Abbildung 4.6.: Vergleich Switch-Case und Methodenansatz

Nach Betrachtung der Unterschiede der beiden Methoden stellt sich die Switch-Case Variante als die geeignetere heraus. Dies ist die Variante, die in SJ Verwendung findet. Vor allem der Nachteil, dass die methodenbasierte Variante nicht für eingebettete Systeme nutzbar ist, führte zu dieser Entscheidung. Aber auch die größere Ähnlichkeit der Switch-Case Variante mit SC spricht dafür.

Die hier vorgestellten funktionstüchtigen Verfahren zur Steuerung des Programmflusses arbeiten mit vielen einzelnen Programmteilen, die jeweils über ihren Namen erreichbar sind. Da es unter Umständen nicht immer ganz intuitiv ist, wann ein neuer Programmteil zu beginnen hat, gibt es dafür in SJ eine Hilfestellung als Teil des verwendeten Designs des Frameworks.

Angelehnt an die gewählte Switch-Case Variante erhält jeder Befehl (jede Methode), der als erste Anweisung nach einem `case` stehen muss, den Buchstaben `C` am Methodenende. Analog dazu steht am Methodenende ein `B`, falls die darauf folgende Anweisung ein `break` sein soll. Daraus folgt, dass die Anweisung für eine Pause in SJ `pauseB()` lautet. Falls eine Anweisung nach einem `case` stehen soll *und* die folgende Anweisung ein `break` sein soll, so endet sie mit `CB`.

Im vorgestellten ABRO Beispiel wurden diese Konventionen bereits verwendet. Zu sehen ist dies in Listing 4.9 in Zeile 17 oder auch Zeile 21.

5. Implementierung

Nachdem die Entscheidungen bezüglich des Designs von SJ getroffen wurden, geht es nun um die konkrete Implementierung. Hierzu wird zunächst eine Implementierung von Signalen in SJ betrachtet. Dabei wird besonders darauf eingegangen, wie vorherige Signalzustände abgefragt werden, Signale deklariert werden, valued Signals implementiert sind und eine Kausalitätsprüfung realisiert ist.

Anschließend findet eine Auseinandersetzung mit der Implementierung des Programmflusses und des Schedulings statt. Danach beschäftigen wir uns mit initialen Warten und Preemptionen als wichtige Bestandteile des Frameworks. Zum Schluss geht es in den Abschnitten zum Debuggen, zur Fehlerbehandlung und zum Testen und Validieren um den Umgang mit Fehlern in SJ.

5.1. Signale

Wie im Unterkapitel 4.4 erläutert und begründet, werden Signale in SJ als eigene Java-Klasse implementiert. Diese Klasse ist in Abbildung 5.1 als UML-Klassendiagramm mit den wichtigsten Variablen und Methoden dargestellt.

Diese Klasse Signal verfügt unter anderem über einen Signalnamen und einen Signalzustand. Die den Signalzustand repräsentierende boolesche Variable *present* ist dann wahr, wenn sie mittels `emit()` emittiert wurde. Um die weiteren Methoden und Variablen sowie ihre Bestimmung besser erläutern zu können, werden im folgenden einige Eigenschaften der Klasse Signal näher beleuchtet.

5.1.1. Abfrage vorheriger Signalzustände

Für die verwendeten Signale soll es in SJ die Möglichkeit geben, die Signalbelegung der vorherigen Ticks abzufragen. Dieses erhöht sowohl Funktionalität als auch die Gemeinsamkeiten mit SC und Esterel.

Als Befehl, um Signale vorheriger Ticks abzufragen, haben wir uns in Tabelle 4.1 im Unterkapitel 4.4 bereits für `pre` entschieden. Die Verwendung erfolgt für ein Signal-Objekt s so, dass `s.pre()` das Signal des vorherigen Ticks zurückgibt.

Da `s.pre()` ein Objekt vom Typ Signal zurückgibt, ist ein erneutes Verwenden von `pre` auf diesem Signal möglich. Zudem können weiter zurückliegende Signalzustände mittels Verketten von Abfragen der Form `s.pre().pre()` abgefragt werden.

Um den Speicherbedarf für vorherige Signalbelegungen möglichst gering zu halten, ist das Abfragen vorheriger Ticks nicht unbegrenzt möglich. Um festzulegen, wie viele Ticks man in die Vergangenheit schauen kann, besitzt einer der Konstruktoren

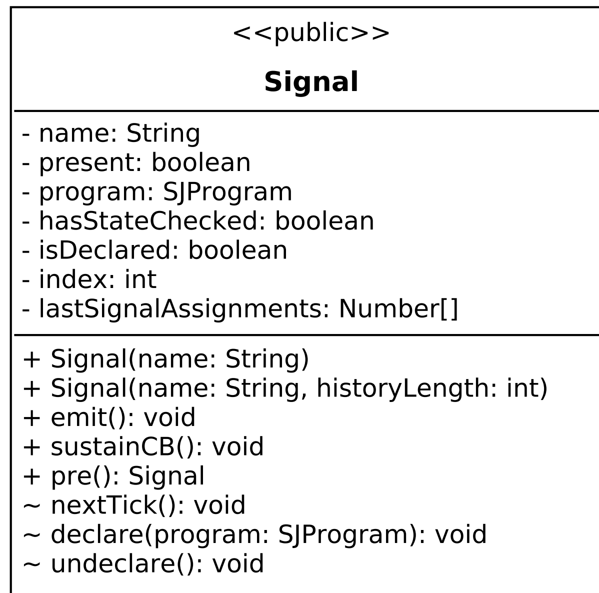

Abbildung 5.1.: UML-Klassendiagramm von `Signal`

von Signal den Parameter `historyLength`. Wird der Konstruktor mit den Wert 1 als `historyLength` aufgerufen, so sind in jedem Tick die Signaleigenschaften des jeweils letzten Zustandes abfragbar. Standardmäßig, das heißt zum Beispiel beim Nutzen des Konstruktors ohne eine Angabe von `historyLength`, ist dieser Wert auf 0 gesetzt.

Für die Speicherung der letzten Zustände innerhalb der Klasse Signal werden die Variablen `index` und `lastSignalAssignments` verwendet. `lastSignalAssignments` ist ein Array von booleschen Werten, welches die letzten Belegungen der Variablen `present` speichert. Die Größe des Arrays entspricht der Anzahl der vergangenen Signalbelegungen die abgefragt werden sollen. Die Variable `index` gibt an, an welche Stelle im Array der nächste Wert gespeichert werden soll. Zu weit in der Vergangenheit liegende Werte werden dabei überschrieben.

5.1.2. Deklaration von Signalen

Um die Funktionalität des Abfragens der vergangenen Signalzustände sicherstellen zu können, müssen bei Ende eines Ticks alle Signale darüber informiert werden, dass ein neuer Tick begonnen hat. Nur so kann der alte Signalzustand rechtzeitig gespeichert werden. Außerdem müssen alle Signale zu Beginn eines neuen Ticks wieder auf *absent* gesetzt werden.

Damit alle Signale beim Ausführen eines neuen Ticks informiert werden, müssen sie in der für die Ausführung zuständigen Klasse deklariert werden. Diese ruft die Methode `declare()` auf und sorgt dafür, dass zu Beginn jedes Ticks `nextTick()`

für alle deklarierten Signale aufgerufen wird.

Zum Deklarieren von Signalen bietet SJ zwei unterschiedliche Möglichkeiten an. Zum einen die Methode `addSignals(Signal... signals)` und zum anderen die Methode `initSignals()`.

