Universität Stuttgart

Analyse expliziter Zustandsverwaltung als Mittel der Synchronisation

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Martin Wittiger
aus Stuttgart

Hauptberichter:	Prof. Dr. rer. nat. Erhard Plödereder
Mitberichter:	Prof. Dr. Michael Leuschel
Tag der mündlichen Prüfung:	17. Mai 2018

Institut für Softwaretechnologie der Universität Stuttgart
2018

Inhaltsverzeichnis

Zusammenfassung

Parallelität ist fester Bestandteil moderner Softwareentwicklung. Nebenläufige Zugriffe auf geteilte Ressourcen müssen synchronisiert werden, da sonst Softwarefehler wie Data-Races entstehen. Zur Synchronisation werden neben etablierten auch häufig »selbstgestrickte« zustandsbasierte Mechanismen eingesetzt. Diese implementieren oft endliche Automaten. Bestehenden leichtgewichtigen, auf Mustererkennung ausgerichteten Ansätzen zur Analyse von Synchronisationseigenschaften expliziter Zustandsverwaltung fehlt es an Mächtigkeit, um solche Muster befriedigend zu analysieren. Diese Abhandlung stellt einen neuen, deutlich schwergewichtigeren Ansatz zur Analyse expliziter Zustandsverwaltung als Mittel der Synchronisation vor. Dabei reduziert statische Analyse in der Sprache C geschriebene Systeme auf im formalen Prozesskalkül CSP verfasste Modelle. Refinement-Checker untersuchen, ob Paare von Zugriffen auf Variablen im Modell ein Data-Race bilden. Lässt sich das Data-Race im Modell ausschließen, ist es dank konservativer Approximierung auch im ursprünglichen System unerreichbar. Die Modellierung und Vorverarbeitung des Modells wird erläutert. Die Evaluation zeigt, dass der neue Ansatz oft eine bessere Einschätzung der Synchronisationseigenschaften expliziter Zustandsverwaltung liefert als bisherige Ansätze. Die Anwendung auf reale eingebettete Systeme aus Automobilen demonstriert, dass der Ansatz praktisch einsetzbar ist.

Abstract

Concurrency is ubiquitous in modern software. To avoid errors like data races, software developers need to synchronize accesses to shared resources. They commonly prefer finite automata implemented in state variables over other means of synchronization. Existing approaches for the analyses of such state-based synchronization are lightweight but lack clout in practise. Their pattern-based methods prove to be insufficient. This dissertation presents a novel and more heavyweight approach for the analysis of state-based synchronization. Static analyses are used to reduce systems written in C to models in the formal language CSP_M. This allows refinement checkers to determine whether specific pairs of accesses form a data race. If a data race can be refuted in the model, it is also unreachable in the original system. A comparison shows the approach to be substantially better at discerning synchronisation properties of state-based synchronisation. Its successful application to real-world embedded systems from the automotive domain demonstrates practicality.

Danksagungen

Ohne die Hilfe und Unterstützung vieler Menschen wäre diese Abhandlung nie entstanden. Diesen Menschen bin ich zu Dank verpflichtet.

An erster Stelle danke ich Prof. Erhard Plödereder für die Betreuung in allen Phasen meines Promotionsvorhabens und für all die Ressourcen, die mir seine Abteilung zur Verfügung gestellt hat.

Ich danke meiner Familie und meinen Freunden für seelische Unterstützung und für ihr Interesse an meinen Erfolgen und Problemen. Ihr gebt mir Rückhalt.

Dank gilt auch allen meinen Kollegen – jetzigen und ehemaligen – für Austausch, Lob und Kritik, für Zusammen- und Zuarbeit in Forschung, Lehre und dem Verfassen dieser Abhandlung und für die angenehme Stimmung, die an unserem Institut herrscht. Ich danke den Bauhausentwicklern, deren jahrelange Vorarbeit, insbesondere im Bereich der Data-Race-Analyse, meine Arbeit erst ermöglicht hat.

Darüber hinaus gilt mein Dank unseren akademischen und industriellen Partnern und all den Menschen, die mich gelehrt und gebildet haben.

Kapitel 1

Einleitung

In der industriellen Anwendung arbeitet Software heute nebenläufig – und diese Nebenläufigkeit ist auch kaum wegzudenken. Programmiersprachen begegnen den Aufgaben, die mit dieser Nebenläufigkeit Einzug halten, auf unterschiedliche Art und Weise.

Viele Sprachen verfügen über Sprachkonstrukte, die Berechnungspfade explizit oder implizit als nebenläufig auszuführen markieren. Beispiele hierfür sind »Tasks« in der Sprache Ada und »Threads« in der Sprache C++, aber auch »magische Interfaces/Klassen« wie »Runnable« in der Sprache Java oder »Promises« in der Sprache Javascript.

In anderen Sprachen werden betriebssystemnahe Bibliotheken verwendet, um Programmteile nebenläufig auszuführen. Ein bekanntes Beispiel hierfür ist die in der Sprache C genutzte Bibliothek »pthreads«.

Eine dritte Variante realisiert Nebenläufigkeit mittels des Ausführungssystems oder durch parallelisierende Compiler. Dabei wird die Eigenschaft der Parallelisierbarkeit durch den Entwickler annotiert oder durch Analysen ermittelt. Diese Analysen müssen in der Regel jedoch ebenfalls auf Annotationen des Entwicklers zurückgreifen. Dabei nehmen sie die vom Benutzer zugesicherten Eigenschaften als gegeben hin und prüfen sie – wenn

überhaupt – nur in einfacher Weise auf Plausibilität. Beispiele hierfür sind »OpenMP« und die Parallelisierung, die der Glasgow Haskell Compiler bietet.

Unabhängig davon, auf welche Weise Sprachen Nebenläufigkeit realisieren, wird sie im Wesentlichen von menschlichen Software-Entwicklern bestimmt und diese machen dabei Fehler.

Zwar können Sprachdesigner durch gutes Sprachdesign die Fehlerträchtigkeit von Programmiersprachen im Bereich Nebenläufigkeit in den von ihnen entwickelten Sprachen reduzieren, viele Fehlerklassen aus dem Bereich der Nebenläufigkeit lassen sich jedoch, zumindest für realistisch komplexe Software, nicht ausschließen. Dem begegnet die Praxis unter anderem mit Software-Test und -Analyse.

Diese Abhandlung beschreibt und evaluiert einen neuen Ansatz zur Analyse von sicherheitskritischen nebenläufigen Echtzeitsystemen, wie sie für Automobile entwickelt werden. Der Ansatz erweitert einen bestehenden Ansatz, der mittels konservativer statischer Software-Analyse eine bestimmte Teilklasse von Fehlern im Bereich Nebenläufigkeit – die Data-Races – findet.

Der in dieser Abhandlung vorgestellte neue Ansatz bewertet in der Praxis vorkommende Muster, die den Systemzustand in expliziter Weise speichern, im Bezug auf Synchronisationseigenschaften. Mittels statischer Software-Analyse modelliert er Systeme konservativ in der formalen Sprache CSP_M. Dadurch lassen sich die oftmals endlichen Automaten ähnelnden Muster präzise analysieren. Es kann insbesondere ermittelt werden, inwieweit sie sich zur Synchronisation eignen.

1.1 Gliederung der Abhandlung

Der erste Teil dieser Abhandlung führt in das Arbeitsgebiet ein. Er definiert die verwendeten Begriffe und erörtert den Stand der Forschung. Dabei definiert er Data-Races und beschreibt Ansätze zur Data-Race-Analyse. Außerdem wird die explizite Zustandsverwaltung definiert und es werden einige Besonderheiten der untersuchten Systeme beschrieben. Weiterhin stellt dieser Teil die verwendete Modellsprache CSP_M vor und erklärt, wie in

dieser Sprache beschriebene Modelle maschinell verarbeitet werden können. Zudem benennt er alternative Herangehensweisen.

Der zweite Teil dieser Abhandlung beschreibt den in der Abhandlung vorgestellten neuen Ansatz und seine Umsetzung, also die Implementierung des Analyse-Systems. Der Aufbau einer Werkzeugkette wird beschrieben. Dabei erläutert dieser Teil, wie sich eine durch statische Software-Analyse gewonnene Abstraktion des Systems im formalen CSP-Kalkül modellieren lässt und wie dies Refinement-Checkern erlaubt Fragen zu Synchronisationseigenschaften zu beantworten. Er beschreibt ferner ein Werkzeug, das CSP_M-Modelle semantikerhaltend komprimiert und damit die Anwendung des Ansatzes auf Systeme realistischer Größe erlaubt.

Der dritte Teil evaluiert den Ansatz der vorliegenden Arbeit. Er erörtert die Eigenschaften des neu entstandenen Verfahrens und grenzt die Forschung von den im ersten Teil beschriebenen verwandten Arbeiten ab. Weiterhin wird gezeigt, wie die Werkzeugkette auf Systeme realistischer Größe angewendet werden kann.

1.2 Konventionen

In dieser Abhandlung werden Abkürzungen nur verwendet, wenn diese etablierter sind als die ausgeschriebene Form. Die ausgeschriebenen Formen aller Abkürzungen sind im Glossar zu finden. Dort werden ebenfalls wichtige Grundbegriffe kurz erläutert.

Mathematische Formalismen werden in dieser Abhandlung gemäß üblichen Konventionen verwendet. Die wichtigsten dieser Konventionen werden im Apparat (Seite 315) erläutert.

Teil I

Arbeitsgebiet

Kapitel 2

Fehlerklasse Data-Races

Abbildung 2.1 zeigt ein einfaches C-Programm. Es wird hier davon ausgegangen, dass die beiden Prozeduren `task_a` und `task_b` des Programms nebenläufig ausgeführt werden. (Dies könnte beispielsweise durch geeignete Compiler-Vorgaben oder die Verwendung der Bibliothek pthreads realisiert sein.) Dieses einfache Programm enthält ein Data-Race. Dieses Kapitel definiert Data-Races und erklärt, warum Data-Races als Software-Fehlerklasse anzusehen sind.

```
1  int c;
2
3  void task_a() {
4     int local = c;
5     // ...
6  }
8  void task_b() {
9     c = 7;
10 }
11
12 int main() {
13    // Setup concurrency
14    // ...
15 }
```

Abbildung 2.1: Einfaches Beispiel für ein nebenläufiges C-Programm mit einem Data-Race

2.1 Tasks

Die Begriffe Task und Thread werden in der Literatur teils synonym verwendet. Ein Task wird hier als ein Ausführungsstrang von Anweisungen angesehen, sowohl in der statischen als auch in der dynamischen Sicht. Wie auch zum Teil in der Literatur wird somit sprachlich nicht zwischen einem zur Ausführung tatsächlich abzuarbeitenden Task (dem »dynamischen Task«) und der abstrakten Sicht darauf (dem abstrakten Task) unterschieden. Dies macht vor allem dann einen Unterschied, wenn ein abstrakter Task mehrfach zu sich selbst parallel ausgeführt wird. Dann kommen auf einen abstrakten Task mehrere dynamische Tasks zur Laufzeit. Dieser Taskbegriff entspricht dem von Keul [Keu11].

2.2 Definitionen für Data-Races in der Literatur

Der Literatur sind verschiedene Definitionen für Data-Races zu entnehmen. Manche der Autoren definieren ein Data-Race als ein bestimmtes Verhalten eines Programms zur Laufzeit.

Andere bezeichnen die Möglichkeit dieses Auftretens als Data-Race und sehen es somit als Eigenschaft eines Programms beziehungsweise eines Paares von Zugriffs-Ausdrücken an, von Data-Races behaftet zu sein.

Definitionen, die sich auf Laufzeitverhalten beziehungsweise Traces von Programmabläufen beziehen, werden als dynamisch bezeichnet; beziehen sich die Definition hingegen auf den Quelltext oder eine statische Sicht darauf, werden sie als statisch bezeichnet.

2.2.1 Nomenklatur von Savage et al.

Savage et al. [SBN+97] definieren im Jahr 1997 im Bezug auf Data-Races Folgendes:

> *A data race occurs when two concurrent threads access a shared variable and when at least one*

> *access is a write and the threads use no explicit mechanism to prevent the accesses from being simultaneous.*
>
> *If a program has a potential data race, then the effect of the conflicting accesses to the shared variable will depend on the interleaving of the thread executions.* «

Hier wird ein Data-Race als ein bestimmtes Laufzeitverhalten definiert. Mit dem Begriff »potential data race« werden Paare von Variablenzugriffen bezeichnet, die im Konflikt stehen, die also zur Laufzeit ein Data-Race auftreten lassen können. Dies ist im Zusammenhang damit zu sehen, dass das von Savage et al. vorgestellte Werkzeug Eraser dynamisch arbeitet, also einzelne Ausführungen von Programmen betrachtet.

2.2.2 Data-Races im C++11-Standard

Auch der C++11-Standard [ISO11] widmet sich der nebenläufigen Ausführung von Programmen und definiert Data-Races im Absatz 1.10.4 ff. folgendermaßen:

> » *Two expression evaluations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location. […]*
>
> *The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.* «

Im ausgelassenen Teil der Definition wird die Happens-Before-Relation klar definiert. Dies dient dazu, die Semantik von Synchronisationsoperationen zu definieren. Damit wird vermieden, den vagen Begriff der »explicit mechanisms« von Savage et al. zu verwenden.

Auch hier ist das Data-Race klar als Laufzeitverhalten charakterisiert.

2.2.3 Definition von Vaziri, Tip und Dolby

Vaziri, Tip und Dolby [VTD06], die Data-Races mit Hilfe statischer Software-Analyse finden, definieren Data-Races zunächst wie folgt, passen ihre Definition dann aber den von ihrer Implementierung tatsächlich erkannten Mustern an.

> *[…] [I]nconsistent results may be computed when two threads access shared data concurrently. In particular, a data race is said to occur when two threads concurrently access some data, where one of these accesses is a write, and where no synchronization exists between the threads.*

2.2.4 Definition für die Programmiersprache Java nach JSR 133

Der JSR 133: Java Memory Model and Thread Specification [JSR04] definiert Data-Races in Abschnitt 3 folgendermaßen:

> *Two accesses (reads of or writes to) the same shared field or array element are said to be conflicting if at least one of the accesses is a write.*
>
> *[…] When a program contains two conflicting accesses that are not ordered by a happens-*

before relationship, it is said to contain a data race. A correctly synchronized program is one that has no data races.

Interessant ist hier, dass nicht eine Ausführung (»execution«) eines Programms ein Data-Race enthalten kann, sondern das Programm selbst.

Die Definition verlangt auf den ersten Blick nicht, dass die beiden im Konflikt stehenden Zugriffe aus unterschiedlichen Tasks (genauer Task-Instanzen) kommen. Dafür scheint jedoch die Happens-Before-Relation so definiert zu sein, dass von zwei Zugriffen aus demselben Task stets einer vor dem anderen steht. Dadurch kann in der Definition auf die Bedingung, dass beide Zugriffe aus unterschiedlichen Tasks herrühren müssen, verzichtet werden.

Es ist im Allgemeinen jedoch nicht möglich, eine innerhalb jedes Tasks totale Happens-Before-Relation aufzustellen, da es leicht möglich ist, ein Programm in Java zu schreiben, bei dem Zugriffe in unterschiedlichen Ausführungen in unterschiedlicher Reihenfolge auftreten. In einer »execution« einer Task-Instanz existiert eine solche totale Happens-Before-Relation jedoch. Daher ist aus der Definition eine implizite existenzielle Quantifizierung im Sinne von ». . . that are not ordered by a happens-before relationship *in any of the program's possible executions* . . . « herauszulesen.

2.3 Verwendete Data-Race-Definitionen

Die in der Literatur vorkommenden Definitionen für Data-Races unterscheiden sich deutlich. In dieser Abhandlung wird eine eigene (statische) Definition des Data-Race verwendet.

Die in dieser Abhandlung beschriebene Implementierung baut auf einer spezifischen Data-Race-Analyse – dem Bauhaus Data Race Detector – auf. Die zugrunde liegende Analyse könnte aber durch eine andere ausgetauscht werden, solange diese ebenfalls konservativ ist und dieselben oder wenigstens ähnliche Speicherzugriffsinformationen liefern kann. Die verwendete

Abbildung 2.2: Drei Begriffswelten für Speicherzugriffe

Definition eines Data-Races ist so generisch gewählt, dass sie die Analyse nicht unnötig einschränkt.

2.3.1 Drei Begriffswelten

Für die in dieser Abhandlung verwendete Definition eines Data-Races sind drei Begriffswelten relevant: der Quelltext, die Traces und die statische Sicht auf das Programm (siehe Abbildung 2.2).

2.3.1.1 Quelltext

Ein nebenläufiges Programm wird definiert über den Quelltext.[1] Darin stehen Zugriffsausdrücke. Dabei handelt es sich im einfachsten Fall um das Lesen oder Schreiben einer Variablen. Statt auf eine Variable kann auch auf eine Komponente eines Verbundtyps (Record) oder auf ein Element eines Arrays zugriffen werden. Ebenso können Zeigerdereferenzierungen Ziel eines Zugriffs sein.

Die Menge der Zugriffsausdrücke eines gegebenen Programms wird hier mit A bezeichnet.

2.3.1.2 Traces

Die Semantik der Sprache erlaubt für jeden Quelltext eine Menge an möglichen Programmabläufen, den Traces. Je nach Programm kann es endlich viele, abzählbar unendlich viele oder sogar überabzählbar unendlich viele Traces geben.

Jede Ausführung eines Zugriffsausdrucks wird dynamischer Zugriff genannt.

Ein Trace eines gegebenen Programms wird mit Θ bezeichnet; die in diesem Trace Θ vorkommenden dynamischen Zugriffe mit T_Θ.

[1] Zusätzlich ist teilweise eine Nebenläufigkeitskonfiguration relevant. Dies kann hier zunächst noch vernachlässigt werden.

2.3.1.3 Statische Sicht

Die verwendete statische Data-Race-Analyse liefert eine statische Sicht auf das Programm. Diese enthält abstrakte Zugriffe. Unter einem abstrakten Zugriff wird ein Speicherzugriff verstanden, so genau er mit der verwendeten statischen Software-Analyse erfasst werden kann. Dies wird bei der Besprechung des Beispiels aus Abbildung 2.7 (Seite 39) näher erläutert.

Die Menge der statischen Zugriffe eines gegebenen Programms wird hier mit Z bezeichnet.

2.3.2 Relationen zwischen den Zugriffsmengen

Für die Definition des Data-Races sind ferner die in Abbildung 2.2 als Pfeile dargestellten Relationen relevant:

- Mit $a_\Theta \subset T_\Theta \times A$ wird die Relation zwischen den dynamischen Zugriffen und ihren zugrunde liegenden Zugriffs-Ausdrücken bezeichnet. Sie ist linkstotal und rechtseindeutig und somit eine Funktion.

  Die Funktion a_Θ ist im Allgemeinen nicht linkseindeutig (injektiv), da zum Beispiel ein in einer Schleife stehender Zugriffsausdruck mehrfach ausgeführt werden kann.

  Die Funktion a_Θ ist im Allgemeinen nicht rechtstotal (surjektiv). Ein in totem Code stehender Zugriffsausdruck wird zum Beispiel niemals ausgeführt werden und hat damit auch kein Urbild in den dynamischen Zugriffen.

- $f_\Theta \subset T_\Theta \times Z$ bezeichnet die Relation zwischen den dynamischen Zugriffen und den abstrakten Zugriffen, von denen sie repräsentiert werden. Sie ist linkstotal und rechtseindeutig und somit eine Funktion.

  Die Funktion f_Θ ist im Allgemeinen nicht linkseindeutig (injektiv), da bei einem nicht-terminierenden Programm der Trace in der Regel unendlich viele dynamische Zugriffe enthält, wogegen Z stets endlich ist (Pigeonhole Principle). Zum Beispiel verursacht ein in einer unendlichen Schleife stehender Zugriffsausdruck möglicherweise unendlich

viele dynamische Zugriffe, wird jedoch durch nur einen oder jedenfalls endlich viele abstrakte Zugriffe dargestellt.

Die Funktion f_Θ ist im Allgemeinen auch nicht rechtstotal (surjektiv). Aus einem Zugriffsausdruck, der in totem Code steht, werden abstrakte Zugriffe generiert, wenn die Analyse den Code nicht als tot erkennt. Natürlich bildet f_Θ aber keinen dynamischen Zugriff auf solche toten abstrakten Zugriffe ab – diese sind also ein Gegenbeispiel für die Eigenschaft der Surjektivität. Da keine Analyse allen toten Code finden kann, ist dies auch im Allgemeinen unvermeidlich.

- Mit $s \subset Z \times A$ wird die Relation zwischen den abstrakten Zugriffen und den dazugehörigen Zugriffs-Ausdrücken bezeichnet. Sie ist linkstotal, da jedem abstrakten Zugriff ein Zugriffsausdruck zugrunde liegen muss. Es wird hier gefordert, dass sie auch rechtseindeutig und somit eine Funktion ist. Es ist damit unzulässig, mehrere Zugriffs-Ausdrücke mit einem abstrakten Zugriff abzudecken. (Dieser eine abstrakte Zugriff kann allerdings mehrere mögliche Ziele haben.)

  Die Funktion s ist im Allgemeinen nicht linkseindeutig (injektiv). Durch »Sensitivitäten« der verwendeten statischen Data-Race-Analyse können mehrere abstrakte Zugriffe für einen Zugriffsausdruck entstehen. So ist der Bauhaus Data Race Detector (und wohl jede sinnvolle Data-Race-Analyse) Task-sensitiv. Das bedeutet zum Beispiel, dass aus einem Assignment-Statement in einer Prozedur `p` für jeden Task, in dessen Aufrufgraph die Prozedur `p` steht, in der Regel ein eigener abstrakter Zugriff entsteht.

  Die Funktion s ist im Allgemeinen auch nicht rechtstotal (surjektiv). Für einen in totem Code stehenden Zugriffsausdruck wird zum Beispiel kein abstrakter Zugriff erzeugt, wenn die Analyse erkennt, dass der Zugriffsausdruck tatsächlich tot ist. Beim Bauhaus Data Race Detector ist dies der Fall, wenn das Assignment-Statement in einer Prozedur steht, die im Aufrufgraphen nicht enthalten ist. Ebenso werden durch `if (0) {...}` oder `if (false) {...}` umgebene Anweisungen als tot erkannt. In untersuchten Systemen werden mit diesem Mus-

ter Programmteile »auskommentiert« oder (etwa zur Konfiguration) »abgeschaltet«.[1]

An die semantische Korrektheit der statischen Sicht wird hier der Anspruch gestellt, dass a_Θ genau die Komposition von f_Θ und s ist ($a_\Theta = f_\Theta \circ s$).

2.3.3 Zugriffsziele und Zeigerdereferenzierungen

Das Ziel eines statischen Zugriffs ist nicht immer leicht zu ermitteln. In dieser Abhandlung wird für die Ziele der Begriff Points-To-Object verwendet. Die Points-To-Objects sind statisch abgebildete reale Speicheradressen.

So werden zum Beispiel bei einem Array dynamischer Größe alle Zellen auf dasselbe Points-To-Objekt abgebildet. Points-To-Objects können auch selbst Zeiger sein. Tabelle 2.1 zeigt eine Gegenüberstellung der Begriffe aus den drei Welten.

Das Ermitteln der relevanten Zugriffsziele wird durch Zeigerdereferenzierungen erschwert. Hier wird von den Zeigeranalysen eine geeignete Approximation durchgeführt.

Weiter erschwert werden kann die Analyse durch unspezifische Typen in der Sprache C (zum Beispiel Dereferenzierung eines Ausdruckes vom Typ `int`) und Union-Typen.

2.3.4 Definition Data-Race-Situation

In der Begriffswelt der Traces lässt sich eine Data-Race-Situation wie folgt definieren: Seien t_1 und t_2 dynamische Zugriffe aus T_θ. In einem Trace Θ ist das Paar dynamischer Zugriffe $\langle t_1, t_2 \rangle$ eine Data-Race-Situation genau dann, wenn

- t_1 und t_2 aus unterschiedlichen (dynamischen) Tasks stammen,

[1] Die in der Automobilindustrie verbreitet Anwendung findenden Misra-C-Richtlinien [MIR08] verbieten zwar durch Regel 14.1 toten Code explizit, fordern aber wiederholt das Vorhandensein von leeren Blöcken und unerreichbarem Kontrollfluss ein – so fordert zum Beispiel Regel 19.4, das »`do ... while (0)`«-Konstrukt zu verwenden. Auch durch Konfiguration unerreichbarer Code wird möglicherweise gestattet.

- mindestens einer der beiden ein schreibender Zugriff ist,
- mindestens einer der beiden ein nicht atomarer Zugriff ist,
- die Speicherbereiche von t_1 und t_2 gleich sind oder überlappen,
- t_1 vor t_2 steht und
- keine Anweisung zwischen t_1 und t_2 aus dem Task von t_1 stammt.

Um plausibel zu machen, dass ein Trace mit einer Data-Race-Situation problematisch ist, kann die Situation betrachtet werden, in der der Zugriff t_1 nicht atomar ist und nur zum Teil ausgeführt ist, wenn der Task unterbrochen wird. Dies erklärt auch, warum Data-Races auf Typen, auf die normalerweise atomar zugegriffen wird, unter Umständen wenig problematisch sind. Hieran erklärt sich auch die letzte Bedingung: Um problematisch zu sein, muss der erste Task tatsächlich direkt nach dem Zugriff unterbrochen werden.

Die von der Definition erwartete Unterbrechung unmittelbar nach dem Zugriff deckt genau auch die Situation ab, auf die eigentlich gezielt wird: eine Unterbrechung während dem Zugriff.

Atomar meint damit über Sprachbestandteile wie `atomic` als atomar gekennzeichnete Zugriffe. Andere Synchronisationsmechanismen als atomar müssen in der Definition nicht explizit erwähnt werden, da sie die Traces ohnehin passend einschränken.

Es ist dabei auch denkbar, bestimmte einfache Zugriffe auf Variablen als atomar zu betrachten, auch wenn der Sprachstandard dies nicht garantiert. Ist die Variable `x` zum Beispiel als `volatile int` deklariert, spricht viel dafür, die schreibenden Zugriffe »`x = 5;`« oder »`x = y;`« als atomar anzusehen.

2.3.5 Definition Data-Race

Die Definition eines (statischen) Data-Races und eines Quelltext-Data-Races lautet hier wie folgt:

Ist $\langle t_1, t_2 \rangle$ eine Data-Race-Situation in einem Trace Θ, wird das Paar abstrakter Zugriffe $\langle f_\Theta(t_1), f_\Theta(t_2) \rangle$ als (statisches) Data-Race und das Paar Zugriffs-Ausdrücke $\langle a_\Theta(t_1), a_\Theta(t_2) \rangle$ als Quelltext-Data-Race bezeichnet.

Tabelle 2.1: Gegenüberstellung von Begriffen aus den drei Welten

Quelltext	**Traces**	**statische Sicht**
Zugriffsausdruck	dynamischer Zugriff	abstrakter Zugriff
Menge A	Menge T_Θ	Menge Z
Variable, ...	Speicheradresse	Points-To-Object
Quelltext-Data-Race	Data-Race-Situation	(statisches) Data-Race

Dabei spielt die Reihenfolge der Zugriffe keine Rolle; $\langle a, b\rangle$ ist dasselbe Data-Race wie $\langle b, a\rangle$. In dieser Abhandlung gilt die Konvention, dass von zwei Zugriffen aus unterschiedlichen Tasks mit unterschiedlichen Prioritäten der Zugriff aus dem Task mit niedrigerer Priorität an erster Stelle steht.

Jedes statische Data-Race lässt sich auf ein Quelltext-Data-Race zurückführen. Das zum statischen Data-Race $\langle z_1, z_2\rangle$ gehörende Quelltext-Data-Race ist $\langle s(z_1), s(z_2)\rangle$.

Umgekehrt kann es jedoch sein, dass zwar das Tupel $\langle s(z_1), s(z_2)\rangle$ ein Quelltext-Data-Race, das Tupel $\langle z_1, z_2\rangle$ aber kein statisches Data-Race ist. Praktisch gesehen bedeutet dies, dass es in der statischen Sicht aus den Sensitivitäten heraus zusätzliche Informationen/Beschränkungen zum Data-Race geben kann. Eine Situation, in der das vorkommt, ist folgende: Seien z_1, z_2, z_3 und z_4 vier unterschiedliche abstrakte Zugriffe, sodass $\langle s(z_1), s(z_2)\rangle = \big\langle s(z_3), s(z_4)\big\rangle$, und $\langle z_1, z_2\rangle$ ein statisches Data-Race ist, $\big\langle z_3, z_4\big\rangle$ aber keines. In Abschnitt 2.3.6.3 wird ein Beispiel hierfür ausführlich besprochen.

2.3.6 Beispiele für Data-Race-Analyseergebnisse

Um plausibel zu machen, dass die vorgestellte Definition eines Data-Races sinnvoll ist, werden im Folgenden drei Beispiele untersucht.

2.3.6.1 Das Programm aus Abbildung 2.1

Zunächst soll das einfache Beispiel aus Abbildung 2.1 (Seite 23) betrachtet werden. Um die Data-Race-Analyse durchführen zu können, werden dem

Task	Z.	Anweisung
a	3	ENTER task_a()
a	4	int local = c;
a	6	RETURN
b	8	ENTER task_b()
b	9	c = 7;
b	10	RETURN

Task	Z.	Anweisung
a	3	ENTER task_a()
a	4	int local = c;
b	8	ENTER task_b()
b	9	c = 7;
b	10	RETURN
a	6	RETURN

Abbildung 2.3: Zwei Traces des Programms aus Abbildung 2.1

Programm einige Zeilen hinzugefügt, die die Nebenläufigkeit mit Hilfe der Bibliothek pthreads aufsetzen.

Die verwendete Data-Race-Analyse liefert sechs abstrakte Zugriffe. Vier davon stammen aus den Quelltextzeilen, welche die Nebenläufigkeit aufsetzen – diese werden hier ignoriert. Die verbleibenden zwei, ein lesender abstrakter Zugriff aus Zeile 4 und ein schreibender aus Zeile 9, betreffen die Variable c. Die beiden Zugriffe werden unterschiedlichen Tasks zugeordnet und haben beide dasselbe Points-To-Objekt als Ziel.

Da auch keine erkennbare Synchronisation vorhanden ist, warnt die Data-Race-Analyse vor einem Data-Race, bestehend aus diesen beiden Zugriffen. Dies ist die einzige Data-Race-Warnung für dieses Programm.

Dass diese Data-Race-Warnung berechtigt ist, weil es sich bei dem Zugriffspaar tatsächlich um ein Data-Race handelt, zeigt sich in Abbildung 2.3.

Der linke Trace enthält keine Data-Race-Situation, da die beiden Tasks hier sequentiell ausgeführt werden. Im rechten Trace lässt sich aber eine Data-Race-Situation erkennen. Nachdem die globale Variable c gelesen wurde, wird der Task a unterbrochen, und bevor er erneut ausgeführt wird, schreibt der Task b in die Variable c.

2.3.6.2 Auswirkungen von Synchronisation

Als zweites Beispiel wird das in Abbildung 2.4 dargestellte Programm betrachtet. Es enthält die Tasks a, b und c sowie den Main-Task (nicht dargestellt),

```
1  mutex_t mutex;
2
3  volatile unsigned int
       a = 5;
4  volatile unsigned int
       b = 0;
5
6  void task_a() {
7    while (true) {
8      a = 1;
9  } }
10
11 void task_b() {
12   unsigned int l1 = 0;
13
14   while (true) {
15     if (a) {
16       lock(&mutex);
17       b = l1;
18       l1++;
19       a = 0;
20       unlock(&mutex);
21 } } }
```

```
23 void task_c() {
24   unsigned int l2;
25
26   while (true) {
27     lock(&mutex);
28     l2 = b;
29     unlock(&mutex);
30
31     printf
32      ("l2: %u\n", l2);
33 } }
34
35 int main() {
36    \\ ...
37 }
```

Abbildung 2.4: Ein komplizierteres Beispiel mit drei Tasks und Synchronisation durch Mutexe

der die anderen Tasks startet. Die Tasks kommunizieren über die beiden Integer-Variablen `a` und `b`. Zusätzlich wird ein Mutex-Lock verwendet.

Das Programm lässt sich übersetzen. Es wurde ausgeführt und, da es offensichtlich nicht terminiert, nach etwa einer Sekunde abgebrochen. Während dieser Ausführung hat es circa 336 000 Zeilen (wohl-formatierte) Ausgabe produziert. Der Wert von `l2` steigt mit jeder Zeile monoton durchschnittlich um etwa 5 – die Schwankungsbreite ist aber sehr groß (0 – 337). Von schädlichen Auswirkungen von Data-Races ist nichts zu bemerken.

Bei einer manuellen Suche nach Data-Races im Programm fällt auf, dass die Zugriffe aus Tasks `b` und `c` auf die Variable `b` durch den Mutex geschützt

Task	Z.	Anweisung
main	3	a = 5;
main	4	b = 0;
main	35	ENTER main()
a	6	ENTER task_a()
a	8	a = 1; ○ ○
b	11	ENTER task_b()
b	12	l1 = 0;
c	23	ENTER task_c()
b	15	if (a) ○
b	16	lock(&mutex);
b	17	b = l1;
b	18	l1++;
b	19	a = 0; ○
b	20	unlock(&mutex);
c	27	lock(&mutex);
...	...	...

Abbildung 2.5: Ein Präfix eines Traces des Programms aus Abbildung 2.4 – die Kringel (○) markieren die Data-Race-Situationen

sind. Die Initialisierung der Variablen b aus dem Main-Task ist bereits abgeschlossen, wenn die Tasks b und c gestartet werden. Die Variable b ist somit nicht von Data-Races betroffen.

Bei der Variablen a verhält es sich anders. Task a greift ohne jeden Schutz auf die Variable a zu. In Task b ist zwar der zweite Zugriff auf die Variable a (die Zuweisung von 0) durch den Mutex geschützt, der erste, lesende Zugriff in der Bedingung jedoch nicht. Ein »einseitiger« Schutz durch einen Mutex hilft aber nicht weiter, sodass die Variable a hier tatsächlich von zwei Data-Races betroffen ist.

Ein Trace genügt, um zwei Data-Race-Situationen darzustellen, welche die beiden Quelltext-Data-Races belegen. Er ist in Abbildung 2.5 dargestellt.

Der Versuch, einen Trace zu finden, der eine Data-Race-Situation für die Variable b enthält, muss jedoch fehlschlagen. Abbildung 2.6 liefert hierfür eine Begründung: Ist der nebenläufig arbeitende Teil der Ausführung erreicht,

Task	Z.	Anweisung
main	3	a = 5;
main	4	b = 0;
main	35	ENTER main()
b	11	ENTER task_b()
b	12	l1 = 0;
b	15	if (a)
b	16	lock(&mutex);
b	17	b = l1;
c	23	ENTER task_c()
c	27	lock(&mutex);
...	...	...

Abbildung 2.6: Versuch, eine Data-Race-Situation für die Variable b im Programm aus Abbildung 2.4 zu finden

wird auf die Variable b nur noch im Schutz des Mutex-Locks zugegriffen. Der abgebildete Trace kann nicht mit weiteren Anweisungen aus Task c fortgesetzt werden, da die `lock`-Anweisung blockiert, bis ein `unlock` ausgeführt wurde. Die Ausführung muss demnach zunächst zum Task b zurückkehren. Auch »andersherum« (das heißt so, dass der Zugriff aus Task c der erste ist) lässt sich keine Data-Race-Situation konstruieren.

2.3.6.3 Quelltext-Data-Race vs. statisches Data-Race

Das dritte Beispiel veranschaulicht den Unterschied zwischen Quelltext-Data-Race und statischem Data-Race.

Dazu wird das Programm der Abbildung 2.7 betrachtet. Es enthält drei Tasks (`low`, `med` und `high`) und diese greifen alle drei durch Aufrufe der Prozedur `reset` auf die globale geteilte Variable `c` zu.

Das Programm enthält nur einen einzigen Zugriffsausdruck, der die Variable `c` betrifft. Daraus folgt aber nicht, dass es kein Data-Race auf der Variablen `c` gibt. Tatsächlich ist $\langle q, q \rangle$ ein Quelltext-Data-Race, wenn q den Zugriffsausdruck aus Zeile 5 bezeichnet, der in die Variable `c` schreibt.

Um Klarheit zu schaffen, wird die statische Sicht auf das Programm be-

```
1  mutex_t mutex;
2  int c = 5;
3
4  void reset() {
5     c = 0;
6  }
7
8  void task_high() {
9     lock(&mutex);
10    reset(); // 1
11    unlock(&mutex);
12 }
14 void task_med() {
15    reset(); // 2
16
17    lock(&mutex);
18    reset(); // 3
19    unlock(&mutex);
20 }
21
22 void task_low() {
23    reset(); // 4
24 }
```

Abbildung 2.7: Quelltext eines C-Programms, bei dem aus einem Zugriffsausdruck mehrere statische Data-Races entstehen

Tabelle 2.2: Die vier abstrakten Zugriffe auf die Variable `c` aus dem Beispiel aus Abbildung 2.7

Nr.	ID	Kontext
1	W887_523	Task: 2 (`task_high`), Lock auf `mutex`
2	W888_523	Task: 4 (`task_med`)
3	W889_523	Task: 4 (`task_med`), Lock auf `mutex`
4	W890_523	Task: 3 (`task_low`)

trachtet. Die zugrunde gelegte Data-Race-Analyse kann alle Zugriffe auf das (eine) von der Variablen `c` erzeugte Points-To-Objekt ausgeben. Die Daten dieser Ausgabe sind Tabelle 2.2 zu entnehmen.

Die IDs der Zugriffe enden alle auf die gleiche Ziffernfolge. Dies ist kein Zufall, sondern Ausdruck dessen, dass alle vier abstrakten Zugriffe von demselben Zugriffsausdruck herrühren.

Die abstrakten Zugriffe unterscheiden sich durch ihren Kontext. Die hier zugrunde gelegte Data-Race-Analyse ist Task- und Synchronisations-sensitiv. Die Zugriffe werden daher nach Task und im behandelten Beispiel durch die Frage charakterisiert, ob ein Lock auf der Variable `mutex` gehalten wird.

Tabelle 2.3: Auflistung aller Paare von abstrakten Zugriffen aus Tabelle 2.2

a	b	Das Paar $\langle a, b \rangle$ ist ...	Begründung
4	4	...kein Data-Race. ✓	Die Multiplizität von Task 3 ist 1.
4	3	...ein Data-Race. ✗	Ein einseitiges Lock schützt nicht.
4	2	...ein Data-Race. ✗	Es gibt keine Synchronisation.
4	1	...ein Data-Race. ✗	Ein einseitiges Lock schützt nicht.
3	3	...kein Data-Race. ✓	Die Multiplizität von Task 4 ist 1.
3	2	...kein Data-Race. ✓	Die Multiplizität von Task 4 ist 1.
3	1	...kein Data-Race. ✓	Das beidseitige Lock schützt.
2	2	...kein Data-Race. ✓	Die Multiplizität von Task 4 ist 1.
2	1	...ein Data-Race. ✗	Ein einseitiges Lock schützt nicht.
1	1	...kein Data-Race. ✓	Die Multiplizität von Task 2 ist 1.

In Tabelle 2.3 sind alle Paare von abstrakten Zugriffen auf die Variable c aufgelistet. Obwohl alle diese Paare auf dasselbe Quelltext-Data-Race abbilden, sind nur vier von ihnen auch statische Data-Races. Die hier zugrunde gelegte Data-Race-Analyse erkennt dies korrekt. Sie listet genau die vier Paare als Data-Race-Warnung, die tatsächlich Data-Races sind.

Das Beispiel belegt auch, dass eine Data-Race-Analyse, die mehr Data-Race-Warnungen ausgibt als eine andere, nicht unbedingt weniger präzise ist. Tatsächlich kann durch Hinzufügen von Sensitivitäten gleichzeitig die Präzision und die Zahl der Data-Race-Warnungen steigen.

2.4 Einordnung von Data-Races als Softwarefehler

Sowohl in der wissenschaftlichen Literatur als auch in der Praxis der Softwareentwicklung wird kontrovers diskutiert, ob Data-Races notwendigerweise als Softwarefehler oder gar als Softwaremängel angesehen werden müssen[1].

[1] Blum und Gibson [BG16] haben einen Abschnitt mit dem Titel *»Philosophy of bugs«* in ihrer Publikation. Sie sehen Data-Races nicht notwendigerweise als Softwarefehler an. Boehm [Boe11] diskutiert diese Fragestellung ebenfalls und vertritt eine gegenteilige Ansicht.

2.4.1 Definitionen

Hier wird den übereinstimmenden Definitionen von Ludewig und Lichter [LL07] und dem DIN-EN-ISO-9000-Standard für Qualitätsmanagementsysteme [DIN05] gefolgt.

2.4.1.1 Softwarefehler

Software ist fehlerhaft (»enthält einen Softwarefehler«), wenn sie mindestens eine Anforderung nicht erfüllt. Dabei ist es unerheblich, ob der Softwarefehler tatsächlich den Gebrauch der Software einschränkt.

2.4.1.2 Softwaremangel

Software ist mangelhaft (»enthält einen Softwaremangel«), wenn sie einen Softwarefehler enthält, der den beabsichtigten oder festgelegten Gebrauch einschränkt.

Mangelhafte Software ist fehlerhaft; fehlerhafte Software ist aber nicht notwendigerweise mangelhaft. Die Unterscheidung zwischen Softwarefehler und Softwaremangel ist rechtlich bedeutsam. (Vergleiche DIN-EN-ISO-9000-Standard für Qualitätsmanagementsysteme [DIN05].)

2.4.2 Race-Conditions

In dieser Abhandlung bezeichnet eine Race-Condition die Eigenschaft eines Programmes nicht-deterministisch, beispielsweise je nach Timing, für den Anwender unterscheidbares Verhalten zu zeigen beziehungsweise für den Anwender unterscheidbare Ausgaben zu erzeugen.

Race-Conditions sind teils unerwünscht. Es kann sich auch um Softwarefehler handeln. Je nach Einsatzszenario können aber auch völlig unterschiedliche Ausgaben korrekte Ergebnisse sein und es völlig akzeptabel sein, dass mal das eine und mal das andere Ergebnis ausgegeben wird.

Data-Races lassen sich aus einem Programm entfernen, indem die einzelnen Zugriffe mit Locks umgeben werden. Dabei kann es passieren, dass

sich die Semantik des Programmes kaum ändert und das ursprüngliche Fehlverhalten unverändert fortbesteht. Aus einem Data-Race ist damit eine Race-Condition geworden.

2.4.3 Untersuchungsgegenstand C99

Die hier untersuchten Programme folgen dem C99-Standard [ISO05]. Dieser beschäftigt sich nicht mit Nebenläufigkeit beziehungsweise Parallelität. Er enthält keinen der Begriffe »task«, »thread«, »concurrency«/»concurrent«, »race«, »parallel« und »multicore«. Nur ganz am Rande wird auf Signale (»signals«) und Interrupt-Service-Routines eingegangen.

Es wird hier davon ausgegangen, dass Data-Races wie in verwandten Sprachen »undefined behavior« haben – ein Programm, das eine Data-Race-Situation erreichen kann, sich dann auf beliebige Weise verhalten darf. Dieses beliebige Verhalten schließt auch ein, sich zunächst scheinbar richtig zu verhalten und später plötzlich und völlig unerwartet fehlzuschlagen.

In der Praxis darf Software ein solches Fehlverhalten nicht aufweisen – auch nicht in ungewöhnlichen oder selten auftretenden Situationen oder als Reaktion auf ungewöhnliche oder seltene Benutzereingaben. Es ist somit eine (möglicherweise auch ungenannte, undokumentierte beziehungsweise implizite) Anforderung an jede Software, frei von solchem Verhalten zu sein.

Zumindest gegen diese Anforderung verstößt ein Programm, das ein Data-Race enthält, da gemäß der Definition die zugrunde liegende Data-Race-Situation immer erreichbar ist. Somit ist eine Data-Race-behaftete Software eine fehlerhafte Software. Diese Abhandlung sieht jedenfalls in jedem Data-Race einen Softwarefehler. Sie sieht Data-Races also als eine Fehlerklasse an.

Es ist jedoch denkbar, dass derart fehlerhafte Software nicht mangelhaft ist. Tritt die Data-Race-Situation nur nach Benutzereingaben auf, die beim Gebrauch tatsächlich nicht getätigt werden, ist die Software zumindest nicht wegen des Data-Races mangelhaft.

Die umfangreiche Forschung an der Erkennung von Data-Races und das dem Autor bekannte Interesse der industriellen Anwendung daran sind ein

Hinweis darauf, dass sich zumindest in sicherheitskritischen eingebetteten Systemen Data-Races oftmals auch als Softwaremängel erweisen.

Kapitel 3

Methoden und Werkzeuge zur Verifikation

Data-Races sind Softwarefehler oder gar Softwaremängel. Es stellt sich daher die Frage, welche Methoden und Werkzeuge zu ihrer Erkennung und Mitigation eingesetzt werden können. Solche Ansätze zur Prüfung eines Programmes beziehungsweise einer Software auf Data-Races sind Softwareverifikationsansätze. Dieses Kapitel motiviert den Einsatz der statischen Software-Analyse zur Erkennung von Data-Races und beschreibt alternative Ansätze.

3.1 Klassifizierung des Data-Race-Problems

Die Frage, ob ein gegebenes Programm ein Data-Race enthält, ist (im Sinne der Berechenbarkeitstheorie) unentscheidbar. Dies leitet sich bereits aus dem Satz von Rice, siehe Rice [Ric53], her, der – informell formuliert – besagt, dass die »wesentlichen« Eigenschaften von Programmen unentscheidbar sind.

```
1  int c;                  9  void task_b() {
2  // ...                 10     if (f(x)) {
3                         11        c = 7;
4  void task_a() {        12     }
5     int local = c;      13     // ...
6     // ...              14  }
7  }
```

Abbildung 3.1: Ein nebenläufiges Programm

Im Folgenden wird das Beispiel in Abbildung 3.1 untersucht. Es handelt sich um eine Abwandlung des einfachen Beispiels aus Abbildung 2.1. Hier ist der schreibende Zugriff auf die geteilte Variable `c` durch eine Bedingung geschützt.

Gemäß der in dieser Abhandlung verwendeten Definition eines Data-Race handelt es sich bei dem Paar aus lesendem und schreibendem Zugriff auf die Variable `c` nur dann um ein Data-Race, wenn es einen Trace gibt, bei dem die Funktion `f` den Wert `true`[1] zurückgibt. Wenn `f` tatsächlich nur den Wert `false` zurückgeben oder nicht aus dem Aufruf zurückkehren kann, handelt es sich bei dem Paar nicht um ein Data-Race.

Das Beispiel kann auch dazu dienen zu zeigen, dass das Halteproblem auf das Data-Race-Problem Turing-reduzierbar ist. Da das Halteproblem unentscheidbar ist, ist auch das Data-Race-Problem unentscheidbar. Das Komplement des Data-Race-Problems, das heißt das Problem, Data-Race-freie Programme zu erkennen[2], ist nicht semi-entscheidbar, da das Komplement des Halteproblems darauf reduzierbar ist.

Wenn gezeigt werden kann, dass das Data-Race-Problem semi-entscheidbar ist, wäre auch sein Komplement präzise klassifiziert. Diesen Beweis zu führen erscheint möglich, liegt aber nicht im Fokus dieser Arbeit.

[1] Untechnisch wird hier von einem booleschen Typ ausgegangen, der nur die Werte `true` und `false` kennt. In C99 würden hier tatsächlich `int`s verwendet, die `0` oder ungleich `0` sind.

[2] Es handelt sich nicht im strengen Sinne um das Komplement, da das Komplement auch die nicht-gültigen C-Programme enthält. Das Wortproblem der C-Programme ist jedoch entscheidbar, sodass diese Ungenauigkeit ohne Auswirkungen bleibt.

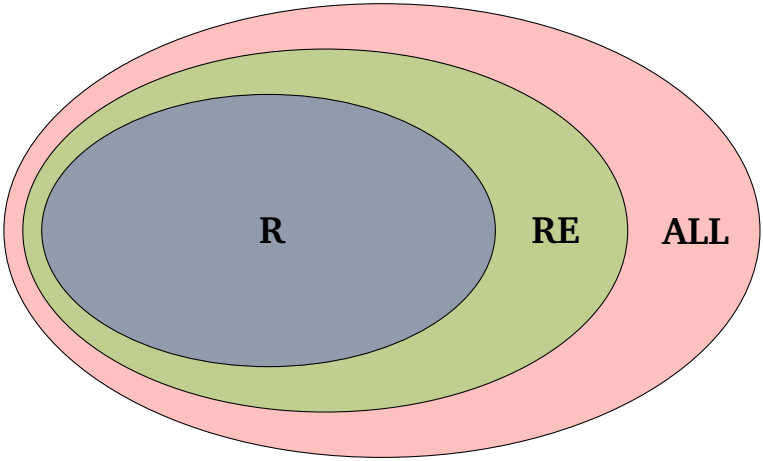


Abbildung 3.2: Die Berechenbarkeitsklassen R: rekursiv (entscheidbar), RE: rekursiv aufzählbar (semi-entscheidbar) und ALL: alle Entscheidungsprobleme

Landi [Lan92] hat sich intensiv mit der Klassifizierung von Datenflussanalysen beschäftigt. Die hier dargestellte Argumentation ist an seine angelehnt. Die Unentscheidbarkeit des Data-Race-Problems wurde bereits von Keul [Keu11] angemerkt.