Die Methode `addSignals(Signal... signals)` arbeitet im Gegensatz zu `initSignals` ohne Java-Reflections und sorgt für ein Deklarieren aller übergebener Signale. Ein Aufrufen von Konstruktoren oder eigenständiges Ermitteln von Signalen findet nicht statt. Zu sehen ist das Verwenden von Signalen auf diese Weise in Listing 5.2.

Listing 5.1: Signaldeklaration in ABRO

```
1  public Signal A, B, R, O;
2
3  public ABRO() {
4    super(ABO, 4);
5    initSignals();
6  }
```

Listing 5.2: Alternative Signaldeklaration in ABRO

```
1  public Signal A, B, R, O;
2
3  public EmbeddedABRO() {
4    super(ABO, 4);
5    A = new Signal("A");
6    B = new Signal("B");
7    R = new Signal("R");
8    O = new Signal("O");
9    addSignals(A, B, R, O);
10 }
```

Beim Nutzen von `initSignals()` in einem SJ-Programms werden alle im Code als `public` gekennzeichneten Signale der Klasse automatisch deklariert, und falls sie noch nicht initialisiert wurden, mit ihren Default-Konstruktor aufgerufen. Zu sehen ist das Deklarieren von Signalen auf diese Weise für das Programm ABRO in Listing 5.1. Das Arbeiten mit Signalen der erbenden Unterklasse, zum Beispiel um deren Konstruktor aufzurufen, geschieht mittels Java-Reflections.

Aufgrund der Tatsache, dass Reflections genutzt werden, wird die Anzahl der nutzbaren Plattformen eingeschränkt. Wie schon im Abschnitt 3.1.2 erläutert, werden Reflections vor allem von Java-Implementierungen für eingebettete Systeme nicht unterstützt.

Damit SJ auch für eingebettete Systeme nutzbar ist, und trotzdem bei verfügbaren Reflections die kurze und elegante Methode `initSignals()` genutzt werden kann, werden zwei Arten von SJ-Programmen unterstützt. Ein SJ-Programm kann sowohl von der Klasse `EmbeddedSJProgram` als auch von `SJProgram` erben. Nur die von der Klasse `SJProgram` erbenden SJ-Programme nutzen Java-Reflections. Allerdings können auch nur diese Programme die Methode `initSignals()` zur Signaldeklaration nutzen. In Abbildung 5.2 wird die hinter diesem Konzept stehende Klassenhierarchie als UML-Klassendiagramm dargestellt.

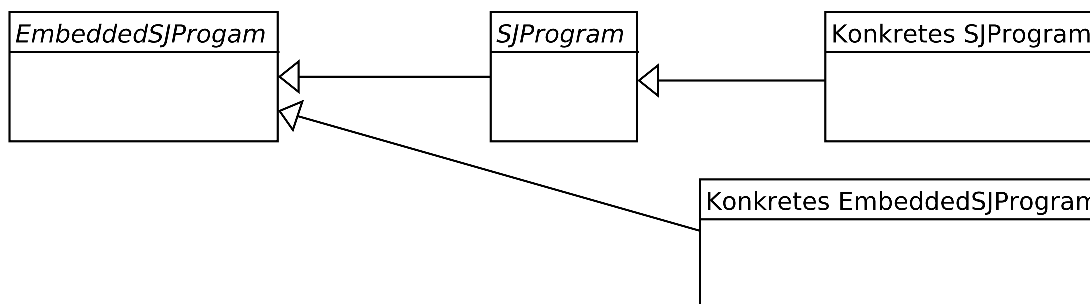

Abbildung 5.2.: Klassenhierarchie von SJ-Programmen

5.1.3. Valued Signals

Um den Sprachumfang von SC und Esterel sowie eine hohe Funktionalität von SJ zu erreichen, gibt es neben einfachen Signalen auch valued Signals. Valued Signals sind Signale, die zusätzlich zu einer Signalbelegung (*present* oder *absent*) auch einen Wert haben. Bei diesem Wert handelt ist sich um eine beliebige Zahl die mit Hilfe der Java-Klasse `Number` implementiert ist.

Da ein valued Signal die Eigenschaften eines Signals erweitert, ist jedes valued Signal auch ein Signal. In der Implementierung von Signalen in SJ erbt die Klasse `ValuedSignal` von der Klasse `Signal`.

Der Wert eines valued Signals kann mittels Nutzen von `emit()` mit einem Parameter verändert werden. Dies kann beliebig oft während eines Ticks geschehen. Existiert beim Setzen eines Wertes für das valued Signal noch kein Wert, so erhält das valued Signal diesen. Existiert bereits ein Wert, so wird dieser mit dem vorherigen Wert kombiniert und der berechnete Wert als neuer Signalwert gespeichert.

Um die Art des Kombinierens von mehreren Werten festzulegen, bietet sich eine individuell festlegbare mathematische Funktion an. Diese sollte zwei Parameter, den bisherigen und den jetzt zu emittierenden Wert, erhalten und daraus den neuen Wert bestimmen.

In Esterel geschieht dies mit einer *comb* genannten Funktion [6, Seite 97], die zwei Werte kombiniert. Diese Funktion wird beim Erstellen eines valued Signals mit angegeben. In SC befinden sich in der zugehörigen Header-Datei[1] die Methoden `EMITINTADD(s, val)`, `EMITINTMUL(s, val)`, `EMITINTMAX(s, val)` und `EMITINTMIN(s, val)`. Diese Funktionen sorgen dafür, dass der existierende Wert mit dem aktuell angegebenen wie vorgesehen kombiniert wird. Das heißt, er wird zum bisherigem Wert addiert, mit ihm multipliziert oder es wird das Maximum bzw. das Minimum der beiden Werte bestimmt.

Um die höchstmöglichste Flexibilität bei der Angabe einer Funktion zur Kombination der emittierten Werte zu erhalten, soll es in SJ die Möglichkeit geben, die

[1] `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/sc/sc.h`

dafür zuständige Funktion dem Signal zu übergeben. Eine Möglichkeit einer Methode Methoden als Parameter zu übergeben gibt es in Java jedoch nicht [18].

Um trotz dieser Beschränkung eine Methode mit einer Berechnungsvorschrift der Kombination zweier Werte einem valued Signal übergeben zu können, gibt es in SJ das Interface `CombinationFunction`. Dieses Interface besitzt die Methode `call(Number oldVal, Number newVal)`. Um eine Berechnungsvorschrift für die Kombination von Werten eines valued Signals angeben zu können, muss eine Instanz der Klasse `CombinationFunction` erzeugt werden. Diese Instanz muss beim Erstellen eines valued Signals im Konstruktor angegeben werden. In der Methode `call` muss die gewünschte Berechnung implementiert werden.

5.1.4. Kausalitätsprüfung

Aufgrund der Tatsache, dass bei Einhaltung der Synchronitätshypothese alle Ereignisse innerhalb eines Ticks als gleichzeitig angesehen werden, müssen Signale während eines Ticks eindeutig bestimmbar sein. Das heißt sie sind entweder *present* oder *absent*.

In Esterel und SyncCharts gilt diese Eindeutigkeit von Signalzuständen. Diese führt dazu, dass bestimmte Programme als nicht korrekt abgelehnt werden. Das in Listing 5.3 abgebildete Esterel-Programm ist beispielsweise nicht korrekt. Ist das Signal A während des Ticks *absent* so wird A emittiert und hat danach den Zustand *present*. Für ein Signal ist allerdings entweder *present* oder *absent* als Zustand während eines Ticks vorgeschrieben. Das Verhalten eines solchen Programms wird als nicht kausal bezeichnet [6].

Listing 5.3: Kausalitätsprobem in Esterel

```
1 present A
2   else emit A
3 end
```

In SJ wurde eine Kausalitätsprüfung als optionales Feature implementiert. Dieses Verhalten deckt sich mit den von SC. Mit Hilfe der Methode `activateCausalityCheck()` lässt sich für ein SJ-Programm diese Prüfung aktivieren. Ist sie aktiviert wird immer wenn ein Signal emittiert wird, nachdem es bereits abgefragt wurde, eine Fehlermeldung geworfen.

Um diese Überprüfung zu implementieren, besitzt jedes Signalobjekt die boolesche Variable `hasStateChecked`, die speichert, ob eine Überprüfung eines Signalzustandes im aktuellen Tick bereits stattfand.

In Listing 5.4 ist ein Auszug eines SJ-Programms abgebildet, das bei aktivierter Kausalitätsprüfung zu einer Fehlermeldung führt. Es ist funktional identisch mit den in Listing 5.3 abgebildeten Esterel-Programm.

Listing 5.4: Kausalitätsprobem in SJ

```
1 if (!s.isPresent) {
2   s.emit();
3 }
```

5.2. Programmfluss

Neben Signalen ist auch der Programmfluss ein zentraler Bestandteil synchroner Programme. Er muss so implementiert werden, dass eine möglichst intuitive Bedienung bei gleichzeitiger Beachtung synchroner Konzepte erreicht wird. Um den Programmfluss eines SJ-Programm besser nachvollziehen zu können, ist dieser in Abbildung 5.3 visualisiert. Abbildung 5.4 zeigt im Vergleich dazu den Programmfluss, wie er in SC implementiert wurde. Es ist zu erkennen, dass die verwendeten Programmzustände identisch sind.

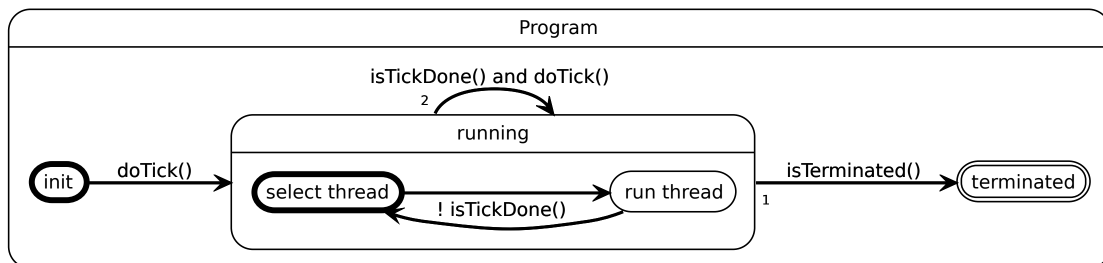

Abbildung 5.3.: Zustände eines SJ-Programms

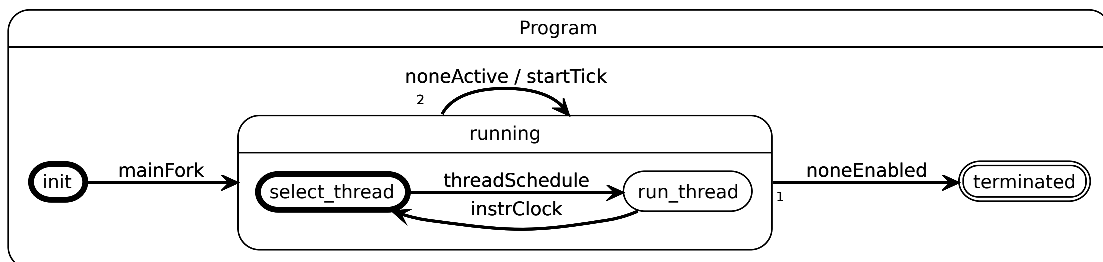

Abbildung 5.4.: Zustände eines SC-Programms nach [9]

Bei Erstellung eines neuen SJ-Programms durch Aufruf des Konstruktors befindet sich das Programm zunächst in einem initialen Zustand (*init*). Dieser kann mit der Methode `isInitialTick()` abgefragt werden.

Mit Aufrufen der Methode `doTick()` wird jeweils der nächste Tick ausgeführt, bis dieser abgeschlossen ist. Voraussetzung dafür ist, dass ein vorangegangener Tick

erfolgreich abgeschlossen wurde. Die Funktionsweise von `doTick` ist in Abbildung 5.5 als Ablaufdiagramm dargestellt.

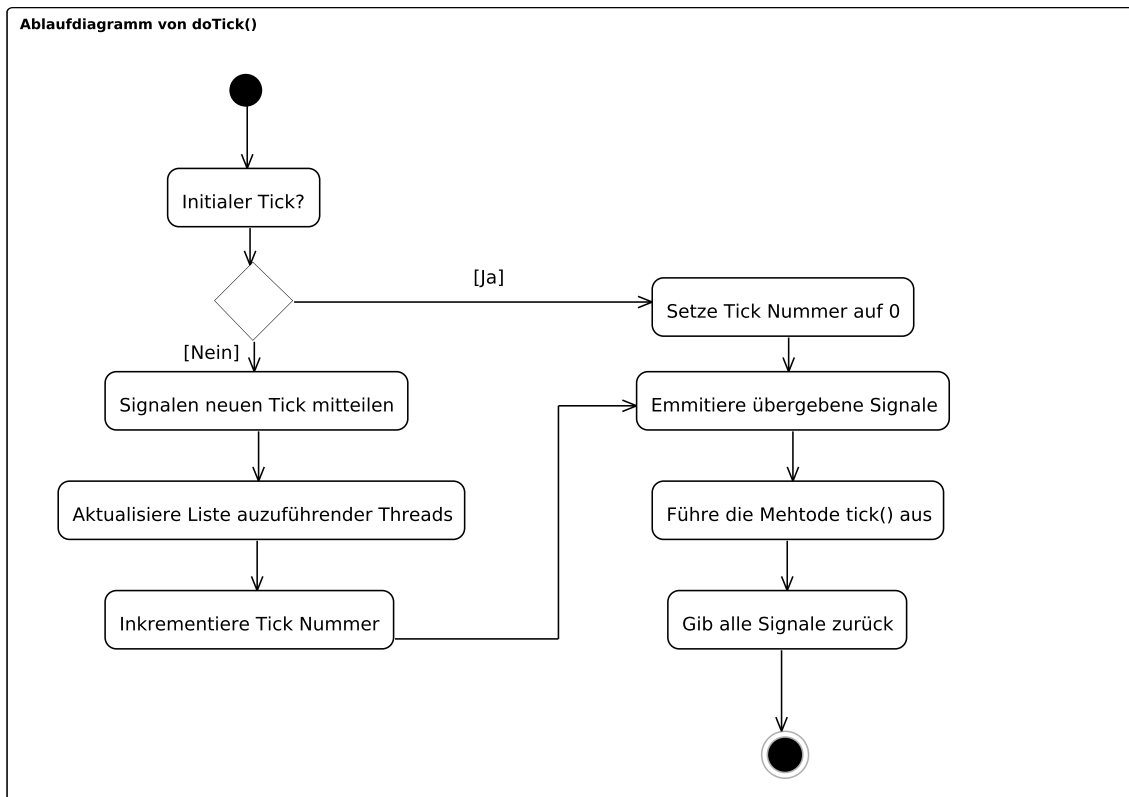

Abbildung 5.5.: Programmablauf von `doTick()`

Die Abbildung zeigt, wie bei einem Aufruf von `doTick()` im Einzelnen vorgegangen wird. Zunächst wird dafür gesorgt, dass Ticknummer und die Liste der in diesem Tick auszuführenden Programmteile aktuell sind. Danach werden die `doTick()` übergebenen Eingabesignale für den Tick emittiert.