Dieses theoretische Ergebnis erlangt in der Praxis Relevanz. Die Unentscheidbarkeit erklärt, warum Verfahren fehlerhafte Befunde liefern: Kein Verfahren kann stets terminieren und dennoch falsch-positive und falsch-negative Befunde ausschließen. Die folgenden Beschreibungen der Verfahren erläutern daher stets, in welche Richtung oder Richtungen die Verfahren approximieren.

3.2 Software-Test

Da Data-Races Softwarefehler sind, lassen sie sich auch prinzipiell durch Software-Test – seien dies Unit-Tests, Integrationstests oder Systemtests – finden.

Ob sich ein Data-Race zur Laufzeit tatsächlich manifestiert, hängt unter anderem vom Timing, genauer von den Interleavings der beteiligten Tasks, ab. Programmiersprachen definieren dieses Timing-Verhalten in der Regel nicht. Es ist indeterministisch und hängt in der Praxis von physischen Gegebenheiten, parallel laufenden und zuvor ausgeführten Programmen und Eingabedaten ab, da all diese das Scheduling beeinflussen können.

Dieser Indeterminismus bedingt, dass Data-Races sporadisch auftreten. Ob sich ein Problem also bei einer spezifischen Ausführung zeigt und damit möglicherweise erkennen lässt, hängt vom Zufall ab.

Data-Races, die häufig, das heißt mit hoher Wahrscheinlichkeit, auftreten, sind möglicherweise leicht zu erkennen. Data-Races, bei denen dies nicht der Fall ist, deren Effekte somit nur selten beobachtet werden können, sind in der Praxis problematischer. Hinzu kommt, dass kleine, scheinbar völlig unabhängige Änderungen am Quellcode oder Änderungen am Compiler beziehungsweise an Compilerschaltern oder am Linker die Auftrittswahrscheinlichkeit verändern und auch beträchtlich steigern können. (Vergleiche Boehm [Boe11].) Tatsächlich können Compiler Data-Races auch komplett verstecken (die Auftrittswahrscheinlichkeit ist dann 0), indem sie Freiheiten ausnutzen, die sie zum Beispiel bei der Codegenerierung haben.

Dies bedeutet, dass Data-Races nicht zuverlässig durch Tests gefunden werden können. Es ist schlicht nicht garantiert oder auch nur sehr wahrscheinlich, dass zum Beispiel ein Unit-Test, der eigentlich einen durch ein Data-Race verursachten Fehler erkennen könnte, bei einer bestimmten Ausführung auch tatsächlich fehlschlägt.

Tests sind in der Regel nicht fehlerklassenspezifisch. Wenn ein Test fehlschlägt, weist dies zwar auf einen Software-Fehler hin, es steht aber typischerweise nicht fest, ob dieser im Zusammenhang mit einem Data-Race, einer allgemeinen Race-Condition oder einem ganz anderem Defekt steht.

3.3 Dynamische Data-Race-Detektoren

Eine Verbesserung im Vergleich zu einem rein Test-basierten Vorgehen sind dynamische Data-Race-Detektoren. Mit Eraser stellen Savage et al. [SBN+97] einen solchen dynamisch arbeitenden Detektor vor.

3.3.1 Das Werkzeug Eraser

Eraser instrumentiert eine ihm übergebene ausführbare Binärdatei so, dass das Werkzeug in der Lage ist, die Verwendung bestimmter Synchronisationsmechanismen zur Laufzeit zu erkennen. Zur Laufzeit können dann im Konflikt stehende Zugriffe erkannt werden, auch wenn diese nicht tatsächlich einen datenzerstörenden Effekt haben. Vereinfachend ausgedrückt, werden so zur Laufzeit nicht nur »echte« Data-Race-Situationen erkannt, sondern auch Beinahe-Data-Race-Situationen.

Aufgrund des dynamischen Ansatzes ist Eraser nicht konservativ – falsch-negative Ergebnisse können prinzipbedingt nicht ausgeschlossen werden. Aus der Tatsache, dass Eraser bei einem Programmlauf oder einer Vielzahl von Programmläufen keine Warnung ausgibt, kann nicht geschlossen werden, dass das Programm frei von Data-Races ist.

Eraser ignoriert auch Zugriffe auf den Hauptspeicher, die relativ zum Stack-Pointer erfolgen, da Savage et al. davon ausgehen, dass dies Stackzugriffe sind und auf dem Stack keine geteilten Variablen liegen.

Eraser gibt auch falsch-positive Warnungen aus. Laut den Autoren ist dies auf die in manchen Bibliotheken verbreitete Wiederverwendung von Hauptspeicher für unterschiedliche geteilte Variablen zurückzuführen. Zudem ist nicht sichergestellt, dass alle Synchronisationsmechanismen zuverlässig erkannt werden.

Prinzipbedingt können außerdem Data-Races, die vom Compiler zum Beispiel durch Optimierungen »versteckt« werden, nicht gefunden werden. Boehm [Boe11] beschäftigt sich ausführlich mit dem Einfluss, den Compiler auf das Auftreten von Data-Races haben.

3.3.2 Methode von O'Callahan und Choi

O'Callahan und Choi [OC03] stellen eine Methode vor, die den Ansatz von Savage et al. erweitert. Sie berücksichtigen zusätzlich eine durch statische Software-Analyse ermittelte Happens-Before-Relation. Sie verbessern dadurch die Laufzeit (den durch die Instrumentierung verursachten Overhead) und reduzieren sowohl die Zahl der falsch-negativen als auch der falsch-positiven Warnungen. Der immer noch dynamische Ansatz hat jedoch weiterhin Fehler in beiden Richtungen.

3.3.3 Race-Condition-Detector von Jannesari et al.

Jannesari et al. [JKS+13] haben einen weiteren dynamischen Ansatz zum Auffinden von Data-Races entwickelt. Ihr Fokus liegt dabei aber auf Race-Conditions, da diese zwar wesentlich zu den Problemen mit Nebenläufigkeit in der Praxis beitragen, aber bislang weniger intensiv wissenschaftlich untersucht wurden. Data-Races sind für Jannesari et al. somit Beifang. Der Ansatz erzeugt sowohl falsch-positive als auch falsch-negative Warnungen (siehe Jannesari et al. [JKS+13]).

In einer frühen Veröffentlichung (Jannesari und Tichy [JT10]) wird ein Algorithmus beschrieben, der ein Werkzeug mit Namen »Helgrind Plus« in die Lage versetzt, von Jannesari und Tichy als *»Ad-hoc synchronisation«* bezeichnete Muster zu erkennen.

Jannesari und Tichy ist aufgefallen, dass Software-Entwickler oftmals selbst gestrickte Muster einsetzen, um nebenläufige Tasks zu synchronisieren. Sie argumentieren, dass solche Muster oftmals aus Zustandsvariablen bestehen, die global sichtbar sind und auf denen einer oder mehrere Tasks ein *»spin-lock«* ausführen. Ein *»spin-lock«* ist für sie dabei eine Schleife, die in der Bedingung von einer Variablen abhängt, deren Wert sich im Schleifenrumpf nicht ändert. Sie gehen optimistisch davon aus, dass solche Zustandsvariablen der Synchronisation dienen und schließen Race-Condition-Warnungen sowohl auf der Zustandsvariablen als auch in von Zugriffen auf dieser Variable umgebenen Ausdrücken aus.

3.4 Statische Software-Analyse

Dynamische Ansätze finden nur Data-Races, die auf Ausführungspfaden liegen, auf die sie bei der Abarbeitung spezifischer Testfälle beziehungsweise Eingabedaten treffen. Dies ist Stärke und Schwäche zugleich. Engler und Ashcraft [EA03] merken berechtigterweise an, dass bei der Abarbeitung realistischer Testfälle durch dynamische Werkzeuge gefundene Data-Races oft eine hohe praktische Relevanz haben, wohingegen es der statischen Software-Analyse leicht fällt, auch Fehler in obskuren Programmpfaden zu finden, die sich durch Tests nur schwer erreichen lassen.

3.4.1 Das Werkzeug RacerX

Engler und Ashcraft [EA03] stellen mit RacerX eine vollständig statische Data-Race-Analyse vor. Der Hauptbeitrag ihrer Publikation ist eine statische Lockset-Analyse, welche erlaubt, Synchronisationsmechanismen zu berücksichtigen. Keul [Keu11], der die vorgestellte Lockset-Analyse adaptiert, zeigt, dass dieses Analysekonzept nicht auf Locks als Synchronisationsmechanismus beschränkt ist, sondern auf weitere Mechanismen (zum Beispiel die später erläuterten Interrupt-Disable/Enable-Synchronisationsmuster) erweitert werden kann.

Mit den Ergebnissen der Lockset-Analyse warnt RacerX sowohl vor möglichen Deadlocks als auch vor Data-Races. Mit dem Werkzeug konnten Fehler in großen Desktop-Software-Projekten gefunden werden.

3.4.2 Ansatz von Vaziri, Tip und Dolby

Vaziri, Tip und Dolby [VTD06] stellen einen Ansatz zum Aufspüren von Fehlern im Zusammenhang mit Nebenläufigkeit vor. Sie implementieren ihren Ansatz prototypisch, um ihn auf Programme anzuwenden, die in der Sprache Java geschrieben sind. Sie definieren eine Menge problematischer Interleavings von Variablenzugriffen. Diese stellt eine Obermenge der Data-Races dar.

Mithilfe statischer Software-Analyse und Benutzerannotationen erkennen Vaziri, Tip und Dolby Programmstellen, an denen Synchronisation hinzugefügt werden muss, um die problematischen Interleavings auszuschließen.

Dieser Ansatz kann als Variante der Data-Race-Analyse gesehen werden. Der Vorschlag, an bestimmten Stellen Synchronisationsmechanismen zu verwenden, um Data-Races auszuschließen, kann auch als Warnung vor den andernfalls an dieser Stelle möglicherweise auftretenden Data-Races gesehen werden.

Am Rande gehen Vaziri, Tip und Dolby darauf ein, dass ihr Ansatz unter nicht näher erläuterten Umständen hinreichend (»sufficient«) ist, korrekte Synchronisation zu garantieren. Damit schlagen sie vor, den nicht ausschließbaren Fehler der statischen Software-Analyse in konservativer Richtung zu beschränken – also zwar falsch-positive, nicht jedoch falsch-negative Befunde zu erlauben.

3.5 Der Bauhaus Data Race Detector

Das Institut für Softwaretechnologie der Universität Stuttgart, an dem der Autor arbeitet, entwickelte mit Bauhaus eine Sammlung von Werkzeugen zur statischen Software-Analyse. Zunächst auf Strukturanalyse, Architekturvalidierung und Reverse Engineering spezialisiert, kamen im Laufe der Entwicklung weitere Schwerpunkte wie Clone-Detection und Analysen zum Auffinden und Ausschließen von Fehlern hinzu. (Vergleiche Czeranski et al. [CEK+00].)

3.5.1 Konzeption

Raza, Vogel und Plödereder [RVP06] beschreiben die Konzeption der in Bauhaus enthaltenen Werkzeuge zur Analyse von nebenläufigen Programmen. Sie fordern dabei von statischen Software-Analysen zur Nebenläufigkeit, insbesondere einer Data-Race-Analyse, Konservativität ein. Das heißt, der Fehler soll sich auf falsch-positive Befunde beschränken. Nur so können die

Werkzeuge dazu verwendet werden, die Fehlerklassen tatsächlich auszuschließen.

> *[Both race conditions and deadlocks] tend to be very difficult to detect or to recreate by test runs; they arise in real-life execution as an inexplicable, sudden, and not recreatable, sometimes disastrous malfunction of the system. For reliable systems it is literally a »must« to impose coding restrictions and to perform a static analysis of the code to ensure the absence of race conditions and deadlocks.*
>
> — *Raza, Vogel und Plödereder [RVP06]*

Ludewig und Lichter [LL07] begründen, warum falsch-negative Resultate in der Praxis des Software-Engineerings besonders problematisch sind:

> *Falsch negative Resultate sind schlimmer als keine Resultate: Sie vermitteln unbegründetes Vertrauen in ein Produkt und erschweren die Fehlersuche, wenn irgendwann auffällt, dass es einen Fehler gibt. Die Fehlerursache wird dann besonders leicht übersehen, weil jeder denkt: »Das haben wir ja schon geprüft!«*
>
> — *Ludewig und Lichter [LL07]*

Die Data-Race-Analyse setzt einige klassische Datenflussanalysen voraus. Einige Ansätze zur Data-Race-Analyse haben Fehler in beiden Richtungen (sowohl falsch-positive als auch falsch-negative Befunde), weil die Analysen, auf denen die Data-Race-Analyse aufbaut, nicht vollständig auf die für Konservativität benötigten Eigenschaften abgestimmt sind.

Eine Schwierigkeit bei der konservativen Auslegung der Basisanalysen ist die Berücksichtigung von Nebenläufigkeit. In Bauhaus sind alle für die

Data-Race-Analyse nötigen Datenflussanalysen so ausgelegt, dass sie auch für nebenläufige Programme die Konservativität nicht verletzen. Für bestimmte Fehlersituationen mit undefiniertem Verhalten werden, wie in dem Bereich üblich, Ausnahmen gemacht. So werden zum Beispiel Ausnahmen von der Konservativität an Stellen gemacht, wo konservatives Verhalten ausschließlich pedantisch und praktisch unsinnig wäre.[1]

3.5.2 Aufbau

Der Bauhaus Data Race Detector setzt sich aus mehreren Werkzeugen der Bauhaus-Werkzeugsammlung zusammen. Nachdem durch das Frontend eine einem Abstrakter Syntaxbaum ähnelnde Zwischendarstellung erzeugt wurde und grundlegende Analysen, wie die Kontrollflussanalyse zur Erstellung von Basic-Blocks, durchgeführt wurden, beginnt der nebenläufigkeitsspezifische Teil der Analysekette. Dazu wird eine feldsensitive, fluss- und kontextinsensitive Zeigeranalyse durchgeführt. Diese ermittelt auch die im weiteren Verlauf wichtigen Points-To-Objects. (Vergleiche Keul [Keu10].)

Die Zeigeranalyse erlaubt es, den interprozeduralen Aufrufgraphen zu erstellen, da durch ihre Ergebnisse indirekte Prozeduraufrufe aufgelöst werden können. Demnach können verschiedene interprozedurale Datenflussanalysen ausgeführt werden. Dabei handelt es sich um eine Reihe von ergebnisverfeinernden Analysen, wie die Escape-Analyse. Die eigentliche Data-Race-Analyse traversiert zum Schluss den gesamten Aufrufgraphen erneut und gibt die Data-Race-Warnung aus. (Vergleiche Keul [Keu10].)

Von der Analyse existieren mehrere Varianten. Zum Teil handelt es sich um gänzlich unterschiedlichen Quelltext und zum Teil können durch Konfiguration experimentelle Features ein- und ausgeschaltet werden. Einige der zum Beispiel in Keul [Keu10] beschriebenen erweiterten Features sind nur

[1] Dieser gedankliche Ansatz wird von Livshits et al. [LSS+15] treffend beschrieben. Livshits et al. bezeichnen das Konzept als *»soundiness«* und Analysen, die es umsetzen, als *»soundy«*. Sie schreiben: *»The concept of soundiness attempts to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately underapproximated handling of a feature subset well recognized by experts. [...] A soundy analysis aims to be as sound as possible without excessively compromising precision and/or scalability«* und stellen fest, dass praktisch alle Werkzeuge, die auf reale Systeme angewendet werden, höchstens soundy sind.

in den experimentellen Varianten vorhanden. Grundlage für die in dieser Abhandlung besprochene Forschung bildet der stabile Kern der Analyse.

3.6 Model-Checking und generelle Software-Verifikation

Es gibt Ansätze, die Model-Checking und andere generelle Software-Verifikationsmechanismen einsetzen, um Data-Races zu finden oder deren Abwesenheit nachzuweisen. Beispielhaft werden hier einige Werkzeuge kurz beschrieben.

3.6.1 CBMC

Clarke, Kroening und Lerda [CKL04] stellen mit CBMC ein Werkzeug zur formalen Verifikation von ANSI-C-Programmen vor. Die eingesetzte Methode (Bounded Model Checking) übersetzt ein gegebenes Programm in eine aussagenlogische Formel. Dabei werden Schleifen (und auf ähnliche Weise auch Rekursion) »abgewickelt« (unwound).

Diese Abwicklung ist in Abbildung 3.4 dargestellt. Eine Schleife wird durch einen oder mehrere ineinander verschachtelte Bedingungsblöcke ersetzt. Am Ende wird eine sogenannte »Unwinding-Assertion« eingefügt. Diese stellt später sicher, dass die Schleife oft genug abgewickelt wurde.

Prozeduraufrufe und Rekursion (sowohl direkte als auch indirekte) werden mittels Tail-Recursion-Elimination in Schleifen umgewandelt oder durch (wiederholtes) Inlining und Unwinding-Assertions ersetzt.

```
1  I := 0;
2
3  while (I < 4) loop
4     Pl ("I: ", I);
5     I := I + 1;
6  end loop;
```

Abbildung 3.3: Ausschnitt eines Programms mit einer Schleife

Das durch Abwicklung entstandene »Spaghetti-Code-Programm« wird in SSA-Form (Static Single Assigment) gebracht und in eine aussagenlogische Formel übersetzt. Dabei werden auch zusätzliche Assertion-Statements eingefügt, die das Programm auf die gewünschten Fehlerzustände überprüfen (beispielsweise Array-Bounds-Checks). Ebenso wird die Erreichbarkeit von fehlschlagenden Assertion-Statements als Formel kodiert.

Die Konjunktion dieser beiden Formeln ist genau dann erfüllbar, wenn es einen Pfad zu mindestens einem fehlschlagenden Assertion-Statement gibt. Ein vom Anwender ausgewählter Sat-Solver kann somit methodisch danach suchen. Wird eine erfüllende Belegung gefunden, kann daraus ein Pfad zu einem fehlschlagenden Assertion-Statement rekonstruiert werden.

Handelt es sich dabei um eine Unwinding-Assertion, wird der Prozess mit einer größeren Abwicklungstiefe wiederholt. Wenn es sich um ein anderes Assertion-Statement handelt, wurde ein Pfad zu einem Fehlerzustand und damit ein Programmfehler gefunden. Kann hingegen kein fehlschlagendes Assertion-Statement erreicht werden, können die Fehlerklassen, für die Assertion-Statements eingebaut wurden, ausgeschlossen werden.

In realen Programmen ist die Zahl der Schleifendurchläufe oft nicht leicht zu begrenzen. Die Grenze kann beispielsweise erst bei »`Integer'Last`« liegen oder schlicht nicht vorhanden sein.

Um die Laufzeit (selbst bei wenige Zeilen großen Testprogrammen) in Grenzen zu halten, werden teilweise unrealistisch kleine Zahlenbereiche für Integer verwendet. (Beispielsweise werden für die C-Typen `short`, `int` und `long`, je nach Konfiguration, Größen von 8, 10 und 12 Bit verwendet.)

3.6.1.1 Bounded Model Checking für nebenläufige Programme

Bounded Model Checking lässt sich auch bei nebenläufigen Programmen anwenden. Rabinovitz und Grumberg [RG05] beschreiben, wie sie CBMC auf nebenläufige Programme erweitern.

Dabei werden Kontextwechsel explizit modelliert. Ihre Zahl ist auf ähnliche Weise wie die der Schleifendurchläufe nach oben begrenzt. Die Maximalzahl an Kontextwechseln wird jedoch vom Anwender vorgegeben, sodass der

```
1  I := 0;
2
3  --  Unwinding #1
4  if (I < 4) then
5     Pl ("I: ", I);
6     I := I + 1;
7
8     --  Potentially more Unwindings  -- or --
9     --  Unwinding Assertion
10    pragma Assert (not (I < 4));
11 end if;
```

Abbildung 3.4: »Abwicklung« (Unwinding) des Programmausschnittes aus Abbildung 3.3

Ansatz nicht mehr streng konservativ ist. In ihren Beispielen beschränken Rabinovitz und Grumberg die Zahl der Unterbrechungen auf 10.

In der von Rabinovitz und Grumberg implementierten Version ist die Zahl der parallelen Tasks auf zwei begrenzt. Sie zeigen aber auf, wie ihr Ansatz auf mehr als zwei Tasks erweitert werden kann. »Ununterbrechbare Abschnitte« (Atomic Sections) und Mutexes müssen explizit modelliert werden.

In Abschnitt 5.1 definieren Rabinovitz und Grumberg den Begriff »Race Condition« so, dass er im Wesentlichen der hier verwendeten Definition eines Data-Race entspricht, und erklären somit, wie mit ihrem Ansatz Data-Races gefunden werden können.

In der vom Autor dieser Abhandlung untersuchten Version 4.9 des CBMC (zum Zeitpunkt des Experiments die neueste verfügbare Version) liefert der Ansatz für sehr kleine sequentielle Programmausschnitte bemerkenswert schnell präzise Ergebnisse. Diese Version und auch später veröffentlichte neuere Versionen enthalten aber keinen Support für nebenläufige Programme.

Schon bei kleinen Testbeispielen für State-Machines in rein sequentiellen Programmen (ca. 25 Zeilen Quelltext) gerät der Ansatz in eine scheinbare Endlosschleife, in der immer wieder der Unwinding-Grenzwert erhöht wird. Dies liegt wohl daran, dass die Tests stets Schleifen oder Rekursion mit

nicht trivial zu findenden Grenzwerten hatten. Eine Anwendung auf Systeme industrieller Größe scheint (auch aufgrund der Mengengerüste) mit der untersuchten Version völlig ausgeschlossen zu sein.

3.6.2 Das Werkzeug Threader

Gupta, Popeea und Rybalchenko [GPR11] stellen das Verifikationswerkzeug Threader vor. Es arbeitet mit nebenläufigen C-Programmen. Das Werkzeug verwendet Model-Checking-Techniken, um gegenseitigen Ausschluss von Coderegionen zu ermitteln. Das Vorgehen kommt ohne Beschränkung der maximal möglichen gegenseitigen Unterbrechungen aus. Gupta, Popeea und Rybalchenko wenden ihr Werkzeug auf *»kleine, aber verworrene«*[1] C-Programme an. Das größte Programm, bei dem sie mit einem ihrer Regelsätze ein Ergebnis erhalten, ist 451 Zeilen lang.

3.6.3 Das Werkzeug Memics

Nowotka und Traub [NT13] sowie Traub [Tra16] stellen ein Werkzeug namens Memics vor. Es verbindet Techniken der abstrakten Interpretation mit dem Einsatz von SMT-Solvern und hat zum Ziel, eingebettete Systeme aus Automobilen untersuchen zu können. Es kann dabei auch Fehler auffinden, die im Zusammenhang mit Nebenläufigkeit entstehen. Die Fehlerklassen, die es dabei abdeckt, sind: Dead-Locks, Double-Locks, Lost Updates und Missing Synchronization (siehe Traub [Tra16], Seite 11). Es produziert dabei sowohl falsch-positive als auch falsch-negative Warnungen. Nowotka und Traub [NT13] schreiben, dass die eingesetzten Techniken *»eine sehr hohe Komplexität haben, was bedeutet, dass die Code-Fragmente, die Memics analysieren kann, nicht allzu groß sind.«*[2] Traub zeigt, dass das Werkzeug Memics in der Lage ist, in einem 62 Zeilen langen Ausschnitt eines eingebetteten Systems in circa 30 Minuten ein tatsächliches Data-Race zu finden, wenn die Größe einiger verwendeteter Arrays auf 500 reduziert wird.

[1] Aus dem Englischen: *»small but intricate«*.

[2] Aus dem Englischen: *»[…] [The] complexity of constraint solving algorithms is very high which means that the code fragments Memics can analyse are not too large.«*

3.6.4 Das Werkzeug Java Race Finder

Mit dem Werkzeug Java Race Finder präsentieren Kim, Yavuz-Kahveci und Sanders [KYS09] eine Erweiterung der Werkzeugsammlung Java Path Finder. Das Werkzeug findet mittels Model-Checking typische durch Nebenläufigkeit bedingte Fehler wie Data-Races und Deadlocks in Java-Bytecode.

Das Werkzeug Java Race Finder ist in der Lage, Lock-freie Algorithmen zu analysieren. Die Autoren empfehlen ihr Werkzeug für die Analyse solcher Algorithmen, da sie klein genug sind, um mittels Model-Checking bearbeitet zu werden.

> *Model checking works very well for [lock-free algorithms]. They are small enough to be feasibly model checked, yet they are very intricate.*
> — *Kim, Yavuz-Kahveci und Sanders [KYS09]*

3.6.5 Bestandsaufnahme von D'Silva, Kroening und Weissenbacher

In einer ausführlichen Bestandsaufnahme vergleichen D'Silva, Kroening und Weissenbacher [DKW08] eine ganze Reihe von Ansätzen und Werkzeugen zur automatisierten formalen Verifikation von Software: Während statische Analyse teils nur einfache Probleme lösen kann, aber dabei sehr effizient vorgeht, ist die Hauptherausforderung, der sich Model-Checking gegenübersieht, der aus der »state-space explosion« entstehende exponentielle Ressourcenbedarf.

> *Most [static analysis/abstract interpretation tools] can be used to analyze large software systems with minimal user interaction. Such tools are extremely robust, meaning that they can cope with large and varied inputs, and are efficient. Conversely, the properties that can be proved are often simple and are usually hard*

coded into the tools, for example, ensuring that array bounds are not exceeded or that arithmetic overflows do not occur [...].

[...] The principal issue in model checking is state-space explosion: the state-space of a software program is exponential in various parameters such as the number of variables, and width of datatypes. [...] Concurrency exacerbates the problem [...].

— D'Silva, Kroening und Weissenbacher

3.6.6 CSP-basierte Vorgehensweisen

Es existieren Vorgehensweisen zur Softwareverifikation, die auf CSP (Communicating Sequential Processes) basieren. Eine ausführliche Einführung die formale Sprache beziehungsweise den Prozesskalkül findet sich in Kapitel 7 (Seite 95).

3.6.6.1 Roscoes Shared-Variable Programs

Roscoe [Ros11b] hat in einer Studie eine sehr einfache imperative Programmiersprache entwickelt, die Kontrollstrukturen, globale Variablen und Nebenläufigkeit bietet. Er hat für diese »Shared-Variable Programs« genannte Sprache einen Compiler[1] in CSP[2] geschrieben. Mit Hilfe des Werkzeuges lassen sich kleine Beispielprogramme (die typischerweise Lock-Free-Algorithmen enthalten) semantikerhaltend transformieren. Der Benutzer kann so Erkenntnisse über die Funktionsweise der Programme gewinnen.

[1] Roscoe verwendet den Begriff »Compiler«. Der Begriff »Interpreter« ist aber möglicherweise passender.

[2] Ja, das Werkzeug ist tatsächlich in CSP geschrieben. Roscoe schreibt dazu: *»CSP [...] seems a most unlikely language to write compilers in. At the time of writing it still has no type of strings, and nor does it have a write or other command to save the result of compilation into a file. It certainly was not designed with this style of application in mind.«*

3.6.6.2 CASPER

Mit CASPER (Compiler for the Analysis of Security Protocols) stellen Lowe et al. [LBDL09] ein Werkzeug zur Analyse von Sicherheitssystemen (Computer Security Systems, zum Beispiel Schlüsselaustauschverfahren) vor. Der Anwender kann in einer formalen Sprache Protokolle spezifizieren. Das Werkzeug übersetzt diese Spezifikationen dann in CSP_M-Modelle. Das erlaubt Eigenschaften der Protokolle maschinell nachzuweisen.

Kapitel 4

Besonderheiten eingebetteter Systeme

Diese Abhandlung beschäftigt sich mit der Erkennung von Data-Races in eingebetteten Systemen. Entwickler solcher – möglicherweise sicherheitskritischer – eingebetteter Systeme haben ihre eigenen Konventionen. Entwicklungsprinzipien und Techniken unterscheiden sich deutlich von denen, die bei der Programmierung von Desktop-Computer-Software eingesetzt werden. Dieses Kapitel beschreibt einige wesentliche Besonderheiten der untersuchten Systeme.

4.1 Misra-C-Richtlinien

Das Misra-Konsortium (The Motor Industry Software Reliability Association) ist ein Zusammenschluss von Herstellern, Zulieferern und technischen Beratern der Automobilindustrie. Das Konsortium stellt Richtlinien zur Entwicklung sicherer (*»safe and secure«*) eingebetteter Systeme zur Verfügung. (Siehe Webseite der Misra [Mir].)

Darunter sind die in der Automobilindustrie verbreitet Anwendung findenden »Misra-C Guidelines for the use of the C language in critical systems« [MIR08], die hier als Misra-C-Richtlinien bezeichnet werden.

Kern der Misra-C-Richtlinien sind zahlreiche Regeln, die zusammen eine Teilmenge der Sprache C definieren, welche sich besser als die eigentliche Sprache C zur Entwicklung eingebetteter Systeme (insbesondere für Automobile) eignen soll.

Die Misra-C-Richtlinien sind für die in der vorliegenden Abhandlung untersuchten Systeme keine universell geltenden Gesetze – die Richtlinien selbst raten dazu, in bestimmten Fällen von den Regeln abzuweichen, und beschäftigen sich mit der Dokumentation solcher Abweichungen (»*deviations*«). Die Richtlinien dokumentieren aber in der Automobilindustrie verbreitete Denkmuster. Wird in dieser Abhandlung auf die Misra-C-Richtlinien verwiesen, so dient der Verweis in der Regel dazu, eine bestimmte Praxis zu belegen.

4.2 Tasks und Scheduling-Prioritäten

In den untersuchten Systemen lassen sich nebenläufig ausführbare Einheiten unterschiedlichen Kategorien zuweisen. Zum Teil überlappen diese Kategorien auch konzeptionell. Dabei existiert in der Praxis keine einheitliche Nomenklatur für die Bezeichnung dieser Einheiten. Diese sind:

- **Hardwareinterrupts** Ausgelöst durch Timer oder physische Ereignisse springt der Kontrollfluss bei Hardwareinterrupts in sogenannte »Interrupt-Service-Routines«. Diese führen teils nur wenige Instruktionen aus und vermerken zum Beispiel in einer globalen Variablen, dass der entsprechende Hardwareinterrupt ausgelöst wurde.
- **Softwareinterrupts** Als Reaktion darauf, dass die Hardwareinterrupts ausgelöst wurden, aktiviert der Scheduler möglicherweise Tasks, die benötigte Aktionen durchführen. Diese werden dann (teils) als Softwareinterrupts bezeichnet. Es gibt prinzipiell auch die Möglichkeit, die Aktivierung von Softwareinterrupts vom Systemzustand abhängig zu machen. Softwareinterrupts werden teils auch als azyklischen Tasks

bezeichnet. Bei Softwareinterrupts prüft der Scheduler periodisch, ob sie ausgelöst werden sollen, (das heißt, in den Zustand bereit versetzt werden) und scheduled den dazugehörigen Code dann gegebenenfalls wie zyklische Tasks auch.

- **Zyklische Tasks** Wesentliche Teile der Arbeit der untersuchten Systeme werden in zyklischen Tasks abgeleistet. Diese sind oftmals nach ihrer Periode benannt. Der Scheduler stellt dabei sicher, dass sie in geeigneter Frequenz aufgerufen werden.

In dieser Abhandlung werden alle diese Kategorien ausführbarer Einheiten als Tasks bezeichnet. Dies steht im Einklang mit der Implementierung des Bauhaus Data Race Detectors.

4.2.1 Fixed-Priority-Scheduling

Die untersuchten eingebetteten Systeme verwenden Varianten des Fixed-Priority-Scheduling. Dabei werden Tasks (und somit auch Software- und Hardwareinterrupts) statische, das heißt zur Übersetzungszeit bekannte, Prioritäten zugeordnet.

In dieser Abhandlung gilt die auch in der praktischen Anwendung oft verwendete Konvention, dass kleinere Zahlen höhere Priorität bedeuten. Hat ein Task A die Priorität 5 und ein Task B die Priorität 3, ist der Task B der höherpriore Task. Task A hat eine niedrigere Priorität.

Dabei merken Burns und Wellings [BW01] zu Recht an:

> *In real-time systems, the »priority« of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity.*
>
> — *Burns und Wellings*

Bei periodischen Tasks wird teilweise die Priorität gleich der Periode gesetzt. Niedrigere Periode, also häufigere Ausführung, bedeutet damit auch höhere Priorität. Dieses Vorgehen bezeichnen Burns und Wellings

als »rate monotonic priority assignment« (vergleiche Burns und Wellings [BW01], Abschnitt »Fixed-Priority Scheduling and rate monotonic priority assignment«).

Die in den untersuchten Systemen verwendeten Scheduler basieren ausschließlich auf harter Unterbrechung (»preemption«). Dies steht im Gegensatz zu Verfahren, die kooperative Unterbrechungen (»yields«) erlauben beziehungsweise erwarten, und solchen, bei denen ein einmal gestarteter Task zu Ende läuft, bevor der nächste gestartet wird.

Die untersuchten Systeme laufen auf Ein-Kern-Prozessoren. Die Scheduler gewähren außerdem Tasks mit höherer Priorität strikten Vorrang. Bei jeder Scheduling-Entscheidung wird derjenige der verfügbaren Tasks ausgewählt, der die höchste Priorität hat. Es kann daher nie geschehen, dass ein Task unterbrochen wird, um einen anderen Task mit niedrigerer Priorität zum Zuge kommen zu lassen. Tasks können nur von Tasks höherer Priorität unterbrochen werden. Dadurch wird die mögliche Parallelität zwischen Tasks stark eingeschränkt. Je nach Konfiguration fasst der Bauhaus Data Race Detector mehrere Tasks gleicher Priorität zu einem einzigen Task mit mehreren möglichen Einsprungspunkten zusammen. Wir verwenden den Bauhaus Data Race Detector stets in dieser Konfiguration. Die für uns sichtbaren Tasks haben somit alle unterschiedliche Prioritäten: Es existieren keine zwei Tasks mit gleicher Priorität.

4.2.2 Interrupts

Die in der vorliegenden Abhandlung zugrunde gelegte Data-Race-Analyse sieht auch Software- und Hardwareinterrupts als Tasks an. Hardwareinterrupts haben höhere Priorität als andere Tasks im System, da sie diese sofort unterbrechen. Es kann jedenfalls kein zyklischer Task eine höhere Priorität als ein Hardwareinterrupt haben. Durch Garantien der Hardware ist unter Umständen sichergestellt, dass sich Interrupt-Service-Routines nicht gegenseitig unterbrechen können. Die Softwareinterrupts können auch eine Priorität haben, die niedriger als die mancher zyklischer Tasks ist.

4.3 Globale Interrupt-Deaktivierung

Wird ein Prozessor eines eingebetteten Systems gestartet oder zurückgesetzt (reset), befindet er sich in einem definierten Startzustand. Watchdogs sind inaktiv, er befindet sich im niedrigsten Prozessor-Level und Interrupts sind abgeschaltet.

Um überhaupt auf Interrupts reagieren zu können, müssen diese eingeschaltet werden. Bei der Hardware der für diese Abhandlung untersuchten Systeme funktioniert das derart, dass zunächst in der Interrupt-Maske, die stets an einer bestimmten Adresse zu finden ist, Klassen von Interrupts freigeschaltet werden müssen und dann in der Prozessor-Status-Maske ein weiteres Flag gesetzt werden muss, um Interrupts zu aktivieren. (Werden auf diese Weise auch Timer-gesteuerte Interrupts eingeschaltet, sollten diese natürlich zuvor passend konfiguriert worden sein.)

Dieser technische Umstand lässt sich nutzen, um nebenläufige Software zu synchronisieren. Um einen kritischen Abschnitt zu schützen, werden global alle Interrupts deaktiviert, bevor in ihn eingetreten wird, und erst wieder aktiviert, nachdem er abgeschlossen ist. Dieses Vorgehen wird als Interrupt-Enable/Disable-Muster bezeichnet. Es ist ein in den untersuchten Systemen häufig verwendetes Synchronisationsmuster. Die Interrupt-Deaktivierung verhindert, durch das Deaktivieren aller Timer-Interrupts, auch, dass der Scheduler Gelegenheit bekommt, einen laufenden Task zu unterbrechen.

Aufgrund der hardwarenahen Implementierung des Interrupt-Disable/Enable-Musters dürfen zwei dieser Muster nicht ineinander verschachtelt werden. Ebenso wenig dürfen die geschützten Abschnitte überlappen.[1] Abbildung 4.1 zeigt eine solche fehlerhafte Anwendung des Musters. Unabhängig davon, ob die dargestellten Interrupt-Abschaltungen als verschachtelt (Zeile 2 gehört zu Zeile 8) oder als überlappend (Zeile 2 gehört zu Zeile 6) intendiert sind, erreichen sie wohl nicht die erwartete Synchronisierung – im Anweisungsblock C sind Interrupts eingeschaltet.

[1] Es existiert die Idee des »Counting Interrupt Disable/Enable«. Dabei darf der Entwickler kritische Abschnitte sowohl verschachteln als auch überlappen lassen. Keines der untersuchten Systeme verwendet aber eine solche Konfiguration.

```
1  void p() {
2     disable_interrupts();
3     // ...  (A)
4     disable_interrupts();
5     // ...  (B)
6     enable_interrupts();
7     // ...  (C)
8     enable_interrupts();
9  }
```

Abbildung 4.1: Fehlerhafte Anwendung des Interrupt-Disable/Enable-Synchronisationsmusters

```
1  int d = 0;
2
3  void task_low() {
4     disable_interrupts();
5     // reset d;
6     d = 0;
7     enable_interrupts();
8  }
```

```
10 void task_high() {
11    // read from d
12    unsigned int l = d;
13    // compute update
14    l = f(l);
15    // write to d
16    d = l;
17 }
```

Abbildung 4.2: Einseitige, aber korrekte und typische Anwendung des Interrupt-Disable/Enable-Synchronisationsmusters

4.3.1 Zusammenspiel mit Prioritäten

Eine Besonderheit des Interrupt-Disable/Enable-Synchronisationsmusters zeigt sich, wenn Prioritäten ins Spiel kommen. Das Beispiel aus Abbildung 4.2 verdeutlicht dies.

In diesem Beispiel sind zwei Tasks dargestellt, die über die Variable `d` kommunizieren. Während der niederpriore Task (`task_low`) seinen Zugriff auf die Variable `d` durch das Abschalten von Interrupts schützt, greift der höherpriore Task (`task_high`) scheinbar ohne Schutz auf die Variable `d` zu.

Dies ist jedoch kein Implementierungsfehler; das Beispiel enthält auch kein Data-Race. Der höherpriore Task `task_high` kann schlicht nicht un-

terbrochen werden, um den Task `task_low` zum Zug kommen zu lassen. Das Scheduling-Verfahren lässt dies nicht zu.

Allgemein lässt sich sagen: Greifen mehrere Tasks auf eine geteilte Variable zu, kann der Task mit der höchsten Priorität auf Synchronisationsmechanismen verzichten, wenn die anderen Tasks konsequent Interrupts[1] abschalten.

Die untersuchten Systeme nutzen diese Eigenschaft des Scheduling-Verfahrens auch konsequent aus. Der höchstpriore Task, der auf einen Speicherbereich zugreift, schaltet Interrupts nicht ab.

4.4 Vermeidung von Heap-Allokationen und Rekursion

Die hier untersuchten sicherheitskritischen eingebetteten Systeme vermeiden bestimmte in Desktop-Software verbreitet genutzte Programmiersprachenfeatures wie Heap-Allokationen und Rekursion. Dies ist vor allem im Zusammenhang damit zu sehen, dass für die Systeme belegt oder gar nachgewiesen werden muss, dass sie mit den ihnen zur Verfügung stehenden begrenzten Ressourcen stets auskommen. Zu den begrenzten Ressourcen gehören unter anderem auch Rechenzeit und Hauptspeicher.

4.4.1 Heap-Allokationen

Wird in einem Programm dynamisch Speicher auf dem Heap allokiert, stellt sich sofort die Frage, wie und wann er wieder freigegeben werden kann. Eingebettete Systeme laufen in der Regel kontinuierlich, sodass die Antwort nicht »beim Beenden des Programms« lauten kann. Auch wenn es Ansätze zum Einsatz von Garbage-Collection in Echtzeitsystemen gibt, ist dieser kaum verbreitet. Ebenso wenig findet Reference-Counting oder die manuelle Freigabe vom Heap-Objekten Verwendung – auch um die Problemklassen wie Heap-Fragmentierung, Memory-Leaks oder Use-after-free gar nicht erst zu ermöglichen. (Vergleiche Burns und Wellings [BW01].)

[1] In den untersuchten Systemen ist dieses Interrupt-Abschalten stets total. Es werden alle Interrupts einschließlich der Timer-Interrupts, die den Scheduler aktivieren, abgeschaltet.

Selbst wenn eine Freigabe des Speichers nicht geplant ist, bleibt noch die Aufgabe, nachzuweisen, dass der für den Heap zur Verfügung stehende Speicher auch ausreicht – schließlich ist der Hauptspeicher jedes Systems nur endlich groß. Wenn bei einer Ausführung versucht wird, mehr Heap-Speicher zu allokieren, als im System verfügbar ist, resultiert dies im besten Fall in einem schwer zu behandelnden Fehlerzustand. Schlimmstenfalls wird gar anderweitig verwendeter Speicher überschrieben. Einen solchen Nachweis zu führen, ist im Allgemeinen schwierig – und dort, wo er leicht möglich ist, hätten in der Regel auch von vornherein der Stack oder globale Variablen verwendet werden können. (Vergleiche Burns und Wellings [BW01].)

Die hier untersuchten Systeme allokieren jedenfalls keinen Speicher auf dem Heap[1]. Nur eines enthält Aufrufe an die bekannte C-Funktion `malloc`. Diese stehen allerdings in totem Code und die Vermutung liegt nahe, dass sie nur zu Testzwecken verwendet wurden.

4.4.2 Datensegment

Globale Variablen, einschließlich der, deren Sichtbarkeit mit dem Schlüsselwort `static` auf die aktuelle Datei begrenzt ist, und lokale Variablen, deren Lebensdauer durch Deklaration mit dem Schlüsselwort `static` die ihres umgebenden Blockes übersteigt, sind Variablen mit statisch bekannter Lebenszeit. Mit diesen kann der Compiler wie in Abbildung 4.3 dargestellt umgehen.

Beim Starten des Systems wird der Datensegment-Speicherblock in den gleich großen Speicherblock für initialisierte Variablen kopiert. Im Datensegment hat der Compiler die Initialisierungswerte der globalen Variablen passend abgelegt. Der Speicherblock uninitialisierter Variablen wird mit Nullen beschrieben, da dies die Initialisierungsgarantien[2] der Sprache C erfüllt. Die konstanten globalen Variablen müssen nicht kopiert werden, da bei diesen ein Überschreiben ohnehin nicht vorgesehen ist.

[1] Siehe hierzu auch Regel 20.4 der Misra-C-Richtlinien [MIR08]: *»Dynamic heap memory allocation shall not be used.«*

[2] Siehe Sektionen 5.1.8 und 6.7.8 in [ISO05].

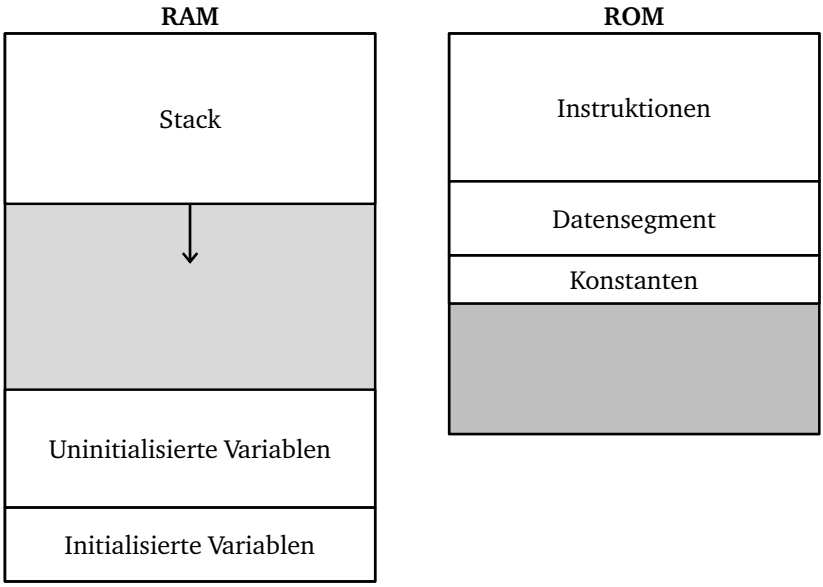


Abbildung 4.3: Vereinfachtes Hauptspeicherlayout für ein eingebettetes System

4.4.3 Rekursion

Einzig für lokale Variablen wird zur Laufzeit Speicher allokiert. Auch direkte und indirekte Rekursion werden weitgehend vermieden[1]. Anders als vereinfacht in Abbildung 4.3 dargestellt, benötigt in einem realen System jeder Task seinen eigenen Stack.

Die untersuchten eingebetteten Systeme werden alle einer Worst-Case-Execution-Time- und einer Worst-Case-Stack-Usage-Analyse unterzogen. Dies ist ein übliches Vorgehen. Damit diese Analyse gelingen kann, muss insbesondere jede Rekursion (und natürlich auch jede Schleife) begrenzt sein und diese Grenze muss durch das Werkzeug ermittelbar sein. (Vergleiche Burns und Wellings [BW01].)

[1] Siehe hierzu auch Regel 16.4 der Misra-C-Richtlinien [MIR08]: *»Functions shall not call themselves, either directly or indirectly.«*

Im Allgemeinen fällt es Analysewerkzeugen schwer, gerade für die Rekursionstiefe eine Grenze zu ermitteln. Dies kann Teil der Motivation dafür sein, Rekursion, sei es direkte oder indirekte Rekursion, zu vermeiden.

Alle hier untersuchten Systeme enthalten aber in zahlenmäßig begrenztem Maße dennoch Rekursion. Dabei handelt es sich ausschließlich um indirekte Rekursion, nie um direkte Rekursion.

Die Rekursion tritt anscheinend hauptsächlich unter extremen Bedingungen auf. So findet sich in einem System eine Prozedur, die Fehler behandelt. Diese Prozedur ruft dabei eine andere Prozedur auf, die in den permanenten Speicher schreibt. Tritt dabei ein Fehler auf, wird erneut die Fehlerbehandlungsprozedur aufgerufen.

4.5 Verwendung von Zeigern

Auch wenn die bisher geschilderten Besonderheiten eingebetteter Systeme den Entwickler von Analysatoren das Leben (im Vergleich zu Desktop-Computer-Software) erleichtern, gilt dies nicht für eine bekannte Quelle von Komplexität: Zeiger. Obwohl keine Heap-Allokationen verwendet werden, sind Zeiger in den hier untersuchten Systemen allgegenwärtig. Sie treten unter anderem auf, wenn Parameter per Referenz übergeben werden. Auch Prozedurzeiger und vereinzelt sogar Zeigerarithmetik werden verwendet.

Um konservativ zu sein, müssen diese Konstrukte also auch konservativ approximiert werden. Der in dieser Abhandlung zugrunde gelegten Data-Race-Analyse gelingt dies auch mit kleinen Einschränkungen: Wird zum Beispiel durch Zeigerarithmetik ein Zeiger von seinem ursprünglichen Ziel weg verschoben, sind die Analyseergebnisse nicht mehr konservativ. Durch Zeigeranalysen können aber Zeigerziele (auch für Funktionszeiger) mit in der Regel akzeptabler Präzision konservativ überapproximiert werden. Der in dieser Abhandlung vorgestellte Ansatz führt hier auch keine zusätzlichen nicht-konservativen Approximationen ein.

Kapitel 5

Einsatz der Data-Race-Analyse und Klassifikation der Data-Race-Warnungen

In den zurückliegenden Kapiteln wurde begründet, warum die Prüfung auf Data-Races Teil der Verifikation von eingebetteten sicherheitskritischen Systemen ist, und dargestellt, dass die statische Data-Race-Analyse ein geeignetes Mittel für diese Prüfung ist. Dieses Kapitel setzt sich mit der Durchführung dieser Prüfung auseinander. Es beschäftigt sich insbesondere auch mit der Klassifikation von Data-Race-Warnungen. Es fasst Teile des Forschungsstands zur Methodik der Data-Race-Analyse zusammen.