Anschließend wird die Methode `tick()` ausgeführt. Diese Methode ist in der Klasse zur Steuerung des Programmflusses (`EmbeddedSJProgram`) als abstrakte Methode definiert. Sie muss vom SJ-Programmierer beim Erben von dieser Klasse implementiert werden. In diese Methode muss das vom SJ-Programmierer entwickelte konkrete SJ-Programm geschrieben werden. Die Methode `doTick()` bietet lediglich die Möglichkeit, den Ablauf dieses Programms einfach zu steuern.

In der Methode `tick()` muss auch sichergestellt werden, dass sie einen kompletten Tick abarbeitet. Außerdem muss die vorher in `doTick()` aktualisierte Liste der auszuführenden Programmteile richtig und vollständig verarbeitet werden. Dem Programmierer steht dafür die Methode `state()` zur Verfügung, welche den jeweils als nächstes auszuführenden Programmteil bestimmt.

Nach dem Abarbeiten von `tick()` werden von `doTick()` alle aktuellen Signalbe-

legungen zurückgegeben. Mit Abschluss des letzten Ticks gelangt ein SJ-Programm in den Zustand *terminated*.

5.3. Scheduling

Im Unterkapitel 4.1 wurde bereits erläutert, dass Nebenläufigkeit mit dynamischen, im SJ-Programm zu setzenden, Prioritäten realisiert wird. Um Nebenläufigkeit mehrerer Programmteile intern in der zuständigen Klasse `SJProgramm` zu realisieren nutzt diese Klasse eine Klasse namens `SJThread`.

Diese Klasse repräsentiert jeweils einen auszuführenden Programmteil. Sie verfügt als hauptsächliche Eigenschaften über ein Label und eine Priorität. Am Label eines SJThreads steht der zum SJThread gehörende Code. Dieses Label kann sich für einen bestimmten SJThread auch ändern, beispielsweise bei einer `goto`-Anweisung. Außerdem hat jeder SJThread eine Priorität, welche die Ausführungsreihenfolge bestimmt. Je höher die Priorität eines Threads ist, desto eher wird er ausgeführt.

Um das Ausführungsmodell der einzelnen Threads besser verstehen zu können, betrachten wir Abbildung 5.6. Dargestellt ist die Zuordnung eines SJThread zu einem von drei Zuständen. Ein SJThread kann deaktiviert (*disabled*), aktiv (*active*) oder inaktiv (*inactive*) sein. Aktive und inaktive SJThreads werden unter der Bezeichnung aktiviert (*enabled*) zusammengefasst. Ein SJThread wird als deaktiviert bezeichnet, wenn er noch nicht oder nicht mehr existent ist.

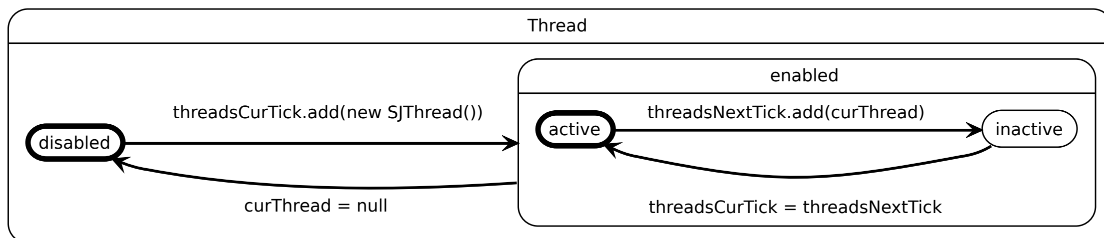

Abbildung 5.6.: Zustände eines SJThreads

In der konkreten Implementierung werden, falls ein Thread in den Zustand *disabled* wechselt, alle Referenzen auf ihn gelöscht. Um aus dem Zustand *disabled* heraus in den Zustand *active* zu wechseln, muss ein SJThread neu erstellt werden und zur Liste der aktiven SJThreads hinzugefügt werden. Ein SJThread gilt als aktiv wenn er im aktuellen Tick noch Arbeit zu tun hat. Alle aktiven Threads werden in der Liste `threadsCurTick` verwaltet. Aus dieser Liste wird, falls ein erneutes Scheduling erforderlich ist, jeweils der SJThread mit der höchsten Priorität ausgeführt. Ein SJThread gilt als inaktiv, falls er nicht beendet wurde und keine Arbeit im aktuellen Tick mehr zu erledigen hat.

In der Implementierung im `SJProgram` wechselt ein Thread dadurch von aktiv zu inaktiv, dass er zu der Liste der im nächsten Thread auszuführenden Threads

(`threadsNextTick`) hinzugefügt wird. Zu Beginn eines neuen Ticks wechseln alle inaktiven Threads automatisch in den Zustand aktiv. Voraussetzung dafür ist, dass es keine aktiven Threads mehr gibt, also die Liste `threadsCurTick` leer ist.

Wie der Programmierer die Zustände eines SJThread beeinflusst, ist zeigt Abbildung 5.7. Dort werden beispielhaft Befehle aufgeführt, welche den Ausführungszustand eines Threads verändern.

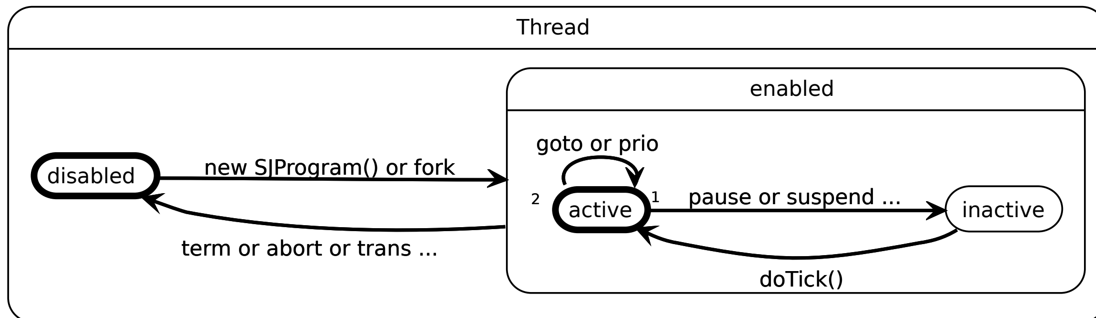

Abbildung 5.7.: Beeinflussung der Zustände eines SJThreads

Ein neuer SJThread lässt sich nur mit `fork` oder beim Start eines neuen Programms erzeugen. Mit der Anweisung `term` kann er beispielsweise beendet werden. Der Befehl `pause` sorgt dafür, dass ein Thread vom Zustand *active* in den Zustand *inactive* wechselt.

5.4. Initiales Warten

Für den Befehl `await` ist ein initiales Warten notwendig. Das heißt beim ersten Erreichen von `await` findet ein automatischer Zustandswechsel des entsprechenden SJThreads von *active* nach *inactive* statt. In jedem folgenden Tick geschieht dies abhängig davon, ob ein Signal anliegt.

Der Befehl `await` muss also wissen, ob er bereits initial gewartet hat. Hierfür wurde in SJThread eine zusätzliche boolesche Variable eingeführt, die angibt, ob für diesen Thread bereits ein initiales Warten stattfand. Nach einem erfolgreichen Durchlaufen von `await`, das heißt bei einem Anliegen des Signals auf das gewartet wurde, muss sichergestellt werden, dass diese Variable wieder zurückgesetzt wird. Bei einem erneuten Ausführen von `await`, beispielsweise durch eine Schleife verursacht, muss schließlich erneut initial gewartet werden.

5.5. Preemptionen

Ein wichtiger Teilbereich von SJ und somit unserer Implementierung sind die Preemptionen. Im konkreten geht es hierbei um die Methoden `abort` und `suspend`.

Der Befehl *abort* sorgt dafür das alle von einem Thread erzeugten Kinder und deren Kinder beendet werden. Das heißt sie wechseln in den Zustand *disabled*. Kinder von einem SJThread werden mittels `fork` und `forke` erzeugt. Der Befehl `suspend` sorgt dafür, dass alle Kinder und deren Kinder des aktuellen SJThread in den Zustand *inactive* wechseln. Das heißt sie spielen erst im Folgetick wieder eine Rolle.

Um diese Preemptionen in SJ zu realisieren besitzt jeder SJThread eine Liste von Kindern. Diese Nachkommen (*descendants*) eines Threads werden, falls sie durch `abort` oder `suspend` manipuliert werden sollen, rekursiv durchgegangen, und die Änderung auf sie angewandt. Dieses rekursive Vorgehen ist notwendig, da nicht nur die Nachkommen des aktuellen Threads, sondern auch deren Nachkommen und deren Nachkommen betrachtet werden müssen.

Beim Aufrufen von `fork` wird jeweils der erstellte Thread zur Liste von Kindern hinzugefügt. Bei einem `abort` wird diese Liste, nachdem alle darin enthaltenden Threads rekursiv beendet wurden, geleert.

5.6. Debugging

Um ein Arbeiten mit SJ zu erleichtern, soll fehlerhaftes Verhalten von SJ-Programmen gut erkennbar und analysierbar sein. Auch ein grafisches Darstellen des Verhaltens sowie ein einfaches Nachvollziehen beliebiger SJ-Programme soll möglichst einfach gewährleistet werden.

Um dies zu erreichen, wurde für SJ eine umfangreiche Protokollierung des Programmablaufes implementiert. Es gibt dazu die Möglichkeit, einem `EmbeddedSJProgram` oder `SJProgramm` im Konstruktor eine Instanz einer Klasse zu übergeben, die das Interface `SJLogger` implementiert. Wird der Wert `null` übergeben, so wird das Protokollieren des Programmablaufs deaktiviert.

Das Interface `SJLogger` besitzt die Methode `log`, welche von in SJ programmierten Programmen selbständig aufgerufen wird. Bei Ausführung eines beliebigen Befehls wird `log` mit einem String, der die Ausführung des Befehls beschreibt, aufgerufen. Es ist ein unmittelbares Handeln, beispielsweise um dieses Ereignis auf der Konsole auszugeben, möglich.

Als Ausgabeformat für das Loggen von Informationen wurde JSON gewählt. Dieses hat wie schon im Unterkapitel 3.2 beschrieben den Vorteil, dass es sowohl für Maschinen als auch für Menschen leicht lesbar ist. Dies soll das Filtern von Informationen, wie beispielsweise das alleinige Ausgeben aller Signale, möglich machen. Auch das grafische Darstellen von protokollierten Informationen, das Senden dieser an ein anderes System oder ein individuelles Ausgabeformat soll so erleichtert werden.

Eine beispielhafte Ausgabe von SJ im JSON-Format zeigt Abbildung 5.5. Es handelt sich hierbei um einen Auszug aus der Ausführung des Programms ABRO.

Listing 5.5: Protokollierung im JSON-Fromat

```
1  "term":{"label":"T2","prio":5}
2  "present":{"label":"L0","prio":4,"param":[{"C":{"present":true}}],"retval":true}
3  "emit":{"label":"L0","prio":4,"param":[{"D":{"present":false}}]}
4  "prio":{"label":"L0","prio":4,"param":["L1",2]}
5  "present":{"label":"T3","prio":3,"param":[{"D":{"present":true}}],"retval":true}
6  "emit":{"label":"T3","prio":3,"param":[{"E":{"present":false}}]}
7  "term":{"label":"T3","prio":3}
8  "present":{"label":"L1","prio":2,"param":[{"E":{"present":true}}],"retval":true}
9  "emit":{"label":"L1","prio":2,"param":[{"T":{"present":false}}]}
10 "term":{"label":"L1","prio":2}
11 "present":{"label":"TMain","prio":1,"param":[{"T":{"present":true}}],"retval":true}
12 "abort":{"label":"TMain","prio":1}
13 "term":{"label":"TMain","prio":1}
14 "signals":[{"A":{"present":true}}{"B":{"present":true}}{"C":{"present":true}}{"D":{
       "present":true}}{"E":{"present":true}}{"T":{"present":true}}]
15 "ticklabels":[]
16 "ticknr":1
17 "signals":[{"A":{"present":false}}{"B":{"present":false}}{"C":{"present":false}}{"D
       ":{"present":false}}{"E":{"present":false}}{"T":{"present":false}}]
18 "signals":[{"A":{"present":false}}{"B":{"present":false}}{"C":{"present":false}}{"D
       ":{"present":false}}{"E":{"present":false}}{"T":{"present":false}}]
19 "ticklabels":[]
20 "programName":"class sj.examples.GRCbal3"
21 "ticklabels":["INIT"]
22 "addedSignals":[{"A":{"present":false}}]
23 "addedSignals":[{"B":{"present":false}}]
24 "addedSignals":[{"C":{"present":false}}]
```

5.7. Fehlerbehandlung

Zusätzlich zum Protokollieren des Programmablaufes als Art der Fehlererkennung wurden zahlreiche Fehlermeldungen in SJ implementiert. Es handelt sich dabei um Fehlermeldungen, die beim falschen Nutzen des Frameworks geworfen werden und den Programmierer darüber informieren. Auch ein Erkennen von fehlerhafter Programmierung soll mit diesen Fehlermeldungen vereinfacht werden.

In Tabelle 5.1 sind einige der in SJ auftretenden Fehlermeldungen aufgeführt. Zusätzlich ist eine häufige Ursache und eine mögliche Lösung für die aufgeführten Fehler angegeben.

Fehlermeldungen wurden in SJ als `RuntimeExceptions` implementiert. Sie müssen also nicht beim Programmieren von SJ für jeden Befehl abgefangen und behandelt werden. Tritt ein Fehler auf, bricht das Programm ab und die Fehlermeldung wird geworfen. Die Klassenhierarchie der in SJ verwendeten Fehlermeldungen ist in Listing 5.8 dargestellt. Da es sich bei den Fehlern meist um Programmierfehler handelt und korrekte SJ-Programme keine Fehler werfen, ist ein Arbeiten mit `RuntimeExceptions` durchaus sinnvoll. Auch andere Programmierfehler, wie ein Zugriff auf ein Array oberhalb der Arraygröße, werden in Java auf diese Weise behandelt.

Ein vorheriges Prüfen auf Programmierfehler ist in SJ nicht realisiert, da beim Schreiben eines Java-Framework keine Syntaxprüfung etwa durch Erweitern des Java-Compilers möglich ist.