5.1 Einsatzszenarien der Data-Race-Analyse

Die konservative statische Data-Race-Analyse kann in unterschiedlichen Phasen der Software-Entwicklung eingesetzt werden, unter anderem:

- als Teil einer routinemäßig vorgesehenen Verifikation vor der Auslieferung eines eingebetteten Systems (vergleiche Raza, Vogel und Plödereder [RVP06]);
- beim Redeployment von Single-Core-Systemen auf Multi-Core-Systeme (Partitioning) (vergleiche Wittiger und Keul [WK12]) oder
- zur gezielten Fehlersuche, wenn bekannt ist, dass ein Mangel der Software vorliegt und vermutet oder zumindest nicht ausgeschlossen werden kann, dass es sich bei dem Mangel um ein Data-Race handelt.

Nach Nowotka und Traub [NT13] kann eine Technik zum Auffinden von Data-Races eingesetzt werden, *»um Sicherheit zu bieten bei 1) der Migration auf Multicore-Hardware und 2) der Entwicklung für Multicore-Hardware.«*

5.2 Umgang mit Ergebnissen der Data-Race-Analyse

Das Ergebnis der in der vorliegenden Abhandlung zugrunde gelegten Data-Race-Analyse ist eine Liste von Data-Race-Warnungen. Eine Data-Race-Warnung ist ein Paar von abstrakten Zugriffen, bei dem es sich möglicherweise um ein Data-Race handelt. Einer Data-Race-Warnung liegt also entweder tatsächlich ein Data-Race zugrunde, oder sie ist ein falsch-positiver Befund (false positive). Für den Umgang mit dieser Ungewissheit braucht es Konzepte.

Eine Möglichkeit ist, dem Programm so lange zusätzliche Synchronisation hinzuzufügen, bis die Data-Race-Analyse keine Data-Race-Warnungen mehr anzeigt. Der Vorteil dieses Vorgehens ist, dass – vorausgesetzt, das eingesetzte Werkzeug ist konservativ – das Ergebnis ein nachweislich Data-Race-freies System ist.

Man fügt bei diesem Vorgehen jedoch in der Regel unnötige Synchronisation ein, da es sich bei den Data-Race-Warnungen zumindest zum Teil um falsch-positive Befunde handelt. Nachteile dieser unnötigen Synchronisation können steigender Code-Umfang, erschwerte Wartung und schlechtere Performanz sein. Außerdem ist die zusätzliche Synchronisation auch Fehler-

quelle: Falsch eingesetzte Synchronisation kann zu Dead-Locks (und anderen Fehlern) führen.

In der Praxis ist dieses Vorgehen aufgrund der dargestellten Nachteile unrealistisch. Dem Autor dieser Abhandlung ist bekannt, dass die Entwickler eingebetteter sicherheitskritischer Systeme für Automobile Synchronisation vermeiden, soweit dies irgend möglich ist. Begründet wird dies mit der Sorge um Dead-Locks und Verschlechterung des Zeitverhaltens der Anwendung. Großflächiges Hinzufügen von möglicherweise unnötiger Synchronisation wird abgelehnt.

Daher muss im praktischen Einsatz ermittelt werden, ob es sich bei den Data-Race-Warnungen tatsächlich um Data-Races handelt oder um falsch-positive Meldungen. Möglicherweise soll auch zwischen unmittelbar zu behebenden Fehlern und zunächst noch tolerierbaren Data-Races unterschieden werden. Dieser Prozess wird in dieser Abhandlung als Klassifizierung bezeichnet. Dabei sieht sich der Anwender unter Umständen mit einer erheblichen Anzahl von falsch-positiven Meldungen konfrontiert (siehe Keul et al. [KPG+10], Felden und Görg [FG13], Degiorgi und Wittiger [DW13] und Koutsopoulos et al. [KNFW15]).

Auch andere (zum Beispiel dynamische) Ansätze, die zum Teil falsch-positive Ergebnisse haben, liefern letztendlich ähnliche Informationen – und auch bei diesen müssen die Data-Race-Warnungen klassifiziert werden. Das gilt unabhängig davon, ob auch falsch-negative Befunde vorkommen können.

Es existiert Forschung darüber, wie diese Klassifizierung bewerkstelligt werden kann und sollte[1]. Dabei werden unterschiedliche Methoden diskutiert. Zudem werden technische Vorschläge gemacht, von denen die Klassifizierung profitieren kann. Die folgenden Abschnitte beschreiben diese Forschungsergebnisse kurz.

[1] Viele der Forschungsarbeiten, die Werkzeuge beschreiben, die möglicherweise falsch-positive Warnungen ausgeben, kommentieren auch den Umgang mit diesen Warnungen. Explizit mit der Klassifizierung beschäftigen sich Koutsopoulos et al. [KNFW15] und Boehm [Boe11]. Letzterer nennt auch weitere Publikationen zu dem Thema. Auch Traub [Tra16] schreibt beispielsweise: *»In order to filter the error candidates into real runtime errors and false positives, a manual review is required. This review has to be made by an expert [...].«*

5.2.1 Allgemeine Methoden

Keul [Keu11] beschreibt, wie durch *»manual evaluation«* Data-Race-Warnungen klassifiziert wurden. Die Untersuchung wurde von einem mit dem System vertrauten Software-Ingenieur durchgeführt. Bei dieser Klassifizierung wurden 80 % der Data-Race-Warnungen als falsch-positiv erkannt. Keul geht aber nicht näher darauf ein, wie die Klassifizierung erfolgte.

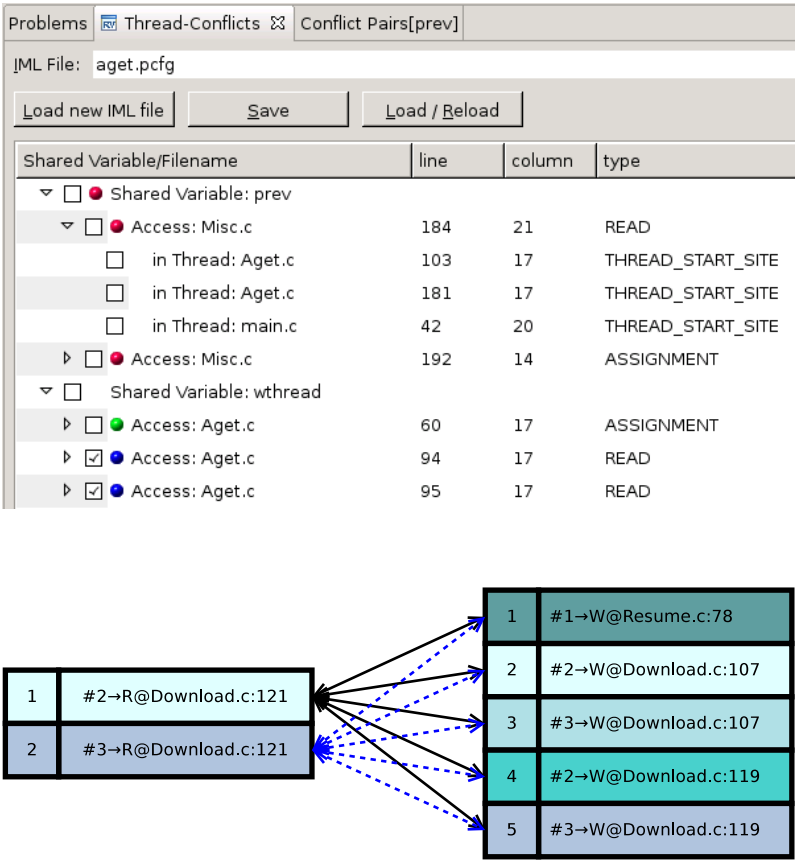
Keul et al. [KPG+10] skizzieren eine Methode zur Klassifizierung und beschreiben ein visuelles Werkzeug, um diese zu unterstützen. Dabei werden geteilte Variablen, die von Data-Race-Warnungen betroffen sind, nach und nach abgearbeitet. In dem Werkzeug werden alle abstrakten Zugriffe auf die zum jeweiligen Zeitpunkt zu untersuchende Variable in einem Baum, gruppiert nach Quelltextzugriffen, dargestellt (siehe Abbildung 5.1 oben). In einer weiteren Darstellung lässt sich erkennen, welche dieser Zugriffe Teil von Data-Race-Warnungen sind.

Der Benutzer des Klassifizierungswerkzeuges untersucht das Umfeld dieser Zugriffe. Hierzu kann er sich Aufrufpfade zu den Quelltextzugriffen anzeigen lassen. Ist die Untersuchung abgeschlossen, markiert er die Data-Race-Warnung entweder als falsch-positiv oder als Softwarefehler. Eine genaue Methode, wie die Untersuchung des Umfeldes zu einer Klassifizierungsentscheidung führt, wird dabei nicht vorgegeben.

5.2.2 Heuristische Bewertung

Degiorgi und Wittiger [DW13] stellen einen Ansatz vor, der den Klassifizierungsprozess ökonomisiert. Der Ansatz bewertet alle Data-Race-Warnungen heuristisch. Dabei soll sich die Klassifizierung von hoch bewerteten Data-Race-Warnungen mehr lohnen als die von niedrig bewerteten. Gute Heuristiken bewerten daher tatsächliche Data-Races statistisch höher als falsch-positive Data-Race-Warnungen. Außerdem bewerten sie Warnungen vor Data-Races, deren Auswirkungen verheerend sind, statistisch höher als Warnungen vor Data-Races mit geringen Auswirkungen.

In der Implementierung wird jedes Data-Race von mehreren unterschied-



Oben: Werkzeug aus Keul et al. [KPG+10].
Unten: Visualisierung aus Koutsopoulos et al. [KNFW15].

Die Graphiken sind der jeweiligen Publikation entnommen.

Abbildung 5.1: Darstellung einer Auswahl von Data-Race-Warnungen mit den Werkzeugen von Keul et al. und Koutsopoulos et al.

lichen Heuristiken bewertet. Die Einzelbewertungen werden zu einer Gesamtbewertung zusammengefasst.

Die Arbeit schlägt mehrere solcher Heuristiken vor und zeigt, dass diese implementiert werden können und bei Systemen industrieller Größe mit Hunderttausenden von Data-Race-Warnungen in akzeptabler Zeit ablaufen. Die Implementierung bewertet 1,261 Millionen Data-Race-Warnungen aus mehreren Systemen in 120 Sekunden, benötigt also weniger als eine zehntel Millisekunde pro Data-Race-Warnung. Der Nachweis, dass diese Heuristiken tatsächlich lohnendere Data-Races erkennen, ist allerdings bislang nicht erbracht. (Vergleiche Degiorgi und Wittiger [DW13].)

5.2.3 Besondere Benutzeroberflächen

Als weitere Möglichkeit, die Klassifizierung zu ermöglichen oder zu unterstützen, wurden geeignete Benutzeroberflächen identifiziert.

Schon Keul et al. [KPG+10] stellen ihrer Methode eine spezielle Benutzeroberfläche zur Seite, die es erlaubt, die Methode mit maschineller Unterstützung anzuwenden. Die wichtigste Fragestellung dabei ist, welche Informationen dem Benutzer neben dem Quelltext angezeigt werden sollen, um ihm die Klassifizierungsentscheidung zu ermöglichen oder zu erleichtern. Keul et al. konzentrieren sich dabei auf Speicherzugriffe und die Kontexte, in denen diese ausgeführt werden, sowie mögliche Aufrufpfade zu der von der Data-Race-Warnung betroffenen Stelle.

Auch Felden und Görg [FG13] wählen einen ähnlichen Ansatz. In der Publikation wird eine Benutzeroberfläche und ihr Einsatz zur Klassifizierung von Data-Race-Warnungen vorgestellt. Statt möglicher Aufrufpfade werden sogenannte »statische Aufrufgraphen« dargestellt. Dazu wird einer der kürzesten Pfade im Aufrufgraphen von der fraglichen Stelle bis zur Startprozedur des jeweiligen Tasks dargestellt. Darin sind dominierende Prozeduren hervorgehoben. Felden und Görg stellen beim Einsatz des Werkzeuges fest, dass diese kürzesten Pfade überwiegend aus dominierenden Prozeduren bestehen und dass gerade diese dominierenden Prozeduren ausschlaggebend für die Klassifizierung sind.

Dies erscheint sinnvoll. Felden und Görg beobachten zum Beispiel folgendes Muster in von ihnen untersuchten Programmen: Ein Task P schreibt zur Initialisierung einmalig einen Wert und ein anderer Task Q liest ihn wiederholt in seiner Arbeitsphase. Um eine aus diesem Szenario resultierende Data-Race-Warnung bei der Klassifizierung als falsch-positiven Befund zu erkennen, müssen drei Aspekte sichergestellt werden:

1. Task P schreibt tatsächlich nur im Kontext der Initialisierung auf die fragliche Speicherstelle,
2. Task Q liest die Speicherstelle nur in seiner Arbeitsphase und
3. die Programmlogik stellt sicher, dass Initialisierung und Arbeitsphase zeitlich getrennt sind.

An dieser Stelle muss der Aufrufgraph beachtet werden: Der erste Aspekt kann durch die Feststellung belegt werden, dass jeder Aufruf der Prozedur, die den Schreibzugriff aus Task P enthält, durch einen Aufruf der Initialisierungsprozedur `init_p()` dominiert ist.

5.2.4 Visualisierung

Einen Schritt weiter als die bisher genannten Arbeiten geht die Arbeit von Koutsopoulos et al. [KNFW15]. Sie versucht, die zur Klassifizierung notwendigen Daten so weit wie möglich visuell darzustellen. Dazu wird auch mit Einfärbung gearbeitet. Abbildung 5.1 (unten) vermittelt einen Eindruck davon. Es wird das Ziel verfolgt, die Data-Race-Analyse in den Entwicklungsprozess zu integrieren.

Wichtig dabei ist, die Menge der dargestellten Daten zu reduzieren und sich dabei auf die wesentlichsten Daten zu beschränken. So können zum Beispiel Aufrufgraphen und Pfade in Aufrufgraphen teilweise auf Dateien (Module) abstrahiert werden. Dass die Darstellung dadurch deutlich übersichtlicher wird, soll den Verlust an Detailinformationen überkompensieren. Zusätzlich werden klassische Techniken wie backward slicing des Kontrollflusses eingesetzt, um die Datenmenge zu reduzieren. Einfach ausgedrückt,

werden alle Anweisungen, die nicht vor dem fraglichen Zugriff ausgeführt werden können, ausgeblendet.

Eine problematische Eigenschaft früherer Benutzeroberflächen ist, dass die Komplexität der Darstellung linear mit der Zahl der Data-Race-Warnungen steigt. Diese ließen sich deswegen gerade bei solchen Systemen, bei denen wegen der hohen Zahl an Data-Race-Warnungen eine Werkzeugunterstützung besonders wünschenswert war, nicht sinnvoll einsetzen. Das Werkzeug von Koutsopoulos et al. hat hingegen das Ziel, mit einer hohen Zahl an Data-Race-Warnungen zurechtzukommen.

Schlussendlich reicht aber ein rein visueller Ansatz nicht aus. Bevor der Anwender eine endgültige Entscheidung trifft, zeigen auch Koutsopoulos et al. den Quelltext des Systems an.

Kapitel 6

Berücksichtigung zustandsbasierter Synchronisation

Wird die im vorangehenden Kapitel beschriebene Methodik zur manuellen Klassifizierung von Data-Race-Warnungen auf eingebettete Systeme aus Fahrzeugen angewandt, stößt der Anwender oft auf explizite Zustandsverwaltung. Diese lässt teils erkennen, dass bestimmte Operationen in unterschiedlichen Laufzeitphasen auftreten. Dieses Kapitel definiert die Begriffe »explizite Zustandsverwaltung« und »zustandsbasierte Synchronisation«. Es beschreibt zwei Ansätze, die Synchronisationseigenschaften von zustandsbasierter Synchronisation zu analysieren und damit die Zahl an Data-Race-Warnungen zu reduzieren.

6.1 Explizite Zustandsverwaltung

Eingebettete Systeme, die von der Industrie für Automobile entwickelt werden, verwalten ihren Systemzustand oft in expliziter Weise. Dies haben bereits Keul [Keu11] und Schwarz et al. [SSVA14] festgestellt.

6.1.1 Definition

Das Programmiermuster explizite Zustandsverwaltung besteht aus zwei Komponenten: der Zustandsvariablen und einer Zustandsmenge aus Zustandswerten. Diese erfüllen die folgenden Bedingungen:

- Die Zustandsvariable ist eine globale Variable.
- Lesende Zugriffe auf die Zustandsvariable kommen unter anderem im Bedingungsteil bedingter Anweisungen/Verzweigungen vor. In der Regel wird der Wert der Zustandsvariablen dabei auf Gleichheit oder Ungleichheit mit einem Element der Zustandsmenge geprüft.
- Die Zustandsmenge besteht aus endlich vielen mit Mitteln der verwendeten Programmiersprache benannten Zustandswerten.
- Mit den Änderungen des Wertes der Zustandsvariablen ändert sich der Systemzustand. In der Regel erfolgt diese Änderung durch direkte Zuweisung eines Zustandswertes in der Form `[Zustandsvariable] = [Zustandswertname];`.

6.1.2 Erläuterungen

Oft wird auf Zustandsvariablen aus mehreren Tasks zugegriffen. Der Bezeichner der Zustandsvariablen ist meist ein klarer Hinweis darauf, dass die Variable zur Zustandsverwaltung eingesetzt wird. Oft ist »`state`« ein Substring des Bezeichners.

Die Zustandsvariablen bilden oft endliche Automaten ab. Endliche Automaten können aber auch auf anderem Wege implementiert werden und Zustandsvariablen sind nicht auf endliche Automaten beschränkt.

```
1  #define C 1
2  const int D = 2;
3  const int E = C * 3;
4  typedef enum {F = 4, G, H} ct;
5  #define I (1 | 2 | 4)
6
7  const int R;
8  int S = 13;
9  volatile const int T = 15;
```

Abbildung 6.1: Verschiedene Möglichkeiten, in der Sprache C Werte zu benennen

Zustandsvariablen haben in den in dieser Abhandlung untersuchten Systemen erwartungsgemäß stets den Typ `int`. Die Zustandswerte sind dazu kompatibel.

Bei den bedingten Anweisungen/Verzweigungen handelt es sich bei eingebetteten Systemen vor allem um `if`- oder `switch`/`case`-Anweisungen, weniger typisch aber auch um Schleifen-Anweisungen.

Die Programmiersprache C hat mehrere Möglichkeiten, Werte zu benennen. Dies wird in Abbildung 6.1 illustriert.

In der Abbildung sind `C`–`I` Namen für die Werte `1`–`7`. `R`, `S` und `T` haben hingegen keinen konstanten Wert. Dabei werden für `C` und `I` Makrodefinitionen verwendet und für `F`, `G` und `H` ein Enumerationstyp definiert. Die Benennung eines Wertes durch Deklaration einer `const`-Variable (wie in `D` und `E`) kommt in der Praxis ebenfalls vor. Die vorgestellte Analyse akzeptiert `D` und `E` auch als benannte Werte – trotz möglicher Schwierigkeiten mit der Initialisierungsreihenfolge.

An Stelle des Begriffs »benannter Wert« wird teils auch der Begriff Konstante verwendet. Diese Begrifflichkeit kollidiert allerdings mit der im C-Standard verwendeten Begrifflichkeit »`constant`«, die eine andere Bedeutung hat.

Anders als in Abbildung 6.1 dargestellt, sind die Namen der Zustandswerte (wie die der Zustandsvariablen) in den hier untersuchten Systemen sprechend gewählt.

Jede explizite Zustandsverwaltung stellt näherungsweise einen determi-

nistischen endlichen Automaten dar. Dabei entsprechen die Zustandswerte den Zuständen des Automaten. Die Zustandsvariable gibt an, in welchem Zustand der Automat zum jeweiligen Zeitpunkt ist. Durch Initialisierung wird der Startzustand bestimmt.

6.2 Zustandsbasierte Synchronisation

Bei der in Kapitel 5 beschriebenen Klassifizierung von Data-Race-Warnungen aus der Analyse von eingebetteten Systemen aus Automobilen stoßen Anwender immer wieder auf explizite Zustandsverwaltung. (Vergleiche Keul [Keu11].) Oft hat es dabei den Anschein, dass die zwei Zugriffe einer Data-Race-Warnung unterschiedlichen Systemzuständen zuzuordnen sind. Dann stellt sich die Frage, ob die Data-Race-Warnung deswegen als falsch-positiv bewertet werden kann.

6.2.1 Synchronisationseigenschaften am Beispiel

In Abbildung 6.2 ist eine solche explizite Zustandsverwaltung dargestellt. Hier wird angenommen, dass eine Data-Race-Warnung für die Zugriffe auf die Variable `i` vorliegt. Task A scheint dabei nur im Zustand `STARTUP` auf die Variable zuzugreifen, wohingegen Task B nur im Zustand `RUN` auf die Variable zugreift. Reicht das aus, um ein Data-Race auszuschließen?

```
1  typedef enum {STARTUP, RUN, SHUTDOWN} st;
2  volatile int swc_state;
```

```
1  // task_a
2  if (swc_state == STARTUP)
3     i = 100;
4  // ...
```

```
1  // task_b
2  if (swc_state == RUN)
3     i++;
4  // ...
```

Abbildung 6.2: Beispiel für explizite Zustandsverwaltung

Diese explizite Zustandsverwaltung ist nicht ausreichend, um ein Data-Race auszuschließen. Es ist davon auszugehen, dass irgendwo im Programm ein Zustandswechsel vom Zustand `STARTUP` in den Zustand `RUN` erfolgt. Unter Umständen, je nachdem, wie dieser Zustandswechsel erfolgt, ist es leicht, eine Data-Race-Situation zu finden. Sie könnte daraus bestehen, dass im Systemzustand `STARTUP` Task A an dem Prädikat vorbeiläuft und anschließend ein anderer Task den Systemzustand auf `RUN` setzt. Von da an stünde der Data-Race-Situation nichts mehr im Wege.

Ebenso denkbar ist aber auch, dass das Data-Race tatsächlich ausgeschlossen werden kann. Wenn Schreibzugriffe auf die Zustandsvariable `swc_state` nur in Tasks erfolgen, die eine niedrigere Priorität als Task A und Task B haben, gibt es keine unerwarteten Zustandsübergänge mehr und das Muster schließt Data-Races auf den dargestellten Zugriffen auf die Variable `i` aus. (Dabei wird davon ausgegangen, dass sowohl Task A als auch Task B die Multiplizität 1 haben und ein übliches prioritätenbasiertes Schedulingverfahren verwendet wird.)

6.2.2 Definition

An dem im letzten Abschnitt behandelten Beispiel ist erkennbar, dass explizite Zustandsverwaltung durchaus geeignet sein kann, um nebenläufige Programme zu synchronisieren. Daher wird hier der neue Begriff »zustandsbasierte Synchronisation« als Teilmenge der expliziten Zustandsverwaltung definiert.

Eine zustandsbasierte Synchronisation ist ein Programmiermuster, das die Kriterien der expliziten Zustandsverwaltung erfüllt und sich zusätzlich dazu eignet, Programme zu synchronisieren. Synchronisieren bedeutet in diesem Zusammenhang, dass zumindest manche Zugriffe, die unterschiedlichen Systemzuständen zuzuordnen sind, aufgrund dieser Zuordnung vor nebenläufiger Ausführung geschützt sind.

Die Intention des Software-Entwicklers/-Architekten spielt dabei keine Rolle. Eine explizite Zustandsverwaltung muss nicht notwendigerweise zur Synchronisation taugen. Es ist daher auch nicht als Fehler anzusehen, wenn

eine explizite Zustandsverwaltung nicht bei der Synchronisation hilft. Ebenso wenig ist aber ausgeschlossen, dass eine explizite Zustandsverwaltung, die nicht dazu gedacht ist, eben doch – quasi zufällig – zur Synchronisation beiträgt.

6.3 Ansätze zur Erkennung zustandsbasierter Synchronisation

Dass explizite Zustandsverwaltung Programme synchronisieren kann, ist in der Literatur bekannt. Das Vorgehen kann als Spezialfall von »lock-free« Algorithmen gesehen werden.

Für den Anwendungsfall eingebetteter Systeme aus Automobilen sind in der Literatur zwei Analyseansätze zur finden. Diese werden in diesem Abschnitt vorgestellt.

6.3.1 Muster von Keul

Keul [Keu11] führt aus, dass in der von ihm untersuchten Software mehrere Instanzen eines einfachen Musters für endliche Automaten vorkommen und diese erkannt und zum Ausschluss von Data-Race-Warnungen genutzt werden können. Prokharau, Gerlach und Keul [PGK11] beschreiben Aspekte der Implementierung einer Analyse, die diese Muster erkennt. Die genauen Kriterien, nach denen die Muster erkannt werden, sind diesen beiden Publikationen jedoch nicht zu entnehmen.

In einem unveröffentlichten Projektbericht [Keu12] und der projektinternen Dokumentation des Bauhaus Data Race Detectors beschreibt Keul die Kriterien, welche die Muster erfüllen müssen. Abschnitt 6.3.1 paraphrasiert und übernimmt in weiten Teilen Inhalte dieser beiden für die Allgemeinheit unzugänglichen Quellen.

6.3.1.1 Kriterien

Keul fordert von zustandsbasierter Synchronisation folgende Eigenschaften ein:

- Die Zustandsvariable muss eine globale, ausschließlich atomar les- und schreibbare Variable sein.
- Die Zustandswerte müssen als Enum-Typ deklariert sein.
- Die Zustandsvariable wird in berücksichtigten Pfadprädikaten nur auf Gleich- und Ungleichheit mit einem Enum-Wert geprüft.
- Alle Änderungen an der Zustandsvariablen sind direkte Zuweisungen eines Enum-Wertes.
- Die Zustandsvariable wird vor der ersten Verwendung initialisiert.

Zusätzlich dürfen Änderungen an der Zustandsvariablen nur nach einem der beiden folgenden Schemen erfolgen. Die Schemen dürfen nicht gemischt werden.

Schema 1:

- Die Zustandsvariable darf während der Initialisierungsphase des Systems, zu der noch keine Nebenläufigkeit vorkommt und somit die eigentlichen Tasks noch nicht gestartet wurden, verändert werden.
- Die Zustandsvariable darf in einem Programmbereich, der von einem Pfadprädikat p auf der Zustandsvariablen dominiert wird, verändert werden wenn
  - alle anderen, möglicherweise nebenläufig dazu ablaufenden Veränderungen an der Zustandsvariablen durch Prädikate $q_1 \dots q_n$ dominiert werden und sich die Prädikate p und q_i jeweils gegenseitig ausschließen;
  - die Zuweisung die letzte Anweisung in dem durch das Prädikat dominierten Programmbereich ist. Sind die Muster verschachtelt, darf nur ein äußerstes Muster den Wert der Zustandsvariablen verändern.

Schema 2:

- Die Zustandsvariable darf während der Initialisierungsphase des Systems, zu der noch keine Nebenläufigkeit vorkommt und somit die eigentlichen Tasks noch nicht gestartet wurden, verändert werden.
- Außerhalb der Initialisierungsphase greift nur ein Task schreibend auf die Zustandsvariable zu. Dieser Task hat eine niedrigere Priorität als alle Tasks, die lesend auf die Zustandsvariable zugreifen. Insbesondere greift er selbst nicht lesend auf die Variable zu.

Bei beiden Schemata gilt, dass Programmbereiche, die durch sich gegenseitig ausschließende Pfadprädikate geschützt sind, nicht nebenläufig ausgeführt werden können. Es gibt also zwischen diesen Bereichen keine Data-Races. Bei Schema 2 sind außerdem Bereiche nach der Zuweisung eines Wertes gegenüber Bereichen mit widersprechendem Prädikat geschützt.

Keuls Kriterien entsprechen einer naheliegenden Implementierungsweise für endliche Automaten und scheinen in Einklang mit der industriellen Praxis zu stehen. Keul gibt aber keine ausführliche Begründung für die Anforderungen und plausibilisiert ihre Sinnhaftigkeit nicht. Die Kriterien sind nicht minimal. Sie scheinen zum Teil dazu zu dienen, eine einfache syntaktische Überprüfung zu ermöglichen.

Für Schema 1 enthält Bauhaus eine Implementierung. Diese überprüft allerdings nicht alle gestellten Kriterien und ist somit nur unter passenden Umständen konservativ. Für Schema 2 ist dem Autor keine Implementierung bekannt.

6.3.2 Muster von Schwarz et al.

Schwarz et al. [SSVA14] stellen einen Ansatz zur Erkennung von expliziter Zustandsverwaltung vor. Das Muster, das sie erkennen, weicht von den beiden von Keul (siehe Abschnitt 6.3.1 (Seite 86)) erkannten Mustern ab. Außerdem verwenden sie eine andere Begrifflichkeit. Ihr Begriff »flag« entspricht der in dieser Abhandlung verwendeten Definition einer Zustandsvariable und

meint dabei insbesondere nicht notwendigerweise ein einzelnes Bit, sondern eine Variable beispielsweise vom Typ `int`.

Hauptbeitrag der Veröffentlichung sind zwei Analysealgorithmen: ein einfacher schneller und ein, in gewissem Sinne, vollständig präziser. Die Analysealgorithmen können die Eignung der erkannten expliziten Zustandsverwaltung zur Synchronisation berechnen.

6.3.2.1 Kriterien

Damit eine Zustandsvariable als solche akzeptiert wird, muss sie einige Kriterien erfüllen. Schwarz et al. akzeptieren eine Variable f als Zustandsvariable genau dann, wenn

1. f nur Konstanten zugewiesen werden,
2. f nur mit Konstanten verglichen wird,
3. die Adresse von f nie ermittelt wird und
4. wenn p_f die niedrigste Priorität ist, sodass es einen Task gibt, der lesend oder schreibend auf f zugreift, alle Tasks, die eine Priorität größer als p_f haben, f *intakt* lassen.

Eine Variable intakt zu lassen bedeutet dabei, dass die Variable beim Beenden des Tasks denselben Wert hat wie beim Start des Tasks. Wird die Variable vom Task verändert, muss der ursprüngliche Wert vor dem Beenden des Tasks demnach wiederhergestellt werden. Die Eigenschaft der Intaktheit ist im Allgemeinen offensichtlich unentscheidbar.

Damit sind die von Schwarz et al. analysierten Muster eher als Abstraktion eines Locks denn als endlicher Automat zu sehen. Bei einem Lock ist es sinnvoll, dass jeder Task das Lock nach getaner Arbeit wieder freigibt. Bei endlichen Automaten ist es kaum einzusehen, dass ein Task, der den Zustand verändert, stets dafür sorgen muss, dass die Änderung schlussendlich wieder rückgängig gemacht wird.

Konstanten müssen bei den Kriterien wohl im Sinne des C-Standards verstanden werden.

Laut den Autoren sind die Operatoren ==, !=, < und > für Vergleiche erlaubt. In den Beispielen und Erklärungen wird aber nur auf == und != eingegangen. Der vorgestellte Algorithmus kann mit < und > nur umgehen, wenn weitere Bedingungen erfüllt sind.

Die Analyse von Schwarz et al. prüft, ob eine Variable die Kriterien 1–3 erfüllt. Erfüllt sie die drei Kriterien, wird sie als Zustandsvariable angesehen und die Analyse geht fortan davon aus, dass auch Kriterium 4 erfüllt ist.

Für eine gegebene Data-Race-Warnung kann der Algorithmus konservativ (und in gewissen Sinne auch ohne Überapproximation) bestimmen, für welche Initialwerte der Zustandsvariablen die Data-Race-Situation[1] erreicht werden kann. Genau dann, wenn diese Menge leer ist, kann die Data-Race-Warnung gestrichen werden.

6.4 Beispiele

Dieser Abschnitt zeigt, wie sich den unterschiedlichen Kriterien entsprechende zustandsbasierte Synchronisation implementieren lässt.

6.4.1 Beispiele nach Keul

Abbildung 6.3 zeigt eine zustandsbasierte Synchronisation, die den Kriterien von Keul (Schema 1) entspricht. Das Beispiel ist eng angelehnt an ein in eingebetteter Software gefundenes Muster.

Wie abgebildet eignet sich das Beispiel zur Synchronisation.. Die hier mit A (Abbildung 6.3) und B (Abbildung 6.3) ersetzten Code-Regionen sind vor nebenläufiger Ausführung geschützt: zwischen ihnen gibt es keine Data-Races.

Es lässt sich erkennen, dass das Schema 1 von Keul immer wohlgeformte endliche Automaten hervorbringt. Die Zustandsmenge ist die Menge der Zustände; die Initialisierung der Zustandsvariablen definiert den Startzustand;

[1] Die Definition von Schwarz et al. für ein Data-Race stützt sich auf ein Konzept, das im Wesentlichen der in dieser Abhandlung verwendeten Definition einer Data-Race-Situation entspricht.

```
1  volatile enum { E_Initial, E_Regular,
       E_Abnormal} E_Software_State;
2
3  void task1(void) {
4     if (E_Software_State == E_Initial) {
5        A;
6        E_Software_State = E_Regular;
7  }  }
8
9  void task2(void) {
10    if (E_Software_State == E_Regular) {
11       B;
12       E_Software_State = E_Abnormal;
13 }  }
14
15 int main(void) {
16    E_Software_State = E_Initial;
17    StartOS();
18 }
```

Codebeispiel übernommen aus Keul [Keu12].

Abbildung 6.3: Beispiel einer zustandsbasierten Synchronisation, die Keuls Kriterien (Schema 1) entspricht

die weiteren Zuweisungen sind Zustandsübergänge. Das System befindet sich (sobald die Initialisierung erfolgt ist) stets in einem definierten Zustand des endlichen Automaten.

Abbildung 6.4 zeigt eine weitere zustandsbasierte Synchronisation. Diese entspricht den Kriterien von Keul nach Schema 2. Auch dieses Muster ist eng an reale Implementierungen angelehnt.

Auch dieses Muster eignet sich mit kleinen Einschränkungen zur Synchronisation. Wie beim Beispiel aus Abbildung 6.3 wären auch hier Bereiche, die von sich widersprechenden Prädikaten dominiert sind, vor nebenläufiger Ausführung geschützt. Solche sind aber nicht dargestellt. Das Beispiel zielt hingegen darauf ab, dass ein Schutzbereich auch mit Zuweisung an die Zustandsvariable beginnen und enden kann. Insbesondere sind hier die Codebereiche, die durch A und B (Zeilen 5 und 10) ersetzt wurden, vor

```
1  volatile enum {E_Inactive = 0, E_Active} E_State;
2
3  void task_highest(void) {
4     if (E_State) {
5        A;
6  }  }
7
8  void task_background(void) {
9     E_State = E_Inactive;
10    B;
11    E_State = E_Active;
12 }
13
14 int main(void) {
15    E_State = E_Inactive;
16    StartOS();
17 }
```

Codebeispiel übernommen aus Keul [Keu12].

Abbildung 6.4: Beispiel einer zustandsbasierten Synchronisation, die Keuls Kriterien (Schema 2) entspricht

nebenläufiger Ausführung geschützt. Das Muster mit Schema 2 eignet sich, um Funktionalität an- und abzuschalten. Programmlogik, die gegebenenfalls Zugriffe auslässt, ist hier als eine Form der Synchronisation anzusehen.

6.4.2 Beispiele nach Schwarz et al.

Abbildung 6.5 zeigt, wie Schwarz et al. zustandsbasierte Synchronisation darstellen. Im Gegensatz zu Keul trennen Schwarz et al. zwischen Synchronisationseigenschaften und Analysierbarkeit.

Das Programm aus Abbildung 6.5 ist für Schwarz et al. analysierbar und sie kommen (richtigerweise) zu dem Schluss, dass die Zugriffe auf die Variable x in Zeile 6 und Zeile 11 frei von Data-Races sind. Das Beispiel ähnelt dem aus Abbildung 6.4. Auch hier wird Funktionalität durch Setzen der Zustandsvariablen an- und abgeschaltet. Wie schon Keul bemerken auch Schwarz et al., dass Taskprioritäten dabei eine essentielle Rolle spielen.

```
1  int f = 0;
2  int x = 0;

4  task_high() {
5     if (f == 0)
6        x--;
7  }

9  task_low() {
10    f = 1;
11    x++;
12    f = 0;
13 }
```

Entspricht `example_flag` aus Schwarz et al. [SSVA14].

Abbildung 6.5: Einfaches Beispiel für explizite Zustandsverwaltung nach Schwarz et al.

In einem Experiment vertauschen die Autoren die Taskprioritäten des Beispiels. Das veränderte Beispiel nennen sie `inverse_flag`. Im veränderten Beispiel kann das Data-Race zwischen den Zugriffen auf die Variable `x` nicht mehr ausgeschlossen werden. Tatsächlich ist es leicht, eine Data-Race-Situation für das Beispiel zu finden.

Schwarz et al. besprechen ein weiteres interessantes Beispiel. Es ist in Abbildung 6.6 dargestellt. Besonders zu beachten ist hier die Zuweisung des Wertes `2` an die Zustandsvariable `f` in Zeile 12. Diese Zuweisung führt dazu, dass sich während der Ausführung des niederstprioren Tasks der Wert der Zustandsvariablen `f` plötzlich auf `2` ändern kann. Dem von Schwarz et al. postulierten Prinzip der Intaktheit folgend, muss der ursprüngliche Wert der Zustandsvariablen `f (1)` dann aber wiederhergestellt werden, bevor der niederstpriore Task seine Ausführung fortsetzt. Dies geschieht hier durch die Zuweisung in Zeile 13.

So wie hier dargestellt, eignet sich die explizite Zustandsverwaltung zur Synchronisation – es gibt kein Data-Race auf der Variablen `x`.

Wird im höchstprioren Task allerdings die Bedingung in Zeile 18 auf `f != 1` abgeändert, zeigt sich die Relevanz der Zuweisung. Die Data-Race-Situation ist dann erreichbar und das Data-Race darf keinesfalls ausgeschlossen werden. Die Analyse von Schwarz et al. ermittelt dies auch korrekt.

```
1  int f = 0;
2  int x = 0;
3
4  task_low() {
5     f = 1;
6     x++;
7     f = 0;
8  }
```

```
10 task_med() {
11    if (f == 1) {
12       f = 2;
13       f = 1;
14    }
15 }
16
17 task_high() {
18    if (f == 0)
19       x--;
20 }
```

Codebeispiel übernommen aus Schwarz et al. [SSVA14].

Abbildung 6.6: Komplexeres Beispiel für explizite Zustandsverwaltung nach Schwarz et al.

Kapitel 7

Modellsprache CSP/CSP$_M$

Der in dieser Abhandlung beschriebene Ansatz geht über die in Kapitel 6 beschriebenen Ansätze von Keul und Schwarz et al. hinaus. Dazu wird im weiteren Verlauf ein Ansatz vorgestellt, der explizite Zustandsverwaltung analysiert, indem bestimmte Fragestellungen in der formalen Sprache CSP$_M$ formuliert werden. Dieses Kapitel beschreibt die Modellsprache CSP$_M$ und erläutert das Verhältnis zum CSP-Kalkül. Es erklärt, wie Werkzeuge mit der formalen Sprache umgehen können, und beschreibt den typischen Einsatz. Außerdem benennt es Werkzeuge, die CSP$_M$ maschinell bearbeiten können.

7.1 Showcasing CSP

Für den in dieser Abhandlung beschriebenen Ansatz wird CSP (Communicating Sequential Processes) als formale Modellsprache eingesetzt. Dabei wird die maschinenlesbare Variante CSP$_M$ (Machine-Readable CSP) verwendet. Bei CSP handelt es sich um einen mathematischen Kalkül zur Modellierung von nebenläufig agierenden und kommunizierenden Aktoren (genannt Prozesse). Der Kalkül wurde zuerst im Jahre 1978 von Hoare [Hoa78] beschrieben. Die umfassendste Beschreibung des seitdem signifikant weiterentwickelten

```
channel a
channel long_signal_name
--  multiple signals in one declaration
channel r, s, t
```

Abbildung 7.1: Signal-Deklarationen in CSP_M

Kalküls findet sich in Hoare [Hoa04]. Auch Roscoe [Ros11b] widmet sich dem Kalkül.

7.1.1 Funktionsumfang der Sprache

Dieser Abschnitt beschreibt kurz einige Bestandteile des CSP-Kalküls. Dabei werden genau diejenigen Mechanismen beschrieben, die später verwendet werden, um der Modellerzeugung verwendet werde. Für den gesamten Abschnitt, sofern nicht anders angegeben, wird auf Hoare [Hoa04] als Beleg und Quelle verwiesen. Dort findet sich auch eine präzise Semantik der besprochenen Mechanismen.

Es werden hier meist die in CSP_M üblichen, aus ASCII-Zeichen zusammengesetzten Operatoren anstelle der eigentlich für CSP angemesseneren »hübschen« Operatorensymbole verwendet, um die Einführung einer weiteren Notation zu vermeiden.

Bei CSP geht es um Prozesse und Signale. Prozesse werden dabei mit Großbuchstaben benannt und Signale mit Kleinbuchstaben. Prozesse ändern ihren internen Zustand, indem sie Signale an ihre jeweilige Umgebung senden oder indem sie einem sogenannten stillen Übergang unterliegen.

7.1.1.1 Signale

Signale haben einen Namen und werden mit dem Schlüsselwort `channel` deklariert. Die Deklaration muss vor der ersten Verwendung in der CSP_M-Datei stehen. Abbildung 7.1 stellt solche Deklarationen dar.

7.1.1.2 Vordefinierte Prozesse

Zwei vordefinierte Prozesse sind für diese Abhandlung sehr wichtig: `STOP` und `SKIP`.

- `STOP` ist der Prozess, der nichts tut. Er kann kein Signal aussenden und keinem stillen Übergang unterliegen.
- `SKIP` ist der Prozess, der nichts tut, außer erfolgreich zu terminieren (*»successful termination«*). Er kann als ein Prozess gesehen werden, der das spezielle Signal ✓ (gesprochen »tick«) aussendet und sich dann wie `STOP` verhält.

7.1.1.3 Präfix

Mit der wichtigste Operator in CSP ist »`->`« (ausgesprochen »then«). Auf der linken Seite des binären Operators steht ein Signal; auf der rechten Seite steht ein Prozess. Das Ergebnis ist dabei ein Prozess, der das links stehende Signal aussenden kann, um zum auf der rechten Seite stehenden Prozess zu werden. Mit »`a -> SKIP`« lässt sich ein Prozess beschreiben, der das Signal `a` aussendet, um anschließend erfolgreich zu terminieren.

7.1.1.4 Sequenz

Ein weiterer hier gebrauchter Operator ist der Sequenz-Operator »`;`« (gesprochen »followed by«). Es ist ein binärer Operator auf Prozessen, der einen neuen Prozess zurückgibt. Dabei ist »`P ; Q`« ein Prozess, der sich zunächst wie der Prozess `P` verhält und wenn dieser erfolgreich terminiert, wie der Prozess `Q`.

7.1.1.5 Parallelität

Zwei Prozesse können parallel ausgeführt werden durch den Alphabetised-Parallel-Operator »`[|` Σ `|]`«. Dabei bezeichnet Σ die Menge der Signale, die der kombinierte Prozess nur aussenden kann, wenn seine beiden Subprozesse darin übereinstimmen.

Spezialfälle des Alphabetised-Parallel-Operators sind der Synchronised-Parallel-Operator »|| « und der Interleaving-Operator »|||«. Bei ersterem enthält die Synchronisationsmenge alle Signale; bei zweiterem ist die Synchronisationsmenge leer.

Der Prozess »`a -> c -> STOP [|a|] a -> b -> STOP`« kann zunächst nur das Signal `a` aussenden und anschließend die Signale `b` und `c` in beliebiger Reihenfolge.

Fügt man hingegen das Signal `c` der Synchronisationsmenge hinzu, das heißt »`a -> c -> STOP [|a, c|] a -> b -> STOP`«, kann das Signal `c` nicht mehr ausgesendet werden und der Prozess verhält sich wie der Prozess »`a -> b -> STOP`«.

7.1.1.6 Verdeckung

Wie auch der Sequenz-Operator, ist der unäre postfix-geschriebene Verdeckungs-Operator »`\ Σ`« (gesprochen »without« oder auch »hiding«, im Englischen als »concealment« oder »hiding operator« bezeichnet) ein durch eine Menge näher bestimmbarer Operator auf Prozessen.

Der Prozess »`P \ {a}`« verhält sich dabei wie der Prozess `P`, sendet aber niemals das Signal `a` aus. An den Stellen, an denen der Prozess `P` das Signal `a` aussenden würde, unterliegt er stattdessen einem stillen Übergang.

7.1.1.7 Auswahl

Eine bekannte Eigenschaft von CSP sind die unterschiedlichen Auswahl-Operatoren. Die beiden wichtigsten davon sind »`[]`« und »`|~|`«. Aufgrund des hier gewählten semantischen Modells für CSP muss zwischen den beiden jedoch nicht unterschieden werden und es wird stets der Operator `|~|` verwendet. Der binäre und assoziative Operator `|~|` ermöglicht die Wahl zwischen zwei Prozessen. Der Prozess »`P |~| Q`« (gesprochen »P choice Q«) kann sich entweder wie der Prozess `P` oder wie der Prozess `Q` verhalten. Dies entspricht dem Verhalten eines Prozesses, der zunächst einem stillen internen Zustandsübergang unterliegt, der sein späteres Verhalten bestimmt.

```
channel up, down, open, close


R = up -> R  |~| down -> R  |~|  open -> close -> R

SPEC1 = R


E = up -> O  |~|  down -> O  |~|  open -> close -> E
O = up -> E  |~|  down -> E

SPEC2 = E
```

Abbildung 7.2: Eine CSP_M-Datei, die die Prozesse `SPEC1` und `SPEC2` mit unterschiedlichen Anforderungen definiert

7.1.1.8 Rekursive Definition

Prozesse werden in CSP definiert, indem ihnen Namen zugewiesen werden. In Abbildung 7.2 werden so die Prozesse `R`, `E`, `O`, `SPEC1` und `SPEC2` definiert. Dabei können die Definitionen direkt rekursiv sein (wie die vom Prozess `R`) oder sogar indirekt rekursiv (wie die von den Prozessen `E` und `O`).

Rekursive Definitionen werden dabei als µ-quantifizierte Ausdrücke gesehen. Die Definition von »`P = a -> P`« ist ein Beispiel dafür. Sie wird interpretiert als »`P = µ X . a -> X`«. Dies soll verstanden werden als: `P` ist der kleinste[1] Prozess, der die Gleichung »`X = a -> X`« erfüllt.

Dabei ist CSP aber nicht abgeschlossen bezüglich der µ-Quantifizierung. Syntaktisch richtige µ-Quantifizierungen ergeben nicht notwendigerweise semantisch Sinn. Im Prinzip können Definitionen, die auf µ-Quantifizierung beruhen, auf zwei Arten misslingen: Es kann keinen Prozess geben, der die gegebene Gleichung erfüllt, und es kann mehrere Prozesse geben, die die Gleichung erfüllen, von denen aber keiner kleiner als alle anderen ist, es existiert also kein eindeutiges Minimum.

[1] Es existiert eine präzise Beschreibung der Größenrelation. Informell ist ein Prozess kleiner als ein anderer, wenn er weniger Verhalten hat. Wenn der Prozess `P` alles tun kann, was der Prozess `Q` tun kann, so ist der Prozess `Q` kleiner oder gleich dem Prozess `P`.

Tatsächlich ist das Entscheidungsproblem, ob ein bestimmter Prozess das Ergebnis einer gegebenen µ-Quantifizierung ist, unentscheidbar.

7.1.2 CSP_M

Der Kalkül CSP wurde zur Beschreibung von Systemen mit »Stift und Papier« beziehungsweise »Kreide und Tafel« entwickelt. Demgegenüber existiert CSP_M. Dabei handelt es sich um eine formale Sprache. Sie ist mittels Grammatiken über dem ASCII-Alphabet definiert und lässt sich somit maschinell parsen und verarbeiten.

Leuschel und Fontaine [LF08] bezeichnen die Entwicklung eines Parsers für CSP_M zwar als *»eine der größten Hürden«* für ihr Projekt und beschweren sich über eine ganze Reihe von Eigenschaften der Sprache – darunter auch das aggressive Wiederverwenden von Operatoren für unterschiedliche Zwecke[1] – schließlich gelingt es ihnen aber, einen Parser zu schreiben.

Im Folgenden wird nicht strikt zwischen der formalen Sprache CSP_M und dem Kalkül CSP unterschieden.

CSP_M ist offensichtlich Turing-vollständig. Tatsächlich, aber in dieser Abhandlung unbewiesen[2], ist auch das hier genutzte Sprachfragment Turing-vollständig, da sich alle CSP_M-Skripte Semantik-erhaltend so umwandeln lassen, dass sie in diesem Sprachfragment liegen.

7.2 Werkzeuge für CSP_M

CSP_M ermöglicht es, Werkzeuge für die Sprache zu bauen. Auch wenn sich »echte« Programme, die mit dem Benutzer interagieren, in CSP_M schreiben

[1] So lässt sich eine Liste aus den booleschen Werten `True`, `True` und `False` schreiben als »`<true, 2>1, false>`«. (Siehe Leuschel und Fontaine [LF08].)