Fehlermeldung	Häufige Ursache	Mögliche Lösung
CausalityException	Auftreten eines Kausalitätsproblem, wie in Abschnitt 5.1.4 beschrieben	Kausalitätsprüfung deaktivieren oder Programm überarbeiten
CurTickNotDoneException	Aktueller Tick beim Versuch einen Neuen zu starten noch nicht beendet	Sicherstellen, dass der alte Tick vor dem Starten eines Neuen beendet ist
NoPreSignalException	Es werden nicht genug vorherige Signalbelegungen gespeichert	Anzahl der gespeicherten vorherigen Signalbelegungen erhöhen
PriorityException	Setzen einer bereits existierenden oder negativen Priorität	Überprüfen aller gesetzten Prioritäten
SignalNotDeclaredException	Arbeiten mit einem nicht deklarierten Signal	Deklarieren aller Signale bei Programmstart
ThreadException	Es sollen Anweisungen ausgeführt werden, ohne dass ein Thread aktiv ist	Sicherstellen, dass alle notwendigen `break` Befehle vorhanden sind und `state()` verwendet wird

Tabelle 5.1.: Fehlermeldungen in SJ

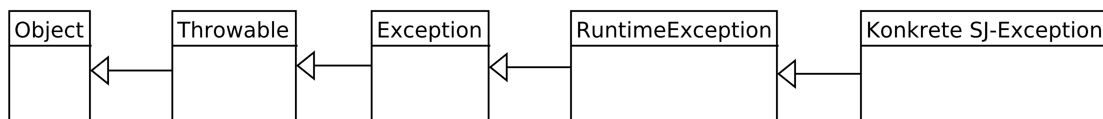

Abbildung 5.8.: SJ Fehlermeldung

5.8. Testen und Validieren

Beim Schreiben eines Frameworks, um synchrone Programmierung in Java zu ermöglichen, ist die Korrektheit eine zentrale Eigenschaft. Damit ein Arbeiten mit SJ sinnvoll möglich ist, und die damit erstellten Programme zuverlässig funktionieren, muss die Implementierung möglichst fehlerfrei sein.

Um dieses für SJ zu erreichen, wurden zahlreiche Testfälle für die unterschiedlichen Komponenten von SJ implementiert. Die Implementierung der Testfälle wurde mithilfe des in Abschnitt 3.1.3 eingeführten JUnit-Framework realisiert. In Abbildung 5.9 ist das Ergebnis automatisierten Testens, so wie es JUnit unter Eclipse ermöglicht, abgebildet.

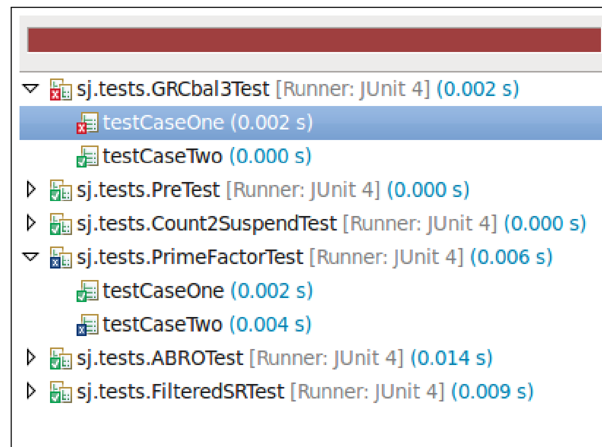

Abbildung 5.9.: Automatisiertes Testen von SJ

In der Abbildung sind unterschiedliche Testfälle im Zusammenhang mit mehreren zu testenden SJ-Programmen dargestellt. Der rot dargestellte Testfall konnte aufgrund einer geworfenen *Exception* nicht korrekt abgearbeitet werden. Bei dem blau dargestellten Testfall stimmen die erwarteten Werte nicht mit den tatsächlichen überein.

Mit Hilfe dieser Testumgebung ist ein ständiges Testen der aktuellen SJ-Implementierung auf die richtige Ausgabewerte für alle implementieren Testprogramme möglich. Bei einer Änderung einzelner Frameworkteile kann sofort überprüft werden ob damit weiterhin alle Programme richtig laufen.

Als Testfälle für SJ wurden mehrere in SC bereits realisierte Programme für SJ angepasst und implementiert. Auch das Überprüfen des Auftretens einer *Exception*, beispielsweise bei mehreren Threads mit identischer Priorität, wurde mit JUnit-Tests implementiert.

6. Bewertung und Ausblick

In dieser Arbeit wurden zunächst verwandte Arbeiten und verwendete Technologien erläutert. Danach wurden unterschiedliche Ansätze zur Realisierung einer synchronen Programmierung in Java betrachtet. Dabei wurde ein Design für SJ ermittelt, welches die unterschiedlichen Anforderungen wie Ausführung auf eingebetteten Systemen oder eine einfache Programmierbarkeit erfüllt.

Im dann folgenden Kapitel wurde die konkrete Implementierung von SJ, so wie sie im Rahmen dieser Arbeit in Java entwickelt wurde, vorgestellt. Es wurde dabei auf zentrale Eigenschaften synchroner Programme eingegangen und deren Umsetzung und Verwendung in SJ erklärt. Auch Themen wie der Umgang mit Fehlern wurden behandelt.

Nach Betrachtung des Designs und der Implementierung von SJ lässt sich festhalten, dass mit dem entwickelten Framework die im Unterkapitel 1.1 beschriebenen Anwendungsszenarien realisierbar sind. SJ erlaubt sowohl eine synchrone Programmierung in Java, als auch eine Abbildung der Struktur und Funktionsweise eines SyncCharts in Java.

Des Weiteren bleibt festzuhalten, dass sich SJ, als in Java integrierte Programmiersprache, wenig wie klassische Programmiersprachen verhält. Es gibt keinen speziellen Compiler, keine vorherige Syntaxprüfung und sämtliche Befehle sind als Java Methoden implementiert.

Das Verwenden von Makros und `goto`-Anweisungen macht SC-Programme relativ schlank und übersichtlich. Das Fehlen dieser Konstrukte in Java sorgt in SJ dafür, dass SJ-Programme teilweise aufgebläht wirken. Vor allem bei kurzen Programmen besteht ein großer Teil des Programms aus Verwaltungs- und Standardbausteinen.

Dennoch lässt sich vermuten, dass SJ, vor allem in Kombination mit SyncCharts, Vorteile mit sich bringt. Schon jetzt lassen sich SyncCharts durch Nutzen von SJ relativ leicht manuell nach Java portieren. Mit einer automatischen Übersetzung wird es auf diese Weise möglich sein, einfacher als bisher synchrone Java-Programme zu schreiben.

Da durch SJ ein Abbilden der Struktur eines SyncCharts als Java-Programm ermöglicht wird, wird es in den erzeugten SJ-Programm möglich sein, weitere Änderungen vorzunehmen. So lassen sich Vorteile von SyncCharts und Java vereinen.

6.1. Anwendungsszenarien

Aufgrund der Tatsache, dass es sich bei SJ um ein Framework zur synchronen Programmierung handelt, decken sich die Einsatzbereiche mit denen klassischer

synchroner Sprachen. Als Einsatzort bietet sich deshalb, neben dem akademischen Bereich, der Bereich der Echtzeitsysteme und eingebetteten Systeme an.

6.2. Mögliche Erweiterungen und Ergänzungen

Für SJ bieten sich eine Reihe von Erweiterungen, Ergänzungen und Verbesserungen an, um die Qualität und die Nutzbarkeit zu steigern.

Automatische Übersetzung Die wohl bedeutendste noch zu realisierende Ergänzung für SJ ist die bereits mehrfach angesprochene automatische Übersetzung von SyncCharts nach SJ.

Weitere SJ-Befehle Zusätzlich zu dieser Übersetzung ist eine Realisierung weiterer Befehle in SJ denkbar. Der jetzt implementierte Befehlssatz behandelt vor allem notwendige Basisbefehle. Für häufig auftretende Befehlsfolgen bietet sich die Möglichkeit eines Realisieren dieser mit einem einzelnen Befehl an. SC unterstützt hier einen umfangreicheren Befehlssatz als SJ.

Lokale Signale In Esterel und SC gibt es zudem das Konzept der lokalen Signale. Dieses erlaubt die Realisierung von Signalen mit lokaler Sichtbarkeit. Eine Realisierung von lokalen Signalen wurde in SJ nicht vorgenommen. Die Suche nach einem geeigneten Konzept dafür, und dessen Implementierung, wäre eine mögliche Erweiterung von SJ für die Zukunft.

Optimierung von Ausführungszeit und Ressourcenverbrauch Bisher nicht näher untersucht wurde die Ausführungsgeschwindigkeit und der Ressourcenverbrauch von SJ-Programmen. Im Zusammenhang damit wären auch andere Implementierungen im Kern des SJ-Framework möglich. Bei Beibehaltung der API zum Arbeiten und Programmieren mit SJ könnten innerhalb des Frameworks benutzte Datenstrukturen und Abläufe verändert werden. So können zum Beispiel, anstelle von komplexen Datenstrukturen, Bitvektoren zum Speichern von Zuständen genutzt werden. Dieses würde den Code zwar zu Lasten der Lesbarkeit und Übersichtlichkeit verändern, aber höchstwahrscheinlich zu besseren Ausführungszeiten und weniger Speicherbedarf von SJ-Programmen führen.

Weitere Beispiele Um die Nutzung von SJ zu demonstrieren und mögliche Anwendungsmöglichkeiten aufzuzeigen, bietet sich eine Implementierung weiterer Bespiele an.

Umfangreichere Validierung Die Validierung von SJ basiert auf dem Realisieren und Testen typischer und möglichst umfangreicher Beispielprogramme. Ein Verwenden weiterer Beispiele, und das Schreiben von Testfällen, dafür ist eine sinnvolle Erweiterung des Validierungsprozesses. Auch das Erstellen und Testen von Programmen, die speziell für den Test einzelner Komponenten

oder typischer Probleme geschrieben wurden, bietet sich an. Hier kann vor allem beim Umfang von Beispielprogrammen und Testfällen noch einiges getan werden.

6.3. Fazit

Mit SJ ist synchrone Programmierung in Java möglich. Zudem wird eine Übersetzung von SyncCharts nach Java erleichtert. Im akademischen Bereich, aber auch bei der Programmierung reaktiver Systeme, bietet sich eine Verwendung von SJ an.

Literaturverzeichnis

[1] AMENDE, Torsten: *Synthese von SC-Code aus SyncCharts*, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Diploma thesis, Mai 2010. – http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf

[2] ANDRÉ, Charles: Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In: *Computational Engineering in Systems Applications (CESA)*. Lille, France : IEEE-SMC, July 1996, S. 19–29. – http://www.i3s.unice.fr/~andre/CAPublis/Cesa96/SyncCharts_Cesa96.pdf

[3] ANDRÉ, Charles: Semantics of SyncCharts / I3S Laboratory. Sophia-Antipolis, France, April 2003 (ISRN I3S/RR–2003–24–FR). – Forschungsbericht

[4] BENVENISTE, Albert ; CASPI, Paul ; EDWARDS, Stephen A. ; HALBWACHS, Nicolas ; GUERNIC, Paul L. ; SIMONE, Robert de: The Synchronous Languages Twelve Years Later. In: *Proceedings of the IEEE, Special Issue on Embedded Systems* Bd. 91, Januar 2003, S. 64–83

[5] BERRY, Gérard: *The Constructive Semantics of Pure Esterel Version 3*. Draft Book, 2004

[6] BERRY, Gérard ; GONTHIER, Georges: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. In: *Science of Computer Programming* 19 (1992), Nr. 2, S. 87–152

[7] BOUSSINOT, Frederic ; SIMONE, Robert de: The ESTEREL Language. Another Look at Real Time Programming. In: *Proceedings of the IEEE* 79 (1991), September, Nr. 9, S. 1293–1304

[8] HALBWACHS, Nicolas ; CASPI, Paul ; RAYMOND, Pascal ; PILAUD, Daniel: The synchronous data-flow programming language LUSTRE. In: *Proceedings of the IEEE* 79 (1991), September, Nr. 9, S. 1305–1320

[9] HANXLEDEN, Reinhard von: SyncCharts in C / Christian-Albrechts-Universität Kiel, Department of Computer Science. Mai 2009 (0910). – Technical Report

[10] HANXLEDEN, Reinhard von: Embedded Real-Time Systems—Lecture 7, Concurrency I. (2010), Mai. – http://rtsys.informatik.uni-kiel.de/teaching/10ss/v-emb-rt/lectures/lecture07-handout4.pdf

[11] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Juni, Nr. 3, S. 231–274

[12] IEEE: IEEE Standards Interpretations for IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988). In: *IEEE Std 1003.1-1988/INT, 1992 Edition* (1992)

[13] Lee, Edward A.: The Problem with Threads. In: *IEEE Computer* 39 (2006), Nr. 5, S. 33–42. – URL http://doi.ieeecomputersociety.org/10.1109/MC.2006.180

[14] Lindholm, Tim ; Yellin, Frank: *The Java Virtual Machine Specification Second Edition*. Prentice Hall, April 1999. – 496 S. – http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html. – ISBN 978-0201432947

[15] Oberlände, Ursula ; Neugebauer, Margit ; Prüfig, Gerhard: *Definition der Nebenläufigkeit*. 1998. – http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/SeminarDidaktik/Nebenlaeufigkeit/nebenlaeufig.htm

[16] Oracle: *Java^{TM} 2 Platform Standard Edition 5.0 API Specification*. http://download.oracle.com/javase/1.5.0/docs/api/. 2004. – [Online; Stand 24. August 2010]

[17] Roessler, P. ; Zauner, M.: Evaluation of an Esterel-based hardware/software co-design flow. In: *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, 8-10 2009, S. 42 –45

[18] Ullenboom, Christian: *Java ist auch eine Insel: Programmieren mit der Java Platform, Standard Edition 6*. 8., aktualisierte und erweiterte Auflage. Bonn : Galileo Press, 2009 (Galileo Computing). – 1475 S. : Ill., graph. Darst. + 1 DVD–ROM (12 cm) S. – ISBN 978-3-8362-1371-4 - 3-8362-1371-0

A. Verwendung des Frameworks

Da es sich bei SJ um ein komplexes Framework mit zugehörigen Programmierkonzept handelt, erfordert das Programmieren damit Wissen über die richtige Nutzung. Um uns mit der Verwendung des Frameworks vertraut zu machen, betrachten wir erneut als Beispiel das Programm ABRO. Die dazu in diesem Kapitel gezeigten Listings sind nicht wie bisher Ausschnitte aus Programmen, sondern komplette Java-Klassen. In Listing A.1 ist eine Klasse mit mit der `main`-Methode für ABRO abgebildet.