[2] Der nicht im Fokus dieser Abhandlung liegende Beweis ließe sich wohl folgendermaßen aufbauen: Leuschel und Fontaine [LF08] wandeln beliebige CSP_M-Skripte in eine einfache Zwischendarstellung um. Diese lässt sich mit moderatem Aufwand in ein CSP_M-Skript aus dem Fragment zurückübersetzen. Ein solches Rückübersetzungsprogramm wurde vom Autor dieser Abhandlung entwickelt. Zu beweisen wäre, dass sowohl die Umwandlung von Leuschel und Fontaine als auch die Umwandlung zurück in das Fragment Semantik-erhaltend sind.

lassen, ist dies nicht der Haupteinsatzzweck der Sprache. Stattdessen wird CSP_M dazu genutzt, Eigenschaften von Modellen zu ermitteln, die in CSP_M geschrieben sind. (Siehe Scattergood und Armstrong [SA11].) Dazu werden zum Teil kleine Prozesse definiert, die als Spezifikation dienen. Die Idee ist dann beispielsweise, dass der Spezifikationsprozess die Gesamtmenge (oder eine ausreichende Teilmenge) des erlaubten beziehungsweise sicheren Verhaltens aufweist, und dass überprüft wird, ob das Modell (das wiederum eine konservative Approximation des tatsächlichen Systems sein kann) nur eine Teilmenge des Verhaltens des Spezifikationsprozesses aufweist. (Vergleiche Hoare [Hoa04].)

7.2.1 Einsatzbeispiel

Das im letzten Abschnitt erläuterte Vorgehen lässt sich anhand von Abbildung 7.2 (Seite 99) erläutern. Angenommen, es ist ein Aufzug zu programmieren. Dieser folgt einem wohl überdachten Algorithmus.

Die Eingaben des Algorithmus sind die Benutzereingaben, die, wie bei Aufzügen üblich, den Aufzug in einer bestimmten Etage anfordern oder den Wunsch symbolisieren, in einer bestimmten Etage auszusteigen.

Die Ausgaben sind, den Aufzug nach oben oder unten zu bewegen sowie die Türen zu öffnen oder zu schließen.

Eine sinnvolle Sicherheitsanforderung könnte dabei sein, den Aufzug nur zu bewegen, wenn die Türen geschlossen sind. Eine dazu passende Spezifikation ist `SPEC1` in der Abbildung. Davon ausgehend, dass die Türen zu Beginn geschlossen sind, kann der Prozess jede Abfolge von Signalen aussenden, die die Regel befolgen, nicht mit geöffneter Türe zu fahren. Der Aufzug kann nach Belieben auf- und abwärts fahren und die Tür öffnen. Wenn aber die Tür geöffnet wurde, muss er sie zunächst schließen, um wieder in der Ausgangszustand zurückzukehren.

Um die Erfüllung der Sicherheitsanforderung nachzuweisen, könnte das tatsächliche System in einem Prozess »`MODEL`« konservativ abschätzend modelliert werden und anschließend könnte ein Refinement-Checker befragt werden, ob das Verhalten des Modells eine Teilmenge des Verhaltens der

Spezifikation ist. Dazu würde man im Modell alle Signale außer den für die Prüfung relevanten (also in diesem Fall `up`, `down`, `open` und `close`) mittels Verdeckungs-Operatoren verstecken.

Auch hinter `SPEC2` versteckt sich ein ähnliches Sicherheitskriterium. Angenommen, der Aufzug hat zwei Türen: eine vorne und eine hinten. Dann könnte eine Sicherheitsanforderung lauten, dass die hintere Türe sich nur in den Stockwerken mit gerader Nummer öffnet, weil sich nur dort ein Ausgang befindet. Genau dieses Verhalten wird von `SPEC2` erlaubt. (Es wird hier davon ausgegangen, dass der Aufzug in einem Stockwerk mit gerader Nummer startet.)

7.2.2 Refinement Checker

Es sind mehrere Refinement-Checker für CSP_M verfügbar. Einige werden in diesem Abschnitt kurz beschrieben.

7.2.2.1 FDR

Der lange Zeit bedeutendste Refinement-Checker ist FDR2. FDR2 wurde an der Universität Oxford entwickelt und wird nun von Formal Systems (Europe) kommerziell vertrieben. Er wird von zahlreichen Projekten verwendet. (Siehe FDR2 User Manual: [FO10].)

Während der Laufzeit dieses Vorhabens wurde FDR3 verfügbar gemacht. (Siehe Gibson-Robinson et al. [GABR14].) Die vielleicht wichtigste Änderung im Vergleich zu FDR2 ist die Nutzung von mehreren Recheneinheiten zum Refinement-Checking (Multicore-Support).

FDR3 wird, während der Verfassung dieser Abhandlung weiterentwickelt, und es werden wiederholt neue Versionen veröffentlicht. Die neuesten Versionen tragen den Name »FDR4«. (Vergleiche Webmanual des FDR-Projektes: Gibson-Robinson et al. [GABR13].)

Zusätzlich zu FDR2 und FDR3 existiert auch FDR. Dieser stand jedoch nicht zur Verfügung.

7.2.2.2 ProB

Forscher der Universität Düsseldorf haben ein Werkzeug mit dem Namen »ProB« entwickelt. (Vergleiche Leuschel und Fontaine [LF08].) Auch bei diesem wurden während der Laufzeit dieses Projektes wiederholt neue Versionen veröffentlicht.

ProB kann nicht nur CSP_M verarbeiten, sondern akzeptiert auch Eingaben in B und LTL (Linear Temporal Logic). Dabei ist es sogar möglich, Modelle, die in der einen Sprache geschrieben sind, gegen Spezifikation zu prüfen, die in einer anderen Sprache geschrieben sind. Dabei hat ProB zu Verarbeitung von CSP_M einen ähnlichen Funktionsumfang wie die unterschiedlichen Versionen der FDR-Refinement-Checker.

7.2.2.3 Andere CSP_M-Werkzeuge

Es existieren weitere Werkzeuge zum Refinement-Checking. Diese wurden aber im Rahmen der Arbeit zu dieser Abhandlung nicht näher untersucht.

Dazu gehören der Adelaide Refinement Checker (vergleiche Esser [Ess02]), geschrieben in Java und zur Verfügung gestellt von der Universität Adelaide. Der Adelaide Refinement Checker ist allerdings, anders als sein Name vermuten ließe, kein Refinement-Checker, sondern ein Werkzeug, das ausschließlich zum Debuggen von CSP_M dient.

Auf der Webseite [Ros11a] zu seinem Buch »Understanding Concurrent Systems« listet Roscoe noch einige weitere CSP_M-Werkzeuge auf.

7.2.3 Andere Werkzeuge

Zusätzlich zu den Refinement-Checkern wurden für die Forschung zu dieser Abhandlung die folgenden Werkzeuge verwendet:

- *cspmchecker*: Ein Syntax- und Typ-Checker für CSP_M, der mit FDR2 mitgeliefert wird.
- *ProBE CSP Animator*: Ein graphisches Werkzeug, das CSP_M-Prozesse in Baumstrukturen darstellt und es erlaubt, ihr Verhalten zu explo-

rieren. Der ProBE CSP Animator ermöglicht auch die Suche in diesen Baumstrukturen.

- Sowohl FDR3 als auch ProB haben ähnliche Funktionalitäten wie die beiden erstgenannten Werkzeuge eingebaut. Beim Debuggen der CSP_M-Modelle half, dass diese Werkzeuge andere, teilweise deutlich besser verständliche Fehlermeldungen lieferten als cspmchecker oder FDR2.
- *TCL-API von FDR2*: FDR2 wird mit einer für die Scriptsprache TCL ausgelegten Programmierschnittstelle ausgeliefert. Um die Arbeitsweise von FDR2 zu untersuchen und um bestimmtes Verhalten von FDR2 nachvollziehen zu können, ist es unerlässlich, sich mit dieser Programmierschnittstelle auseinanderzusetzen. Die in dieser Abhandlung abgebildeten generalisierten linearen Transitionssysteme wurden zum Teil mittels dieser Programmierschnittstelle erhoben. (Freitas und Woodcock [FW07] beschreiben diese Programmierschnittstelle in einem Reverse-Engineering-Paper.)

7.3 CSP-Semantik-Modelle

Damit Refinement-Checker entscheiden können, was genau das Verhalten eines Prozesses ist, muss die Semantik der Operatoren und Grundprozesse klar definiert sein.

Dazu wurden für CSP mehrere sogenannte Semantik-Modelle entwickelt. Diese definieren jeweils eine formale Semantik für CSP.

7.3.1 Algebraische Semantik

Hoare [Hoa04] listet für die vordefinierten Prozesse und Operatoren sogenannte Gesetze auf. Beispiele dafür sind die in Abbildung 7.3 dargestellten Gesetze L2 – L4 des Operators $|\sim|$.

Mit diesen Gesetzen lässt sich eine algebraische Semantik für CSP definieren. Darin sind genau die Prozesse gleich, die sich durch die Gesetze ineinander umformen lassen. Das Ergebnis ist aber unbefriedigend und un-

It does not matter in which order the choice is presented

L2 $P \sqcap Q = Q \sqcap P$ (symmetry)

A choice between three alternatives can be split into two successive binary choices. It does not matter in which way this is done

L3 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ (associativity)

The occasion on which a nondeterministic choice is made is not significant. A process which first does x and then makes a choice is indistinguishable from one which first makes the choice and then does x

L4 $x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q)$ (distribution)

Ausschnitt aus Hoare [Hoa04] (Seite 83)

Abbildung 7.3: Die Gesetze (Laws) L2 – L4 zum Nondeterministic-Choice-Operator |~|. Insgesamt listet Hoare [Hoa04] acht Gesetze auf.

terscheidet Prozesse, die nicht unterschieden werden sollen. (Vergleiche Hoare [Hoa04].)

In der Praxis und für die Refinement-Checker ist diese Semantik kaum relevant. Die Gesetze gelten aber in allen weiteren hier angegebenen Semantik-Modellen, das heißt, sind zwei Prozesse algebraisch gleich, so werden sie auch durch die folgenden Semantik-Modelle zu gleicher Semantik abgebildet. (Vergleiche Hoare [Hoa04].)

7.3.2 Traces

Ein CSP-Trace (hier kurz als Trace bezeichnet) ist eine endliche oder unendliche Folge von Signalen. Traces sind die Grundlage des Traces-Modells. Das Traces-Modell ist die wohl einfachste Möglichkeit, eine denotationale Semantik für CSP zu definieren. Die Traces eines Prozesses werden über der Struktur der Prozesse definiert. Sie werden als die Menge aller endlichen Präfixe von Traces angesehen, welche die an die Umgebung ausgesendeten Signale des Prozesses umfassen.

Für die Operatoren und vordefinierten Prozesse lässt sich leicht eine Semantik auf den Traces definieren. So gilt zum Beispiel $Tr(\texttt{STOP}) = \emptyset$ und $Tr(\texttt{P} \ |\sim| \ \texttt{Q}) = Tr(\texttt{P}) \cup Tr(\texttt{Q})$. In Abbildung 7.4 sind die Traces von weiteren Prozessen angegeben.

Das Traces-Modell setzt dabei viele Prozesse gleich, die andere semantische Modelle unterscheiden. So sind die Prozesse P und Q aus Abbildung 7.4 im Traces-Modell identisch, während alle anderen hier besprochenen semantischen Modelle sie unterscheiden.

Das Traces-Modell eignet sich dabei besonders, um Sicherheitsaspekte nachzuweisen, das heißt zu zeigen, dass »nichts Schlimmes« passieren kann.

7.3.3 Stable Failures

Das Failures-Modell, auch als »Stable-Failures-Modell« bezeichnet, erlaubt genauere Unterscheidungen von Prozessen. Hier werden den Traces Mengen von Signalen zur Seite gestellt, deren Aussendung der Prozess nach dem Trace verweigern kann.

Ein solches Tupel aus einem endlichen Trace und einer Menge an Signalen wird als »refusal« bezeichnet.

Betrachtet wird dazu Abbildung 7.4. Zu Beginn, bevor überhaupt ein Signal ausgesendet wurde, können (und werden) die Prozesse P und Q beide das Aussenden des Signals b ablehnen. Dies zeigt sich an dem sogenannten »Refusal« $\langle \epsilon, \{\texttt{b}\}\rangle$, das ein Element von beiden Failures ist.

Vielschichtiger ist, was beim Trace a passiert. Das Refusal $\langle \texttt{a}, \{\texttt{b}\}\rangle$ ist Element der Failures von Prozess P. Das bedeutet, dass sich der Prozess P, nachdem er das Signal a ausgesendet hat, weigern kann, das Signal b auszusenden. Die Failures des Prozesses Q enthalten dieses Refusal hingegen nicht. Der Prozess Q muss, nachdem er das Signal a ausgesendet hat, schlussendlich auch das Signal b aussenden. Die Prozesse P und Q unterscheiden sich also nicht darin, was sie tun können, sondern darin, was sie zu tun ablehnen können. Je nachdem, ob sich der Prozess P für die linke oder die rechte Seite seines Auswahl-Operators entscheidet, folgt auf das Signal a das Signal b oder eben nicht.

Seien die Prozesse P, Q und R wie folgt definiert über dem Alphabet Σ:

```
P = a -> STOP |~| a -> b -> STOP
Q = a -> b -> STOP
R = a -> (STOP |~| b -> STOP)
```

$$\Sigma = \{a; b\}$$

Man erkennt, dass sich P und R mittels dem Gesetz L4 des Nondeterministic-Choice-Operators (siehe Hoare [Hoa04] beziehungsweise Abbildung 7.3) ineinander umformen lassen. Ihre Traces, Failures und Divergences sind also gleich, da alle drei Modelle die algebraischen Gesetze respektieren.

Die Prozesse P und Q haben die folgenden Traces, Failures und Divergences:

$$Tr(P) = \{\epsilon; a; ab\} = Tr(Q)$$

$$\begin{aligned} F_\Sigma(P) = {} & \{\langle \epsilon, \emptyset\rangle; \langle \epsilon, \{b\}\rangle\} \cup \\ & \{\langle a, \emptyset\rangle; \langle a, \{a\}\rangle; \langle a, \{b\}\rangle; \langle a, \{a; b\}\rangle\} \cup \\ & \{\langle ab, \emptyset\rangle; \langle ab, \{a\}\rangle; \langle ab, \{b\}\rangle; \langle ab, \{a; b\}\rangle\} \end{aligned}$$

$$\begin{aligned} F_\Sigma(Q) = {} & \{\langle \epsilon, \emptyset\rangle; \langle \epsilon, \{b\}\rangle\} \cup \\ & \{\langle a, \emptyset\rangle; \langle a, \{a\}\rangle\} \cup \\ & \{\langle ab, \emptyset\rangle; \langle ab, \{a\}\rangle; \langle ab, \{b\}\rangle; \langle ab, \{a; b\}\rangle\} \end{aligned}$$

$$Div_\Sigma(P) = \emptyset = Div_\Sigma(Q)$$

Abbildung 7.4: Vergleich der Semantik von Beispielprozessen in unterschiedlichen semantischen Modellen

Deutlich wird, dass Prozesse, die im Failures-Modell gleich sind, auch stets im Traces-Modell gleich sind, der Umkehrschluss aber nicht zutrifft.

Das Failures-Modell eignet sich besonders, um Deadlocks-Freiheit zu demonstrieren.

7.3.4 Failures/Divergences

Im letzten betrachteten Modell, dem Failures-Divergences-Modell, kommen zu den Failures noch die Divergences hinzu. Dabei wird ein Prozess als divergent bezeichnet, wenn er einer unbegrenzten Anzahl an internen Zustandsübergängen unterliegen kann, bevor er wieder ein Signal an seine Umgebung aussendet.

Divergenz gilt dabei als unerwünschte Eigenschaft eines Prozesses. Es gibt einen engen Zusammenhang[1] zwischen Live-Locks und Divergenz. Das Modell setzt alle divergenten Prozesse gleich. Es gilt nicht, dass Prozesse, die im Failures-Divergences-Modell gleich sind, auch im Failures-Modell oder im Traces-Modell gleich sind.

Die in Abbildung 7.4 dargestellten Prozesse divergieren nicht. Die Divergences sind somit stets die leere Menge.

Das Failures-Divergences-Modell eignet sich, um Probleme mit Live-Locks, Liveness und Fairness auszuschließen. Im Failures-Divergences-Modell lässt sich ermitteln, ob ein Prozess stets Fortschritt erzielen wird.

7.4 Generalisierte kantenbeschriftete Transitionssysteme

Generalisierte kantenbeschriftete Transitionssysteme sind Graphen, deren Kanten mit Signalen oder dem griechischen Buchstaben τ beschriftet sind. Sie dienen manchen Werkzeugen als Zwischendarstellung zur maschinellen Verarbeitung von CSP_M. Sie können als Verallgemeinerung von nichtdeterministischen endlichen Automaten gesehen werden. Dabei entsprechen

[1] Ein Prozess ist in einem Live-Lock gefangen, wenn er zwar immer weiter »arbeitet«, aber keinen Fortschritt mehr erzielt. Um einen CSP-Prozess auf Live-Locks zu untersuchen, können alle Signale versteckt werden, die keinen Fortschritt darstellen. Wenn dann kein divergierender Zustand erreicht werden kann, ist der Prozess frei von Live-Locks.

die τ-Transitionen den ϵ-Übergangen in nicht-deterministischen endlichen Automaten.

Jeder CSP_M-Prozess kann als generalisiertes kantenbeschriftetes Transitionssystem mit einem definierten Startzustand dargestellt werden. Für das Failures-Modell und das Failures-Divergences-Modell werden komplexere generalisierte kantenbeschriftete Transitionssysteme benötigt, für das Traces-Modell reichen die hier beschriebenen einfachen generalisierten kantenbeschrifteten Transitionssysteme aus. Der Zusammenhang der Traces mit dem generalisierten kantenbeschrifteten Transitionssystem ist einfach: Die Menge der Traces ist genau die Menge der auf Signale reduzierten endlichen Pfade im generalisiertem kantenbeschrifteten Transitionssystem. Die τ-Transitionen sind dabei stille Übergänge.

Die später eingesetzten Refinement-Checker FDR2 und FDR3 wandeln CSP_M-Prozesse in generalisierte kantenbeschriftete Transitionssysteme um, um Fragen zu ihren Eigenschaften zu beantworten.

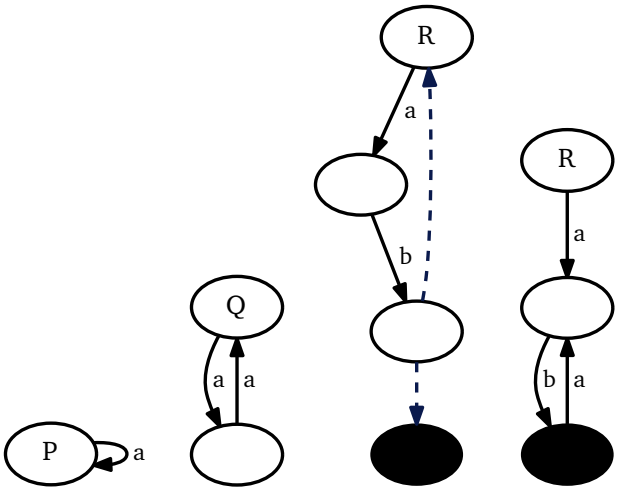
7.4.1 Kompression

Insbesondere generalisierte kantenbeschriftete Transitionssysteme, die viele τ-Transitionen enthalten, lassen sich komprimieren. Komprimieren bedeutet, dass der Refinement-Checker die Transitionssysteme verkleinert, ohne die Semantik, das heißt die aus ihnen hervorgehenden Traces, zu verändern.

Eine der aufwendigsten Kompressionen, die FDR2 durchführen kann, ist die »Normalisierung«. Die Normalisierung *»kann teuer sein und das System sogar vergrößern«*. (Siehe Reference Manual von FDR2 [FO10].)

Bei der Normalisierung handelt es sich, zumindest wenn das Traces-Modell zum Einsatz kommt, um die Komposition der Potenzmengenkonstruktion zur Umwandlung eines nicht-deterministischen in einen deterministischen endlichen Automaten und der Minimierung endlicher Automaten. (Vergleiche Schöning [Sch09], dort beschrieben als Potenzmengenkonstruktion nach dem Satz von Rabin und Scott – Seite 24 – und DFA-Minimierung nach dem Satz von Myhill und Nerode – Seite 38.) Roscoe [Ros11b] erläutert auch die komplexeren Verfahren, die bei anderen Modellen eingesetzt werden.

```
1  channel a, b
2
3  P = a -> P
4  Q = a -> a -> Q
5  R = a -> b -> (SKIP |~| R)
```



Oben: Eine kleine CSP$_M$-Datei, die die Prozesse `P`, `Q` und `R` definiert.

Unten: Generalisierte kantenbeschriftete Transitionssysteme, die von FDR3 erzeugt wurden (zum Teil normalisiert).

Dabei wurde ganz links, links der Mitte und rechts der Mitte jeweils auf Normalisierung verzichtet.

Das generalisierte kantenbeschriftete Transitionssystem von `P` ändert sich durch Normalisierung nicht. Das von `Q` wird zu dem von `P` (ganz links). Das von `R` wird kleiner (ganz rechts).

Abbildung 7.5: Einige CSP$_M$-Prozesse und zugehörige generalisierte kantenbeschriftete Transitionssysteme

7.4.2 Darstellung

Abbildung 7.5 zeigt einige generalisierte kantenbeschriftete Transitionssysteme. Knoten, die erfolgreicher Terminierung unterliegen können, sind schwarz ausgefüllt. Die gestrichelten Kanten sind τ-Transitionen. Die Knotenbeschriftungen dienen hier nur der Verdeutlichung der dargestellten Prozesse. Normalerweise sind die Knoten nicht beschriftet.

Die Tatsache, dass sich die generalisierten kantenbeschrifteten Transitionssysteme der Prozesse `P` und `Q` zu demselben generalisierten kantenbeschrifteten Transitionssystem komprimieren lassen, bedeutet insbesondere, dass sie im Traces-Modell identisch sind.

Teil II

Ansatz und Umsetzung

Kapitel 8

Vision und Weichenstellung

Die vorangehenden Kapitel haben Data-Races definiert und erklärt, dass sie mit statischer Software-Analyse gefunden werden können. Es wurde motiviert, dass gerade konservative Analysen sinnvoll sind und dass bei diesen die Klassifizierung der Data-Race-Warnungen nötig ist. Werden die geschilderten Verfahren auf eingebettete Systeme aus Automobilen angewendet, stößt der Anwender dabei oft auf explizite Zustandsverwaltung und es gilt die Synchronisationseigenschaften dieser expliziten Zustandsverwaltung zu ermitteln. Dazu wurden Ansätze von Keul und Schwarz et al. vorgestellt. Dieses Kapitel stellt eine Vision für ein alternatives Vorgehen zur Analyse expliziter Zustandsverwaltung vor. Aus dieser Vision wird ein Ansatz entwickelt. Die Umsetzung und Evaluation dieses Ansatzes ist der Kern dieser Abhandlung.

8.1 Vision

Es gibt eine Werkzeugkette, die explizite Zustandsverwaltung analysiert und ermitteln kann, welche Data-Race-Warnungen durch sie ausgeschlossen werden. Diese Werkzeugkette entspricht der folgenden Beschreibung.

8.1.1 Eingabe

Die Haupteingabe für die Werkzeugkette ist der vollständige Quelltext eines Programmes. Von dem Programm wird erwartet,

- dass es nebenläufig arbeitet, also mehr als einen Task enthält,
- dass es ein eingebettetes System aus Automobilen ist oder zumindest die in diesem Bereich vorherrschenden Konventionen einhält (siehe Kapitel 4) und
- dass es explizite Zustandsverwaltung enthält, deren Eigenschaften ermittelt werden sollen.

Eine weitere Eingabe ist die zum Quelltext passende Nebenläufigkeitskonfiguration des Systems. Die Analysekette benötigt diese Eingabe, da aus dem Quelltext alleine nicht alle benötigten Informationen zu Tasks hervorgehen. Die Nebenläufigkeitskonfiguration enthält Informationen zu den Einsprungspunkten von Tasks und Interrupts, sowie Schedulinginformationen, wie die Priorität der Tasks. Die Nebenläufigkeitskonfiguration enthält nur aus ohnehin vorhandenen Dokumentationsartefakten ablesbare Fakten über das System.

8.1.2 Ablauf

Die Anwendung der Werkzeugkette läuft folgendermaßen ab:

1. Die Werkzeugkette analysiert das System weitgehend automatisch. Dabei führt es auch eine Data-Race-Analyse durch. Das Ergebnis dieser Data-Race-Analyse ist eine Liste von Data-Race-Warnungen.

2. In dieser Liste der Data-Race-Warnungen identifiziert der Anwender das zu untersuchende Muster, indem er einer oder mehrere Zustandsvariablen angibt, und wählt Data-Race-Warnungen aus. Die Analysekette ermittelt dann, welche der ausgewählten Data-Race-Warnungen unter Berücksichtigung des identifizierten Musters ausgeschlossen werden.

Dabei garantiert der Benutzer durch Auswahl des Musters kein besonderes Verhalten des Programms. Die Werkzeuge dürfen insbesondere keine Annahmen darüber treffen, dass mit Zustandsvariable auf bestimmte Weise umgegangen wird. Dasselbe gilt auch für die Auswahl der Data-Race-Warnungen.

8.1.3 Ausgabe und Konservativität

Es wird für jede ausgewählte Data-Race-Warnungen das Ergebnis der Analyse ausgegeben. Bei der Berechnung soll der Fehler in eine Richtung begrenzt sein. Die Ausgabe soll wie im Folgenden beschrieben konservativ sein.

Es gibt drei mögliche Ergebnisse:

Ausschluss: Das Data-Race kann unter Berücksichtigung der explizite Zustandsverwaltung sicher ausgeschlossen werden.

Timeout: Die Frage des Ausschlusses ließ sich nicht innerhalb der gegebenen Zeitbeschränkung beantworten. Dieses Ergebnis ist unerwünscht, da es die eigentlich gewünschte Information nicht liefert. Aufgrund der Komplexität der Aufgabe ist jedoch nicht zu erwarten, dass immer eine Antwort gefunden werden kann.

Erreichbar: Eine Data-Race-Situation kann in der getroffenen Abstraktion erreicht werden. Das Muster schließt in der erkannten Ausdehnung das Data-Race nicht aus.

8.2 Laufzeitverhalten von Verifikationsmechanismen

Von Werkzeugen, die statische Software-Analyse zur Fehlererkennung einsetzen, wird in der Regel erwartet, dass jedes System, das eine für den

jeweiligen Einsatz realistische Größe hat, in annehmbarer Zeit analysiert werden kann. Dies zeigt sich zum Beispiel in einer Worst-Case-Laufzeit in $\widetilde{\mathcal{O}}\left(n^2\right)$,[1] wobei n die Größe des Quelltextes ist. Dabei wird in Kauf genommen, dass im Einzelfall die gelieferten Ergebnisse zu stark approximiert sind, um noch nützlich zu sein.

Wie lange dabei »annehmbare Zeit« ist, hängt von vielen Faktoren ab und ist letztlich auch kaum objektiv festzulegen. Bei einem interaktiven Einsatz, zum Beispiel zur Auto-Vervollständigung, mögen fünf Sekunden bereits unannehmbar wirken. Bei einem einmaligen Einsatz zur abschließenden Verifikation eines Systems, wenn Gewissheit herrscht, dass die Analyse terminiert, können hingegen auch drei Tage annehmbar sein.

Die Erwartungen an Ansätze aus dem Bereich Model-Checking beziehungsweise allgemeine Software-Verifikation sind andere. Hier gilt es oft als ausreichend zu zeigen, dass der Ansatz für manche, speziell ausgewählte Programme terminiert und dabei nützliche Informationen liefert. Dafür wird oft eine höhere Präzision geliefert; es werden teils deutlich komplexere Aufgaben behandelt. (Darstellung in Anlehnung an D'Silva, Kroening und Weissenbacher [DKW08])

8.2.1 Aufteilung in Phasen

Die Werkzeugkette ist nicht auf statische Software-Analyse beschränkt. Das Laufzeitverhalten wird daher weniger streng eingeschränkt, als dies bei anderen Werkzeugen der Fall ist. Dabei soll aber nicht bis auf das realitätsferne Niveau mancher Model-Checking-Ansätze hinabgestiegen werden.

Der hier vorgestellte Ansatz läuft in zwei Phasen ab:

- Die erste Phase setzt statische Software-Analyse ein und hat auch entsprechende Anforderungen. Hier werden eher Kompromisse bei der Präzision gemacht, als eine prohibitive Laufzeit in Kauf zu nehmen. Es wird das gesamte System auf einmal untersucht.

[1] Für Erläuterungen der mathematischen Konventionen siehe Seite 315.

- Die zweite Phase erlaubt allgemeinere Verifikationstechniken. Diese bearbeiten komplexere Fragestellungen und liefern dabei eine höhere Präzision. Diese beeinträchtigt die Laufzeit und Zuverlässigkeit: Die Algorithmen werden nicht immer in annehmbarer Zeit terminieren. Da in dieser Phase nur einzelne Warnungen bearbeitet werden, ist dieses Verhalten akzeptabel. Es wird aber zumindest bei einem gewissen Anteil der Data-Race-Warnungen erwartet, Antworten zu erhalten.

8.2.2 Anforderungen

Wie beschrieben, ist ausdrücklich eine zweite Phase erlaubt, die auf die Data-Race-Analyse und weiteren benötigten statischen Software-Analysen folgt. In dieser Phase werden, statt dem ganzen System, nur noch einzelne Data-Race-Warnungen betrachtet. In der zweiten Phase gelten losere Anforderungen an die Laufzeit als in der ersten, um mächtigere Lösungskonzepte zu erlauben.

Es werden demnach folgende Anforderungen formuliert:

- Die nach der Data-Race-Analyse noch notwendigen statischen Software-Analysen sollen, in der praktischen Anwendung, stets in annehmbarer Zeit terminieren. Die Laufzeit gilt jedenfalls als annehmbar, wenn sie geringer ist als die der Data-Race-Analyse.
- Die Analysen müssen insbesondere mit dem in der verwendeten Hardware verfügbaren Hauptspeicher auskommen.
- In der zweiten Phase ist die Laufzeit nicht mehr strikt begrenzt. Der Ansatz soll bei manchen Data-Race-Warnungen in annehmbarer Zeit terminieren. Es kann dabei eine Zeitspanne definiert werden, nach deren Ablauf das System abbricht und das Ergebnis Timeout liefert.

Es sollen sich somit Systeme realistischer Größe prinzipiell analysieren lassen. An den Teil der Analyse, der für alle Data-Race-Warnungen benötigt wird, werden also strikte Laufzeitanforderungen gestellt. Wenn dieser nicht terminiert, stehen keine Ergebnisse zur Verfügung. Wenn sich für einzelne Data-Race-Warnungen in Kombination mit bestimmten Zustandsvariablen nicht immer ein Ergebnis berechnen lässt, ist dies weniger problematisch.

Dabei bleibt das Interesse des Nutzers im Blick. Bei einer Vielzahl an Warnungen ist jede Antwort wertvoll und man ist nicht darauf angewiesen, immer eine Antwort zu erhalten.

8.3 Architekturansatz

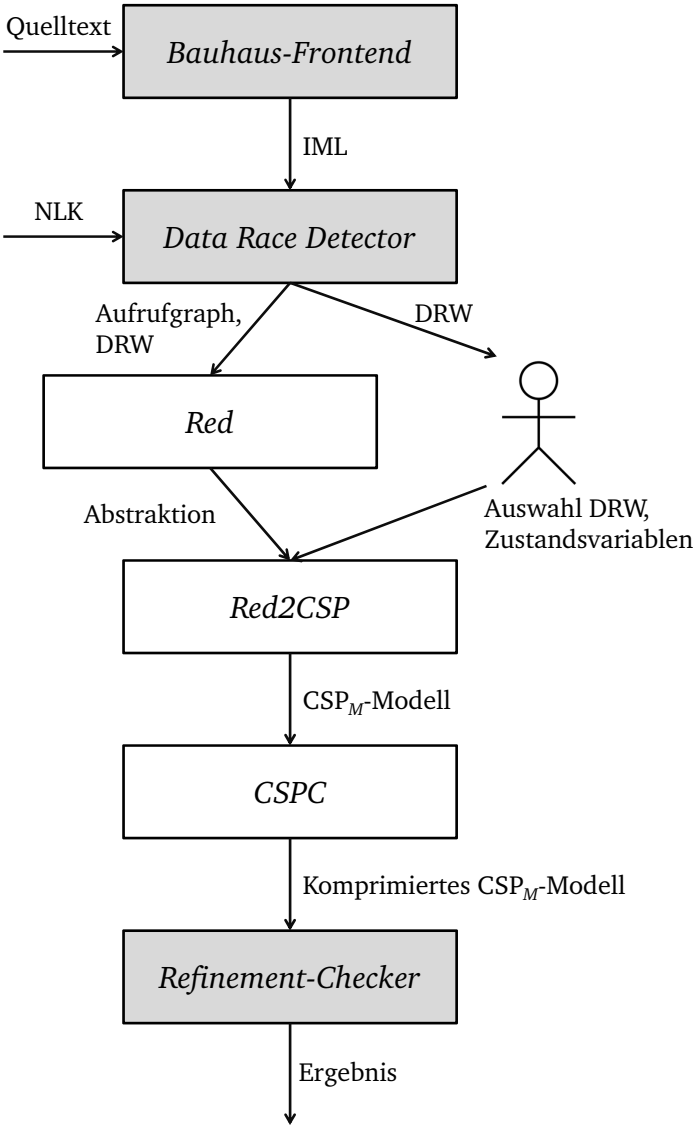
Die hier entwickelte Werkzeugkette soll Teil der am Institut des Autors entwickelten Werkzeugsammlung Bauhaus sein. In dieser existiert bereits die Implementierung einer Data-Race-Analyse mit der Bezeichnung Bauhaus Data Race Detector. Auf dieser soll hier aufbaut werden. Abbildung 8.1 zeigt den Aufbau der Werkzeugkette; die folgenden Abschnitte erläutern die Aufgabe der einzelnen Werkzeuge. Die folgenden Kapitel werden die von dem Autor dieser Abhandlung selbst erstellten Werkzeuge (die weißen Kästchen in Abbildung 8.1) beschreiben.

8.3.1 Anknüpfungspunkt Bauhaus Data Race Detector

Der Bauhaus Data Race Detector nutzt das gemeinsame Bauhaus-Frontend für die Sprache C. Dieses parst C-Code, führt einfache Analysen durch und liefert einen mit semantischen Informationen annotierten abstrakten Syntaxbaum in der Bauhaus Intermediate Language.

Zusätzlich liest der Bauhaus Data Race Detector die Nebenläufigkeitskonfiguration ein und führt zahlreiche statische Software-Analysen durch. Ergebnis davon sind zum einen die Liste der Data-Race-Warnungen (und einige weitere Daten, die aus dieser Liste abgeleitet werden können wie die Liste aller Variablen, die von Data-Race betroffen sind) und zum anderen eine weitere angereicherte Zwischendarstellung wiederum im Format der Bauhaus Intermediate Language. Die umfangreichste öffentlich zugängliche Beschreibung zum Bauhaus Data Race Detector findet sich in Keul [Keu11].

Die angereicherte Zwischendarstellung ist eine natürliche Wahl als Anknüpfungspunkt für die Arbeiten im Rahmen dieser Abhandlung. Sie enthält nicht nur alle hier wesentlichen Informationen aus dem Quelltext, sondern



IML: Bauhaus Intermediate Language, NLK: Nebenläufigkeitskonfiguration, DRW: Liste der Data-Race-Warnungen

Abbildung 8.1: Werkzeugkette für die Analyse von expliziter Zustandsverwaltung

ist bereits mit Analyseergebnissen zur Synchronisation und Nebenläufigkeit angereichert.

An dieser Stelle sei daran erinnert, dass der Bauhaus Data Race Detector konservativ ist. Naheliegenderweise ist dabei nicht nur die Ausgabe der Data-Race-Warnungen konservativ, sondern auch die angereicherte Zwischendarstellung. So enthält der Aufrufgraph beispielsweise bei Prozeduraufrufen über die Dereferenzierung von Funktionszeigern eine konservativ abgeschätzte Liste an Zielen.

Auch wenn hier auf ein spezifisches Werkzeug aufgebaut werden soll, ließe sich der Ansatz wohl auch auf einer anderen Data-Race-Analyse aufbauen, sofern diese ähnliche Analyseergebnisse zur Verfügung stellen kann.

8.3.1.1 Abgrenzung Definition

Die in dieser Abhandlung formulierte und verwendete Definition eines Data-Race »passt« zum Bauhaus Data Race Detector. Soweit dem Autor bekannt, hat Keul nie eine vollständige und präzise Definition eines Data-Races veröffentlicht. Aus seiner vorliegenden Implementierung geht jedoch hervor, dass er wohl einer ähnlichen oder im Wesensgehalt identischen Definition gefolgt ist. Jedenfalls warnt Keul (abgesehen von kleinen Einschränkungen, siehe Abschnitt 3.5.1 (Seite 52)) vor allen Zugriffspaaren, die nach der hier verwendeten Definition ein Data-Race sind.

Ohne dass auf eine Publikation verwiesen werden kann, sind somit Teile der Ideen, die hinter der hier angegebenen Definition stehen, Keul zuzusprechen. Der Autor der Abhandlung vereinnahmt jedoch die Erstellung der präzisen Definition für sich.

8.3.2 Red

Das erste selbst entwickelte Werkzeug der Analysekette heißt »Red«. Red führt auf der angereicherten Zwischendarstellung eigene statische Software-Analysen durch. Es wird sich zum Beispiel als notwendig erweisen, Konstantenpropagierung und -faltung durchzuführen. Hauptaufgabe des Werkzeuges

ist aber Abstraktion und Reduktion der Daten auf das für spätere Analysen notwendige Maß.

Dieser Zwischenschritt der Reduktion der Datenmenge erscheint vielleicht zunächst widersinnig. Können in der Kette folgende Werkzeuge für sie überflüssige Daten nicht schlicht ignorieren?

Tatsächlich hat der Autor zunächst einen Ansatz verfolgt, der daraus bestand, die Bauhaus Intermediate Language weiter anzureichern. Das Bild, das sich dem Entwickler des folgenden Werkzeuges damit bot – eine Mischung aus grob einem Dutzend Programmierschnittstellen, entwickelt in zwei Jahrzehnten von diversen Programmierern mit unterschiedlicher Qualifikation – erschien schlicht zu komplex, um vertrauenswürdige Software zu entwickeln.

Daher wurde auf folgende Weise verfahren. Das Werkzeug Red reduziert per Whitelisting, nachdem es die statischen Software-Analysen durchgeführt hat, die zur Verfügung stehenden Informationen auf das gerade benötigte Maß. Wenn bei der Entwicklung folgender Werkzeuge weitere Details benötigt wurden, musste zunächst Red so angepasst werden, dass diese nicht der Reduktion anheim fielen. Die bei diesem Prozess gemachten Erfahrungen haben weitere Forschung am Institut des Autors inspiriert – siehe beispielsweise Felden und Wittiger [FW16].

8.3.2.1 Alternative Festlegung der Zustandsvariablen

Entgegen der hier dargestellten Architektur ist auch beim Werkzeug Red die Möglichkeit implementiert, sich bereits auf wenige oder nur eine Zustandsvariable zu beschränken. Beim Aufruf der folgenden Werkzeuge in der Werkzeugkette werden dann keine Zustandsvariablen genauer spezifiziert, sondern alle in der Datei vorhandenen verwendet. Dieses Vorgehen hat sich in der Praxis bewährt und wird auch teilweise bei den im Ergebnisteil beschriebenen Experimenten angewendet. Dabei wird somit ein Durchlauf des Werkzeuges Red pro Muster, das heißt üblicherweise ein Durchlauf pro Zustandsvariable, durchgeführt. Dieses Vorgehen sorgt für deutlich übersichtlichere Ausgabedateien.

8.3.3 Formale Sprache für Modelle

Angesichts der Erfolge, die sich durch den praktischen Einsatz von Solvern für Constraint-Sprachen und diverse Logiken erzielen ließen, sollte ein ähnlicher Weg beschritten werden.

8.3.3.1 Motivation des Einsatzes eines Solvers

Nachdem statische Software-Analysen die notwendigen Informationen zusammengetragen haben, ist es möglich, die immer noch schwierige Frage, ob explizite Zustandsverwaltung ein bestimmtes Data-Race ausschließt, in einer formalen Sprache zu formulieren und diese dann an einen passenden »Solver«[1] weiterzuleiten.

Bei diesem Vorgehen lässt sich genau das in den vorherigen Abschnitten geforderte Laufzeitverhalten erreichen. Die statischen Software-Analysen terminieren zuverlässig auch bei Systemen realistischer Größe. Der eigentliche Solver hingegen wird mit einem schwierigen oder gar unentscheidbaren Entscheidungsproblem konfrontiert, für das er manchmal eine Antwort findet und manchmal auch nicht.

Als formale Sprache wurde CSP beziehungsweise CSP_M ausgewählt. Der Beschreibung dieser Sprache ist das Kapitel 7 gewidmet.

8.3.3.2 Umgang mit dem Ergebnis des Solvers

Im hier verwendeten Ansatz soll das Ergebnis des Solvers relativ direkt dem Anwender kommuniziert werden. Dazu ist nur eine sprachliche Anpassung der Ausgabe nötig. Das in Abschnitt 13.1.1 (Seite 222) erörterte Framework, das die Werkzeuge der Werkzeugkette ausführt, muss nur textuelle Ersetzungen an der Ausgabe der Refinement-Checker vornehmen.

[1] Hier wird Solver als Überbegriff für eine nicht genau eingrenzbare Klasse von Programmen verwendet, die Eigenschaften von Konstrukten ermitteln, die durch formale Sprachen spezifiziert werden. Dabei kann an Sat-Checker für Aussagenlogik, LTL oder andere Logiken, Constraint-Solver für Linear Programming oder Integer Programming oder Model-Checker für Sprachen wie B gedacht werden – oder eben an Refinement-Checker für CSP_M.

So ersetzt es zum Beispiel »`... *is not* a trace-refinement ...`« durch »`Erreichbar`«.

8.3.3.3 Red2CSP

Um den Einsatz von Refinement-Checkern umzusetzen, wurde ein Werkzeug entwickelt, das CSP_M-Code erzeugt, nachdem Red die Datenmenge reduziert hat. Das Red2CSP genannte Werkzeug folgt einem klaren Konzept zur konservativen Modellierung des ursprünglichen Programms in CSP_M. Eine wichtige Eigenschaft der Architektur ist dabei, dass der Ansatz an dieser Stelle Data-Race-warnungsspezifisch wird. Es wird für jede zu untersuchende Warnung eine eigene CSP_M-Datei generiert.

8.3.4 Komprimierung des Modells

Die von Red2CSP generierten Dateien können bei kleinen Beispiel-Programmen direkt als Input für Refinement-Checker verwendet werden. Es hat sich allerdings gezeigt, dass die Refinement-Checker mit der Ausgabe, die für große Programme erzeugt wird, nur schlecht zurechtkommen. Das Verhalten bei großen Programmen lässt sich verbessern, indem das CSP_M-Modell Semantik-erhaltend komprimiert wird. Dazu wurde das Programm CSPC (von: »CSP Compressor«) entwickelt.

Kapitel 9

Das Werkzeug Red

Dieses Kapitel beschreibt die Implementierung des Werkzeuges Red, dessen Aufgabe es ist, die benötigten Datenflussanalysen (vor allem eine kombinierte Konstantenpropagierung und -faltung) durchzuführen und die für die CSP_M-Generierung nötigen Informationen zusammenzutragen. Es bietet außerdem Funktionen, die dem Anwender bei der Auswahl von Zustandsvariablen zur Seite stehen. Ferner abstrahiert und sequentialisiert das Werkzeug Red die bestehende Zwischendarstellung. Dabei werden auch die Pfadprädikate analysiert und behandelt.

9.1 Ausgangssituation

Die in dieser Abhandlung zugrunde gelegte Analyse (der Bauhaus Data Race Detector) stellt Datenstrukturen zur Verfügung, auf denen im Folgenden aufgebaut wird. Bei den Datenstrukturen handelt es sich zum einen um einen Task- und synchronisationssensitiven Aufrufgraphen, der das gesamte Programm erschließt, und zum anderen um einen Mechanismus, um die Data-Race-Warnungen zu erhalten.

Außerdem werden wertvolle Zwischenergebnisse der Data-Race-Analyse

zur Verfügung gestellt: zum Beispiel ein Mechanismus zur Auflösung von Zeigerzielmengen und Speicherbereichen und eine Datenstruktur, die die Charakterisierung von Variablen von »local« bis »communication« erlaubt.

9.1.1 Liste der Data-Race-Warnungen

Als Eingabe für das Werkzeug Red liegt eine Zwischendarstellung in der Bauhaus Intermediate Language vor. Darin ist die Liste der Data-Race-Warnungen nicht explizit gespeichert. Es werden vielmehr asymptotisch kleinere Datenstrukturen angelegt und ein Modul zur Verfügung gestellt, das daraus die gesamte Liste der Data-Race-Warnungen erzeugt.

Dieses Modul wurde im Rahmen der Implementierung angepasst. Dabei wurden die folgenden Änderungen vorgenommen:

- Die Data-Race-Warnungen werden nun stets in derselben Reihenfolge generiert. Die Reihenfolge war zuvor nicht-deterministisch.
- Jede Data-Race-Warnung erhält eine eindeutige ID.
- Werden in dem untersuchten System Prioritäten eingesetzt, so steht in dem Zugriffspaar stets der Zugriff vorne, der die niedrigere Priorität hat – sofern die Prioritäten nicht gleich sind.

Die Änderungen erleichtern Vergleiche von Ausgaben von verschiedenen Werkzeugen. Sie helfen außerdem dabei, die spätere Ausgabe einfach und effizient zu gestalten.

9.1.2 Aufrufgraph

Die wichtigste von der Implementierung genutzte Datenstruktur ist ein Task- und synchronisationssensitiver Aufrufgraph bzw. Kontrollflussgraph.

Der Aurufgraph ist ein bipartiter Graph. Die Knoten sind zum einen die Prozeduren (beziehungsweise Funktionen), zum anderen die Prozeduraufrufe. Prozeduren verweisen auf die in ihnen enthaltenen Prozeduraufrufe. Die Prozeduraufrufe verweisen wiederum auf die Prozeduren, die sie aufrufen.

Verfeinert wird dieser Graph durch die Task- und Synchronisationssensitivität. Die Arbeitsweise der Verfeinerung zeigt sich in dem im Folgenden besprochenen Beispiel.

Ein Prozeduraufruf kann auf mehrere Prozeduren verweisen, wenn er auf der Dereferenzierung eines Funktionszeigers basiert. In diesem Fall wird die Zielmenge konservativ approximiert.

Die Einsprungspunkte von Tasks werden in dem normalerweise unzusammenhängenden Graphen gesondert ausgewiesen.

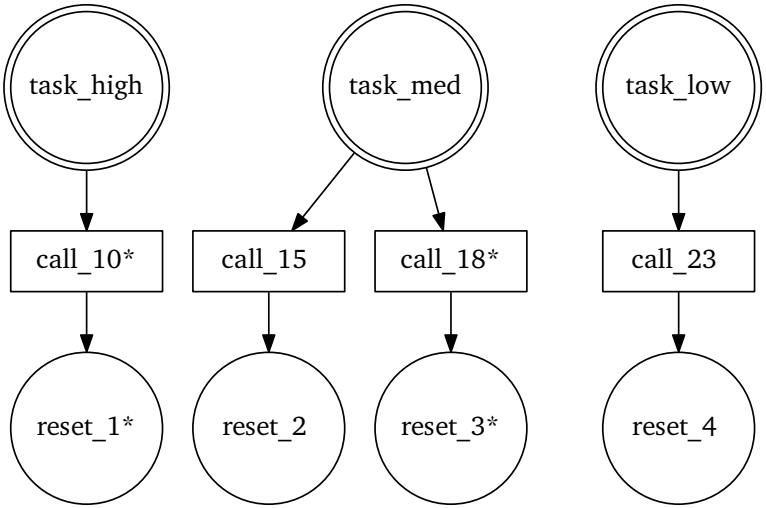
9.1.2.1 Beispiel

Betrachtet wird das Beispielprogramm aus Abbildung 2.7 (Seite 39). Es sei daran erinnert, dass die Prozedur `Reset` aus unterschiedlichen Kontexten heraus aufgerufen wird – alle vier Aufrufe sind dabei zu unterscheiden.

In Abbildung Abbildung 9.1 ist der Aufrufgraph zu diesem Beispiel dargestellt. In der Abbildung lässt sich erkennen, wie die Tasksensitivität des Aufrufgraphen dafür sorgt, dass er unzusammenhängend ist. Es ist sichergestellt, dass es keine Kanten zwischen den einzelnen Aufrufgraphen der Tasks gibt.

Ferner erkennt man, dass Prozeduren auch nach Synchronisationskontext unterschieden werden. Dazu wird der Task mit mittlerer Priorität betrachtet. Die beiden Aufrufe aus den Zeilen 15 und 18 verweisen auf unterschiedliche Knoten, da sich die Synchronisationseigenschaften an den Aufrufstellen unterscheiden: einmal wird ein Lock gehalten, einmal nicht.

Alle Prozeduraufrufknoten haben hier genau eine ausgehende Kante, da das Aufrufziel hier statisch eindeutig zu ermitteln ist. Wären hingegen Prozedurzeiger involviert, könnte es auch mehrere Aufrufziele geben – nämlich in dem Fall, dass die Zeigeranalyse kein eindeutiges Ziel ermittelt. In Randsituationen kann es auch kein Aufrufziel geben. Dies wäre zum Beispiel der Fall, wenn der Funktionszeiger uninitialisiert bliebe.



Mit dem Symbol * markierte Knoten stehen unter dem Schutz eines Locks.

Abbildung 9.1: Aufrufgraph des Programms aus Abbildung 2.7 (Seite 39)

9.1.3 Intraprozedurale Kontrollflussgraphen

Zusätzlich zum Aufrufgraphen enthält die Zwischendarstellung, die das Werkzeug Red als Eingabe erhält, auch intraprozedurale Kontrollflussgraphen. Jeder Prozedurknoten verweist auf seinen Start-Basic-Block. Basic-Blocks haben entweder einen unbedingten Nachfolger, eine Bedingung und zwei bedingte Nachfolger oder sind als Exit-Basic-Block markiert. Exit-Basic-Blocks signalisieren das Ende der Prozedur.

Nahezu alle[1] Anweisungen sind genau einem Basic-Block zugeordnet. Es gibt wichtige zur Datenstruktur des Kontrollflussgraphen passende Hilfsmittel. Diese Hilfsmittel sind für die Implementierung von Bedeutung. Dazu gehören in beide Richtungen traversierbare Verbindungen zwischen den Anweisungen und dem Basic-Block, dem sie zugeordnet sind, und einige

[1] Neben totem Code werden in Bauhaus Ausnahmen gemacht für Anweisungen, die in Bedingungen stehen, insbesondere bei `switch`-Verzweigungen, die sogenannte Common-Subexpressions enthalten. Für manche Analysen mag dieses Verhalten sinnvoll erscheinen; für die hier beschriebene ist es möglich und nötig, an diesem Umstand vorbei zu arbeiten.

Visitor-/Iterator-Implementierungen, die es erlauben, Anweisungen und die darin enthaltenen Ausdrücke in sinnvoller Reihenfolge abzulaufen.

9.2 Konstantenpropagierung und -faltung

Die vom Bauhaus Data Race Detector zur Verfügung gestellten Datenstrukturen sind gut geeignet, um Datenflussanalysen darauf aufzubauen. Dies ist natürlich kein Zufall – schließlich besteht die Data-Race-Analyse hauptsächlich aus Datenflussanalysen.

Während Data-Race-Analysen im Wesentlichen ohne Wissen über die Werte, die Ausdrücke annehmen, arbeiten können,[1] ist die Analyse von expliziter Zustandsverwaltung darauf angewiesen, Werte möglichst oft statisch berechnen zu können. Statisch bekannte Werte bilden schließlich eine Grundlage des Modells.

9.2.1 Motivation

Um effizient mit statisch bekannten Werten arbeiten zu können – und nicht etwa bei jeder Bedingung einzeln durch Traversierung herausfinden zu müssen, ob der jeweilige Ausdruck einen statisch ermittelbaren Wert hat – wurde ein umfassender Ansatz gewählt. Es wird stets eine systemweite Konstantenpropagierung und -faltung durchgeführt.

Dabei hat es sich als wichtig herausgestellt, tatsächlich auch eine Konstantenfaltung und nicht nur eine reine Propagierung zu implementieren. Dies hat zwei Gründe:

- Der erste ist technischer Natur: Schon beim Aufbau des abstrakten Syntaxbaumes in Bauhaus werden gerade Literale von dem Frontend oftmals in Kaskaden von impliziten Casts eingehüllt, sodass sie nahezu nie »nackt« im abstrakten Syntaxbaum stehen. Hinzu kommen bei

[1] Einzige Ausnahmen im Bauhaus Data Race Detector sind Zeiger und die Erkennung von totem Code.

```
1  #define PREAMBLE_LENGTH [...]
2
3  #ifdef LEGACY
4     #define CHECKSUM_LENGTH ((uint16_t) 2)
5  #else
6     #define CHECKSUM_LENGTH ((uint16_t) 4)
7  #endif
8
9  #define PAYLOAD_LENGTH (sizeof(some_type_t) * num)
10 [...]
11 #define MESSAGE_LENGTH \
12       (PREABLE_LENGTH \
13    +  ((uint16_t) PAYLOAD_LENGTH) \
14    +  CHECKSUM_LENGTH)
15 [...]
16 const size_t num = 6;
17 [...]
```

Abbildung 9.2: Definition eines benannten »Wertes« mittels konstanter Ausdrücke

eingebetteten Systemen aus Automobilen dann oft noch explizite Casts, die durch Makros eingefügt werden.[1]

- Der zweite, wichtigere Grund ist Dokumentation und Konfiguration. Gerade durch Makros benannte Werte sind oftmals tatsächlich konstante Ausdrücke unterschiedlicher Größe. Abbildung 9.2 stellt beispielhaft dar, wie das aussehen kann. Im Programmcode taucht dabei schlicht die »Konstante« `MESSAGE_LENGTH` auf. Nach Expansion der Makros (und damit auch im abstrakten Syntaxbaum) steht dort dann ein komplexer Ausdruck aus möglicherweise Hunderten Bausteinen.

Anscheinend verwenden die Entwickler diese Makros, als seien sie einfache Konstanten, und verlassen sich dabei darauf, dass dieses Vorgehen kaum

[1] Diese teils unnötig wirkenden expliziten Casts stellen sicher, dass der entsprechende Code-Abschnitt nicht gegen Regel 10.1 der Misra-C-Richtlinien [MIR08]: *»The value of an expression of integer type shall not be implicitly converted to a different underlying type [...]«* verstößt.

Performance-Nachteile beim Betrieb des Systems mit sich bringt. Aus Sicht des Autors auch durchaus zu Recht gehen sie davon aus, dass Konstantenpropagierung und -faltung im Compiler (eventuell in Verbindung mit weiteren Analysen) dafür sorgen, dass sich derlei Ausdrücke fast wie Literale verhalten. Um mit den Erwartungen mithalten zu können, müssen die Entwickler von Analysen demnach dieselben Mittel anwenden.

Anzumerken ist, dass das Beispiel wenig mit expliziter Zustandsverwaltung zu tun hat, da `MESSAGE_LENGTH` wohl kaum dazu verwendet wird, den Systemzustand zu repräsentieren. Das Prinzip gilt aber auch für die explizite Zustandsverwaltung. Auch wenn in den Beispielen, die in dieser Abhandlung aufgeführt werden, meist einfache Enums und Literale verwendet werden, stehen in realen Systemen an ihrer Stelle teils auch konstante Ausdrücke.

9.2.2 Implementierung

Die Analyse wurde als Task-, Kontext- und flussinsensitive Datenflussanalyse implementiert. Aufgrund der Insensitivität kann bei richtig gewählter Auswertungsreihenfolge auf Iteration verzichtet werden.

Das Analyseergebnis ist eine Datenstruktur, die für alle im analysierten Programm vorkommenden Ausdrücke speichert, ob der Ausdruck einen statisch bestimmten Wert hat und welcher dies ist.

Zusätzlich sind die Definitionen von Variablen, die als `const` aber nicht `volatile` deklariert sind und initialisiert werden, mit dem Wert dieser (Pseudo-)Konstante annotiert. Lesende Zugriffe auf Variablen haben in der Analyse genau dann einen statisch bestimmbaren Wert, wenn sie auf eine solche Variable zugreifen.[1]

Makros sind zum Zeitpunkt der Analyse bereits durch den Makroexpansionsmechanismus des Frontends aufgelöst, sodass für durch Makros benannte Werte zunächst kein besonderes Analyseverhalten notwendig ist.

Literale erhalten ihren Literalwert als statisch bekannten Wert. Ebenso

[1] Es ist eine Eigenart der Sprache C, dass eine mit »`const int c = 3;`« deklarierte und initialisierte »Konstante« nicht dazu führt, dass der Ausdruck »`c`« eine »constant expression« im Sinne des Sprachstandards ist.

einfach können Enumwerte und artifizielle Zahlkonstanten[1] behandelt werden.

Ferner werden für das Constant-Folding einige ausgewählte Funktionen beziehungsweise Operatoren ausgewertet, wenn alle (relevanten) Argumente bekannt sind.

Mit dem hier beschriebenen Analyseaufbau können alle der in Abbildung 6.1 (Seite 83) dargestellten Möglichkeiten zur Benennung von Werten richtig erkannt und behandelt werden.

9.3 Zustandsvariablenerkennung

Um dem Anwender die Auswahl von Zustandsvariablen und damit Mustern zur expliziten Zustandsverwaltung zu erleichtern, sammelt Red Informationen zu Variablen und gibt diese aus.

Gemäß der Definition von expliziter Zustandsverwaltung aus Kapitel 6 gibt es naheliegende und leicht prüfbare Kriterien, die Zustandsvariablen erfüllen müssen, um zur Synchronisation beizutragen.

Das Werkzeug Red kann eine Liste von Variablen ausgeben, die die nachfolgenden Kriterien erfüllen. Der Anwender kann sich daran orientieren, wenn er die Zustandsvariablen auswählt, die tatsächlich verwendet werden sollen.

Die Variable wird mindestens einmal in einem Pfadprädikat in geeigneter Weise mit einem statisch bekannten Wert verglichen.

Variablen, bei denen dies nicht der Fall ist, können zwar tatsächlich zur Synchronisation beitragen, es ist jedoch in dem konservativ approximierten Modell nicht möglich, diesen Beitrag zu erkennen.

Der Variablen wird mindestens einmal ein konstanter Wert zugewiesen.

Zwar lassen sich leicht Beispiele konstruieren, bei denen dies nicht der Fall ist und dennoch Synchronisation hergestellt wird, jedoch dürfte

[1] Dabei kann es sich um Zahlen handeln, die im Quelltext nur implizit auftauchen. Solche kommen – dann in der Form von `0` und `1` – zum Beispiel bei booleschen Ausdrücken vor.

es sich dabei um Randfälle handeln. Durch das Kriterium werden im Einsatz an realen Systemen aus Automobilen ohnehin kaum Variablen ausgeschlossen, da dort fast alle Variablen explizit zu konstanten Werten (in der Regel Literalen) initialisiert werden.

Der vom Bauhaus Data Race Detector der Variablen zugewiesene Escape-Status ist größer als »thread-local«.

Wird auf eine Variable sicher nur aus einem Task zugegriffen, ist dies ein klarer Hinweis darauf, dass sie nicht zur Synchronisation vorgesehen ist. Zwar lassen sich auch hier Beispiele konstruieren, bei denen Thread-lokale Variablen zur Synchronisation beitragen, die gefundenen Beispiele wirken aber künstlich.

***(optional)* Der deklarierte Bezeichner der Variablen hat `state` als Substring.**

Dieses alternative Kriterium wurde mit Erfolg dazu eingesetzt, Zustandsvariablen zu identifizieren. Es ist eine Binsenweisheit der Softwareentwicklung, dass die Variablenbezeichner den Zweck einer Variablen erkennen lassen. Dass in den Systemen sprechende Bezeichner gewählt werden, hilft dem Anwender hier dabei, Zustandsvariablen zu identifizieren.

9.4 Sequentialisierung

In der Sprache C gibt es auch außerhalb des Bereichs Nebenläufigkeit Möglichkeiten, Anweisungen so aufzuschreiben, dass die Reihenfolge ihrer Ausführung entweder gar nicht definiert ist oder zumindest die Reihenfolge der (lokalen) Ausführung nicht mit der Reihenfolge im abstrakten Syntaxbaum übereinstimmt.

Der Sequentialisierungsschritt des Werkzeuges Red bringt mit der in Abschnitt 9.5.1 (Seite 138) beschriebenen Methode (Neben-) Effekte in eine klar definierte Reihenfolge.

9.4.1 Beispiele

Die für die hier vorliegende Modellierung relevanten Nebeneffekte haben nur Zuweisungen[1] und Funktions- beziehungsweise Prozeduraufrufe. Wenn solche nebeneffektbehafteten Ausdrücke innerhalb von anderen Ausdrücken stehen, greift die Sequentialisierung ein.

Die Sequentialisierung erfolgt insbesondere in folgenden Fällen:

- Der Wert einer Zuweisung wird weiterverwendet. Dies ist in der Sprache C generell erlaubt, da Zuweisungen immer auch Ausdrücke sind. Ein Beispiel hierfür ist der Ausdruck »`x = i++;`«.
- Ein Parameterausdruck hat Nebeneffekte. Beispiele hierfür sind die Ausdrücke »`f(g(x));`« und »`p(x = 1);`«.
- In Bedingungsausdrücken kommen Nebeneffekte vor. Beispiele hierfür sind »`if (x = 0) [...]`« und »`if (p(a)) [...]`«.

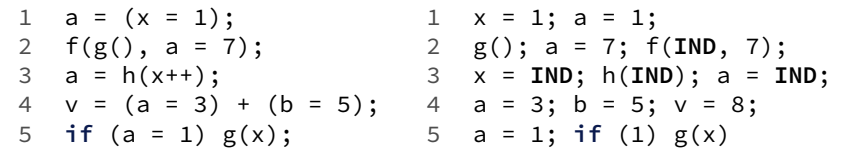
Ein bekannter Sonderfall sind außerdem Funktionsaufrufe, in denen mehrere Parameterausdrücke Nebeneffekte haben. Ein Beispiel hierfür ist der Ausdruck »`p(x = 1, f(x));`«.

9.4.2 Sequentialisierungsalgorithmus

Der Algorithmus zur Sequentialisierung setzt die innerhalb eines Top-Level-Ausdruckes auftretenden Nebeneffekte in In-Order-Reihenfolge vor den jeweiligen Ausdruck. Top-Level-Ausdrücke sind dabei die durch Strichpunkte und Code-Blöcke getrennten Ausdrücke. Ausdrücke in den Bedingungen von Basic-Blocks mit bedingten Nachfolgern, die möglicherweise Nebeneffekte haben, werden hinter die anderen Anweisungen des Basic-Blockes gesetzt.[2]

[1] Dies ist auch deshalb der Fall, weil hier – in Übereinstimmung mit anderen Bauhaus-Werkzeugen – alle wert-ändernden Ausdrücke als Zuweisungen betrachtet werden. Das gilt insbesondere auch für Ausdrücke der Form »`i++`«.

[2] Tatsächlich bedarf es im Werkzeug Red keiner Sonderbehandlung von gemeinsamen Teilausdrücken, wie sie zum Beispiel bei `switch`-Statements auftreten, da das C-Frontend und vorgelagerte Analysen bereits passende Maßnahmen ergreifen.



Auf der rechten Seite ist bereits der Effekt der Konstantenpropagierung und -faltung berücksichtigt (siehe Abschnitt 9.5.1.1).

Abbildung 9.3: Illustration der Sequentialisierung

Durch diese Sequentialisierung wird eine Left-To-Right-Evaluationsreihenfolge postuliert. Dies verletzt streng genommen die Konservativität der Analyse, da der C-Standard in vielen Fällen nicht spezifiziert, in welcher Reihenfolge derartige Nebeneffekte auftreten.[1]

Um die Sequentialisierung zu implementieren, wurden in Bauhaus bereits vorhandene Iteratoren geeignet konfiguriert und kombiniert. Da unterschiedliche Analysen Ausdrücke in unterschiedlicher Weise traversieren, sind die Iteratoren konfigurierbar.

9.4.3 Ergebnis

Abbildung 9.3 illustriert den Effekt der Sequentialisierung zusammen mit Teilen der in Abschnitt 9.5 dargestellten Abstrahierung. Die Abstrahierung ist an den artifiziellen Ausdrücken `IND` zu erkennen. In der Anwendung auf eingebettete Systeme aus Automobilen sind die Effekte der Sequentialisierung wenig dramatisch: Die vom Autor der Abhandlung untersuchten Systeme scheinen einem Style-Guide zu folgen, der zu simplen Anweisungen führt. Die meisten Funktionen haben den Rückgabewert `void`, sind somit tatsächlich Prozeduren. Auch gibt es nur wenige Funktionen mit mehr als einem Parameter. Nebeneffektbehaftete Statements stehen meist in einem eigenen Top-Level-Ausdruck.

[1] Sinnvollerweise verbieten daher auch die Misra-C-Richtlinien [MIR08] mit Regel 12.2 alle Ausdrücke, deren Wert von der Evaluationsreihenfolge abhängt.

9.5 Abstrahierung

Der Abstrahierungsschritt von Red lässt den Aufrufgraphen und den Kontrollflußgraph strukturell intakt und projiziert die in den Basic-Blocks enthaltenen Top-Level-Ausdrücke (darunter auch Funktionsaufrufe) und die Pfadprädikate auf die später vom Modell benötigten Eigenschaften. Zudem werden die Positionen von Zugriffen markiert, die Teil von Data-Race-Warnungen sind.

Die Abstrahierung ist konservativ in dem Sinne, dass jedes im Eingabeprogramm mögliche Verhalten auch im Ergebnisprogramm möglich ist. Allerdings ist nicht jedes im Ergebnisprogramm mögliche Verhalten auch im Eingabeprogramm notwendigerweise erlaubt. Die Überapproximierung des Verhaltens kann zum Beispiel bei zu `IND` abstrahierten Bedingungen auftreten.

9.5.1 Zuweisungen

Die Sequentialisierung sorgt dafür, dass die im Eingabeprogramm stehenden Zuweisungen nur noch auf ihrer linken Seite Effekte haben, die zu berücksichtigen sind. Der Effekt der Zuweisung besteht in der Änderung eines Hauptspeicherbereiches, der durch den Ausdruck auf der linken Seite bestimmt wird.

Bei den Zuweisungen werden die linke Seite (das Zuweisungsziel) und die rechte Seite (der zugewiesene Wert) getrennt voneinander abstrahiert.

9.5.1.1 Rechte Seite

Die Abstraktion der rechten Seite ist einfach: Wurde durch die Konstantenpropagierung und -faltung ein statisch bekannter Zahlenwert gefunden, der den Ausdruck ersetzen kann, wird dieser verwendet. Wenn dies nicht der Fall ist, wird ein spezieller Wert `IND` verwendet. Dabei steht `IND` für »indeterminable value« – ein unbekannter, möglicherweise bei jeder Ausführung unterschiedlicher Wert. Im später erzeugten Modell ist nach der Zuweisung von `IND` an eine Variable nichts mehr über ihren Wert bekannt.

9.5.1.2 Linke Seite

Die Abstraktion der linken Seite ist aufwendiger als die Abstraktion der rechten Seite. Zwar stehen (gerade in den untersuchten eingebetteten Systemen) oftmals einfache Variablen auf der linken Seite der Zuweisungen, im Allgemeinen können hier jedoch auch andere Konstrukte stehen.

Das Ziel der Zuweisung lässt sich zum Beispiel bei der Dereferenzierung von Zeigern (zum Beispiel »`*p = INIT;`«) und der Indexierung von Arrays (zum Beispiel »`a[x] = SHUTDOWN;`«) nicht unbedingt eindeutig statisch ermitteln.

Die von der zugrunde gelegten Analyse zur Verfügung gestellten Datenstrukturen zu Points-To-Objekten helfen hier weiter. Die linke Seite der Anweisungen wird durch eine Menge von Points-To-Zielen repräsentiert. Der Gedanke hierbei ist, dass die Anweisung bei jeder Ausführung genau eines der Points-To-Ziele beschreibt.

Die Anweisungen werden als Strong- und Weak-Updates angesehen. Ein Strong-Update setzt das Ziel auf einen neuen Wert, während ein Weak-Update nur einen neuen möglichen Wert der Wertemenge hinzufügt. In Abhängigkeit von der Zusammensetzung der Zuweisungszielmenge und davon, welche Points-To-Objekte als Zustandsvariablen ausgewählt werden, müssen unterschiedliche Zuweisungen generiert werden. Es ergeben sich drei zu unterscheidende Fälle:

- Keines der Points-To-Objekte steht für eine Zustandsvariable. In diesem Fall steht fest, dass die Anweisung keine Zustandsvariable beeinflusst, sie kann also (für diesen Schritt) ignoriert werden.
- Eines (oder mehrere) der Points-To-Objekte steht für eine Zustandsvariable, es kommen aber auch andere Points-To-Objekte in Frage. Dies entspricht einem Weak-Update auf den jeweiligen Zustandsvariablen.
- Es befindet sich genau ein Points-To-Objekt auf der linken Seite und dieses steht für eine Zustandsvariable. In diesem Fall ist die Zuweisung ein Strong-Update der Zustandsvariable.

Hier lauern allerdings einige Fallstricke, die großräumig umgangen wer-

den. In bestimmten Fällen können die Points-To-Ziele überlappen. Es muss beispielsweise sichergestellt werden, dass zwischen einem Zugriff auf ein ganzes Array und dem auf ein einzelnes Feld unterschieden wird. Im Code könnte zum Beispiel eine als `int` deklarierte Variable in ein `char`-Array gecastet werden, bei dem dann die erste Komponente zugewiesen wird. Dies hat einen anderen Effekt, als der ursprünglichen Variable denselben Wert zuzuweisen.

Red verweigert daher die Nutzung von Variablen als Zustandsvariable, die nicht als »einfache« Variable definiert sind (zum Beispiel `A[3]`) und von Variablen, die nicht als ein Zahltyp[1] definiert sind.

Ganz ausschließen lassen sich überlappende Points-To-Objekte damit allerdings nicht. In einigen verbleibenden Randfällen können Nebeneffekte von Red falsch approximiert werden. Die eingesetzten Compiler erlauben es zum Beispiel, globalen Variablen fixe Speicheradressen zu geben. Ein sinnvoller Einsatzzweck dieses Mechanismus ist es, bei hardwarenaher Programmierung Register zugänglich zu machen. Würde er versehentlich oder absichtlich dazu genutzt, einfachen Variablen überlappende Speicherbereiche zuzuweisen, könnte dies, wenn diese Variablen als Zustandsvariablen verwendet werden, zu fehlerhaften Ergebnissen der Analysekette führen.

9.5.2 Prozeduraufrufe

In ähnlicher Weise wie die Zuweisungen werden auch die Prozeduraufrufe abstrahiert. Einfache Prozeduraufrufe benennen schlicht eine Zielprozedur. Dabei bleiben die Sensitivitäten der zugrunde gelegten Data-Race-Analyse erhalten. Siehe dazu auch Abbildung 9.1 (Seite 130).

Allgemeiner besteht ein Prozeduraufruf in der Abstraktion aber aus einer Menge von möglichen Aufrufzielen. Diese Menge enthält bei Aufrufen mittels Dereferenzierung von Funktionszeigern möglicherweise mehrere Einträge.

[1] Dabei folgen wir `typedefs`. Dies ist in den untersuchten Systemen auch dringend notwendig – reine Typen wie `int` tauchen dort nur selten auf. Siehe hierzu auch Regel 6.3 der Misra-C-Richtlinien [MIR08]: *»typedefs that indicate size and signedness should be used in place of the basic numerical types.«*

Bei jedem Aufruf wird genau eines der Elemente dieser Menge tatsächlich aufgerufen.

9.5.2.1 Extremfälle

In Extremfällen kann die Menge der Zeigerziele leer sein. Dies ist etwa dann der Fall, wenn uninitialisierte Funktionszeiger oder `NULL`-Funktionszeiger in Aufrufausdrücken dereferenziert werden. Das Verhalten des Programmes ist in diesem Fall undefiniert.

Im Modell kann[1] ein Prozeduraufruf ohne Ziel als der Prozess `STOP` modelliert werden. Dies ist eine natürliche Erweiterung der sonstigen Vorgehensweise, da der Prozess `STOP` im Traces-Modell ein neutrales Element des Non-deterministic-Choice-Operators `|~|` ist. Das Verhalten des Modells kommt damit dem Verhalten des realen Programms nahe, das wohl schlicht abstürzen würde. Diese Modellierung kann als konservative Approximierung gesehen werden.

Ebenso ist es möglich, dass ein bestimmter Aufrufausdruck, der einen Funktionszeiger dereferenziert, bei mehrmaliger Ausführung entweder einen Funktionszeiger mit validem Ziel oder den Null-Funktionszeiger dereferenziert. Dieser Aufrufausdruck hat im hier verwendeten Modell dann genau ein valides Ziel und er wird so modelliert, dass jedes Mal dieses Ziel aufgerufen wird.

Dies verletzt streng genommen die Konservativität der Analyse. Eingebettete Systeme aus Automobilen verwenden zwar zum Teil Funktionszeiger, stellen aber, um katastrophales Fehlverhalten zu vermeiden, sicher, dass niemals uninitialisierte Funktionszeiger oder Null-Funktionszeiger dereferenziert werden. Bei der Untersuchung der Systeme wurden auch keine Hinweise auf derart invalide Funktionszeigerdereferenzierungen gefunden.

Der industriellen Anwendung stehen zahlreiche Methoden und Werkzeuge zur Verfügung um solche Fehler auszuschließen – darunter vor allem

[1] In den Tests der Werkzeugkette kamen derartige Aufrufstatements nicht vor. Die in dieser Abhandlung evaluierte Version der Werkzeugkette beendet sich mit einer Fehlermeldung selbst, wenn sie auf ein derartiges Aufrufstatement trifft – ein Implementierungsfehler.

statische Software-Analyse und abstrakte Interpretation. (Vergleiche Misra-C-Richtlinien [MIR08], insbesondere Regel 21.1.)

9.5.3 Pfadprädikate

Als Letztes werden zur Abstrahierung der Programme noch Pfadprädikate abstrahiert. Nebeneffekte, die möglicherweise in den Pfadprädikaten vorkommen, müssen dabei nicht mehr berücksichtigt werden, da die Sequentialisierung sie bereits ausgelagert hat.

In der Abstrahierung werden die Prädikate zunächst in durch boolesche Operatoren verbundene Grundausdrücke aufgeteilt. Dabei werden, wo dies möglich ist, auch in C vorhandene »unkonventionelle« boolesche Ausdrücke wie `*`, `==` und `!=` berücksichtigt. Dabei entspricht der Operator `*` der Konjunktion, der Operator `==` der Äquivalenz und der Operator `!=` der Antivalenz. Siehe dazu auch Abbildung 9.6.

Die Grundausdrücke werden dann abstrahiert. Dazu gibt es drei Möglichkeiten:

- Der Wert des Grundausdruckes lässt sich statisch ermitteln. Dies ist der Fall, wenn die Konstantenpropagierung und -faltung den Ausdruck zu einem konstanten Wert – `0` oder `1` – reduziert hat[1]. In diesem Fall wird der Grundausdruck zu `TRUE` oder `FALSE`.
- Der Grundausdruck ist ein Vergleich mittels der Vergleichsoperatoren `==` oder `!=` zwischen einer als Zustandsvariable in Frage kommenden einfachen Variablen und einem anderen Ausdruck, der sich mittels Konstantenpropagierung und -faltung auf einen konstanten Wert reduzieren lässt. In diesem Fall wird der Grundausdruck zu einem Vergleich der Form $(x \circ n)$ abstrahiert. (x ist eine Variable, $\circ$ ist ein Vergleichsoperator und n eine ganze Zahl.)

[1] Es kommen wirklich nur die Werte `0` und `1` in Frage, obwohl die Sprache C keinen eigenen Datentyp für boolesche Werte hat und stattdessen der Typ `int` verwendet wird. Immer dann, wenn der Wert nicht ohnehin offensichtlich im booleschen Bereich liegt, spezifiziert der Standard eine Semantik, die einem impliziten »`(_ != 0)`« entspricht. Das hier verwendete Frontend löst dieses Verhalten auf. Der Ausdruck »`if (7) [...]`« hat im abstrakten Syntaxbaum fast die gleiche Repräsentation wie der Ausdruck »`if ((7 != 0)) [...]`«.

Tabelle 9.1: Beispiele für die Abstraktion von Prädikaten

Nr.	Ursprüngliches Prädikat	Abstraktes Prädikat
1	`a == 1`	$a = 1$
2	`5 != b`	$b \neq 5$
3	`(a == 1) | (0 == b)`	$(a = 1) \vee (b = 0)$
4	`x == 7`	`IND`
5	`g(a, 7)`	`IND`
6	`a == 2 & x == 7`	$(a = 2) \wedge$ `IND`
7	`1`	`true`
8	`(a == 7) != (5 != b)`	$(a = 7) \oplus (b \neq 5)$
9	`(x == f(3)) | (x == 13)`	`IND` $\vee$ `IND`

In dieser Tabelle sind `a` und `b` Zustandsvariablen und `x` ist keine.

- Ansonsten wird der Grundausdruck zu `IND` abstrahiert. `IND` steht für einen Konditionalausdruck, über dessen Wert nichts bekannt ist und der sich auch von Auswertung zu Auswertung unterscheiden kann.

9.5.3.1 Beipiele

In Tabelle 9.1 sind Beispiele für die Abstraktion von Prädikaten dargestellt. Dabei ist die Anwendung der soeben beschriebenen Regeln zu erkennen.

In der Tabelle enthält die Spalte »Ursprüngliches Prädikat« das Prädikat, wie es sich der Analyse nach Makro-Expansion und Konstantenpropagierung und -faltung darstellt.

Die mechanische Regelanwendung lässt sich zum Beispiel in Zeile 9 erkennen. Hier könnte auch ein einfacherer Ausdruck – nämlich schlicht `IND` – generiert werden. Eine derartige Optimierung findet aber nicht statt.

9.5.3.2 Partiell unbestimmbare Ausdrücke

Die im vorhergehenden Abschnitt dargestellten Regeln erlauben, dass Ausdrücke partiell unbestimmt bleiben. Dabei sind partiell unbestimmte Ausdrücke nicht mit dem völlig unbestimmbaren Ausdruck `IND` gleichzusetzen. Dies zeigt sich bei Betrachtung der in Tabelle 9.2 dargestellten Beispiele.

Tabelle 9.2: Beispiele für implizite Bedingungen, die sich aus den Regeln zur Abstraktion von Bedingungen für Then- und Else-Zweige ergeben

Nr.	Ursprüngliches Prädikat	Implizierte Bedingung für… …Then-Zweig	…Else-Zweig
1	`a == 1`	$a = 1$	$a \neq 1$
2	`5 != b`	$b \neq 5$	$b = 5$
3	`a == 1 || 0 == b`	$a = 1$ oder $b = 0$	$a \neq 1$ und $b \neq 0$
4	`x == 7`	Wahr	Wahr
5	`a == 2 && x == 7`	$a = 2$	Wahr
6	`x == 1 || b == 1`	Wahr	$b \neq 1$
7	`7 + 2 == 1`	Falsch	Wahr
8	`(3 == a) != (b == 7)`	$a = 3$ g. d. w. $b \neq 7$	$a = 3$ g. d. w. $b = 7$
9	`x == 2 | f(x)`	Wahr	Wahr

In dieser Tabelle sind `a` und `b` Zustandsvariablen und `x` ist keine.

Die Tabelle zeigt implizite Bedingungen, die sich für Then- und Else-Zweige, insbesondere auch im Zusammenhang mit `IND` ergeben.

In der Tabelle stehen in der ersten Zeile einige Beispielprädikate, wie sie so (nach der Makroexpansion) möglicherweise im Quelltext stehen. Dem gegenübergestellt werden die impliziten, im späteren Modell geltenden Bedingungen für den Einstieg in den Then- beziehungsweise Else-Zweig.

Die dargestellten Bedingungen sind dabei notwendige, aber im Allgemeinen nicht hinreichende Bedingungen. Zum Einstieg in den Then-Zweig muss die dargestellte Bedingung erfüllt sein. Nur daraus, dass die Bedingung erfüllt ist, lässt sich aber nicht folgern, dass in den Then-Zweig eingestiegen wird. Die Bedingung des Else-Zweiges könnte auch erfüllt sein.

Im einfachsten Fall ist die ganze Bedingung bestimmbar. Dies ist in Tabelle 9.2 in den Zeilen 1–3, in Zeile 7 und in Zeile 8 der Fall. Die implizierten Bedingungen des Then- und Else-Zweigs sind dabei Negationen voneinander. Ist dann der Zustand der Zustandsvariablen bekannt, ist auch bekannt, in welchen Zweig eingestiegen wird.

Ist hingegen die gesamte Bedingung unbestimmbar – wie es in Zeile 4

und Zeile 9 in Tabelle 9.2 der Fall ist –, sind die implizierten Bedingungen des Then- und Else-Zweigs beide stets erfüllt. Selbst wenn der Zustand der Zustandsvariablen bekannt ist, lässt sich also nicht ermitteln, welcher der Zweige eingeschlagen wird.

Die interessantesten Fälle sind die partiell unbestimmbaren Ausdrücke. Beispiele dafür finden sich in Zeile 5 und in Zeile 6 in Tabelle 9.2. Hier »überlappen« die Bedingungen[1]. Je nach der tatsächlichen Wertebelegung der involvierten Zustandsvariablen kann bekannt oder unbekannt sein, in welchen Zweig eingestiegen wird.

9.5.4 Zugriffe aus Data-Race-Warnungen

In Data-Race-Warnungen enthaltene abstrakte Zugriffe werden durch Markierungen ersetzt. Da später nur einzelne Data-Races untersucht werden, muss hier nicht auf die Reihenfolge innerhalb der Ausdrücke geachtet werden.

Da jede Data-Race-Warnung bereits eine eindeutige ID zugewiesen bekommen hat und stets klar definiert ist, welcher der beiden Zugriffe »vorne steht« (der mit der niedrigeren Priorität[2]), lassen sich die abstrakten Zugriffe der Data-Race-Warnungen eindeutig durch die ID und die Angabe identifizieren, ob es sich um die linke oder rechte Seite des Zugriffspaares handelt.

9.6 Ausgabe in der Zwischensprache

Das Ergebnis des Werkzeuges Red ist eine Textdatei, in der das Programm in abstrahierter Form in einer Zwischensprache vorliegt.

9.6.1 Design

Die verwendete Zwischensprache dient der Kommunikation des Werkzeuges Red mit dem in der Werkzeugkette nachfolgenden Werkzeug. Beim Design

[1] Gemeint ist hier: Die Konjunktion der Ausdrücke ist erfüllbar.

[2] Es kann bei den angenommenen Scheduling-Verfahren kein Data-Race zwischen Tasks gleicher Priorität geben, da diese sich nicht gegenseitig unterbrechen können. Aus diesem Grund werden Tasks gleicher Priorität direkt vom Bauhaus Data Race Detector vereint. Abschnitt 4.2.1 (Seite 65).

der Zwischensprache wurde darauf geachtet, dass diese leicht zu parsen ist und (nahezu) nur die wirklich notwendigen Informationen enthält.

Es wurde eine einfache Textdatei verwendet, um die Gesamtkomplexität des Ansatzes zu beschränken. Zudem ist die Sprach- und Plattformunabhängigkeit von geeignet spezifizierten Textdateien unübertroffen.

Das Format ist auch (eingeschränkt) menschenlesbar. Gerade bei kleinen Beispielen war es bei der Entwicklung ausgesprochen hilfreich, sich die Ausgabe in einem Texteditor anschauen zu können und dabei unmittelbar zu erkennen, welche Auswirkungen Änderungen an der Implementierung hatten.

9.6.2 Syntax und Semantik

Abbildung 9.4 stellt die syntaktische Struktur der Zwischensprache in erweiterter Backus-Naur-Form dar.

Weder das produzierende Werkzeug Red, noch das konsumierende Werkzeug Red2CSP, siehe Kapitel 10 (Seite 157), nutzt die Grammatik strikt. Die dargestellte Grammatik ist als Kompatibilitätsstandard zu verstehen. Während die Ausgabe tatsächlich nur eine Teilmenge der Sprache verwendet, also strengere Regeln befolgt als die hier angegebenen, akzeptiert der Parser tatsächlich eine Obermenge.

Die Ausgabe orientiert sich stark an der abstrakten Struktur. Hauptbestandteil der Ausgabe sind die ⟨basic-block⟩-Paragrafen. Die Basic-Blocks haben ⟨statements⟩, die entweder Zuweisungen, Prozeduraufrufe oder die Positionsmarkierungen eines Data-Races sind. Ziele von Aufrufen sind Basic-Blocks, da Prozeduren durch ihren jeweiligen Start-Block repräsentiert werden. Die End-Basic-Blocks haben den Nachfolgetyp `SKIP`. Das bedeutet, dass der Kontrollfluss zum Aufrufer zurückkehrt oder endet, wenn es sich um die Start-Prozedur des Tasks handelt.

Die ⟨context-notification⟩-Paragrafen sind streng genommen redundant. Sie vereinfachen die Verarbeitung der Ausgabe und verbessern die Lesbarkeit leicht. Die auf eine ⟨context-notification⟩ folgenden Basic-Blocks gehören alle dem in der ⟨context-notification⟩ angegebenen Task an. Wird in einen

```
⟨file⟩ ::= { ⟨paragraph⟩ }*

⟨paragraph⟩ ::=
          | ⟨context-notification⟩
          | ⟨basic-block⟩
          | ⟨scc-component⟩
          | ⟨task-info⟩

⟨context-notification⟩ ::= ‚CONTEXT‘ ‚TGI‘ Number

⟨basic-block⟩ ::= ‚BLOCK‘ ‚(‘ Block-Id ‚)‘ ↩
                  ‚{‘ { ⟨statement⟩ }* ‚}‘ ‚(‘ ⟨successor⟩ ‚)‘

⟨scc-component⟩ ::= [...]

⟨task-info⟩ ::= [...]

⟨statement⟩ ::=
          | ⟨assignment⟩
          | ⟨method-call⟩
          | Data-Race-Mark

⟨successor⟩ ::=
          | ‚UNCOND‘ ‚(‘ Block-Id ‚)‘
          | ‚COND‘ ⟨condition⟩ ↩
                ‚(‘ ‚FALSE‘ Block-Id ‚)‘ ‚(‘ ‚TRUE‘ Block-Id ‚)‘
          | ‚(‘ ‚SKIP‘ ‚)‘

⟨assignment⟩ ::= ‚ONE OF‘ { ⟨variable⟩ }* ‚IS ASSIGNED‘ ⟨value⟩

⟨method-call⟩ ::= ‚CALL‘ ‚(‘ { Block-Id }* ‚)‘

⟨condition⟩ ::= [...]

⟨variable⟩ ::=
          | Variable-Id
          | ‚OTHER‘

⟨value⟩ ::=
          | Number
          | ‚UNKNOWN‘
```

Abbildung 9.4: Syntaktische Struktur der Zwischensprache in erweiterter Backus-Naur-Form

```
        |  (capital (capital | digit | ,_‘ )* )
⟶ Block-Id

        |  (lowercase (lowercase | digit | ,_‘ )* )
⟶ Variable-Id

        |  ( (,.‘ | ,:‘ ) (lowercase | capital | digit)+ )
⟶ Data-Race-Mark

        |  ( ,−−‘ (,−‘ | , ‘ ) (no_linebreak)* )
⟶ (skip)
```

Abbildung 9.5: Auszug aus der lexikalischen Struktur der Zwischensprache

anderen Task gewechselt, wird zunächst eine weitere ⟨context-notification⟩ ausgegeben.

In Abschnitt 10.1.5 (Seite 165) wird auf eine Komplikation bei nicht-trivialen starken Zusammenhaltsgruppen im Aufrufgraphen hingewiesen. Um diese zu lösen, wurde die Grammatik angepasst. Diese Anpassungen sind in Abbildung 9.4 nur angedeutet. Die ⟨scc-component⟩s rühren von dieser Komplikation der Modellsprache CSP_M her.

Es gibt unterschiedliche Namensräume für Identifier. Diese können vom Lexer unterschieden werden. In der Grammatik werden die Tokens Block-Id und Variable-Id verwendet. In der syntaktischen Struktur nicht dargestellt sind Kommentare. Diese sind erlaubt und werden wie der Whitespace vom Lexer entfernt. Die lexikalische Struktur ist in Auszügen in Abbildung 9.5 dargestellt. Der verwendete Parser- und Lexer-Generator folgt bestimmten Konventionen bezüglich Zeilenumbrüchen. Die Zwischensprache muss diesen Konventionen ebenfalls folgen.

9.6.3 Bedingungen

Abbildung 9.6 stellt die in Abbildung 9.4 ausgelassene syntaktische Struktur der Bedingungen dar. Aus dieser ergibt sich der Umfang der Sprache.

⟨condition⟩ ::=
| ‚(‘ ⟨condition⟩ ‚)‘
| ‚IND‘
| ‚TRUE‘
| ‚FALSE‘
| ⟨equality-comparison⟩
| ⟨inequality-comparison⟩
| ⟨boolean-negation⟩
| ⟨boolean-conjunction⟩
| ⟨boolean-disjunction⟩
| ⟨boolean-antivalence⟩
| ⟨boolean-eqivalence⟩

⟨equality-comparison⟩ ::= ‚==‘ Variable-Id Number

⟨inequality-comparison⟩ ::= ‚!=‘ Variable-Id Number

⟨boolean-negation⟩ ::= ‚NOT‘ ⟨condition⟩

⟨boolean-conjunction⟩ ::= ‚AND‘ ⟨condition⟩ ⟨condition⟩

⟨boolean-disjunction⟩ ::= ‚OR‘ ⟨condition⟩ ⟨condition⟩

⟨boolean-antivalence⟩ ::= ‚XOR‘ ⟨condition⟩ ⟨condition⟩

⟨boolean-eqivalence⟩ ::= ‚XNOR‘ ⟨condition⟩ ⟨condition⟩

Abbildung 9.6: Syntaktische Struktur der Bedingungen der Zwischensprache in erweiterter Backus-Naur-Form

Sicherheitshalber ist die Ausgabe der Bedingungen stets vollständig geklammert, obwohl durch Verwendung der Präfix-Notation bereits sichergestellt ist, dass die Grammatik eindeutig ist.

9.6.4 Ausgabe am Beispiel

Im Folgenden wird die tatsächliche Ausgabe eines Beispielprogramms betrachtet. In Abbildung 9.7 ist der Programmcode eines Tasks samt den

```
1  volatile unsigned int a = 5;
2  volatile unsigned int b = 0;
3  unsigned int x;
4  volatile unsigned int *p;
5
6  void task_a() {
7     a = (19 - 16) << 1;
8     b = 4;
9     while (true) {
10       if (b == 0 && a != 5) {
11          a = 0;
12          b = b + 7;
13       }
14       else {
15          *p = 22;
16          x = x + 3;
17       }
18    }
19 }
```

Abbildung 9.7: Ausschnitt eines nebenläufigen Programms mit Zustandsvariablen

zugehörigen Deklarationen dargestellt. In diesem Beispiel werden die Variablen a und b als Zustandsvariablen ausgewählt. Die Variable x ist keine Zustandsvariable. (Dabei wurde die alternative Festlegung von Zustandsvariable angewandt. Siehe dazu Abschnitt 8.3.2.1 (Seite 123).)

Ferner wird hier davon ausgegangen, dass es einen weiteren, nebenläufigen Task gibt, der ohne jeden Schutz auf die Variable x zugreift. Somit sind die Zugriffe auf x in Abbildung 9.7 Teile von Data-Races und damit natürlich auch Teile von Data-Race-Warnungen. Der Programmcode wurde so gestaltet, dass einige Eigenschaften der Reduktion sichtbar werden. Abbildung 9.8 zeigt den Teil der Ausgabe, der aus diesem Task hervorgeht.

```
CONTEXT TGI 2
BLOCK (C933_B871) { } (UNCOND C933_B873)

BLOCK (C933_B873) {
   ONE OF o80__a IS ASSIGNED 6
   ONE OF o81__b IS ASSIGNED 4
} (UNCOND C933_B874)

BLOCK (C933_B874) { } (COND (TRUE)
      (FALSE C933_B877)
      (TRUE C933_B876))

BLOCK (C933_B876) { } (COND ((== o81__b 0))
      (FALSE C933_B883)
      (TRUE C933_B882))

BLOCK (C933_B882) { } (COND ((!= o80__a 5))
      (FALSE C933_B881)
      (TRUE C933_B880))

BLOCK (C933_B880) {
   ONE OF o80__a IS ASSIGNED 0
   ONE OF o81__b IS ASSIGNED UNKNOWN
} (UNCOND C933_B884)

BLOCK (C933_B884) { } (UNCOND C933_B874)

BLOCK (C933_B881) {
   ONE OF o80__a o81__b IS ASSIGNED 22
   .c .d
   ONE OF OTHER IS ASSIGNED UNKNOWN
} (UNCOND C933_B884)

BLOCK (C933_B883) { } (UNCOND C933_B881)

BLOCK (C933_B877) { } (UNCOND C933_B872)

BLOCK (C933_B872) { } (SKIP)

TASK "task_a" TGI 2 START C933_B871 PRIO 5
```

Abbildung 9.8: Ausgabe des Werkzeuges Red für den Task `task_a` aus Abbildung 9.7

9.6.4.1 Leere Blöcke

Bei Betrachtung der Abbildung 9.8 fällt zunächst auf, dass viele der Blöcke leer sind. Manche dieser leeren Blöcke haben sogar einen unbedingten Nachfolger. Tatsächlich sind auch in realistischen Systemen viele der Blöcke leer. Dies hat mehrere Gründe.

Zum einen ist die Basic-Block-Generierung von Bauhaus nicht darauf ausgelegt, Basic-Blöcke nur sparsam anzulegen. Vielmehr scheinen diese auf Vorrat in den Kontrollflussgraphen eingebaut zu werden, möglicherweise in der Absicht, später noch artifizielle Anweisungen in diese einbauen zu können. Bekannte Techniken, die Zahl der Basic-Blocks durch Minimierung des Kontrollflussgraphen später zu reduzieren, werden nicht eingesetzt. Stattdessen wird das Vorhandensein leerer Blöcke sogar explizit gefordert und spezifiziert. Es wird zum Beispiel garantiert, dass alle Start- und End-Blöcke von Prozeduren leer sind, und auch bei Kontrollstrukturen wird sichergestellt, dass bestimmte Blöcke nicht zusammengeführt werden, auch wenn dies ersichtlich möglich ist und ohne die Zusammenführung leere Blöcke entstehen. So ist zum Beispiel sichergestellt, dass auf alle Schleifen stets noch ein »normaler« Basic-Block (also nicht direkt der leere End-Block) folgt. Der Block `C933_B877` ist ein Beispiel hierfür.

Zum anderen können (in Randfällen) Blöcke auch tatsächlich nur in der Abstraktion leer sein, aber im Konkreten effektfreie Anweisungen enthalten. Auch die in Bauhaus vorhandenen Algorithmen zur Eliminierung von totem Code lassen unnötig komplexe Kontrollflussgraphen mit leeren Blöcken zurück.

9.6.4.2 Kontrollfluss

Aus der Abstraktion lässt sich der Kontrollflussgraph des Tasks ablesen. Zum besseren Verständnis des Ausschnitts und zur Illustration ist dieser Kontrollflussgraph in Abbildung 9.9 dargestellt.

Da die Bedingung von Block `C933_B874` schlicht `true` ist, sind der Block `C933_B877` sowie der Exit-Block `C933_B872` tatsächlich gar nicht zu errei-

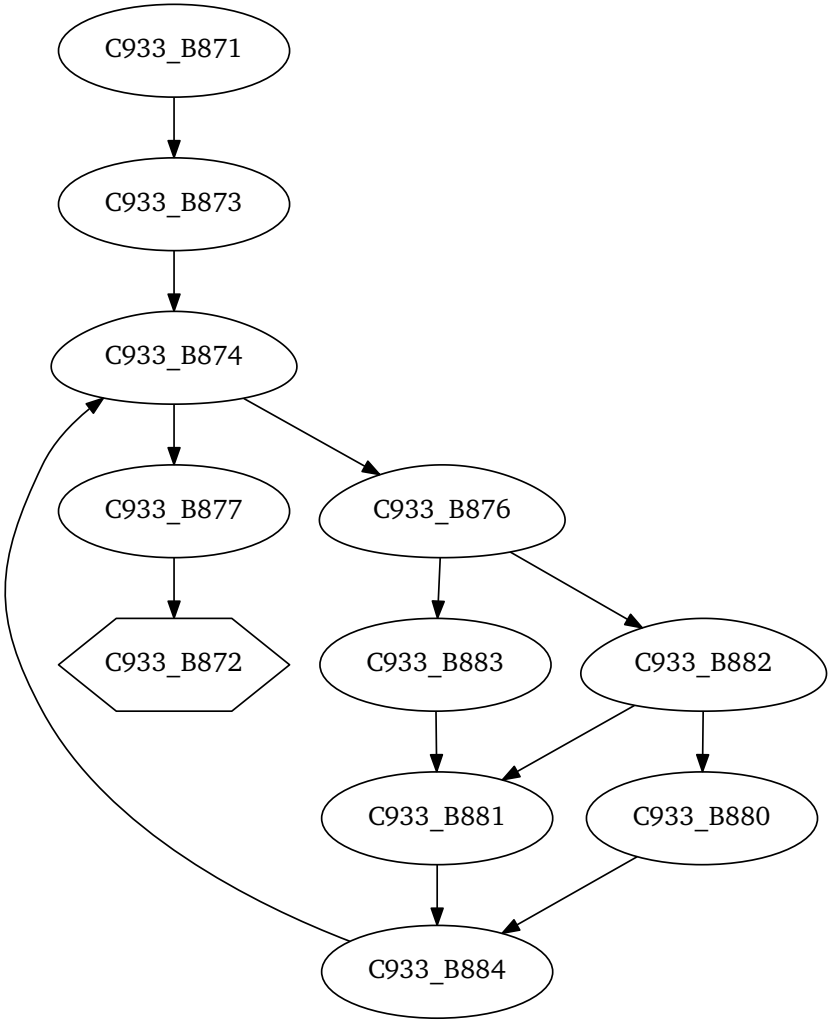


Abbildung 9.9: Kontrollflussgraph des Tasks `task_a` aus Abbildung 9.7, extrahiert aus der Abstraktion dargestellt in Abbildung 9.8

chen. Man erkennt auch, dass die &&-Verknüpfung (non-strict) im Gegensatz zu einer &-Verknüpfung (strict) im Kontrollflussgraphen weitere Blöcke verursacht.