Listing A.1: `main`-Methode zum Steuern von ABRO

```java
package sj.examples;

import java.io.*;
import java.util.*;
import sj.*;
import sj.exceptions.SignalNotDeclaredException;

public class ABROMain {

  public static void main(String[] args) throws IOException {

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    ABRO program = new ABRO();

    while (!program.isTerminated()) {

      ArrayList<Signal> inputSignals = new ArrayList<Signal>();
      System.out.print("Please enter the signals for the current tick: ");
      for (String s : in.readLine().split(" ")) {
        try {
          inputSignals.add(SignalConverter.string2Signal(program, s));
        } catch (SignalNotDeclaredException e) {
          System.out.println("ERROR: The signal " + s
              + " is not part of " + program.getName());
        }
      }

      Signal[] signalArray = program.doTick(inputSignals
          .toArray(new Signal[inputSignals.size()]));

      System.out.println("Signals after tick");
      for (Signal s : signalArray) {
        System.out.println(s);
      }

    }
  }
}
```

Listing A.2: Komplettes ABRO in SJ

```
package sj.examples;

import sj.SJProgram;
import sj.Signal;
import sj.examples.ABRO.StateLabel;
import static sj.examples.ABRO.StateLabel.*;

public class ABRO extends SJProgram<StateLabel> {

  enum StateLabel {
    ABO, AB, WaitA, WaitB, ABMain, ABOMain
  }

  public Signal A, B, R, O;

  public ABRO() {
    super(ABO, 4);
    initSignals();
  }

  @Override public void tick() {
    while (!isTickDone()) {
      switch (state()) {
        case ABO:
          fork(AB, 1);
          forkeB(ABOMain);
          break;

        case AB:
          fork(WaitA, 2);
          fork(WaitB, 3);
          forkeB(ABMain);
          break;

        case WaitA:
          if (awaitDoneCB(A)) {
            termB();
          }
          break;

        case WaitB:
          if (awaitDoneCB(B)) {
            termB();
          }
          break;

        case ABMain:
          if (joinDoneCB()) {
            O.emit();
            termB();
          }
          break;

        case ABOMain:
          if (awaitDoneCB(R)) {
            transB(ABO);
          }
          break;

      }
    }
  }
}
```

Ziel der einzigen Methode der abgebildeten Klasse `ABROMain` ist das Steuern des Programmflusses der Programms ABRO. Hierzu werden die im Unterkapitel 5.2 vorgestellten Methoden `isTerminated()` (Zeile 15) und `doTick(...)` (Zeile 28) verwendet. Vor Aufruf von `doTick(...)` werden jeweils die für diesen Tick aktuellen Eingabesignale von der Konsole eingelesen. Hierzu müssen die gelesenen Signale in ein Array von Signalobjekten umgewandelt werden. Zu sehen ist dies in den Zeilen 17 bis 26. Nach jedem erfolgreichen Tick wird die aktuelle Signalbelegung auf der Konsole ausgegeben.

Für unterschiedliche Umgebung und Verwendungszwecke muss die Steuerung des Programms individuell programmiert werden. Die hier vorgestellte Variante zum Steuern von ABRO steht beispielhaft dafür, wie eine Steuerung mittels Konsole realisiert werden kann.

In Listing A.2 ist das Programm ABRO, implementiert in SJ, vollständig abgebildet. Das Verhalten von ABRO auf unterschiedliche Eingaben wurde bereits im Unterkapitel 2.3 ausführlich erläutert. In den Zeilen 10 bis 12 werden die im Switch-Statement verwendeten Labels als Enumerations deklariert. Dieses ist nötig, damit diese typsicher mittels Java Generics in der Oberklasse verwendet werden können. Ein Mitteilen an die Oberklasse, welche Label das Programm besitzt, geschieht in Zeile 8.

Im Anschluss an das Deklarieren aller Labels werden die Signale deklariert. Danach folgen Konstruktor und Tickfunktion, die nacheinander die einzelnen Befehle ausführen. Das Schreiben eines synchronen Programms mit SJ ist in Abbildung A.1 als Flowchart dargestellt. Dort werden Schritt für Schritt die Bestandteile aufgeführt, die für ein SJ-Programm erstellt werden müssen.

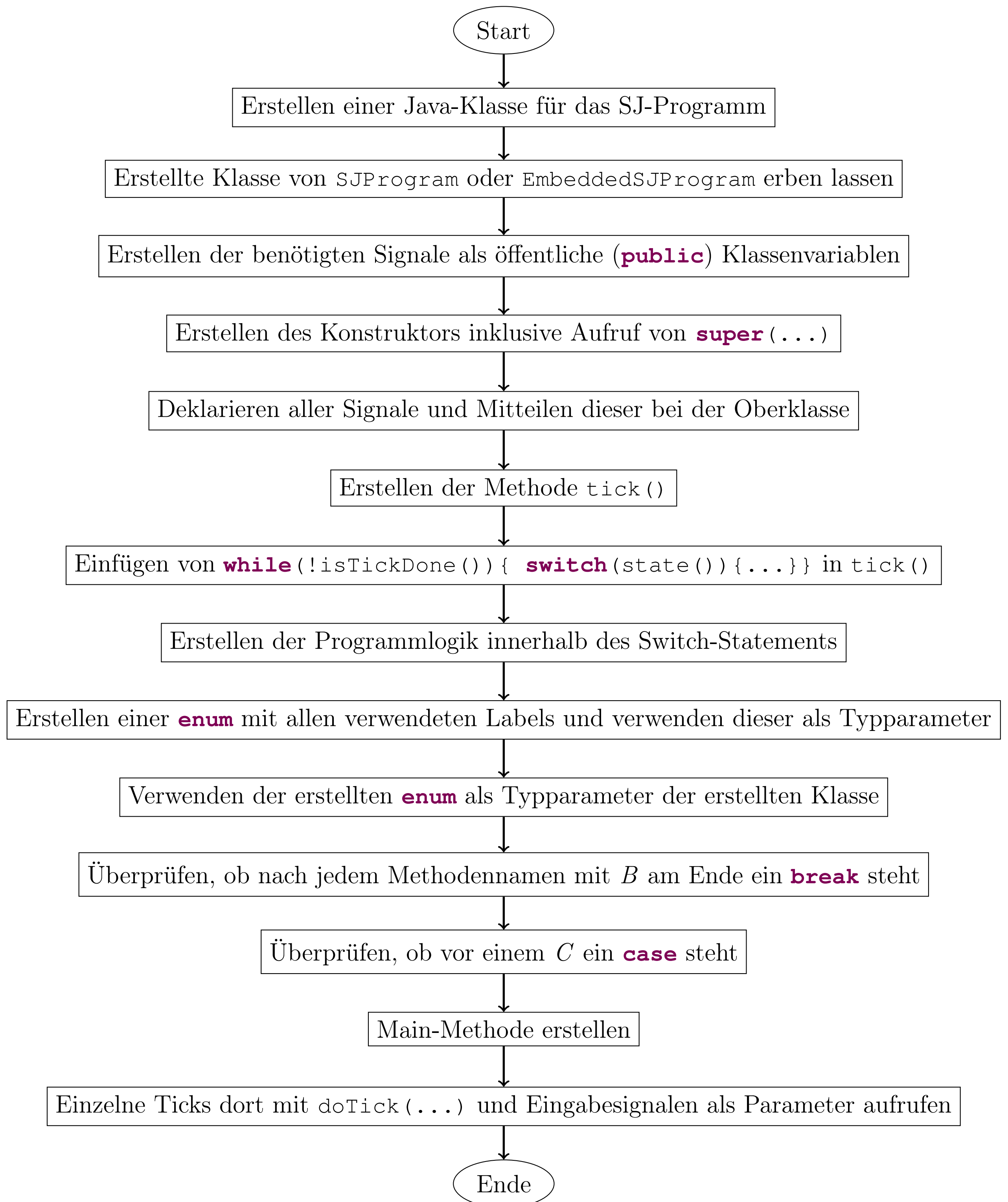

Abbildung A.1.: Erstellen eines SJ-Programms