9.6.4.3 Konstantenpropagierung und -faltung

Anhand der Zeile 7 der Abbildung 9.7 lässt sich der Effekt der Konstantenpropagierung und -faltung erkennen. Der – vielleicht etwas willkürlich gewählte – Ausdruck »`(19 - 16) << 1`« ist in der Ausgabe (Zeile 5 in Abbildung 9.8) zum Wert 6 reduziert worden.

9.6.4.4 Zuweisungen und Data-Race-Markierungen

Es sind auch mehrere unterschiedliche Formen von Zuweisungen in der Ausgabe zu erkennen. Diese sind:

- Der vielleicht einfachste Fall steht in Zeile 5: Hier wird einer genau bestimmten Zustandsvariablen (`a`) ein statisch bekannter Wert (`6`) zugewiesen.
- In der Zuweisung an `p*` aus Zeile 29 ist das Zuweisungsziel nicht eindeutig bekannt: Die Analyse kann hier nicht entscheiden, ob der Zeiger `p` auf die Variable `a` oder die Variable `b` zeigt. Die Analyse schließt aber aus, dass der Zeiger auf einen anderen Speicherbereich zeigt – wäre dies nicht so, würde das Ziel `OTHER` als mögliches Zuweisungsziel auftauchen. Dieses Analyseergebnis wurde durch eine geeignet ausgestaltete, hier nicht dargestellte `Main`-Prozedur induziert.
- Im Gegensatz dazu ist in Zeile 23 zwar das Zuweisungsziel bekannt (die Variable `b`), nicht aber der zugewiesene Wert. Die Zuweisung weist konservativ approximierend den Pseudowert `UNKNOWN` zu.

  Bei aufmerksamer Betrachtung könnte hier die Frage aufkommen, ob eine flusssensitive Konstantenpropagierung und -faltung hier möglicherweise den Wert zu `7` ermitteln könnte – schließlich ist die Zuweisung ja von dem Prädikat »`b == 0`« direkt dominiert. Dies wäre

aber wohl nur dann möglich, wenn sich der Wert von der Variablen `b` nicht durch eine nebenläufig ausgeführte Zuweisung verändern könnte. Gerade bei Zustandsvariablen wäre es aber wohl besonders schwer, solche nebenläufigen Zuweisungen auszuschließen.

- Ein Extremfall, der in realen Programmen allerdings sehr häufig vorkommt, steht in Abbildung 9.8. Hier ist nur eines über die Zuweisung bekannt: Sie betrifft keinen der als Zustandsvariablen in Frage kommenden Speicherbereiche. In diesem Fall ist die Zuweisung von zwei Data-Race-Warnungen betroffen. Dies zeigt sich an den Markierungen `.c` und `.d`. Gäbe es diese Markierungen nicht, könnten die folgenden Werkzeuge diese Zeile ignorieren.

9.6.4.5 Bedingungen

Zu betrachten bleiben die Bedingungen. Dazu ist wenig anzumerken. Zeile 13 zeigt den Standardfall. In Zeile 9 ist der Fall eines statisch bekannten Wertes zu erkennen.

9.6.5 Implementierung

In der tatsächlichen Implementierung sind die Abstrahierung und die Erzeugung der beschriebenen Ausgabe sowohl im Quelltext als auch in der Ausführung verzahnt.

Zunächst wird der Aufrufgraph mehrfach abgelaufen, um Informationen zu propagieren beziehungsweise zu sammeln. Anschließend wird in einem Durchlauf durch den Aufrufgraphen (und die verlinkten intraprozeduralen Kontrollflussgraphen) abstrahiert und die Ausgabe erstellt.

9.7 Steckbrief

Eine Übersicht über die Implementierung des Werkzeuges CSPC gibt der Steckbrief, der in Abbildung 9.10 dargestellt ist.

Red

Bauhaus-Werkzeug, das Systeme zu Modellen reduziert

Importierte Bibliotheken/Technologien

Bauhaus Intermediate Language
Zeigeranalyse und weitere Teile des Bauhaus Data Race Detectors
Bauhaus-Datenflussanalyse-Framework
Kontrollfluss- und Aufrufgraphtraversierung

Werkzeugimplementierung

Ada-Implementierung	31 Dateien 4427 Zeilen

Eigene Bibliotheken

Data-Race-IDs	4 Dateien 1949 Zeilen
Konstantenpropagierung und -faltung	4 Dateien 1036 Zeilen
Modellausgabe	9 Dateien 394 Zeilen
IML-Erweiterungen	2 Dateien 368 Zeilen
Starke Zusammenhaltsgruppen in Aufrufgraphen	2 Dateien 249 Zeilen

Test

Systemtest-Eingaben (C-Dateien)	153 Dateien 4422 Zeilen
Systemtest-Framework (Bash-Scripte)	3 Dateien 267 Zeilen

Ressourcennutzung

Laufzeit reale Systeme	10 s – 20 s
Laufzeit »kleine Beispiele«	< 1 s

(ermittelt auf dem Testrechner, siehe Tabelle 16.3 (Seite 271))

Abbildung 9.10: Steckbrief des Werkzeuges Red

Kapitel 10

CSP_M-Erzeugung

Aus der vom Werkzeug Red erzeugten Ausgabe sollen CSP_M-Dateien erzeugt werden, die die Fragestellung enthalten, ob eine bestimmte Data-Race-Warnung durch die explizite Zustandsverwaltung ausgeschlossen wird. Dieses Kapitel beschreibt, wie sich diese CSP_M-Datei aus verschiedenen Teilen zusammensetzen lässt.

10.1 Übersetzung der Statements und des Kontrollflusses

Der (mengenmäßige) Hauptinhalt der Ausgabe des Werkzeuges Red besteht üblicherweise aus Basic-Blocks. Siehe dazu die Grammatik der Zwischensprache in Abbildung 9.4 (Seite 147) und eine Beispielausgabe in Abbildung 9.7 (Seite 150).

Für jeden Basic-Block wird ein CSP_M-Prozess erzeugt, der mit dem Namen des jeweiligen Blocks benannt wird. Nicht zufällig sind die IDs der Basic-Blocks bereits passende CSP_M-Prozess-Bezeichner und durch die in diesen Bezeichnern enthaltenen Indexes ist auch sichergestellt, dass es keine Namenskonflikte gibt. Für jeden in der Ausgabe vorkommenden Basic-Block wird somit eine CSP_M-Definition erzeugt, die die Form »`<ID> = ...`« hat.

Der Inhalt der Definition ergibt sich aus der seriellen Aneinanderreihung der Übersetzungen der im Basic-Block vorhanden Statements. Der »Abschluss« ist die Übersetzung des ⟨successor⟩s. Gemäß der Definition von Statement in der Grammatik der Zwischensprache – siehe Abbildung 9.4 (Seite 147) – müssen Übersetzungen für ⟨assignment⟩s, ⟨method-call⟩s und data-race-marks erzeugt werden.

10.1.1 Übersetzung der Assignments

Die Übersetzung der Assignments funktioniert wie folgt:

- Für jedes der normalen Zuweisungsziele des Assignments wird ein Prozess erzeugt, der die Form »`set_<ID>_to_<VALUE> -> SKIP`« hat.
- Ist in den Zuweisungszielen `OTHER` enthalten, wird der Prozess `SKIP` dieser Prozessmenge noch hinzugefügt.
- Die Übersetzung des Assignments ist dann die Auswahl aus allen diesen Prozessen, gefolgt vom Sequenz-Operator »`;`«.

10.1.1.1 Effekt der Übersetzung

Die erzeugten Übersetzungen der Assignments sind unvollständige Ausdrucke. Auf sie muss ein CSP_M-Konstrukt folgen, das einen Prozess darstellt, damit gültige Konstrukte entstehen. Das folgende Konstrukt ist entweder das nächste Statement oder die Übersetzung des ⟨successor⟩s. Diese ist in Abschnitt 10.2 beschrieben. In passender Einbettung sorgen die unvollständigen Ausdrücke dafür, dass genau die zu erwartenden Signale in den Traces auftauchen.

10.1.1.2 Beispiele

Um den Effekt der Übersetzung zu verstehen, bietet es sich an, einige Beispiele zu betrachten. Dafür werden die in Tabelle 10.1 aufgezählten Zuweisungen verwendet.

Tabelle 10.1: Unterschiedliche Zuweisungen und die aus ihnen erzeugten CSP_M-Fragmente

#	**Zuweisung**	**CSP_M-Fragment (Γ)**	Tr(B ; Γ A)
1	`ONE OF OTHER IS ASSIGNED UNKNOWN`	`SKIP ;`	$\langle\!\langle \texttt{before} \cdot \texttt{after} \rangle\!\rangle^{\mathrm{Pref}}$
2	`ONE OF a IS ASSIGNED 5`	`(set_a_to_5 -> SKIP) ;`	$\langle\!\langle \texttt{before} \cdot \texttt{set\_a\_to\_5} \cdot \texttt{after} \rangle\!\rangle^{\mathrm{Pref}}$
3	`ONE OF a b IS ASSIGNED 3`	`(set_a_to_3 -> SKIP |~| set_b_to_3 -> SKIP) ;`	$\langle\!\langle \texttt{before} \cdot \texttt{set\_a\_to\_3} \cdot \texttt{after} \rangle\!\rangle^{\mathrm{Pref}}$ $\cup$ $\langle\!\langle \texttt{before} \cdot \texttt{set\_b\_to\_3} \cdot \texttt{after} \rangle\!\rangle^{\mathrm{Pref}}$
4	`ONE OF a IS ASSIGNED UNKNOWN`	`(set_a_to_unknown -> SKIP) ;`	$\langle\!\langle \texttt{before} \cdot \texttt{set\_a\_to\_unknown} \cdot \texttt{after} \rangle\!\rangle^{\mathrm{Pref}}$
5	`ONE OF a OTHER IS ASSIGNED 7`	`(set_a_to_7 -> SKIP |~| SKIP) ;`	$\langle\!\langle \texttt{before} \cdot \texttt{set\_a\_to\_7} \cdot \texttt{after} \rangle\!\rangle^{\mathrm{Pref}}$ $\cup$ $\{\langle \texttt{before} \cdot \texttt{after} \rangle\}$

Die Prozesse `B` und `A` seien definiert durch
`B = before -> SKIP` und
`A = after -> STOP`.

Um Traces angeben zu können, werden die Fragmente in eine passende Umgebung eingebettet, sodass aus Fragmenten vollständige Prozesse werden. Diese Umgebung besteht aus den Prozessen `B` und `A`. Diese sind so gewählt, dass das fragliche Verhalten von den Signalen `before` und `after` eingeschlossen wird.

Zuweisung 1: Zuweisungen dieser Form haben keinen Effekt. Dies lässt sich an den Traces erkennen: Zwischen den Signalen `before` und `after` steht in keinem der Traces ein weiteres Signal. Der Prozess, von dem die Traces stammen, lässt sich vereinfachen. Am Ergebnis lässt sich erkennen, dass die Traces korrekt angegeben wurden:

```
  B ; (SKIP ; A)
= (before -> SKIP) ; (SKIP ; (after -> STOP))          (Def.)
= before -> (SKIP ; (SKIP ; (after -> STOP)))   (Seq.L2, Seq.L4)
= before -> (after -> STOP)                            (Seq.L1)
```

Üblichen Konventionen folgend, bezeichnet hier Seq.L2 das Gesetz L2 des Sequentialisierungsoperators, wie es in Hoare [Hoa04] auf Seite 157 angegeben ist. Dabei ist Seq.L2 das Assoziativgesetz des Operators `;` und Seq.L4 besagt, dass die Operatoren `;` und `->` miteinander assoziieren. Seq.L1 bestimmt `SKIP` als neutrales Element des Sequenzoperators.

Zuweisung 2: Aus dieser Zuweisung wird ein Strong-Update der Zustandsvariablen. In den Traces steht vor dem Signal `after` stets das Signal `set_a_to_5`.

Zuweisung 3 enthält erwartungsgemäß eine Alternative: Es wird entweder das Signal `set_a_to_3` oder das Signal `set_b_to_3` ausgesandt.

Zuweisung 4 zeigt, dass der Wert `UNKNOWN` an dieser Stelle keiner Sonderbehandlung bedarf.

Bei **Zuweisung 5**, die ein Weak-Update erzeugt, lässt sich durch Vereinfachung plausibel machen, dass das Verhalten den Erwartungen entspricht.

```
  B ; ((set_a_to_7 -> SKIP) |~| SKIP ; A)
= B ; (((set_a_to_7 -> SKIP) ; A) |~| SKIP ; A)          (Seq.L2A)
= B ; ((set_a_to_7 -> ; A) |~| A)                    (Seq.L1, Seq.L4)
= (B ; (set_a_to_7 -> A)) |~| (B ; A)                    (Seq.L2B)
```

Entweder steht zwischen den Signalen `before` und `after` das Zuweisungssignal `set_a_to_7` – oder eben nicht.

10.1.1.3 Komplexitätsbetrachtungen

An den vorangehenden Beispielen wird deutlich, dass bei der maschinellen Bearbeitung von CSP_M-Prozessen Vorsicht geboten ist. Stehen zum Beispiel N Weak-Updates in Folge in einem Prozess, so hat die Definition des Prozesses eine Länge in $\mathcal{O}(n)$. Daraus ergeben sich aber unter Umständen $\mathcal{O}(2^n)$ zu unterscheidende Prozesszustände. Zum Teil lässt sich bei der maschinellen Verarbeitung eine Explosion des Zustandsraumes vermeiden. Erfolgt zum Beispiel eine Expansion so spät wie möglich (»lazy«-Evaluation von Prozessausdrücken), stellt sich möglicherweise heraus, dass die Expansion gänzlich unterbleiben kann, weil der Ausdruck ohnehin unerreichbar ist.

10.1.2 Übersetzung der Prozeduraufrufe

Die Übersetzung der als ⟨method-call⟩ bezeichneten Statements, also der Prozeduraufrufe, ist sehr einfach. Der Prozeduraufruf enthält eine Liste der Start-Basic-Blocks aller in Frage kommenden Aufrufziele.

Die Übersetzung eines Prozeduraufrufs ist die Auswahl aus den Prozessen, die die Start-Basic-Blocks repräsentieren, gefolgt vom Sequenz-Operator »`;`«. Aufgrund der Namensgleichheit des Prozesses und der Basic-Blöcke muss hier noch nicht einmal ein Transfer der Bezeichner erfolgen.

Tabelle 10.2: Übersetzung von Call-Statements: keine Überraschungen

#	**Prozeduraufruf**	**CSP$_M$-Fragment**
1	`CALL C123B1234`	`C123B1234 ;`
2	`CALL C123B1234 C123B2345`	`(C123B1234 |~| C123B2345) ;`
3	`CALL C8B12 C8B23 C8B34 ...`	`(C8B12 |~| C8B23 |~| C8B34 |~| ...) ;`

10.1.2.1 Beispiele

Die Beispiele zu Prozeduraufrufen sind entsprechend einfach. Tabelle 10.2 stellt mögliche Übersetzungen dar.

Prozeduraufruf 1: Dies ist der Standardfall. In den untersuchten Systemen sind einfache Aufrufe die Regel. Die Aufrufausdrücke haben somit in der Regel genau ein Ziel.

Prozeduraufruf 2: Bei der Dereferenzierung von Funktionszeigern ist es möglich, dass mehrere Aufrufziele vorliegen. Die Übersetzung ist dann die Auswahl aus den jeweiligen Zielen. Auch wenn die Liste der Aufrufziele sehr lang ist, muss nicht von diesem Muster abgewichen werden. Siehe dazu Prozeduraufruf 3.

10.1.3 Übersetzung der Data-Race-Marks

Als Statement kommen in den Basic-Blocks auch Data-Race-Marks vor. Zum Zeitpunkt der Übersetzung steht schon fest, welches Zugriffspaar untersucht werden soll. Data-Race-Marks, die nicht zu diesem Paar gehören, werden einfach ignoriert. Die Übersetzung der relevanten Data-Race-Marks wird in Abschnitt 10.4 (Seite 182) erläutert.

10.1.4 Übersetzung des Kontrollflusses zwischen den Basic-Blocks

Jeder Basic-Block hat eine Klausel, die bestimmt, wohin der Kontrollfluss nach dem Basic-Block führt. Die Klausel kann drei unterschiedliche Formen annehmen:

1. Es gibt einen unbedingten Nachfolger (`UNCOND`).
2. Es gibt zwei mögliche Nachfolger und eine Bedingung, die darüber entscheidet, welcher der beiden zum Zuge kommt (`COND`).
3. Es handelt sich um den Exit-Basic-Block der Prozedur (`SKIP`). Der Kontrollfluss geht dann zum Aufrufer zurück beziehungsweise endet, wenn es sich um die Start-Prozedur handelt.

Alle drei Formen werden unterschiedlich übersetzt.

10.1.4.1 Unbedingter Nachfolger

Beim unbedingten Nachfolger ist die Übersetzung einfach: Die ID des Nachfolgers ist der Bezeichner des Prozesses, der das Verhalten des Basic-Blocks enthält. Die ID kann ohne Anpassung als CSP_M-Prozess ausgegeben werden.

10.1.4.2 Bedingter Nachfolger

In Abschnitt 10.2 (Seite 168) wird eine Funktion zur Übersetzung von Bedingungen beschrieben. Diese erzeugt aus einer Bedingung, einem Then- und einem Else-Nachfolger einen CSP_M-Prozess. Diese Funktion wird verwendet, um den hier benötigten Prozess zu erhalten. Dabei wird der Bedingungsausdruck einfach übergeben und für den Then- und den Else-Nachfolger werden die gelieferten IDs verwendet.

10.1.4.3 Rücksprung zum Aufrufer

Es lässt sich ermitteln, was benötigt wird, um zum Aufrufer zurückzukehren, indem erneut betrachtet wird, wie Prozeduraufrufe gehandhabt werden. Ein

```
1  BLOCK (C1_B11) { } (UNCOND C1_B12)
2
3  BLOCK (C2_B22) { } (COND (TRUE)
4        (FALSE C2_B23)
5        (TRUE C2_B24))
6
7  BLOCK (C3_B33) { } (SKIP)
```

```
1  C1_B11 = C1_B12
2
3  C2_B22 = C2_B24
4
5  C3_B33 = SKIP
```

Oben: Drei leere Basic-Blocks in der Zwischensprache

Unten: Die Übersetzung in CSP_M

Abbildung 10.1: Übersetzung der drei Formen von Nachfolgern

Prozeduraufruf überträgt zunächst den Kontrollfluss auf den Start-Basic-Block der aufzurufenden Prozedur. Mittels des Sequentialisierungsoperators `;` wird dann auf die »successful termination« gewartet. Um zum Aufrufer zurückzukehren, muss man also zum Prozess `SKIP` werden – dem Prozess, der nichts anderes tut, als erfolgreich zu terminieren. Die Übersetzung ist somit einfach »`SKIP`«.

10.1.4.4 Beispiele für Nachfolger

Abbildung 10.1 zeigt Beispiele für alle drei Formen von Nachfolgern. Die drei oben dargestellten leeren Basic-Blocks werden wie unten dargestellt übersetzt. Die zweite Übersetzung ist sehr einfach und gleicht der ersten. Dies ist der Tatsache geschuldet, dass die Bedingung hier einfach `TRUE` ist und es somit trotz Bedingung einen feststehenden Nachfolger gibt. Die Übersetzung komplexerer Bedingungen erläutert der Abschnitt 10.2.

```
1  volatile int a;
2  volatile int b;
3
4  void p() {
5     a = 1;
6     p();
7     b = 1;
8  }
9
10 void q() {
11    a = 1;
12    if (IND) q();
13    b = 1;
14 }
```

```
1  channel a, b
2
3  P = a -> P ; b -> SKIP
4  Q = a -> (Q |~| SKIP) ; b -> SKIP
```

Links: Direkt-rekursive C-Prozeduren p und q
Rechts: Versuch der Umsetzung eines ähnlichen Verhaltens in CSP_M

Abbildung 10.2: Versuch, direkte und indirekte Rekursion umzusetzen

10.1.5 Komplikationen mit starken Zusammenhaltsgruppen im Aufrufgraphen

Tatsächlich wurde eine Komplikation bei der Übersetzung des Aufrufgraphen bisher übergangen. Rekursive Prozeduren beziehungsweise Funktionen, insbesondere wenn diese indirekt rekursiv sind, sorgen unter Umständen wie im Folgenden dargelegt für Komplikationen.

Hoare [Hoa04] definiert auf Seite 77 »Konstruktivität« für Funktionen auf Traces. Eine stetige und monotone Funktion F auf Traces ist konstruktiv genau dann, wenn das Verhalten von $F(X)$ auf den ersten $(n+1)$ Schritten nur von den ersten n Schritten von X abhängt. (Er bezieht sich dabei auf »die Fixpunkt-Theorie von Scott«. Gemeint ist wohl Scott [Sco72].)

Ist F definiert auf Traces (Traces sind stets ein vollständiger Verband) eine stetige und monotone Funktion, dann existiert der kleinste Fixpunkt $\mu X.F(X)$ (Fixpunktsatz).

Hoare zeigt, dass, wenn F auch konstruktiv ist, ein eindeutiger Fixpunkt existiert. Die Refinement-Checker FDR2 und FDR3 nutzen dies anscheinend,

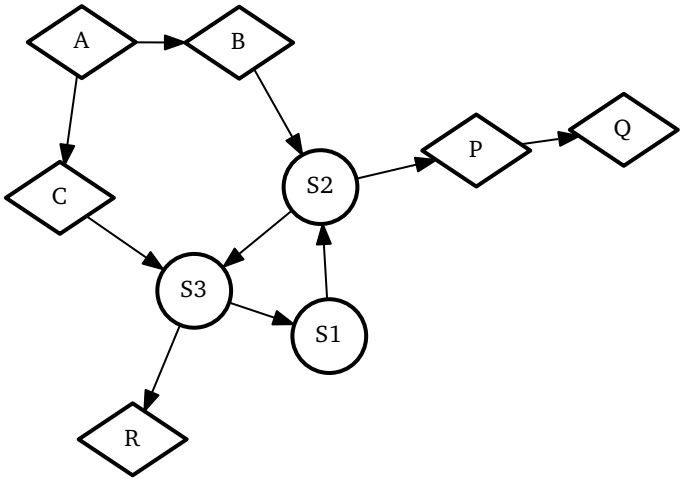


Abbildung 10.3: Beispiel eines Aufrufgraphens mit einer starken Zusammenhangskomponente

um endliche generalisierte kantenbeschriftete Transitionssysteme für mittels μ-Rekursionen definierte Prozesse mit unendlich großen Traces zu finden und diese somit handhabbar zu machen.

Hoare listet CSP-Operatoren, aus denen nur konstruktive Funktionen entstehen. Der Sequenz-Operator `;` gehört nicht dazu.

Dies begründet, warum die in Abbildung 10.2 dargestellten Versuche, die Prozesse `P` und `Q` zu definieren, Probleme bereiten. Weder FDR2 noch FDR3 kann die dargestellte CSP_M-Datei sinnvoll verarbeiten.

10.1.5.1 Lösung

In den untersuchten Systemen kommt indirekte Rekursion vor (siehe Abschnitt 4.4). Das Werkzeug Red erkennt die durch indirekte Rekursion verursachten starken Zusammenhangskomponenten im Aufrufgraphen mittels des Algorithmus von Tarjan.

Alle schreibenden Zugriffe aus Basic-Blocks von Prozeduren in nichttrivialen starken Zusammenhangskomponenten werden jeweils in einer Liste

zusammengefasst. Für jede starke Zusammenhangskomponente entsteht somit eine solche Liste. Ebenso werden Aufrufe, die die starke Zusammenhangskomponente verlassen, der Liste hinzugefügt. Aufrufe innerhalb der starken Zusammenhangskomponente können jedoch ignoriert werden.

Bei der Anwendung auf die hier untersuchten Systeme sind die Listen tatsächlich normalerweise leer. Sie enthalten also weder Aufrufe noch schreibende Zugriffe.

Für jede der nicht-trivialen starken Zusammenhangskomponenten wird ein Prozess erzeugt, der eine beliebige Teilmenge der aufgelisteten Schreibzugriffe und Prozeduraufrufe in beliebiger Reihenfolge und beliebig oft ausführen kann und danach zum Prozess `SKIP` wird. Der Kontrollfluss innerhalb der Prozeduren der starken Zusammenhangskomponenten wird somit stark konservativ approximiert.[1]

Aufrufe an Prozeduren aus nicht-trivialen starken Zusammenhangskomponenten verweisen auf den jeweiligen Prozess. Für die zu starken Zusammenhangskomponenten zusammengefassten Prozeduren wird kein eigener Prozess generiert.

Das Vorgehen approximiert das Verhalten konservativ. Es kann auch nur dann zu einem Genauigkeitsverlust kommen, wenn innerhalb der nicht-trivialen starken Zusammenhangskomponenten Zustandsvariablen geschrieben werden. Probleme mit Fixpunkten werden umgangen.

Abbildung 10.3 zeigt ein Beispiel für einen Aufrufgraph auf Prozeduren. Hier bilden die Prozeduren `S1`, `S2` und `S3` eine starke Zusammenhangskomponente. Der für diese generierte Prozess – hier »`SCC_1`« genannt – führt beliebig häufig in beliebiger Reihenfolge die Aufrufe der Prozeduren `P` und `R`

[1] Man kann die Frage stellen, ob ein mögliches Abgleiten der Ausführung in eine unendliche Rekursion – sei sie nun direkt oder indirekt – nicht explizit berücksichtigt werden muss. Dies ist zu verneinen. Das eingesetzte Traces-Modell vermeidet es zwischen Prozessen zu unterscheiden, die an einem bestimmten Punkt aufhören können zu kommunizieren und solchen die weitermachen müssen. Dies zeigt sich deutlich an den Beispielprozessen `P` und `Q` aus Abbildung 7.4 (Seite 107). Ein Abgleiten in eine unendliche Rekursion kann, wenn innerhalb der Aufrufe keine nach außen sichtbaren Signale ausgesendet werden, mit dem Prozess `STOP` modelliert werden. Diesen Prozess könnte man also der Auswahl hinzufügen. Im Traces-Modell würde das aber die Semantik nicht ändern!
Andere Semantik-Modelle von CSP_M – insbesondere das Failures-Modell und das Failures-Divergences-Modell – sind explizit dafür ausgelegt, solche Prozesse zu unterscheiden.

aus. Die beiden Aufrufe aus den Prozeduren `B` und `C` werden beide in CSP_M mit dem Ausdruck »`SCC_1 ;` « modelliert.

10.2 Übersetzung der Bedingungen

Zur Übersetzung der Bedingungen von Basic-Blocks mit bedingtem Nachfolger wird eine Funktion Ψ definiert, siehe dazu Abbildung 10.4. Der Funktion werden zwei geklammerte CSP_M-Prozess-Ausdrücke und die Bedingung aus der Zwischendarstellung übergeben.

Mit geklammerten CSP_M-Prozess-Ausdrücken sind CSP_M-Ausdrücke gemeint, die den Typ Prozess haben, und

- atomar sind, also eine Prozessvariable oder ein vordefinierter Prozess wie `SKIP` oder `STOP` sind, oder
- ein von Klammern umschlossener Ausdruck sind.

Die übergebenen Ausdrücke sind die Then- und Else-Nachfolger der Bedingung. Das Ergebnis der Funktion ist ein geklammerter CSP_M-Prozess-Ausdruck, der in die Then- und Else-Nachfolger genau nach Prüfung der für sie geltenden impliziten Bedingungen übergehen kann.

10.2.1 Effekt und Beispiele

Der Effekt der Übersetzung lässt sich gut an Beispielen erkennen.

10.2.1.1 Beispiel 1

Das erste betrachtete Beispiel ist das des abstrakten Prädikats $\mathtt{a} = 1$. In Tabelle 9.2 (Seite 144) kann bei Nummer 1 die Erwartung für dieses Beispiel an den Bedingungen von Then- und Else-Zweig abgelesen werden.

Die Bedingung wird in ein generisches Beispiel eingebettet. Siehe dazu Abbildung 10.5.

Drei Basic-Blöcke sind hier relevant: Der Basic-Block B, der einen bedingten Nachfolger hat und mit der Auswertung des Prädikats endet, der Basic-Block

Sei C die Menge geklammerter CSP_M-Prozess-Ausdrücke und P die Menge der Ausdrücke in der Prädikatensprache. Dann lässt sich $\Psi : (C \times C \times P) \to C$ über der Struktur der Prädikate definieren:

$$
\begin{aligned}
\Psi\big(T, F, \underline{\texttt{x = 4}}\big) &= \texttt{(x\_is\_4 -> } T \texttt{ |\~| x\_isnot\_4 -> } F\texttt{)} \\
\Psi\big(T, F, \underline{\texttt{x} \neq \texttt{4}}\big) &= \Psi\big(F, T, \underline{\texttt{x = 4}}\big) \\
\Psi\big(T, F, \underline{\neg\alpha}\big) &= \Psi\big(F, T, \underline{\alpha}\big) \\
\Psi\big(T, F, \underline{\alpha \wedge \beta}\big) &= \Psi\Big(\Psi\big(T, F, \underline{\beta}\big), F, \underline{\alpha}\Big) \\
\Psi\big(T, F, \underline{\alpha \vee \beta}\big) &= \Psi\Big(T, \Psi\big(T, F, \underline{\beta}\big), \underline{\alpha}\Big) \\
\Psi\big(T, F, \underline{\alpha \oplus \beta}\big) &= \Psi\big(T, F, \underline{(\alpha \vee \beta) \wedge \neg(\alpha \wedge \beta)}\big) \\
\Psi\big(T, F, \underline{\alpha \equiv \beta}\big) &= \Psi\big(T, F, \underline{\neg(\alpha \oplus \beta)}\big) \\
\Psi\big(T, F, \underline{\texttt{IND}}\big) &= \texttt{(}T\texttt{ |\~| }F\texttt{)} \\
\Psi\big(T, F, \underline{\texttt{true}}\big) &= T \\
\Psi\big(T, F, \underline{\texttt{false}}\big) &= F
\end{aligned}
$$

»x« steht für eine beliebige Zustandsvariable, »4« für eine beliebige Integer-Konstante. Die prädikatenlogischen Symbole haben ihre übliche Bedeutung. Das Symbol $\oplus$ steht für Antivalenz (`xor`); das Symbol $\equiv$ steht für Äquivalenz (`xnor`). Reihenfolge der Booleschen-Operatoren entspricht der Reihenfolge in Abbildung 9.6.

Abbildung 10.4: Übersetzungs-/Interpretationsfunktion

```
1  /* B */ ;
2  if (Prädikat) ; /* T */
3  else          ; /* F */ ;
```

Abbildung 10.5: Generisches Beispiel für Pfadprädikate

T, der den Then-Zweig repräsentiert, und der Basic-Block F, der den Else-Zweig repräsentiert. Drei CSP_M-Ausdrücke werden definiert, die hier ein erkennbares Verhalten für die Basic-Blöcke beinhalten:

```
B = before -> SKIP
T = then -> STOP
F = else -> STOP
```

Im Folgenden wird das Verhalten des CSP_M-Ausdruckes »`B ; Ψ(T, F, a = 1)`« untersucht. Die Funktion Ψ lässt sich hier direkt auswerten. Damit entsteht dieser Ausdruck:

```
B ; (a_is_1 -> T |~| a_isnot_1 -> F)
```

Ohne weitere Vereinfachung sind diesem Prozess seine Traces anzusehen:

$$\langle\!\langle \texttt{before} \cdot \texttt{a\_is\_1} \cdot \texttt{then} \rangle\!\rangle^{\text{Pref}} \cup \langle\!\langle \texttt{before} \cdot \texttt{a\_isnot\_1} \cdot \texttt{else} \rangle\!\rangle^{\text{Pref}}$$

Das ist genau das gewünschte Verhalten. Der Prozess kann nach Prüfung der Bedingung in den Then-Zweig und nach Prüfung der negierten Bedingung in den Else-Zweig absteigen.

10.2.1.2 Beispiel 2

Das zweite Beispiel ist das des abstrakten Prädikats `IND ∨ b = 1`. In Tabelle 9.2 (Seite 144) können wieder, diesmal bei Nummer 6, die Erwartungen abgelesen werden. Verwendet wird dasselbe Setup wie beim ersten Beispiel, so dass folgender CSP_M-Ausdruck von Interesse ist:

```
B ; Ψ(T, F, IND ∨ b = 1)
```

Die Auswertung der Funktion Ψ ist:

$$
\begin{aligned}
&\texttt{B ; } \Psi\big(\texttt{T, F, } \underline{\texttt{IND} \vee \texttt{b = 1}}\big) \\
=\ &\texttt{B ; } \Psi\big(\texttt{T, } \Psi\big(\texttt{T, F, } \underline{\texttt{b = 1}}\big)\texttt{, } \underline{\texttt{IND}}\big) \\
=\ &\texttt{B ; (T |\~| (b\_is\_1 -> T |\~| b\_isnot\_1 -> F))} \\
=\ &\texttt{B ; (T |\~| b\_is\_1 -> T |\~| b\_isnot\_1 -> F)}
\end{aligned}
$$

Auch für diesen Prozess lassen sich leicht Traces ermitteln:

$$
\begin{gathered}
\left\langle\!\!\left\langle \underline{\texttt{before} \cdot \texttt{then}} \right\rangle\!\!\right\rangle^{\text{Pref}} \cup \\
\left\langle\!\!\left\langle \underline{\texttt{before} \cdot \texttt{b\_is\_1} \cdot \texttt{then}} \right\rangle\!\!\right\rangle^{\text{Pref}} \cup \\
\left\langle\!\!\left\langle \underline{\texttt{before} \cdot \texttt{b\_isnot\_1} \cdot \texttt{else}} \right\rangle\!\!\right\rangle^{\text{Pref}}
\end{gathered}
$$

Auch diese entsprechen weitgehend den Erwartungen: In den Then-Zweig kann bedingungslos abgestiegen werden. Dies entspricht dem Fall, in dem `IND true` ist. In den Else-Zweig hingegen kann nur abgestiegen werden, nachdem geprüft wurde, dass die Zustandsvariable `b` nicht `1` ist. Zusätzlich gibt es einen dritten Fall: Dieser ist hier in der Mitte zu erkennen. Hier wird die Bedingung geprüft und dann in den Then-Zweig eingestiegen. Er ist semantisch unnötig, da er durch den mächtigeren unbedingten Abstieg überschattet wird[1].

10.3 Durchsetzung der Zustandsvariablensemantik

Die im vorhergehenden Abschnitt beschriebene Übersetzung der Statements und des Kontrollflusses berücksichtigt die Semantik der Zustandsvariablen nicht. Wie im Folgenden zu sehen ist, ignoriert die Übersetzung zunächst den Zustand, das heißt den Wert der Zustandsvariablen. Sie verhält sich

[1] Einige Optimierungen auf Bedingungsausdrücken wurden ausprobiert. Diese hatten bei Anwendung auf den realen Systemen kaum positive Auswirkungen, weil die dort auftretenden Bedingungen fast immer (wohl aus stilistischen Gründen) sehr einfach gehalten sind. Solche Optimierungen könnten aber prinzipiell den hier beschriebenen Ausdruck verkleinern.

```
1  void p (void) {
2     a = 1;
3     while (a == 1)
4        if (IND)
5           a = 2;
6  }
```

```
1  P  = B1
2  B1 = set_a_to_1 -> B2
3  B2 = a_is_1 -> B3 |~| a_isnot_1 -> B5
4  B3 = B4 |~| B2
5  B4 = set_a_to_2 -> B2
6  B5 = SKIP
```

Oben: C-Quelltext, der mit der Zustandsvariablen a arbeitet.
Unten: Vereinfachter CSP_M-Output (Übersetzung) des Quellcodes. Die Übersetzung erfolgt hier Zeile für Zeile.

Abbildung 10.6: Vergleich einer C-Funktion mit dem vereinfachten CSP_M-Output

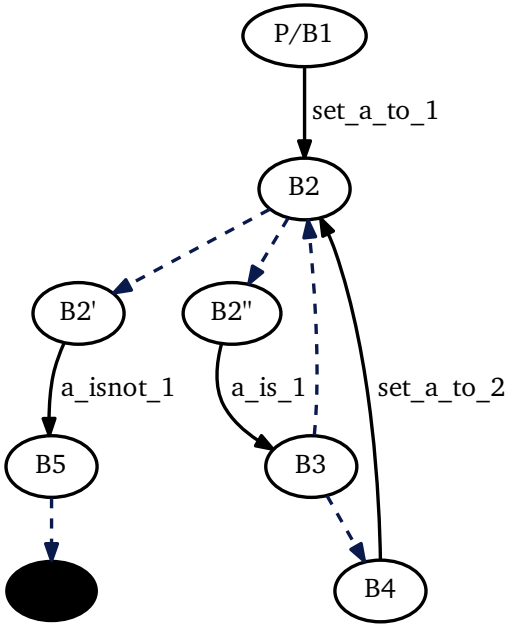
»erinnerungslos«. Dieser Abschnitt beschreibt, wie sich die Zustandsvariablensemantik, also der sich ändernde Zustand, in CSP_M durchsetzen lässt.

10.3.1 Erinnerungsloses Verhalten

Zunächst wird das Verhalten ohne die Durchsetzung der Zustandsvariablensemantik betrachtet, das heißt das erinnerungslose Verhalten. Hierzu wird die in Abbildung 10.6 dargestellte Prozedur p betrachtet.

Abbildung 10.6 zeigt auch – in vereinfachter Form – den aus der Prozedur erzeugten CSP_M-Output. Um das Verhalten zu veranschaulichen, werden diesmal nicht die Traces des Prozesses betrachtet, sondern ein generalisiertes kantenbeschriftetes Transitionssystem (siehe dazu Abschnitt 7.4 (Seite 108)). Es ist in Abbildung 10.7 dargestellt und wird in dieser Form von FDR2 erzeugt, wenn keine Kompression gefordert ist.

Während sich ohne Normalisierung klar ein Zusammenhang zwischen dem Graphen und dem ursprünglichen Programmcode erkennen lässt, ist



Gestrichelte Pfeile stellen τ-Transitionen dar; die schwarz ausgefüllten Knoten entsprechen dem Zutand `SKIP`; Knotenbeschriftungen sind informell die zugehörigen CSP_M-Prozesse.

Abbildung 10.7: Generalisiertes kantenbeschriftetes Transitionssystem, erzeugt von FDR2 aus dem Prozess `P` aus Abbildung 10.6

das komprimierte generalisierte kantenbeschriftete Transitionssystem kleiner und somit möglicherweise einfacher zu verstehen. Das Ergebnis der Normalisierung des generalisierten kantenbeschrifteten Transitionssystems aus Abbildung 10.7 durch FDR2 ist in Abbildung 10.8 dargestellt.

10.3.1.1 Erinnerungslosigkeit

Im Folgenden werden einige Pfade in den generalisierten kantenbeschrifteten Transitionssystemen betrachtet. Ein Pfad im normalisierten Transitionssystem ist natürlich auch ein Pfad im nicht normalisierten. Es ist schließlich eine

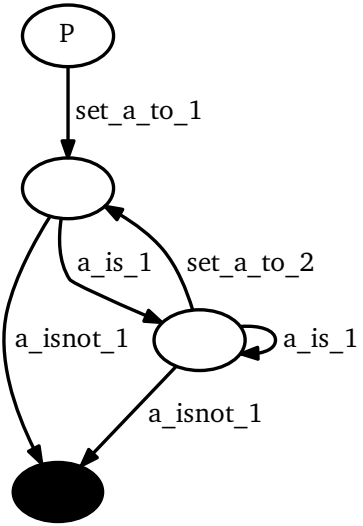


Abbildung 10.8: Normalisierung des generalisierten kantenbeschrifteten Transitionssystems aus Abbildung 10.7 durch FDR2

Eigenschaft der Kompression, genau das zu gewährleisten. Die Pfade sind – ebenfalls gemäß der Konstruktion – Traces des Prozesses aus Abbildung 10.6 (Seite 172).

Der Pfad[1] $\langle$`set_a_to_1 · a_isnot_1`$\rangle$ führt zum `SKIP`-Zustand. Auch der Pfad $\langle$`set_a_to_1 · a_is_1 · a_isnot_1`$\rangle$ führt zum `SKIP`-Zustand. Beide Pfade passen zum Kontrollfluss der in Abbildung 10.6 (Seite 172) dargestellten Prozedur, sind aber nicht mit der Semantik der Variablenzuweisung kompatibel. Der Prozess »erinnert« sich nicht an zuvor ausgeführte Zuweisungen und Vergleiche, wenn er das nächste Signal aussendet.

10.3.2 Wertsensitives Verhalten

Um ein semantisch sinnvolles Verhalten der Zustandsvariablen zu gewährleisten, wird für jede Zustandsvariable ein Prozess erzeugt, der genau die

[1] Es werden identische Darstellungen für Pfade und Traces verwendet, da es zwischen ihnen einen offensichtlichen Isomorphismus gibt.

```
1  channel a_is_0, a_isnot_0, set_a_to_0
2  channel a_is_1, a_isnot_1, set_a_to_1
3  channel set_a_to_unknown
4
5  A_0   = A_SET |~| a_is_0 -> A_0 |~| a_isnot_1 -> A_0
6  A_1   = A_SET |~| a_is_1 -> A_1 |~| a_isnot_0 -> A_1
7  A_OTH = A_SET |~| a_isnot_0 -> A_OTH |~| a_isnot_1 ->
       A_OTH
8
9  A_SET = set_a_to_0 -> A_0 |~| set_a_to_1 -> A_1 |~|
       set_a_to_unknown -> A_UNKNOWN
10
11 A_UNKNOWN = A_0 |~| A_1 |~| A_OTH
```

Abbildung 10.9: Variablenprozess für eine Variable a, die auf die Werte 0 und 1 gesetzt werden kann

Semantik einer Variablen repräsentiert. Später wird damit dann der Prozess synchronisiert, der den Kontrollfluss repräsentiert. Der dadurch erzeugte Prozess kann dann Signale, die zur jeweiligen Variablen gehören, genau dann aussenden, wenn dies sowohl durch den Kontrollfluss als auch durch die Wertesemantik des Prozesses erlaubt wird.

10.3.2.1 Variablenprozesse

Der Variablenprozess wird mit Hilfe des Wissens erzeugt, welche statisch bekannten Werte der Zustandsvariablen zugewiesen werden können. Damit können die Variablenprozesse klein gehalten werden. Abbildung 10.9 zeigt einen Variablenprozess für eine Variable a. Die Generierung enthält weitere Einsparmechanismen. Kommt zum Beispiel das Signal `a_isnot_0` in keinem Prädikat vor, so wird es auch bei der Generierung direkt ausgelassen.

10.3.2.2 Erstellung des Variablenprozesses

Während der Übersetzung des Kontrollflusses merkt sich das Übersetzungswerkzeug die vorkommenden statisch bekannten Werte und die benötigten Vergleichssignale. Es wird dann ein Codefragment generiert, das den Variablenprozess enthält.

Dazu werden zunächst die benötigten Signale deklariert (siehe Zeilen 1 – 3 in Abbildung 10.9). Dann werden Zustandsprozesse für alle vorkommenden statisch bekannten Werte und den Spezialwert `OTHER` erzeugt (siehe Zeilen 5 – 7). Diese können jeweils beliebig oft passende Vergleichssignale aussenden oder mit einem `set`-Signal in den passenden (neuen oder alten) Zustand wechseln. Um Wiederholung zu vermeiden, wird dieses überall gleiche Verhalten in einen eigenen Prozess ausgelagert (siehe Zeile 9). Zuletzt wird noch der Prozess `UNKNOWN` generiert. Er besteht aus der Auswahl unter allen Zustandsprozessen (siehe Zeile 11).

10.3.2.3 Generalisierte kantenbeschriftete Transitionssysteme der Variablenprozesse

Auch wenn das Schema der Variablenprozesse recht einfach ist, sind die resultierenden generalisierten kantenbeschrifteten Transitionssysteme aufgrund der großen Anzahl an Kanten recht unübersichtlich. Abbildung 10.10 zeigt ein solches Beispiel.

Das generalisierte kantenbeschriftete Transitionssystem wird so tatsächlich nicht von FDR2 oder FDR3 erzeugt. Ohne Kompressionen erzeugt FDR3 ein generalisiertes kantenbeschriftetes Transitionssystem mit 19 Knoten und zahlreichen τ-Transitionen. Schon mit einfachen Kompressionen sinkt die Zahl der Knoten auf 4, dieses generalisierte kantenbeschriftete Transitionssystem enthält dann aber keine τ-Transitionen, dafür jedoch deutlich mehr Kanten als das hier dargestellte.

In der Praxis sind die zahlreichen Kanten unproblematisch. Zum einen hängt die Laufzeit von Refinement-Checkern im Wesentlichen von der Zahl der Knoten und nicht der Zahl der Kanten ab (siehe Reference Manual von

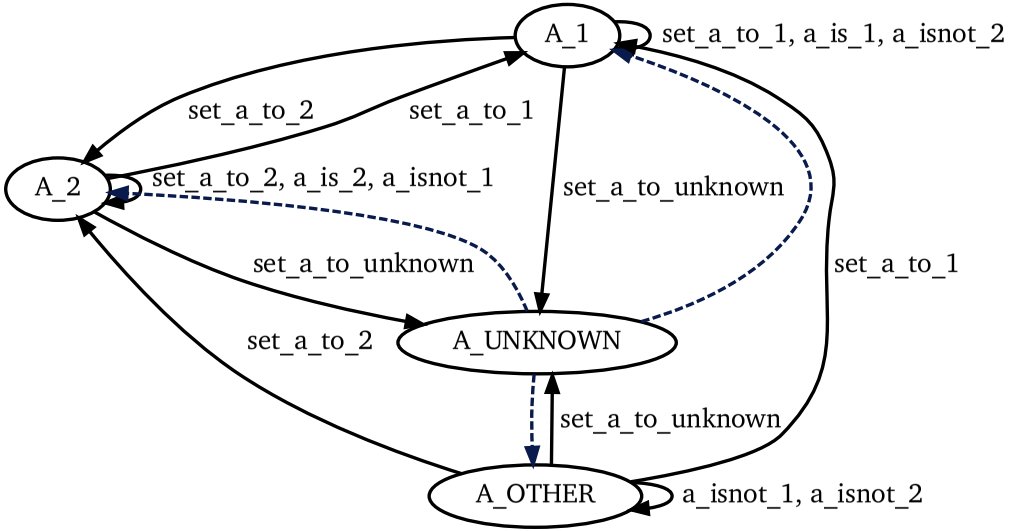


Abbildung 10.10: Ein generalisiertes kantenbeschriftetes Transitionssystem, das Prozesse aus Abbildung 10.9 repräsentiert

```
1  volatile unsigned
       int a;
2
3  void p (void) {
4     if (a == 1)
5        a = 2;
6     else if (a == 2)
7        a = 1;
8     else
9        f();
10 }
```

Abbildung 10.11: C-Programm, in dem die Zustandsvariable `a` den Wert `OTHER` annehmen kann

FDR2 [FO10]) und zum anderen können in realen Programmen viele der Kanten weggelassen werden, weil es im Programm keine entsprechenden Anweisungen oder Bedingungen gibt. Das Werkzeug Red2CSP nutzt dies konsequent aus. Es werden nur die benötigten Kanten eingefügt.

10.3.2.4 Spezialwert »OTHER«

Dass man beim Einsparen von Zuständen und Signalen bei den Variablenprozessen nicht zu weit gehen darf, zeigt sich am Spezialwert `OTHER`.

Um dessen Existenz zu motivieren, wird das Codebeispiel in Abbildung 10.11 betrachtet.

Gäbe es hier den Zustand `OTHER` im Modell nicht, könnte der Refinement-Checker durch einen dann zulässigen »tertium non datur«-Schluss den Aufruf an die Prozedur `f` als tot betrachten.

Dies lässt sich durch genaues Studium von Abbildung 10.11 in Verbindung mit Abbildung 10.10 nachvollziehen: Dazu wird parallel durch das Programm und durch das generalisiertes kantenbeschriftetes Transitionssystem geschritten. Begonnen wird mit dem Einstieg in die Prozedur `P` und im Zustand `A_UNKNOWN`. Geendet werden soll im Else-Zweig. Die erste Bedingung

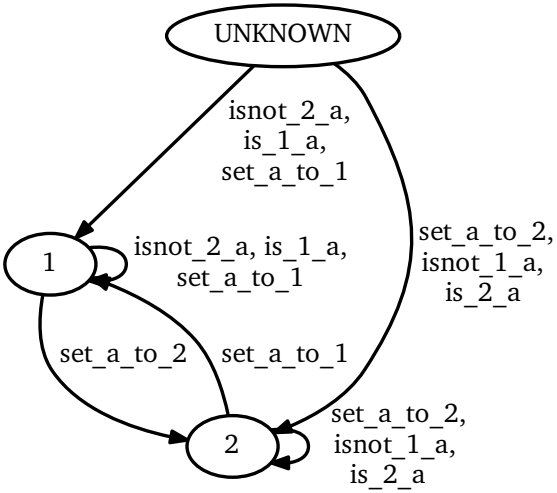


Abbildung 10.12: Generalisiertes kantenbeschriftetes Transitionssystem, das eine Variable ohne den Spezialwert OTHER zeigt

muss demnach falsch sein, es muss das Signal a_isnot_1 ausgesendet werden. Dies ist nur möglich, indem entweder zunächst der τ-Transition in den Zustand A_2 gefolgt wird oder indem zunächst der τ-Transition in den Zustand A_OTHER gefolgt wird. Aber nur in letzterem Fall kann danach auch das Signal a_isnot_1 ausgesandt werden, was geschehen muss, um in Else-Zweig zu landen. Im Zustand UNKNOWN lässt sich somit genau dann der Pfad $\langle \overline{\texttt{isnot\_1} \cdot \texttt{isnot\_2}} \rangle$ beschreiten, welcher beschritten werden muss, um zum Else-Zweig zu gelangen, wenn man zunächst der τ-Transition zum Knoten OTHER folgt.

Weiterhin lässt sich in Abbildung 10.12 erkennen, wie sich der fehlende OTHER-Zustand auswirkt. In der Abbildung ist ein durch Normalisierung entstandenes generalisiertes kantenbeschriftetes Transitionssystem abgebildet, das aus einer passend zum Programm in Abbildung 10.11 ausgerüsteten Zustandsvariable hergeleitet wurde. Umwandlung und Komprimierung erfolgten hier durch FDR3. Es wurde allerdings der Spezialwert OTHER entfernt. Hier können Fehlschlüsse entstehen. Unabhängig vom Ausgangszustand,

```
1  --  Synchronisation mit dem Variablenprozess
2  PROG = C456B567 [|Σa|] A_UNKNOWN
3
4  --  Synchronisation mit dem Variablenprozess
5  --  einschliesslich Verdeckung der Variablensignale
6
7  PROG = (C456B567 [|Σa|] A_UNKNOWN) \ Σa
```

Abbildung 10.13: Die Synchronisation mit dem Variablenprozess und die Verdeckung der Variablensignale

führt das Signal `isnot_1_a` zum Zustand 2. Dabei könnte der Wert der Variablen in einem realen Programm auch beispielsweise 3 sein.

10.3.2.5 Synchronisation

Die den Kontrollfluss repräsentierenden Prozesse werden mit dem Variablenprozess mit dem Alphabetised-Parallel-Operator »[| Σ |]« synchronisiert. Die Menge Σ ist die Menge der zu der jeweiligen Variablen gehörenden Signale. Die in Σ enthaltenen Signale kann der kombinierte Prozess also nur aussenden, wenn seine beiden Subprozesse darin übereinstimmen. Alle anderen Signale – dies betrifft nur den Kontrollflussprozess – können unabhängig voneinander ausgesendet werden.

In Abbildung 10.13 wird diese Synchronisation dargestellt. Dabei ist Σ_a die Menge der Variablensignale.

Als Beispiel ist erneut das Programm aus Abbildung 10.6 zu betrachten. Der Kontrollflussprozess ist in Abbildung 10.7 (Seite 173) und in Abbildung 10.8 (Seite 174) als generalisiertes kantenbeschriftetes Transitionssystem dargestellt. Als Variablenprozess eignet sich der Prozess `A_UNKNOWN` aus Abbildung 10.9. Da Variablen im hier verwendeten Modell zunächst uninitialisiert sind, wird `A_UNKNOWN` als Startprozess genutzt. Variablenprozesse starten damit immer mit dem allgemeinsten, das heißt am wenigsten verfeinerten Prozess.

Beide Prozesse werden synchronisiert und es wird FDR2 verwendet, um das Ergebnis zu komprimieren. Abbildung 10.14 zeigt das Ergebnis. Man

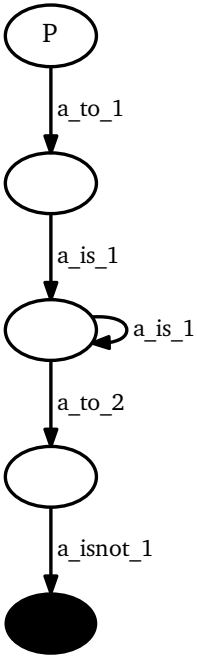


Abbildung 10.14: Komprimiertes generalisiertes kantenbeschriftetes Transitionssystem des Prozesses `P [|Σ|] A_UNKNOWN`, erzeugt von FDR2

erkennt, dass das Programm die Schleife durchläuft und immer wieder verifiziert, dass die Zustandsvariable noch den Wert 1 hat, bis der Wert der Zustandsvariablen auf 2 gesetzt wird und es folglich die Schleife verlässt.

10.3.2.6 Verdeckung

Tatsächlich werden die Variablensignale nicht nur zur Synchronisation verwendet, sondern nach der Synchronisation auch direkt mit dem Verdeckungs-Operator versteckt. Daher entspricht die zweite Definition in Abbildung 10.13 dem tatsächlich verwendeten Muster. Durch das Verstecken werden aus Kanten, die mit Signalen beschriftet sind, τ-Transitionen.

Dieses Vorgehen ist erlaubt, weil die Variablensignale zu diesem Zeitpunkt bereits all ihre beschränkende Wirkung entfaltet haben und zur Beantwortung der Frage nach Data-Races nicht mehr gebraucht werden.

Das Verhalten ist zur Steigerung der Performance ausgesprochen sinnvoll – geradezu essentiell –, da die meisten Komprimierungsstrategien der Refinement-Checker FDR2 und FDR3 nur mit τ-Transitionen effizient funktionieren (siehe Reference Manual von FDR2 [FO10]). Die Effekte der Komprimierung, die durch das Verstecken von Signalen ermöglicht werden, stellen Roscoe et al. [RGG+95] eindrucksvoll dar.

Das Verstecken der Variablensignale führt gemeinsam mit einigen anderen ähnlichen Maßnahmen allerdings auch dazu, dass keine hilfreichen Traces mehr ausgegeben werden können. Der Refinement-Checker liefert somit die Angabe, dass ein Pfad zu einer Data-Race-Situation existiert, aber nicht den tatsächlichen Pfad. Refinement-Checker machen sich typischerweise zu Nutze, dass es oftmals (zumindest scheinbar) einfacher ist, die Existenz eines Pfades zu beweisen, als den Pfad tatsächlich zu finden. (Vergleiche Hoare [Hoa04].) Mit der Vision ist dies kompatibel: Zwar ist das eigentliche Analyseergebnis nicht beeinträchtigt, in der Anwendung wären aussagekräftige Traces allerdings nützlich.

10.4 Prioritätenbasiertes Scheduling und CSP_M-Fragestellung

Bisher wurde Kontrollfluss innerhalb eines Tasks modelliert. Von Interesse ist aber tatsächlich der taskübergreifende Kontrollfluss.

Es ist einfach, Tasks in CSP nebenläufig ablaufen zu lassen. So können zum Beispiel die jeweiligen Startprozeduren mittels des Interleaving-Operators »|||« parallelisiert werden. Das Ergebnis wäre ein System, in dem sich die Tasks zu jedem Zeitpunkt unterbrechen können, also in beliebigen Interleavings ablaufen.

Dieses Vorgehen ist hier ungeeignet. Zum einen gehört Interleaving zu den teuersten Operationen in CSP, und zum anderen stellt es eine deutliche Überapproximation des tatsächlichen Scheduling-Verhaltens dar. Tatsächlich

```
1  TASK_P1 = C123B234
2  TASK_P2 = C234B345
3  TASK_P3 = (C345B456 |~| C567B678)
4  TASK_P4 = C456B567
5
6  INT_P1 = SKIP
7  INT_P2 = ((INT_P1 |~| TASK_P1) ; INT_P2) |~| SKIP
8  INT_P3 = ((INT_P2 |~| TASK_P2) ; INT_P3) |~| SKIP
9  INT_P4 = ((INT_P3 |~| TASK_P3) ; INT_P4) |~| SKIP
```

Abbildung 10.15: Beispielhafte Definition von Preemption-Prozessen in einem System mit vier Tasks

können in den von uns untersuchten Systemen Tasks nur unterbrochen werden, um Tasks höherer Priorität zum Zuge kommen zu lassen. Es wird daher Code generiert, der dieses Schedulingverhalten explizit modelliert.

10.4.1 Preemption-Prozesse

Das Werkzeug Red normalisiert für die Erzeugung der Preemption-Prozesse die Task-Prioritäten. Nach der Normalisierung hat der höchstpriore Task die Priorität 1 und die niedrigeren Prioritäten stehen dicht (ohne Lücken) darunter. Diese Informationen stehen im ⟨task-info⟩-Block der Zwischensprache. In diesem Block sind auch die Startprozeduren der Tasks vermerkt. Es gibt keine zwei Tasks gleicher Priorität. Siehe dazu auch Abschnitt 4.2.1 (Seite 65).

Aus diesen Informationen wird eine Reihe von Prozessen generiert. Diese Generierung folgt dem in Abbildung 10.15 dargestellten Muster. Im Beispiel hat der Task mit der Priorität 3 zwei Einsprungspunkte. Grund dafür kann sein, dass hier zwei unterschiedliche Tasks der Priorität 3 vereint wurden.

Für jeden Task wird zunächst ein Prozess definiert, der auf die Startprozeduren verweist. Zusätzlich wird für jeden Task ein »Unterbrechungs-Task« generiert. Dieser fasst all das Verhalten zusammen, das während einer Unterbrechung des jeweiligen Tasks auftreten kann.

Kurz gefasst, ist dies das beliebig häufige, in beliebiger Reihenfolge erfol-

gende Ablaufen von Tasks höherer Priorität. Hat ein Task höherer Priorität einmal begonnen zu laufen, muss er auch zu Ende laufen, da er nicht unterbrochen werden kann, um einen Task niedrigerer Priorität zum Zuge kommen zu lassen. Der Task höchster Priorität kann niemals unterbrochen werden. Sein Unterbrechungs-Task ist einfach `SKIP`.

Damit diese Generierung effektiv ist, muss sie bei der Übersetzung des Kontrollflusses bereits berücksichtigt werden. Immer dann, wenn nach außen sichtbares Verhalten emittiert wird, folgt darauf der Kontrollflusstransfer an den passenden Unterbrechungs-Task. Dieser erlaubt dann die Unterbrechung an dieser Stelle. (Aus Gründen der Übersichtlichkeit wurde dies in den Darstellungen bisher stets ausgelassen.)

Anstatt »`set_a_to_2 -> `« auszugeben, wird daher tatsächlich ein Fragment wie dieses ausgegeben: »`(set_a_to_2 -> SKIP ; INT_T2) ;`« Nach dem Aussenden des Signals kann der Prozess von einigen – oder auch keinen – höherprioren Tasks unterbrochen werden.

10.4.2 Data-Race-Marks

Für Data-Race-Marks wird folgendes CSP_M-Fragment generiert, wenn diese für den aktuellen Durchlauf relevant sind:

```
((dr_φ -> INT_Tβ ; STOP) |~| SKIP) ;
```

Dabei ist ϕ entweder »`low`« oder »`high`«, je nachdem, ob es sich bei dem Zugriff um das erste oder das zweite Data-Race-Mark handelt. An Stelle des β steht die Nummer des aktuellen Tasks. Der Prozess `INT_T`β bezeichnet den Prozess, der die Unterbrechungsmöglichkeiten des Tasks β enthält.

Die Erreichbarkeit der Data-Race-Situation wird gezeigt, indem der Prozess, wenn es ihm möglich ist, das Signal `dr_low` aussendet. Anschließend wird er direkt unterbrochen. Der Kontrollfluss kann nicht mehr in diesem Task fortgesetzt werden, da auf den Unterbrechungsprozess der Prozess `STOP` folgt.[1] Vielmehr muss durch Ablaufen der höherprioren Tasks auch

[1] Dieses Verhalten modelliert nicht das Verhalten des realen Programms. Es dient hingegen dazu, das Erreichen der Data-Race-Situation erkennbar zu machen.

das Signal `dr_high` ausgesendet werden. Wenn das geschehen kann, darf die untersuchte Data-Race-Warnung nicht ausgeschlossen werden. Wenn es aber nicht geschehen kann, handelt es sich um eine falsch-positive Data-Race-Warnung. Sowohl das Signal `dr_low` als auch das Signal `dr_high` kann nur einmal ausgesendet werden. Es ist leicht nachzuvollziehen, dass eben dieses Verhalten durch das dargestellte Fragment erreicht wird.

10.4.3 Einsprungspunkt

Als letzte Frage bleibt noch die nach dem Einsprungspunkt in das Programm. Technisch ausgedrückt: Welcher der zahlreichen Prozesse, die bei der Generierung erzeugt werden, ist nun das Modell des Programms?

Als Einsprungspunkt wird der Beginn des Tasks gewählt, in dem der niederpriore Teil des Data-Races steht. Die Definition des Prozesses `PROG` wird dabei genutzt, um die Zustandsvariablenprozesse hinzu zu synchronisieren. Dies wird in Abbildung 10.13 bereits angedeutet.

Dieser Einsprungspunktprozess, bei dem alle Zustandsvariablen synchronisiert sind, sendet keine Variablensignale mehr aus, da diese alle verdeckt wurden. Das einzige verbleibende Verhalten geht dann von den Data-Race-Marks und den Umständen aus, unter denen diese erreichbar sind.

Mit der Wahl dieses Tasks wird konservativ überapproximiert. Zum Start des Tasks ist der Zustand der Zustandsvariablen im allgemeinsten aller möglichen Zustände. Würde, was vielleicht zunächst naheliegender erscheint, mit dem tatsächlichen Start des Programms (der in Bauhaus als »Above-Main« bezeichneten Initialisierungsprozedur) begonnen werden, könnte der Zustand nur spezifischer, nicht aber allgemeiner sein.

Es ist möglich, dass durch die Überapproximierung Data-Races nicht ausgeschlossen werden, die ohne sie ausgeschlossen würden. Solche Beispiele lassen sich leicht konstruieren.

In der Praxis dürfte so ein Verhalten aber selten sein. Dadurch, dass ein höherpriorer Task einen niederprioren jederzeit unterbrechen kann, kann bei seinem Start die Zustandsvariable jeden Wert haben, den sie während der Ausführung eines niedrigerprioren Tasks hat. Damit die Approximierung die

```
1  void high (void) {
2     if (s == C) {
3        dr = IND;
4        s = A;
5     }
6  }
7
8  void mid (void) {
9     if (s == B) {
10       dr = f();
11       s = C;
12    }
13 }
```

```
15 void low (void) {
16    if (IND)
17       s = A;
18
19    if (s == A)
20       s = B;
21    else
22       s = C;
23 }
24
25 int main (void) {
26    s = C;
27 }
```

Abbildung 10.16: Beispielsystem

Präzision verringert, muss mindestens ein Wert der Zustandsvariablen Tasks mit hoher Priorität vorbehalten sein. Diese müssen den Wert also setzen und, bevor sie zu Ende gelaufen sind, die Zustandsvariable wieder zurücksetzen. Solche Muster sind in praktischen Systemen nicht geläufig.

Die Approximierung hat Vorteile: Ein Teil des Modells beeinträchtigt die Performance nicht mehr, da es sich um unerreichbare Definitionen handelt.

10.4.3.1 Beispiel

Ein Beispielsystem, bei dem sich die Vorteile der Wahl des Einsprungspunktes zeigen, ist in Abbildung 10.16 dargestellt. Wird hier das Data-Race zwischen den Zugriffen aus den Zeilen 3 und 10 untersucht, ist der Einsprungspunkt des Modells der Aufruf an die Prozedur `mid`. Die gesamte Modellierung des auf der rechten Seite abgebildeten Quelltextes ist dadurch nicht zu erreichen. Ein Präzissionsverlust ergibt sich nicht.

10.4.4 Prüfbedingung

Um nun die Frage, ob im Modell des Programms eine Data-Race-Situation erreichbar ist, von FDR2 beantworten zu lassen, wird sie in Form einer

```
channel dr_low, dr_high

DR = dr_low -> dr_high -> STOP

NODR = dr_low -> dr_low -> RUN_DR |~| dr_high -> RUN_DR
RUN_DR = dr_low -> RUN_DR |~| dr_high -> RUN_DR
```

Abbildung 10.17: Definition von CSP_M-Prozessen zur Prüfung auf Data-Race-Freiheit

Prüfbedingung gestellt. Prüfbedingungen sind Anweisungen an Refinement-Checker, eine bestimmte Eigenschaft eines CSP_M-Prozesses zu prüfen. Sie stehen in CSP_M-Dateien. Sie haben, wenn sie sich auf das Traces-Modell beziehen, die Form »`assert P [T= Q`«.

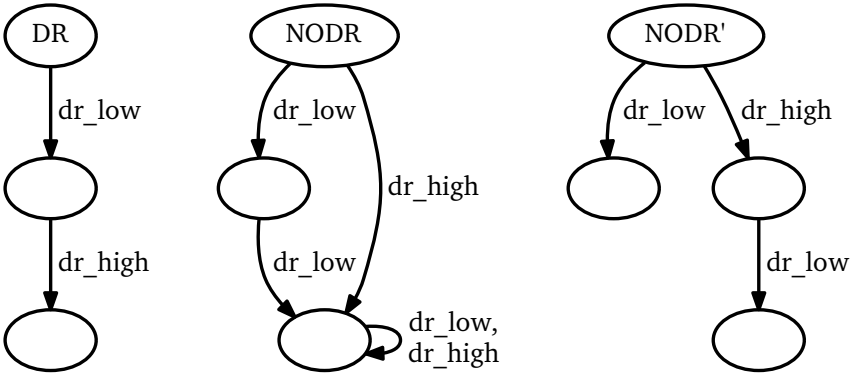
10.4.4.1 Traces-Verfeinerung

Da hier im Traces-Modell gearbeitet wird, ist die Traces-Verfeinerung (traces refinement) von Interesse. Analog zum FDR2 Reference Manual [FO10] wird hier die Traces-Verfeinerung wie folgt definiert:

Seien `I` und `S` CSP_M-Prozesse. Dann ist `S` Traces-feiner als (eine Traces-Verfeinerung von) `I`, geschrieben `I [T= S`, genau dann, wenn jedes Verhalten von `S` auch `I` möglich ist. Formal gilt demnach:

$$\texttt{I [T= S} \quad \text{g.d.w.} \quad \boldsymbol{Tr}(\texttt{S}) \subseteq \boldsymbol{Tr}(\texttt{I})$$

Eine einfache Möglichkeit, die Data-Race-Prüfbedingung zu formulieren, ist folgende: Ein Prozess `DR` wird definiert, der genau die beiden Data-Race-Signale in der richtigen Reihenfolge aussendet (siehe Abbildung 10.17). Mit `PROG` wird hier ein geeigneter Einsprungspunkt in das Modell des Systems bezeichnet. An dieser Stelle sei daran erinnert, dass alle zu Zustandsvariablen gehörenden Signale mittlerweile versteckt wurden, sodass das Alphabet von `PROG` nur noch die Data-Race-Signale enthält. Wenn nun gezeigt werden kann, dass der Prozess `DR` Traces-feiner als der Prozess `PROG` ist, also `PROG [T= DR` gilt, so ist gezeigt, dass der Prozess `PROG` den Trace



Produziert durch die Graphausgabe von FDR3. Beim Prozess `NODR` kam die Normalisierung zum Einsatz.

Abbildung 10.18: Generalisiertes kantenbeschriftetes Transitionssystem für die Prozesse `DR` (links) und `NODR` (mitte) aus der Abbildung 10.17 sowie für einen möglichen alternativen Prozess `NODR'`

$\langle \texttt{dr\_low} \cdot \texttt{dr\_high} \rangle$ enthält. Interessanter ist also der Fall, wo dies nicht der Fall ist. In diesem Fall ist bewiesen, dass `PROG` den Trace nicht enthält. Die Data-Race-Situation ist unerreichbar und somit ist die zugrundeliegende Data-Race-Warnung falsch-positiv.

Praktisch funktioniert diese Prüfbedingung aber nur bei sehr kleinen Programmen. FDR2 beginnt jede Prüfung mit der Normalisierung der linken Seite. Die Normalisierung ist eine Kompressionsstrategie, die teuer sein kann und nicht auf dem komplexen Programm, sondern auf einer einfachen Spezifikation ausgeführt werden sollte (siehe Reference Manual von FDR2 [FO10]). Wir wollen die Prüfbedingung also »invertieren«.

Diese Invertierung der Prüfbedingung gelingt durch Definition eines Prozesses, der auf dem Alphabet der Data-Race-Signale alles Verhalten erlaubt, das nicht mit einer Data-Race-Situation beginnt. Ein solcher Prozess ist der in Abbildung 10.17 definierte Prozess `NODR`. Ein derartiger Prozess lässt sich formal herleiten, indem Techniken angewendet werden, die aus dem

Bereich endlicher Automaten stammen.[1] Eine formale Herleitung wird hier nicht dargestellt. Abbildung 10.18 zeigt generalisierte kantenbeschriftete Transitionssysteme der Prozesse.

Die Prüfbedingung mit dem Prozess `NODR` lautet nun `NODR [T= PROG`. Dabei ist `NODR` leicht zu normalisieren. Eine affirmative Antwort von FDR2 bedeutet nun, dass das Data-Race ausgeschlossen ist. Mit dem Wissen, dass der Prozess `PROG` beide Signale sicher nur einmal ausführen kann, könnte auch der Prozess `NODR'` an Stelle des Prozesses `NODR` verwendet werden.

10.5 Steckbrief

Eine Übersicht über die Implementierung des Werkzeuges CSPC gibt der Steckbrief, der in Abbildung 10.19 dargestellt ist.

[1] Der Pfad kann sein: regulärer Ausdruck, nicht-deterministischer endlicher Automat, deterministischer endlicher Automat, Minimierung, Invertierung, regulärer Ausdruck, CSP_M-Prozess, Vereinfachung.

Red2CSP

Werkzeug, das CSP_M-Modelle generiert

Importierte Bibliotheken/Technologien

Lexer-Generator Ocamllex · Parser-Generator Ocamlyacc
String-Processing-Bibliothek str

Werkzeugimplementierung

Lexer	1 Datei 98 Zeilen
Parser	1 Datei 202 Zeilen
Werkzeug (OCaml)	14 Dateien 542 Zeilen
Build-Scripte	2 Dateien 99 Zeilen

Ressourcennutzung

Laufzeit reale Systeme	2 s – 5 s
Laufzeit »kleine Beispiele«	< 0,5 s

(ermittelt auf dem Testrechner, siehe Tabelle 16.3 (Seite 271))

Abbildung 10.19: Steckbrief des Werkzeuges Red2CSP

Kapitel 11

Werkzeug CSPC

Die vom Werkzeug Red2CSP erstellten CSP_M-Modelle sind teilweise zu groß, um sie sinnvoll mit Refinement-Checkern zu verarbeiten. Es wurde daher ein Werkzeug entwickelt, dass die CSP_M-Modelle Semantik-erhaltend komprimiert. Dabei hilft das Wissen um den üblichen Aufbau der generierten CSP_M-Modelle. Dieses Kapitel beschreibt dieses CSPC genannte Werkzeug und die Transformationen, die es durchführt. Außerdem werden getroffene Designentscheidungen erläutert.

11.1 Einsatzszenario und Anforderungen

Die Refinement-Checker FDR2 und FDR3 haben aufgrund ihrer Funktionsweise Schwierigkeiten mit verbosem generiertem Code. Beide Refinement-Checker wandeln in einem der ersten Verarbeitungsschritte die Prozessdefinitionen in generalisierte kantenbeschriftete Transitionssysteme um und achten dabei vor allem auf semantische Korrektheit und nicht auf minimale Größe des erzeugten Systems. Erst später, nachdem die Systeme aufgebaut wurden, werden sie komprimiert und damit möglicherweise verkleinert (vergleiche Gibson-Robinson et al. [GABR14]). Wenn schon der erste Schritt

aufgrund der Größe der entstehenden Systeme fehlschlägt, können die Kompressionen gar nicht erst eingesetzt werden. Dies motiviert das folgenden vorgestellte Einsatzszenario.

11.1.1 Einsatzszenario von CSPC

Das Werkzeug CSPC wird eingesetzt, um von Red2CSP erzeugte CSP_M-Modelle Semantik-erhaltend zu komprimieren. Da das Einsatzszenario beschränkt ist, muss das Werkzeug nicht mit allgemeinen CSP_M-Dateien umgehen können, sondern nur mit von Red2CSP erzeugbaren.

Die Anforderungen an das Werkzeug sind so gestellt, dass die Transformation zwar im Allgemeinen korrekt ist, und zwar unabhängig davon, ob die Eingabedatei von Red2CSP erzeugt wurde oder nicht, das Werkzeug aber die Möglichkeit hat, Dateien abzulehnen, die erkennbar nicht von Red2CSP stammen. Es werden hier daher stark beschränkende Anforderungen an die Ein- und Ausgabedateien des Werkzeuges formuliert.

Korrektheit wird in dieser Abhandlung wie folgt definiert: Eine Transformation einer CSP_M-Datei ist genau dann korrekt, wenn sie Semantik-erhaltend bezüglich der in der Datei enthaltenen Prüfbedingungen ist. Das bedeutet beispielsweise, dass eine Datei, die zuvor drei Prüfbedingungen enthielt, nach der Transformation auch wieder genau drei Prüfbedingungen enthält und diese (auch in derselben Reihenfolge) genauso wahr oder falsch sind wie zuvor. Das Werkzeug CSPC kann also insbesondere solche Definitionen entfernen, die weder direkt noch indirekt von den Prüfbedingungen verwendet werden.

11.1.2 Anforderungen an Ein- und Ausgabedateien

Folgende Anforderungen werden an Ein- und Ausgabedateien des Werkzeuges CSPC gestellt:

- Als Eingabedatei werden nur CSP_M-Dateien mit einem oder mehreren auf das Traces-Modell bezogenen Prüfbedingungen (Assertions)

akzeptiert. Eingabedateien, die keine Prüfbedingungen oder Prüfbedingungen mit Bezug zu anderen semantischen Modellen haben, dürfen zurückgewiesen werden.

- Die Eingabe ist auf das in Kapitel 7 (Seite 95) beschriebene CSP_M-Fragment beschränkt. Es müssen also insbesondere nur die dort beschriebenen Operatoren erkannt werden. Werden andere Sprachbestandteile verwendet, darf die Datei zurückgewiesen werden.
- Die Ausgabe wird durch die Prüfbedingungen beschränkt. Es müssen nur Prozessdefinitionen ausgegeben werden, die direkt oder indirekt durch die Prüfbedingungen verwendet werden. Es müssen nur die tatsächlich verwendeten Signale deklariert werden.
- Die Transformation ist im bereits beschriebenen Sinne korrekt. Andere Beschränkungen für die Transformation gibt es nicht. Insbesondere dürfen sowohl Prozess- als auch Signal-Bezeichner beliebig verändert werden, solange die ursprüngliche Semantik erhalten bleibt.

11.1.3 Anforderungen an Ressourcennutzung

Das Werkzeug CSPC arbeitet mit CSP_M-Modellen. Da diese bereits Data-Race-Warnungensspezifisch sind, wäre es der üblichen Denkweise folgend möglich, dass die Komprimierung nur bei einem bestimmten Anteil der Eingaben in akzeptabler Zeit zu Ende läuft. Dies soll hier aber ausdrücklich nicht die Richtlinie sein. Hingegen soll die »harte« Arbeit den Refinement-Checkern überlassen werden, die darauf ausgelegt sind. Der Komprimierungsschritt soll einfache Transformationen ausführen, die den Einsatz der Refinement-Checker gerade ermöglichen.

Eine strikte Laufzeitkomplexität wird nicht vorgegeben. Die Vorgabe an die Laufzeit soll vielmehr sein, dass sie kurz genug für den interaktiven Einsatz ist. Das bedeutet, dass die Transformation in der Regel nach weniger als 5 Sekunden abgeschlossen ist.

11.2 Arbeitsweise des Werkzeuges und Designentscheidungen

Die Gesamttransformation, die CSPC durchführt, ist aus mehreren Teiltransformationen zusammengesetzt. Diese werden wiederholt ausgeführt, um ein möglichst gutes Ergebnis zu erhalten. Die einzelnen Teiltransformationen werden in den folgenden Abschnitten beschrieben.

11.2.1 Auswahl und Entwicklung der Transformationen

Noch bevor die Arbeit an CSPC begann, wurde im Rahmen der Arbeit zu dieser Abhandlung ein Werkzeug mit dem Namen Obf entwickelt. Dieses hat in ähnlicher Art und Weise wie das Werkzeug CSPC CSP_M-Dateien transformiert und komprimiert. Alle jetzt in CSPC enthaltenen Teiltransformationen waren bereits im Werkzeug Obf enthalten.

Das Werkzeug, geschrieben in der Sprache OCaml, diente der Evaluation der Transformationen. Die Architektur des Werkzeuges war zwar technisch ineffizient, erlaubte es aber, Ansätze zur Komprimierung der Modelle schnell zu implementieren und zu vergleichen. Die Transformationen, die nach Abwägung von Laufzeit, Komplexität und Komprimierungserfolg am geeignetsten erschienen, wurden dann in einem eigenen Werkzeug (CSPC) implementiert.

Die für CSPC ausgewählten Teiltransformationen haben alle eine Laufzeitkomplexität in $\widetilde{\boldsymbol{\mathcal{O}}}(\boldsymbol{n})$. Die Gesamttransformation führt diese Teiltransformationen aus. Wenn diese »Fortschritt erzielen«, setzen sie ein globales Flag. Ist dieses Flag nach Ablauf aller Teiltransformationen gesetzt, wird es zurückgesetzt und alle Teiltransformationen werden erneut ausgeführt. Als obere Grenze für die Zahl der Ausführungswiederholungen wurde 50 ausgewählt. Normalerweise sind aber beträchtlich weniger Durchläufe nötig. Durch diese obere Grenze ist die Laufzeitkomplexität der Gesamttransformation in der Theorie ebenfalls in $\widetilde{\boldsymbol{\mathcal{O}}}(\boldsymbol{n})$.

11.2.1.1 The Road Not Taken

Zu den Teiltransformationen, die nicht von Obf übernommen wurden, gehören diese beiden:

- Eine Subexpression-Elimination-Transformation. Diese hat, in gegensätzlicher Richtung zum Inlining arbeitend, gemeinsame Teilausdrücke in eigene Prozesse ausgelagert. Dafür wurde ein probabilistischer Algorithmus verwendet. Die Transformation hat zwar zu einer weiteren Komprimierung der Datei beigetragen, hatte jedoch eine Komplexität schlechter als $\mathcal{O}(n^2)$ und hat erheblich Zeit beansprucht.
- Eine komplexere Erkennung toter Signale. Diese Transformation hat mit einem quadratischen Algorithmus nach toten Signalen gesucht. Die Laufzeit der Transformation war hoch und der praktische Erfolg gering.

11.2.2 Technischer Aufbau

CSPC ist in objektorientiertem Java geschrieben. Das Frontend verwendet den Parser-Generator Antlr. Als Zwischendarstellung wird ein moderat semantisch attribuierter abstrakter Syntaxbaum (AST) verwendet. Dessen Aufbau ergibt sich abgesehen von einer Ausnahme, die in Abschnitt 11.4.2 geschildert wird, direkt aus der Grammatik von CSP_M.

Die eigentliche Arbeit wird, wie bereits dargestellt, durch die Teiltransformationen erledigt. Das Backend, welches im Wesentlichen ein Pretty-Printer ist, kann die Zwischendarstellung, die ja ein Abstrakter Syntaxbaum ist, ohne Schwierigkeiten ausgeben.

11.3 Transformation der Bezeichner von Prozessen und Signalen

Die Bezeichner von Prozessen und optional auch Signalen werden durch die in diesem Abschnitt beschriebene einfache Transformation konsistent umbenannt.

Von der Transformation der Bezeichner von Prozessen und Signalen ist keine deutliche Performanzverbesserung zu erwarten.[1] Die Refinement-Checker dürften die Bezeichner zu Tokens reduzieren und als einen der ersten Schritte eine Namensauflösung durchführen. Wenn so vorgegangen wird, sollte die Laufzeit der Verarbeitung der Modelle kaum von der Länge der Bezeichner abhängen.

Da es zur Idee von CSPC gehört, dass die ausgebenden Dateien von Hand geschriebenen möglichst ähnlich sind, ist es auch sinnvoll, die langen Bezeichner zu kürzen. Die ursprünglichen Bezeichner verlieren ohnehin weitgehend ihren einstmaligen Sinn durch die Transformationen. Die Bezeichner werden auch nicht für Rückmeldungen an den Benutzer benötigt: Für diesen ist nur das Ergebnis des Refinement-Checks relevant.

11.3.1 Funktionsweise

Die ursprünglichen Bezeichner von Prozessen werden nur zur Namensbindung herangezogen. Die im Folgenden dargestellten Transformationen führen teils neue Prozessdefinitionen ein. Diese bekommen zunächst keinen Bezeichner zugewiesen. Bei der Ausgabe werden neue Bezeichner für die Prozesse erstellt und diese somit konsistent umbenannt. Die neuen Prozessbezeichner bestehen aus dem Buchstaben »P«, gefolgt von einer natürlichen Zahl. Um einen neuen Bezeichner zu erhalten, wird die zuletzt vergebene Zahl inkrementiert. Die neuen Bezeichner sind oft kürzer als die ursprünglichen.

Optional können auch die Signale nach einem ähnlichen Schema umbenannt werden. Die Signale haben dann Bezeichner, die aus »a« und einer natürlichen Zahl bestehen. Zu Debug-Zwecken ist es allerdings sinnvoll, die ursprünglichen Bezeichner der Signale zu erhalten.

In der Ausgabe werden auf jeden Fall alle Deklarationen von Signalen zusammengefasst und an den Anfang der Datei geschrieben. Dabei werden

[1] Dies wurde auch praktisch in einem kleinen Experiment bestätigt. Es wurden bis auf die Bezeichner identische CSP_M-Scripte erzeugt. Die Dateien mit langen (200 Zeichen) Bezeichnern benötigten bei keinem der verwendeten Refinement-Checker (FDR2, FDR3, FDR4, ProB) deutlich länger als die Dateien mit kurzen (5 Zeichen) langen Bezeichnern.

```
channel a, b, c

UNREF = c -> UNREF
Q = a -> b -> Q
P_LONGNAME = a -> Q

channel x, y
C12345B12345 = x -> y -> SKIP
R = a -> C12345B12345

assert R [T= P_LONGNAME
```

(davor)

```
channel y, a, b, x

P1000 = (a -> P1001)
P1001 = (a -> (b -> P1001))
P1002 = (a -> (x -> (y -> SKIP)))

assert P1002 [T= P1000
```

(danach)

Abbildung 11.1: Die Änderung von Prozess- und Signalbezeichnern durch CSPC

nur die Signale deklariert, die (rein syntaktisch) in den Prozessen auch vorkommen.

Da die folgenden Transformationen Definitionen zum Teil als tot erkennen und entfernen, kann es gelegentlich geschehen, dass bestimmte Signale schon rein syntaktisch nicht mehr in den noch vorhandenen Definitionen vorkommen. Diese müssen dann auch nicht mehr deklariert werden.

11.3.2 Beispiel

Abbildung 11.1 stellt die Transformation an einem Beispiel dar. Die Umbenennung der Signale ist hier ausgeschaltet. Zu erkennen ist, dass die Ausgabe vollständig geklammert ist und dass nicht referenzierte Definitionen – wie

hier die des Prozesses `UNREF` – nicht ausgegeben werden. (Außerdem werden Auswirkungen von anderen Transformationen deutlich – zum Beispiel wird der Prozess `C12345B12345` in den Prozess `R` inline-expandiert.)

11.3.3 Korrektheit und Komplexität

Die Transformation der Bezeichner von Prozessen und Signalen hat eine Komplexität in $\widetilde{\boldsymbol{\theta}}(\boldsymbol{n})$. Die Transformation wird nur einmal ausgeführt. Die Größe des abstrakten Syntaxbaumes bleibt durch die Transformation gleich oder verringert sich, wenn Signale nicht mehr deklariert werden müssen.

Sind die Bezeichner von Prozessen und Signalen in der Eingabe allerdings sehr kurz, kann die Ausgabe durch die Transformation textuell länger werden. Die von Red2CSP üblicherweise verwendeten Bezeichner sind aber meistens länger als die von CSPC generierten, sodass dies in der Praxis kein Problem darstellt. Der Effekt ist auch in Abbildung 11.1 zu erkennen.

Die Transformation ist auch erkennbar korrekt: Wie in den meisten anderen formalen Sprachen auch ändert in CSP_M eine konsequente Umbenennung von Bezeichnern die Semantik nicht.

11.4 Transformationen für Teilausdrücke

Die hier dargestellten Transformationen für Teilausdrücke vereinfachen und verkleinern Teilausdrücke in Prozessdefinitionen. Dabei werden passende Teilausdrücke durch kleinere, aber semantisch identische Teilausdrücke ersetzt. Zusätzlich wird der Auswahloperator `|~|` im abstrakten Syntaxbaum auf eine spezielle Art und Weise repräsentiert, die auch zu Optimierungen führen kann.

11.4.1 Funktionsweise der Teilausdruckersetzung

Die Transformation von Teilausdrücken traversiert alle von einer Prüfbedingung erreichbaren Definitionen und kommt so an allen relevanten Prozessausdrücken vorbei. Innerhalb der Definitionen werden die Teilausdrücke in

Tabelle 11.1: Teilausdruckvereinfachungsregeln und die zugrundeliegenden Gesetzmäßigkeiten

Nr.	Ausdruck	Vereinfachung	Gesetze
1	`P \ {}`	`P`	Concealment.L1
2	`STOP \ Σ`	`STOP`	Concealment.L4
3	`SKIP \ Σ`	`SKIP`	Concealment.L5
4	`SKIP ; P`	`P`	Seq.L1
5	`P ; SKIP`	`P`	Seq.L1
6	`STOP ; P`	`STOP`	Seq.L5
7	`(a -> STOP |~| SKIP) ; P`	`(a -> STOP |~| P)`	Seq.L2A, Seq.L1, Seq.L5

In den Regeln steht `P` für einen beliebigen Prozessausdruck und `a` für ein beliebiges Signal.

Alle angegebenen Vereinfachungen sind im relationalen Modell gültig und daher nicht auf das Traces-Modell beschränkt. Ihre Gültigkeit lässt sich direkt mit den angegebenen Gesetzen beweisen. Die Gesetze sind Hoare [Hoa04] entnommen.

Post-Order-Reihenfolge bearbeitet. Jeder Ausdruck wird auf Übereinstimmung mit den in Tabelle 11.1 aufgezählten Mustern verglichen. Stimmt er überein, wird der abstrakte Syntaxbaum entsprechend der Vereinfachungsregel angepasst.

Die Korrektheit der Ersetzungsregel mit der Nummer 7 in Tabelle 11.1 lässt sich folgendermaßen beweisen:

```
  (a -> STOP |~| SKIP) ; P
= a -> STOP ; P |~| SKIP ; P          (Seq.L2A)
= a -> STOP |~| SKIP ; P               (Seq.L5)
= a -> STOP |~| P                      (Seq.L1)
```

Die Beweise der anderen Regeln sind noch einfacher. Im Wesentlichen entsprechen die Transformationen den in der Tabelle angegebenen Gesetzen.

11.4.2 Funktionsweise der Mengenmodellierung des Auswahloperators

Zusätzlich zur Ersetzung von Teilausdrücken werden Ausdrücke umgestellt, die den Auswahloperator `|~|` enthalten. In der Zwischendarstellung wird statt dem eigentlich binären Auswahloperator ein Auswahloperator verwendet, der auf einer Menge von Ausdrücken arbeitet. Dies ist zulässig, da der Auswahloperator assoziativ (NDChoice.L3)[1], kommutativ (NDChoice.L2) und idempotent (NDChoice.L1) ist. Verschachtelte Auswahloperatoren werden zusammengefasst. Dabei wird der Prozess `STOP` als leerer Auswahloperator betrachtet (NDChoice.L4). Tabelle 11.2 stellt die Auswirkungen dieser Modellierung dar.

Nummer 1 ist der einfachste Fall. Aus der einfachen binären Anwendung wird eine zweielementige Menge. Nummer 2 zeigt, wie verschachtelte Anwendungen des binären Operators zusammengefasst werden.

Ist die ursprüngliche Anwendung redundant, führt die Mengenmodellierung des Auswahloperators zu Optimierungen. Dieses Verhalten zeigt sich in den Nummern 3 – 7. Ein Auswahloperator mit einer einelementigen Menge

[1] Wie überall in dieser Abhandlung stammen die Gesetze aus Hoare [Hoa04].

Tabelle 11.2: Darstellung des Auswahloperators im abstrakten Syntaxbaum

Nr.	Konkrete Syntax	AST-Darstellung
1	`(P |~| Q)`	⊓ {P; Q}
2	`(P |~| (Q |~| R))`	⊓ {P; Q; R}
3	`(P |~| (STOP |~| R))`	⊓ {P; R}
4	`(P |~| (Q |~| P))`	⊓ {P; Q}
5	`(P |~| STOP)`	P
6	`(P |~| P)`	P
7	`(STOP |~| STOP)`	STOP
8	`(P |~| Q) |~| (R |~| S))`	⊓ {P; Q; R; S}

In der Darstellung stehen `P`, `Q` und `R` jeweils für einen beliebigen Prozessausdruck.

wird durch das eine Element dieser Menge ersetzt (Nummern 5 und 6). Ein Auswahloperator mit einer leeren Menge, der in Verbindung mit anderen Operationen entstehen kann, wird durch `STOP` ersetzt (Nummer 7). Diese Darstellung führt also unter Umständen auch zu einer Verkleinerung der Ausgabe. Bei der Umstellung werden verschachtelte Operatoren beliebiger Tiefe zusammengefasst.

11.4.3 Korrektheit und Komplexität

Eine wichtige Eigenschaft der Transformationen für Teilausdrücke ist, dass durch ihre Anwendung der abstrakte Syntaxbaum strikt kleiner wird. Es ist somit ausgeschlossen, dass die Transformationen bei mehrfacher Ausführung in eine Endlosschleife geraten. Der abstrakte Syntaxbaum wird durch Anwendung einer Reihe solcher Einzeltransformationen nicht wachsen. Auch die Zusammenfassung der Auswahloperatoren kann nur zu einer Verkleinerung des abstrakten Syntaxbaumes führen. Die Komplexität eines einzelnen Durchlaufs liegt in $\widetilde{\boldsymbol{\sigma}}(\boldsymbol{n})$. Die Transformationen sind auch korrekt. Dies wurde in den vorangehenden beiden Abschnitten bereits begründet. Beide

Transformationen sind unabhängig von dem verwendeten Modell korrekt, da sie sich allein aus den Gesetzen von Hoare herleiten lassen.

11.5 Transformation der Prüfbedingungen

Auch die Prüfbedingungen haben Freiräume zur Umstellung. Die Transformation der Prüfbedingungen nutzt diese, um die CSP_M-Modelle zu optimieren und technisch leichter verarbeitbar zu machen.

11.5.1 Funktionsweise

Aus praktischen Erwägungen heraus werden die Prüfbedingungen bei dieser Transformation so umgestellt, dass sowohl linke als auch rechte Seite aus einem Prozessbezeichner bestehen. Stehen dort in der Originaldatei komplexe Ausdrücke, werden diese als eigene neue Prozesse definiert und dann werden die Prüfbedingungen so umgestellt, dass sie auf diese neuen Prozesse verweisen.

Verweist der rechte Prozess einer Prüfbedingung auf den `STOP`-Prozess, ist die Antwort des Refinement-Checkers immer »Ja«, da im Traces-Modell `STOP` der verfeinertste aller Prozesse ist. Die Prüfbedingung wird also durch die triviale, immer erfüllte Prüfbedingung »`STOP [T= STOP`« ersetzt.[1]

11.5.2 Beispiel

Beide Aktionen dieser Transformation kommen in dem in Abbildung 11.2 dargestellten Beispiel vor.

11.5.2.1 Korrektheit und Komplexität

Die Vereinfachung ist aufgrund der Definition der Traces-Verfeinerung und der Tatsache, dass CSP_M deklarativ ist, korrekt.

Die Transformation lässt sich mit einer Laufzeit in $\mathcal{O}(1)$ implementieren.

[1] Die Prüfbedingung wird nicht ganz entfernt, da die Refinement-Checker beim Fehlen von Prüfbedingungen auch keine Antwort liefern. Eine solche wird aber stets erwartet.

```
channel a

COMPLEX = [...]

assert a -> COMPLEX [T= STOP
```

Die ursprüngliche Datei. In der Fragestellung stehen komplexe Ausdrücke.

```
channel a

COMPLEX = [...]
P_TMP1 = a -> COMPLEX
P_TMP2 = STOP

assert P_TMP1 [T= P_TMP2
```

Für die Ausdrücke der Fragestellungen wurden die Prozesse `P_TMP1` und `P_TMP2` ausgelagert.

```
assert STOP [T= STOP
```

Die Fragestellung wurde als trivial erkannt und entsprechend ersetzt. Alle Prozessdefinitionen und Signale werden nun als tot erkannt.

Abbildung 11.2: Die Transformation der Prüfbedingungen

11.6 Prozessdefinitionsübergreifende Transformationen

Es werden einige prozessdefinitionsübergreifende Transformationen durchgeführt. Dazu gehören die Vereinfachung von Kopieketten, die Vereinfachung von speziellen Auswahldefinitionen und eine Inlining-Transformation.

11.6.1 Vereinfachung von Kopieketten

Es kommt bei der Übersetzung nach CSP_M oft vor, dass ein Prozess als Kopie eines anderen definiert wird. Als Kopie wird eine Prozessdefinition definiert, die die Form »`P = Q`« hat, wobei `P` und `Q` beliebige ungleiche Prozessbezeichner sind. Vergleiche dazu auch Abbildung 9.8 (Seite 151). Aus allen dort abgebildeten leeren Basic-Blocks werden von der Red2CSP Kopien erzeugt.

Aus Kopien können auch Ketten entstehen, wenn zum Beispiel auf die Definition »P = Q« die Definition »Q = R« folgt. Solche Kopien werden durch eine Transformation reduziert, die an eine Copy-Propagation erinnert.

Der Algorithmus, der dazu verwendet wird, ist eine Abwandlung des Union-Find-Algorithmus (vergleiche Tarjan.1975). Der Algorithmus traversiert die von Prüfbedingungen erreichbaren Prozessdefinitionen. Stößt er dabei auf eine Kopie der Form P = Q, werden die Prozesse P und Q als Kopien voneinander vereint.

Dabei wird eine Art Hierarchie eingehalten. Der Stellvertreter der Teilmenge soll in diesem Fall der Prozess Q sein, da dieser eventuell Funktionalität implementiert. Es muss sichergestellt sein, dass der Stellvertreter einer Menge von Kopien das »Original« ist, wenn dieses existiert.

Nach der Traversierung werden erneut die von Prüfbedingungen erreichbaren Prozessdefinitionen traversiert. Dabei wird jeder Prozessverweis auf der rechten Seite einer Prozessdefinition durch den jeweiligen Stellvertreter ersetzt. Die Kopie-Definitionen sind nun nicht mehr erreichbar und werden daher von zukünftigen Transformationen nicht mehr beachtet.

11.6.1.1 Komplikation: Zyklen

Eine nicht unwesentliche Komplikation sind Zyklen, die in den Kopieketten auftreten können. Der einfachstmögliche Zyklus hat die Form »P = P« – aber welche Semantik hat ein solcher Zyklus?

Der Prozess wird als »Kopie« seiner selbst definiert. In CSP muss dies als µ-Quantifizierung aufgefasst werden: »P = μX.X« Der Prozess P ist also der kleinste Prozess, der die Gleichung »P = P« erfüllt. Da jeder Prozess diese triviale Gleichung erfüllt, soll P der kleinste Prozess überhaupt sein. Dies ist (in allen in dieser Abhandlung besprochenen Modellen) der Prozess STOP. Es hätte somit auch gleich »P = STOP« definiert werden können.

Nicht unerwähnt bleiben soll, dass in Hoare [Hoa04] diese leere µ-Quantifizierung für CSP_M explizit verboten wird. Gründe dafür werden nicht genannt, können aber erschlossen werden.

In von Hand geschriebenem Code dürfte eine solche leere µ-Quantifizierung

fast immer ein Fehler sein. Wenn ein Nutzer einen Prozess als STOP definieren möchte, wird er dies vernünftigerweise in expliziter Weise tun und seinen Definitionswunsch nicht durch eine leere µ-Quantifizierung verschleiern.

Obwohl sie so einfach ist, ist die dieser µ-Quantifizierung zugrundeliegende Funktion – die Identität – nicht konstruktiv. (Siehe zum Begriff der Konstruktivität Abschnitt 10.1.5 (Seite 165).) Die Refinement-Checker FDR2 und FDR3 können aber nur mit auf konstruktiven Funktionen basierenden µ-Quantifizierungen zuverlässig umgehen. Auch deshalb ist es sinnvoll, auf solche Konstrukte zu verzichten.

Wird CSP_M-Code allerdings automatisiert generiert, kann eine solche leere µ-Quantifizierung in Randfällen[1] leicht auftreten. Sie bei der Generierung zu vermeiden, ist aufwändig. Ansatz ist es daher hier, sie zunächst zu erlauben und durch CSPC zu entfernen.

Das Entfernen der leeren µ-Quantifizierung wird leicht von der hier dargestellten Kopieentfernung mit erledigt. Stößt diese auf einen Zyklus in den Kopiedefinitionen, wählt sie als Stellvertreter der Gruppe einfach den STOP-Prozess.

11.6.1.2 Beispiel

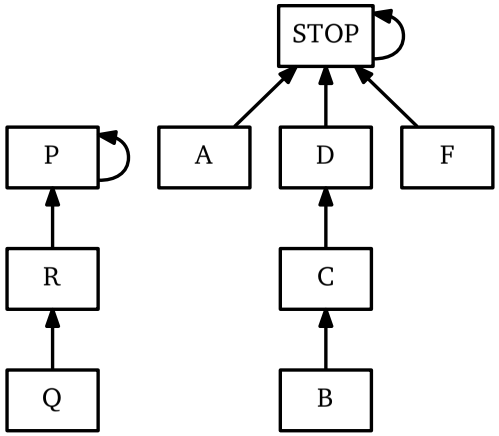
Im Folgenden wird ein Beispiel für die Vereinfachung von Kopieketten analysiert. Abbildung 11.3 zeigt links ein Beispiel-Script mit zahlreichen Kopiedefinitionen. Dabei sind zwei getrennte Strukturen erkennbar.

Bei der ersten Struktur, die der typischen Verwendung der Transformation entspricht, geht es um den Prozess P. In der Mitte ist die Union-Find-Datenstruktur zu sehen, wie sie bei der ersten Traversierung entsteht. Im Wesentlichen werden die Kopiedefinitionen zu Kanten in dem gerichteten Graphen. Selbstkanten markieren, wie bei Union-Find-Datenstrukturen üblich, den Vertreter der Teilmenge. Erkennbar ist, dass die Prozesse Q und R Kopien des Prozesses P sind. Dementsprechend werden alle Vorkommen dieser Prozesse in der zweiten Traversierung durch den Prozess P ersetzt.

[1] Ein solcher ist zum Beispiel eine leere Endlosschleife: `while (1) {};`.

```
1  channel a
2
3  P = a -> R
4
5  A = A
6
7  B = C
8  C = D
9  D = B
10
11 F = B
12
13 Q = R
14 R = P
15
16 assert F [T= R
```

davor



Union-Find-Datenstruktur
(ohne Pfadkomprimierung)

```
1  channel a
2
3  P = a -> P
4
5  A = STOP
6
7  B = STOP
8  C = STOP
9  D = STOP
10
11 F = STOP
12
13 Q = P
14 R = P
15
16 assert STOP [T= P
```

danach

Abbildung 11.3: Die Anwendung der Vereinfachung von Kopieketten

Die zweite Struktur enthält die wenig typischen zyklischen Kopien. Einer der Zyklen entsteht, indem der Prozess `A` als Kopie seiner selbst definiert wird. Ein zweiter Zyklus besteht aus den Prozessen `B`, `C` und `D`. Wird ein Zyklus erkannt, wird der Prozess `STOP` zum Vertreter der beteiligten Gruppe. Dies zeigt sich darin, dass alle verbundenen Prozesse nun auf den Prozess `STOP` verweisen.

Rechts in der Abbildung ist das Ergebnis der Transformation zu sehen. Die Vereinfachung zeigt sich hier besonders prägnant darin, dass alle Definitionen, außer der des Prozesses `P`, nun nicht mehr referenziert werden. Sie müssen fortan nicht mehr berücksichtigt werden.

11.6.1.3 Korrektheit und Komplexität

Da CSP_M deklarativ ist, ist die Transformation korrekt. Der Umgang mit Zyklen in den Kopieketten erweitert in angemessener Art und Weise den Definitionsbereich von CSP_M. Die Komplexität der Transformation liegt in $\tilde{\mathcal{O}}(n)$.

11.6.2 Vereinfachung spezieller Auswahldefinitionen

Eine recht spezielle und auch nur für das Traces-Modell gültige Transformation besteht aus der Vereinfachung bestimmter Auswahldefinitionen. Dabei sind Definitionen von Interesse, welche die folgende Form haben:

$$\texttt{P} = \sqcap \Phi, \text{ wobei } \texttt{P} \in \Phi$$

Darin lässt sich dann der gerade definierte Prozess aus der Auswahl entfernen. So wird aus

```
P = A |~| P
```

dann

```
P = A
```

Derartige Prozessdefinitionen werden zum Beispiel aus dem folgenden Code-Segment generiert: `void f() {while (IND) {}; a = 1;}`.

11.6.2.1 Korrektheit und Komplexität

Im Allgemeinen ist diese Transformation zur Vereinfachung spezieller Auswahldefinitionen nicht Semantik-erhaltend. Da von dem Werkzeug CSPC aber nur Dateien mit Traces-Prüfbedingungen bearbeitet werden, reicht es aus, wenn die Transformation in diesem Modell gültig ist. Die Korrektheit im Traces-Modell lässt sich wie im Folgenden dargestellt zeigen:

Sei der Prozess `P` definiert durch

$$\texttt{P} = \texttt{Q} \ |\sim| \ \texttt{P}$$

für einen beliebigen Prozess `Q`. Dann gilt[1]:

$$\begin{aligned} \boldsymbol{Tr}(\texttt{P}) &= \boldsymbol{Tr}\left(\mu \texttt{X} . (\texttt{Q} \ |\sim| \ \texttt{X})\right) \\ &= \mu \boldsymbol{S} . (\boldsymbol{Tr}(\texttt{Q}) \cup \boldsymbol{S}) \\ &= \boldsymbol{Tr}(\texttt{Q}) \end{aligned}$$

Es ergeben sich also für `P` genau die Traces, die sich auch ergeben würden, wenn der Prozess `P` direkt mit »`P = Q`« definiert worden wäre.

11.6.2.2 Beispiele

Die Beispiele für die Transformation spezieller Auswahldefinitionen sind schnell erläutert. Tabelle 11.3 zeigt für drei Beispiele jeweils die konkrete Syntax und die dazugehörige Zwischendarstellung vor und nach der Transformation.

Die Nummern 1 und 2 zeigen dabei die Standardfälle. Bei Nummer 1 fällt die Auswahl ganz weg; bei Nummer 2 wird sie nur kleiner.

Nummer 3 wird in der realen Ausführung nicht von der Transformation spezieller Auswahldefinitionen behandelt, sondern als Kopiekette entfernt. Es ist zu erkennen, dass beide Regeln zum gleichen Ergebnis führen.

[1] Die verwendeten Definitionen für Traces lassen sich in Hoare [Hoa04] nachlesen.

Tabelle 11.3: Drei Beispiele für die Vereinfachung spezieller Auswahl-Definitionen

Nr.	Konkreter Syntax	AST-Darstellung	Transformations-ergebnis
1	`P = P |~| Q`	`P = ⊓ {P; Q}`	`P = Q`
2	`P = Q |~| P |~| R`	`P = ⊓ {P; Q; R}`	`P = ⊓ {Q; R}`
3	`P = P |~| P`	`P = P`	`P = STOP`

In der Darstellung stehen `Q` und `R` jeweils für einen beliebigen Prozessausdruck.

11.6.3 Inlining

Da die Sprache CSP_M deklarativ ist, ist es stets erlaubt, einen Verweis auf einen Prozess durch die rechte Seite der Definition dieses Prozesses zu ersetzen. Im besten Fall wird dadurch die ursprüngliche Definition nicht mehr referenziert und kann daher entfernt werden. Im schlechtesten Fall entsteht dadurch nur zusätzliche Komplexität und keine der Definitionen kann wegfallen. Es gilt bei dieser Transformation also gut auszuwählen, an welchen Stellen eine solche Expansion (Inlining) vorzunehmen ist.

11.6.3.1 Regeln

Für die Entscheidung, ob eine Expansion (»Inlining«) durchgeführt wird, gibt es mehrere Regeln:

1. Ein Prozess wird niemals in seine eigene (rekursive) Definition hinein expandiert, da durch dieses Vorgehen nichts gewonnen werden kann, sondern die Definition nur aufgebläht wird.

   Einen Verstoß gegen diese Regel ist in Zeile 1 der Tabelle 11.4 zu sehen. Dabei wird deutlich, wie der neue Prozess länger wird, ohne dass neue Chancen zur Optimierung entstehen.

2. Ein Prozess, der selbst nur eine Kopie eines vordefinierten Prozesses ist, also `SKIP` oder `STOP`, wird immer expandiert.

Tabelle 11.4: Die Effekte der Inlining-Transformation unter unterschiedlichen Bedingungen

1	`1 P = a -> P`	P	`1 P = a -> a -> P`	✗
2	`1 P = STOP` `2 Q = P |~| R`	P	`1 Q = R`	✓
3	`1 P = SKIP` `2 Q = P ; a -> P`	P	`1 Q = a -> SKIP`	✓
4	`1 P = b -> STOP` `2 Q = a -> P`	P	`1 Q = a -> b -> STOP`	✓

5	`1 P = b -> Q`
	`2 Q = a -> P`
	`3 R = Q`
		Q	`1 P = b -> a -> P`
			`2 R = a -> P`
				✗

6	`1 P = b -> Q`
	`2 Q = a -> P`
	`3 R = Q`
		P	`1 Q = a -> b -> Q`
			`2 R = Q`
				✓

7	`1 P = A |~| SKIP`
	`2 Q = P |~| SKIP`
	`3 R = B |~| P`
	`4 S = C |~| P`
		(P)	`1 P = A |~| SKIP`
			`2 Q = A |~| SKIP`
			`3 R = B |~| P`
			`4 S = C |~| P`
				✓

In den Spalten von links nach rechts: Die Nummer des Beispiels, der ursprüngliche Code-Ausschnitt, der zu expandierende Prozess, das Ergebnis der Transformation und die Bewertung des Erfolges der Transformation.

Die Zeilen 2 und 3 in Tabelle 11.4 zeigen, was hier im besten Fall geschieht: Durch das Expandieren wird zwar die Zwischendarstellung zunächst kaum kleiner, es ergeben sich aber neue Optimierungsmöglichkeiten für andere Transformationen.

3. Ein Prozess, der nur einmal referenziert wird, wird expandiert, solange dies nicht gegen Punkt 1 verstößt.

   Wird der Prozess mehrfach referenziert, würde durch seine Expansion

der abstrakte Syntaxbaum möglicherweise beträchtlich wachsen. In Zeile 5 aus Tabelle 11.4 sind die Konsequenzen zu sehen. In den Zeilen 4 und 6 wird der expandierte Prozess nur einmal referenziert und das Ergebnis ist zufriedenstellend.

4. Eine spezielle Regel gilt für Prozesse, die als Auswahl zwischen einem Prozess und `SKIP` definiert sind, deren Definition also die Form `P = Q |~| SKIP` hat. Werden diese Prozesse nicht ohnehin schon wegen Punkt 3 expandiert, werden sie an allen Stellen expandiert, wo sie in Auswahloperatoren stehen, in denen ohnehin bereits ein `SKIP` enthalten ist. Dadurch wächst der abstrakte Syntaxbaum nicht, da das zusätzliche `SKIP` später wegoptimiert wird.

   Ein Beispiel dafür steht in Zeile 7 der Tabelle 11.4. Hier wird der Verweis auf `P` nur in der Definition von `Q` ersetzt.

11.6.3.2 Implementierung

Die Transformation wird ausgeführt, indem der erreichbare Teil der Prozessdefinitionen mehrfach durchlaufen wird. Dabei werden die Regeln genutzt, um zu entscheiden, welche Prozesse expandiert werden.

Im ersten Durchlauf werden die Verwendungen der Prozesse gezählt. Es entsteht eine Menge der nur einmal verwendeten Definitionen. In zwei weiteren Durchläufen werden die Prozesse nach den Regeln 2 und 4 expandiert. (Dies könnte auch in einem Durchlauf erfolgen.) Im dritten Durchlauf werden dann die Expansionen nach Regel 3 durchgeführt.

Im letzten Durchlauf gibt es das Problem, dass die Prozessdefinitionen während ihrer Traversierung verändert werden. In bestimmten Konstellationen kann es dabei passieren, dass relevante Teile des abstrakte Syntaxbaumes nicht traversiert werden und das Inlining somit unvollständig ist. Im praktischen Einsatz hat dies kaum Auswirkungen und wird ignoriert. Ohnehin passiert es häufig, dass sich durch Inline-Expansionen weitere Optimierungsmöglichkeiten ergeben, welche dann wiederum neues Inlining ermöglichen. Es ist nicht erforderlich, die Inline-expandierten Prozessdefini-

```
channel is_0_o31_a
channel isnot_0_o31_a
channel set_o31_a_to_0, set_o31_a_to_1
channel dr_low, dr_high

C82_B361 = C82_B363
C82_B363 = C89_B312 ; C82_B362
C82_B362 = SKIP
C89_B312 = C89_B315
C89_B315 = C89_B317
C89_B317 = (is_0_o31_a -> INT_T2 ; C89_B320 |~|
    isnot_0_o31_a -> INT_T2 ; C89_B317)
C89_B320 = (set_o31_a_to_1 -> SKIP ; INT_T2) ; C89_B324
C89_B324 = ((dr_high -> INT_T2 ; STOP)  |~| SKIP) ; (
    C89_B327 |~| C89_B329)
C89_B327 = C89_B329
C89_B329 = C89_B330
C89_B330 = (set_o31_a_to_0 -> SKIP ; INT_T2) ; C89_B333
C89_B333 = C89_B313
C89_B313 = SKIP

INT_T2 = SKIP

TASK_2_ENTRY = C82_B361
TASK_2 = TASK_2_ENTRY

assert STOP [T= TASK_2
```

(davor)

```
channel dr_high, is_0_o31_a, isnot_0_o31_a,
    set_o31_a_to_0, set_o31_a_to_1

P1000 = STOP
P1001 = ((isnot_0_o31_a -> P1001)  |~|  (is_0_o31_a ->
    set_o31_a_to_1 -> ((set_o31_a_to_0 -> SKIP)  |~|  (
    dr_high -> STOP))))

assert P1000 [T= P1001
```

(danach)

Abbildung 11.4: Die Anwendung aller Teiltransformationen

tionen zu entfernen. Werden diese nicht mehr benötigt, sind sie auch nicht mehr erreichbar und werden von den folgenden Transformationen ignoriert. Es gilt das Prinzip: Gelegenheiten, die bei dieser Ausführung verpasst werden, nutzt vielleicht die nächste.

Die Implementierung hat pro Durchlauf eine Laufzeit in $\widetilde{\boldsymbol{\sigma}}(\boldsymbol{n})$.

11.6.3.3 Beispiel

Nachdem alle Teiltransformationen beschrieben wurden, wird ein Beispiel betrachtet, das den Gesamteffekt des Werkzeuges darstellt. Abbildung 11.4 zeigt einen Ausschnitt eines CSP_M-Modells. (Tatsächlich handelt es sich bei dem hier dargestellten `TASK_2` um den hochprioren Task des Beispiels aus Abbildung 13.1 (Seite 222). Dieser Umstand ist hier aber unwichtig.)

Es wurde dem Task eine Prüfbedingung zur Seite gestellt, damit er referenziert wird. Da es sich um einen höchstprioren Task handelt, ist sein Unterbrechungsprozess `SKIP`.

Die Abbildung zeigt, dass sich das Modell durch das Werkzeug CSPC erheblich verkleinern lässt.

11.7 Steckbrief

Eine Übersicht über die Implementierung des Werkzeuges CSPC gibt der Steckbrief, der in Abbildung 11.5 dargestellt ist.

CSPC

Werkzeug, das CSP_M-Modelle komprimiert

Importierte Bibliotheken/Technologien

Lexer/Parser-Generator Antlr · Antlr-Runtime-Bibliothek

Werkzeugimplementierung

Frontend – Java	2 Dateien 300 Zeilen
Frontend – Antlr-Grammatik	1 Datei 75 Zeilen
Transformationen	26 Dateien 1036 Zeilen
Backend	1 Datei 225 Zeilen

Eigene Bibliotheken

Zwischendarstellung + Traversierung (Abstrakter Syntaxbaum)	25 Dateien 460 Zeilen
Initialdarstellung + Traversierung (generierter konkreter Syntaxbaum)	4 Dateien 1460 Zeilen

Test

Systemtest-Eingaben (Handgeschriebene CSP_M-Dateien)	108 Dateien 2682 Zeilen
Systemtest-Eingaben (Generierte CSP_M-Dateien)	345 Dateien 46207 Zeilen
Systemtest-Framework (Bash-Scripte)	6 Dateien 268 Zeilen

Ressourcennutzung

Laufzeit reale Systeme	1 s – 3 s
Laufzeit »kleine Beispiele« (siehe Abbildung 15.1 (Seite 260))	< 0,5 s

Abbildung 11.5: Steckbrief des Werkzeuges CSPC

Teil III

Ergebnisse, Evaluation und Diskussion

Kapitel 12

Übersicht über die Ergebnisse

Dieser dritte Teil der Abhandlung benennt die Ergebnisse, evaluiert den im zweiten Teil beschriebenen Ansatz und diskutiert die gewonnenen Erkenntnisse. Dieses Kapitel ist eine Übersicht über die Inhalte der folgenden Kapitel. Es stellt die Zusammenhänge der dort geschilderten Ergebnisse dar.

12.1 Forschungsinteresse

Im ersten Teil dieser Abhandlung wurde festgestellt, dass die Data-Race-Analyse, insbesondere die konservative statische, ein etabliertes Forschungsfeld mit hoher praktischer Relevanz ist. Bei der Arbeit an diesem Themenfeld ist der Autor der vorliegenden Abhandlung auf Programmiermuster gestoßen, die hier als explizite Zustandsverwaltung bezeichnet werden.

Solche Muster sind Jannesari et al. [JKS+13] in Desktop-Applikationen aufgefallen. Sie bezeichnen das Konzept als *»ad-hoc Synchronization«*. Sowohl Keul [Keu12] als auch Schwarz et al. [SSVA14] finden solche Muster in

eingebetteter Steuersoftware aus Automobilen. Schwarz et al. verwenden dabei den Begriff *»Value-Dependent Synchronization«*.

Obwohl diese Muster wiederholt erkannt wurden, gibt es insbesondere für den Bereich eingebetteter Systeme aus Automobilen keine befriedigende Lösung, um sie zu analysieren. Sowohl Keul als auch Schwarz et al. haben effiziente und relativ leichtgewichtige Analyseansätze entwickelt, mussten dafür aber Einschränkungen in Kauf nehmen. Das hier beschriebene Forschungsanliegen entwickelt und untersucht einen schwergewichtigen Ansatz zur Analyse expliziter Zustandsverwaltung. Es soll die Qualität seiner Ergebnisse und seine Anwendbarkeit an realen Systemen untersucht werden.

12.2 Betrachtungen zur Mächtigkeit

Der Ansatz zur Analyse expliziter Zustandsverwaltung geht technisch über einfache Mustererkennung und herkömmliche Ansätze zur Data-Race-Erkennung deutlich hinaus. Zusätzlich werden an ihn nur schwache Forderungen bezüglich Laufzeit und Ressourcennutzung gestellt. Die Erwartung liegt daher nahe, dass er hochwertigere und differenziertere Ergebnisse liefert, als das mit einfacheren Mitteln möglich wäre.

In Kapitel 13: »Mächtigkeit des Ansatzes« wird daher die Mächtigkeit des Ansatzes ergründet. Dazu wird der Ansatz an relativ kleinen handgeschriebenen Programmen, die explizite Zustandsverwaltung enthalten, ausprobiert. Nebenbei wird das einfach gehaltene Framework gezeigt, mit dem die Implementierungen des Ansatzes in geeigneter Verschachtelung ausgeführt werden können. Das Framework wird dabei durch Screenshots dokumentiert.

In Kapitel 14: »Abgrenzung zu Keul und Schwarz et al« werden die Ergebnisse des Ansatzes mit denen der Ansätze von Keul und Schwarz et al.verglichen. Dazu wird der Ansatz auf Beispiele angewendet, die von Keul und Schwarz et al. selbst geliefert werden. Zusätzlich werden anhand von speziell ausgewählten Programmen Grenzen, Erfolge und Schwächen aller drei verglichenen Ansätze aufgezeigt. Es gelingt damit zu zeigen, dass

der hier vorgestellte Ansatz eine wesentlich tiefere Analyse von explizite Zustandsverwaltung ermöglicht und damit in vielen Fällen eine bessere Einschätzung von explizite Zustandsverwaltung liefert. Das Kapitel enthält auch eine tabellarische Übersicht über die diskutierten Ansätze.

12.3 Betrachtungen zur Anwendungsfähigkeit

Für den praktischen Einsatz ist neben der Qualität der Ergebnisse vor allem auch die Möglichkeit, diese in der Praxis zu erlangen, von großer Bedeutung. Während bei den im Wesentlichen Muster-basierten Ansätzen kaum Zweifel an der Praxistauglichkeit aufkommen, ist bei schwergewichtigeren Ansätzen eine differenziertere Betrachtung nötig.

Wie nicht anders erwartet zeigen sich zunächst Schwierigkeiten, wenn die beim Einsatz an realen Systemen erzeugten CSP_M-Modelle direkt den Refinement-Checker übergeben werden. War bei den Betrachtungen zur Mächtigkeit keine Vorverarbeitung nötig, ist diese beim Einsatz an realen Systemen zwingend notwendig.

In Kapitel 15: »Mit CSPC zu großen Systemen« wird daher der Einsatz des Vorverarbeitungswerkzeuges CSPC begründet und sein Effekt untersucht. Da die korrekte Funktionsweise des Werkzeuges von herausragender Bedeutung für den gesamten Ansatz ist, wird die Verifizierung und Validierung des Werkzeuges näher beschrieben. Es lässt sich zeigen, dass CSPC für den Einsatz an realen Systemen schnell genug ist und selbst bei Eingaben, die größer als die in der Praxis zu erwartenden sind, die Laufzeit im akzeptablen Bereich bleibt.

Ausgerüstet mit dem Werkzeug CSPC widmet sich Kapitel 16: »Anwendung auf reale Systeme« dem Einsatz des Ansatzes an realen Systemen. Es wird gezeigt, wie sich der Ansatz bei Anwendung auf Eingaben aus einem Testkorpus bestehend aus realen Beispielen verhält. Es wird die Eignung der unterschiedlichen Refinement-Checker verglichen. Außerdem werden einige in den realen Systemen beobachtete Muster dokumentiert und ihre Synchronisationseigenschaften besprochen.

Es zeigt sich, dass der Ansatz der vorliegenden Abhandlung trotz seiner schwergewichtigen Techniken auch an realen Systemen einsetzbar bleibt. Es lassen sich brauchbare Kompromisse aus Laufzeit und Antwortwahrscheinlichkeit erzielen.

Kapitel 13

Mächtigkeit des Ansatzes

Dieses Kapitel diskutiert die Mächtigkeit des implementierten Ansatzes zur Analyse von expliziter Zustandsverwaltung. Es zeigt Möglichkeiten und Grenzen auf und belegt diese anhand von praxisnahen, aber klein gehaltenen Beispielen. Es zeigt sich dabei, dass der Ansatz geeignet ist, Zustandsverwaltungsmuster präzise zu analysieren. Weitere Beispiele für Anwendungen des Ansatzes werden in Wittiger und Felden [WF15] dargestellt.

13.1 Zustandsvariablen als Mutexe?

Das wohl bekannteste Synchronisationsverfahren ist das der Mutexe. Es liegt nahe zu versuchen, ein Mutex-Muster mit Zustandsvariablen aufzubauen und zu untersuchen, ob der in dieser Abhandlung vorgestellte Ansatz zur Analyse von expliziter Zustandsverwaltung hinreichend mächtig für die Analyse eines solchen Musters ist.

Abbildung 13.1 zeigt einen Quelltext indem ein Mutex-ähnliches Muster implementiert ist. Die Variable `a` wird darin wie ein Mutex behandelt. Dazu werden die C-Makros `lock(.)` und `unlock(.)` definiert und verwendet. Bevor der Ablauf im Detail besprochen wird, sei das Ergebnis genannt: So

```
1  #define lock(X) do {while (X) {}; X = 1;} while (0)
2  #define unlock(X) do X = 0;  while (0)
3
4  /* the 'mutex' */
5  volatile enum {UNLOCKED = 0, LOCKED} a;
6
7  /* the dr variable */
8  int dr;
9
10 void task_low (void) {
11    lock(a);
12
13    dr = 30;
14
15    unlock(a);
16 }
17
18 void task_high (void) {
19    lock(a);
20    if (dr != 0) dr += 20;
21    unlock(a);
22 }
```

Abbildung 13.1: Aus Enums und Zustandsvariablen selbstgestrickte Mutexe

wie hier dargestellt, wird die implementierte Werkzeugkette alle Data-Races auf der Variablen `dr` ausschließen.

Im Beispiel wird die Mutexvariable nicht initialisiert. Wird dagegen die Variable zum Wert `UNLOCKED` initialisiert, würde keine Data-Race-Warnung mehr als falsch-positiv erkannt werden. Es scheint hier also mehr zu passieren, als man auf den ersten Blick erwartet.

13.1.1 Ausführung der Werkzeuge

Um die Werkzeugkette, die für diese Anhandlung entwickelt wurde, einfach ausführen zu können, wurde ein kleines Framework entwickelt, das die einzelnen Konsolenwerkzeuge der Werkzeugkette konsistent konfiguriert

und ausführt. Dieses Framework ist in den Abbildungen 13.2 und 13.3 dargestellt.

Der Build-Befehl (siehe Abbildung 13.2 – oben) »baut« dabei das in einer Datei mit dem Namen `mutex_macro.c` stehende Programm aus Abbildung 13.1. Dazu verwendet er eine Nebenläufigkeitskonfiguration, welche die beiden Prozeduren `task_low` und `task_high` als Taskeinsprungspunkte auszeichnet und ihnen Prioritäten zuweist. Dabei wird der höherpriore Task `h` und der niedrigerpriore Task `l` genannt. Das prioritätenbasierte Scheduling wird somit berücksichtigt. Es gibt in dem Programm (hier nicht dargestellt) noch eine `main`-Prozedur. Diese hat einen leeren Rumpf.

Mit dem Show-Befehl werden dann die ermittelten Data-Race-Warnungen angezeigt (siehe Abbildung 13.2 – unten). Dabei ist auch die Zustandsvariable Subjekt einiger Data-Race-Warnungen. Dies ist zu erwarten, da die verwendete Data-Race-Analyse Warnungen unabhängig von der Deklaration der Variablen auflistet. Die Tatsache, dass die Variable `a` hier als `volatile enum` deklariert ist, hat keinerlei Einfluss.

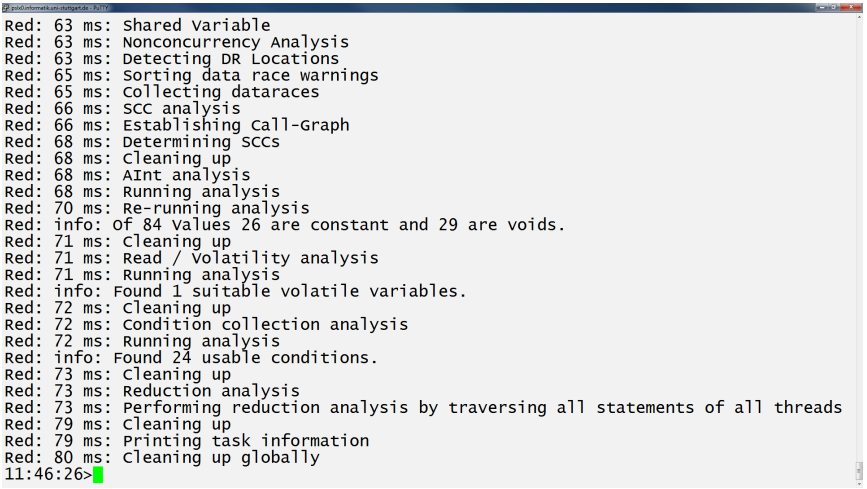
Wichtiger sind die Data-Race-Warnungen auf der Variablen `dr`. Diese sollen durch die zustandsbasierte Synchronisation ausgeschlossen werden.

Dafür wird der Check-Befehl verwendet. (siehe Abbildung 13.3 – oben). Dieser nutzt die vom bereits ausgeführten Werkzeug Red bereitgestellte Datei, um die benötigten drei CSP_M-Modelle zu erzeugen. Durch Konfiguration wurde festgelegt, dass als Zustandsvariablen alle Variablen verwendet werden, die die Heuristik für geeignet hält, und dass hier, weil das Beispiel ohnehin nur sehr klein ist, auf den Einsatz von CSPC verzichtet wird.

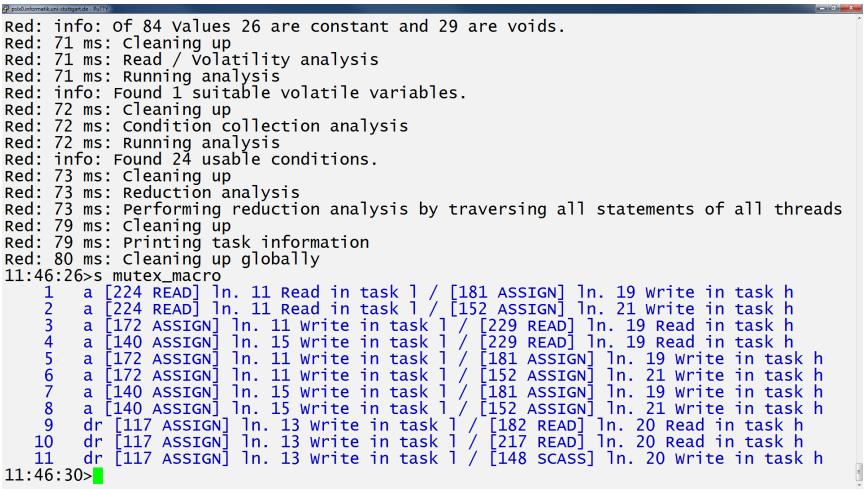
Die erzeugten CSP_M-Modelle werden dann mitsamt den Prüfbedingungen dem Refinement-Checker FDR2 übergeben. Dieser kann in allen drei Fällen ein Data-Race ausschließen.

13.1.2 Synchronisationsverhalten

An dieser Stelle könnte die Betrachtung abgeschlossen werden, wäre da nicht das seltsame Verhalten, wenn die Zustandsvariable initialisiert wird. Stattdessen soll im Detail betrachtet werden, wie die Tasks interagieren.



Mit dem Befehl »`b mutex_macro`« (build) wurden mehrere Bauhaus-Werkzeuge (unter anderem der Parser und der Bauhaus Data Race Detector) ausgeführt. Zuletzt wurde das Werkzeuges Red ausgeführt, dessen Statusmeldungen zum Teil noch sichtbar sind. Die Ausführung benötigt circa 2 Sekunden.



Mit dem Befehl »`s mutex_macro`« (show) wurden die Data-Race-Warnungen angezeigt. Es werden elf Data-Race-Warnungen ausgegeben – drei davon auf der Variablen `dr` und acht auf der später als Zustandsvariable ausgewählten Variablen `a`.

Abbildung 13.2: Screenshots der Analyse I

```
Red: 72 ms: Cleaning up
Red: 72 ms: Condition collection analysis
Red: 72 ms: Running analysis
Red: info: Found 24 usable conditions.
Red: 73 ms: Cleaning up
Red: 73 ms: Reduction analysis
Red: 73 ms: Performing reduction analysis by traversing all statements of all threads
Red: 79 ms: Cleaning up
Red: 79 ms: Printing task information
Red: 80 ms: Cleaning up globally
11:46:26>s mutex_macro
    1   a [224 READ] ln. 11 Read in task 1 / [181 ASSIGN] ln. 19 Write in task h
    2   a [224 READ] ln. 11 Read in task 1 / [152 ASSIGN] ln. 21 Write in task h
    3   a [172 ASSIGN] ln. 11 Write in task 1 / [229 READ] ln. 19 Read in task h
    4   a [140 ASSIGN] ln. 15 Write in task 1 / [229 READ] ln. 19 Read in task h
    5   a [172 ASSIGN] ln. 11 Write in task 1 / [181 ASSIGN] ln. 19 Write in task h
    6   a [172 ASSIGN] ln. 11 Write in task 1 / [152 ASSIGN] ln. 21 Write in task h
    7   a [140 ASSIGN] ln. 15 Write in task 1 / [181 ASSIGN] ln. 19 Write in task h
    8   a [140 ASSIGN] ln. 15 Write in task 1 / [152 ASSIGN] ln. 21 Write in task h
    9   dr [117 ASSIGN] ln. 13 Write in task 1 / [182 READ] ln. 20 Read in task h
   10   dr [117 ASSIGN] ln. 13 Write in task 1 / [217 READ] ln. 20 Read in task h
   11   dr [117 ASSIGN] ln. 13 Write in task 1 / [148 SCASS] ln. 20 Write in task h
11:46:30>c mutex_macro 9 11
mutex_macro / 9 (m): true (Ausschluss: Kein Data-Race)
mutex_macro / 10 (n): true (Ausschluss: Kein Data-Race)
mutex_macro / 11 (p): true (Ausschluss: Kein Data-Race)
Completed
11:46:56>
```

Mit dem Befehl »`c mutex_macro 9 11`« (check) wurden die Data-Race-Warnungen 9 – 11 überprüft. Die Werkzeugkette ist dabei so konfiguriert, dass alle von der Heuristik als tauglich bewerteten Variablen – in diesem Fall nur die Variable `a` – als Zustandsvariable verwendet werden. Der Refinement-Checker FDR2 schließt in allen drei Fällen ein Data-Race aus. Die Ausführung benötigt circa 1 Sekunde.

```
Red: 32 ms: Performing reduction analysis by traversing all statements of all threads
Red: 34 ms: Cleaning up
Red: 34 ms: Printing task information
Red: 34 ms: Cleaning up globally
11:47:18>s mutex_macro
    1   a [230 READ] ln. 11 Read in task 1 / [112 INIT] ln. 5 Write in task main
    2   a [230 READ] ln. 11 Read in task 1 / [186 ASSIGN] ln. 19 Write in task h
    3   a [230 READ] ln. 11 Read in task 1 / [155 ASSIGN] ln. 21 Write in task h
    4   a [235 READ] ln. 19 Read in task h / [112 INIT] ln. 5 Write in task main
    5   a [177 ASSIGN] ln. 11 Write in task 1 / [235 READ] ln. 19 Read in task h
    6   a [143 ASSIGN] ln. 15 Write in task 1 / [235 READ] ln. 19 Read in task h
    7   a [177 ASSIGN] ln. 11 Write in task 1 / [112 INIT] ln. 5 Write in task main
    8   a [143 ASSIGN] ln. 15 Write in task 1 / [112 INIT] ln. 5 Write in task main
    9   a [186 ASSIGN] ln. 19 Write in task h / [112 INIT] ln. 5 Write in task main
   10   a [155 ASSIGN] ln. 21 Write in task h / [112 INIT] ln. 5 Write in task main
   11   a [177 ASSIGN] ln. 11 Write in task 1 / [186 ASSIGN] ln. 19 Write in task h
   12   a [177 ASSIGN] ln. 11 Write in task 1 / [155 ASSIGN] ln. 21 Write in task h
   13   a [143 ASSIGN] ln. 15 Write in task 1 / [186 ASSIGN] ln. 19 Write in task h
   14   a [143 ASSIGN] ln. 15 Write in task 1 / [155 ASSIGN] ln. 21 Write in task h
   15   dr [118 ASSIGN] ln. 13 Write in task 1 / [187 READ] ln. 20 Read in task h
   16   dr [118 ASSIGN] ln. 13 Write in task 1 / [223 READ] ln. 20 Read in task h
   17   dr [118 ASSIGN] ln. 13 Write in task 1 / [151 SCASS] ln. 20 Write in task h
11:47:20>c mutex_macro 15 17
mutex_macro / 15 (t): false (Erreichbar: Data-Race nicht ausgeschlossen)
mutex_macro / 16 (u): false (Erreichbar: Data-Race nicht ausgeschlossen)
mutex_macro / 17 (v): false (Erreichbar: Data-Race nicht ausgeschlossen)
Completed
11:47:31>
```

Erneute Ausführung aller drei Befehle, nachdem der Quelltext so verändert wurde, dass die Zustandsvariable initialisiert wird.

Abbildung 13.3: Screenshots der Analyse II

Es zeigt sich, dass das in Abbildung 13.1 abgebildete Muster nur wenig Ähnlichkeit mit einem echten Mutex hat. Wenn der hochpriore Task startet und die Zustandsvariable bereits im Zustand `LOCKED` ist, wartet er zunächst darauf, dass sie wieder zurückgesetzt wird. Fraglich ist jedoch, wie das geschehen kann. Der einzige Task, der der die Zustandsvariable zurücksetzen könnte, ist der niederpriore Task. Dieser wird aber nicht zur Ausführung kommen, solange der hochpriore Task nicht zu Ende gelaufen ist. Das System würde in diesem Fall in ein Dead-Lock – oder je nach genauer Definition ein Live-Lock – geraten. Der Refinement-Checker erkennt dies auch. Er erkennt jedenfalls, dass sich keine Data-Race-Situation erreichen lässt, und schließt folgerichtig ein Data-Race aus.

Die Synchronisationsmechanismus hat eine weitere beunruhigende Eigenschaft. Der Lock-Prozess ist nicht hinreichend atomar. Wenn ein Prozess unterbrochen wird, nachdem er verifiziert hat, dass die Zustandsvariable den Wert `UNLOCKED` hat, aber bevor er diese auf den Wert `LOCKED` setzt, kann der ihn unterbrechende Prozess ein Lock ausführen. Wenn die Ausführung dann aber zum ursprünglichen Task zurückkehrt, wird dieser ohne weitere Überprüfung die Zustandsvariable auf den Wert `LOCKED` setzen, auch wenn sie diesen Wert mittlerweile bereits hat. Es können also unter bestimmten Umständen zwei Tasks in eigentlich geschützte Bereiche eintreten.

Ein unerwünschtes doppeltes Eintreten in einen geschützten Bereich setzt aber voraus, dass der höherpriore Task sein Lock über seine (vorläufige) Beendigung hinaus behält. Gibt er sein Lock stets vor Beendigung wieder frei – so wie im Beispiel – besteht das Problem nicht. Dies ist ein Hinweis darauf, dass die von Schwarz et al. geforderte »Intaktheit« von Tasks (siehe Abschnitt 6.3.2 (Seite 88)) in solchen Konstellationen durchaus eine wünschenswerte Eigenschaft ist.

Dass die Synchronisation mit Mutexen in Dead-Locks endet, ist nicht der kruden Implementierung geschuldet. Auch wenn hier »echte«, also mit speziellen Prozessoranweisungen implementierte, Mutexe zum Einsatz kämen, bestünde das Problem fort. Es braucht komplexere Schedulingverfahren, um Mutexe sinnvoll einzusetzen. Tasks müssten in einen Blocked-Zustand übergehen können, um es dem Halter des Locks zu erlauben, dieses wieder

freizugeben. Um dann Problemen mit »Priority Inversion« zu begegnen, sollten außerdem »Priority Inheritance«-Protokolle angewandt werden (siehe Burns und Wellings [BW01]).

Bei den verwendeten Schedulingverfahren bleibt einem Task, der auf eine gesperrte Zustandsvariable trifft, nur, die entsprechende Funktionalität in diesem Durchgang auszulassen. Dies käme dann dem zweiten Muster von Keul (siehe Abschnitt 6.3.1 (Seite 86)) recht nahe.

Dass die Zustandsvariable zunächst uninitialisiert ist, stellt kein Problem dar. Hätte die Variable zu Beginn einen Wert ungleich 0, wäre sie von Anfang an gesperrt, da in C99 jeder Wert ungleich 0 als `true` gilt. Dann könnte keiner der Data-Race-Zugriffe je ausgeführt werden. Der Refinement-Checker geht also automatisch vom »schlechtesten« Fall aus, nämlich dem, dass die Zustandsvariable zu Beginn den Wert 0 hat.

13.1.3 Initialisierungen im Main-Task

Wenn der Refinement-Checker ohnehin dazu gezwungen ist, davon auszugehen, dass die Zustandsvariable zu Beginn den Wert 0 hat, wie kann dann eine Initialisierung zu genau diesem Wert dazu führen, dass die Data-Race-Warnungen plötzlich nicht mehr zurückgewiesen werden können?

Der Grund dafür ist technischer Natur. Ein Hinweis ist, dass mit der Initialisierung weitere Data-Race-Warnungen auf der Zustandsvariablen gemeldet werden (siehe Abbildung 13.3 – unten). Es gibt in dem dargestellten System nämlich drei Tasks. Der Main-Task, der die (hier leere) Main-Prozedur aufruft, ist ein Task mit höchster Priorität. Er arbeitet aber in einer in Bauhaus als Below-Main bezeichneten Prozedur auch die Initialisierungen der Variablen ab.

Wenn nichts anderes konfiguriert ist, gehen der Bauhaus Data Race Detector und auch die Erweiterung konservativ approximierend davon aus, dass der Main-Task parallel zu den anderen Tasks laufen kann. Die Initialisierung der Variablen kann erfolgen, nachdem der niederpriore Task das Lock schon erhalten hat. In der Folge wird das Muster zerstört.

Sowohl im Bauhaus Data Race Detector als auch in der für diese Ab-

Tabelle 13.1: Wirkung des Compare-and-Swap-Mechanismus

Wert `*m`	Wert `alt`	Wert `neu`	Wirkung	Ergebnis
0	0	0	—	success
0	0	1	`*m = 1`	success
0	1	0	—	fail
0	1	1	—	fail
1	0	0	—	fail
1	0	1	—	fail
1	1	0	`*m = 0`	success
1	1	1	—	success

Die Tabelle stellt Wirkung und Ergebnis der Operation `CAS(&m, alt, neu)` dar.

handlung entwickelten Werkzeugkette gibt es daher die Möglichkeit, in der Phase, in der die Parallelität stattfindet, den Main-Task unberücksichtigt zu lassen. So konfiguriert werden auch mit Initialisierung die Data-Races ausgeschlossen.

13.1.4 Echte Mutexe

Es verbleibt die Frage, wie der Ansatz dieser Abhandlung mit echten Mutexen umgehen kann. Zur Implementierung von Mutexen werden normalerweise spezielle CPU-Instruktionen (»Compare-and-Swap«, CAS) verwendet. Diese werden als Assembler-Instruktionen verwendet[1] und sind nicht mit normalem C-Code schreibbar.

Tabelle 13.1 stellt Wirkung und Ergebnis einer Compare-and-Swap-Operation dar. Das erste Argument der Operation ist ein Zeiger auf eine Speicheradresse. Ist der darin gespeicherte Wert gleich dem zweiten übergebenen Argument, wird er mit dem Wert des dritten Arguments überschrieben, und die Operation hat den Rückgabewert »`success`«. Ist er ungleich, wird nichts verändert, und der Rückgabewert ist »`fail`«.

[1] Siehe dazu zum Beispiel die Implementierung von `pthread_mutex_lock` einsehbar unter anderem unter `ftp://sourceware.org/pub/pthreads-win32/sources/pthreads-w32-2-9-1-release/pthread_mutex_lock.c`.

```
channel a_is_0, a_isnot_0, set_a_to_0
channel a_is_1, a_isnot_1, set_a_to_1
channel set_a_to_unknown

A_0 = A_CAS0 |~| a_is_0 -> A_0 |~| a_isnot_1 -> A_0
A_1 = A_CAS1 |~| a_is_1 -> A_1 |~| a_isnot_0 -> A_1
A_OTH = A_CASOTH |~| a_isnot_0 -> A_OTH |~| a_isnot_1
    -> A_OTH

A_CAS0 = A_SET |~|
  a_cas_00_succ -> A_0 |~| a_cas_01_succ -> A_1 |~|
  a_cas_11_fail -> A_0 |~| a_cas_10_fail -> A_0
A_CAS1 = A_SET |~|
  a_cas_11_succ -> A_1 |~| a_cas_10_succ -> A_0 |~|
  a_cas_00_fail -> A_1 |~| a_cas_01_fail -> A_1
A_CASOTH = A_SET |~|
  a_cas_00_fail -> A_OTH |~| a_cas_01_fail -> A_OTH |~|
  a_cas_10_fail -> A_OTH |~| a_cas_11_fail -> A_OTH

A_SET = set_a_to_0 -> A_0 |~| set_a_to_1 -> A_1 |~|
    set_a_to_unknown -> A_UNKNOWN

A_UNKNOWN = A_0 |~| A_1 |~| A_OTH
```

(Für bessere Lesbarkeit wurden Zeilenumbrüche eingefügt.)

Abbildung 13.4: Eine Zustandsvariable, die Compare-and-Swap-Operationen erlaubt

Eine solche Operation ließe sich leicht in das hier verwendete Variablenmodell integrieren. Abbildung 13.4 zeigt, wie das erfolgen könnte. Es handelt sich bei den CAS-Operationen im Prinzip um Vergleichsoperationen, die unter bestimmten Umständen den Wert der Zustandsvariablen verändern.

Die Implementierung des Autors unterstützt solche Operationen bislang nicht. Zum einen werden in den untersuchten Systemen ohnehin keine Mutexe verwendet; zum anderen dürfte sich die Erkennung der Inline-Assembler-Anweisungen als technisch aufwändig erweisen.

```
1  void lock(int *p) {
2     while (*p) {};
3     *p = 1;
4  }
5  void unlock(int *p) {
6     *p = 0;
7  }
```

Abbildung 13.5: »Mutexe« mit Prozeduren statt Makros

13.2 Zeiger auf Zustandsvariablen

Im Folgenden werden in Anknüpfung an das Beispiel der Mutexe – versehen mit einer Abwandlung – Zeiger auf Zustandsvariablen behandelt. Es ist möglich, die `lock`- und `unlock`-Makros in Prozeduren umzuwandeln. Abbildung 13.5 zeigt, wie dies bewerkstelligt werden kann.

Die Prozeduren müssen ihr Ziel per Referenz übergeben bekommen, da sie sonst keine globale Wirkung erzielen können. Das heißt, die Prozeduren werden folgendermaßen aufgerufen: `lock(&a)` und `unlock(&a)`.

Untersucht man das veränderte Programm, erhält man nahezu gleiche Data-Race-Warnungen, und die Data-Races auf der Variablen `dr` werden auch nach wie vor ausgeschlossen. Der Quelltext der entstehenden CSP_M-Modelle ähnelt sogar stark dem bisherigen. Daraus folgt aber nicht, dass die Verwendung von Zeigern kein Problem für den neuen Ansatz darstellt. Werden zum Beispiel im höherprioren Task die folgenden weiteren Zeilen C-Code eingefügt, können die Data-Races nicht mehr ausgeschlossen werden.

```
1  int b = 0;
2  lock(&b);
3  unlock(&b);
```

Die Zeilen haben zwar erkennbar kaum Einfluss auf das Programmverhalten, erschweren aber die Zeigeranalyse. Die im Bauhaus Data Race Detector verwendete Zeigeranalyse ist weder Kontext- noch flusssensitiv. Durch die Zeilen wird daher (richtigerweise) die globale Zeigerzielmenge des Zeigers

p vergrößert. Zunächst einmal steht somit weder bei der lock- noch der unlock-Operation das Ziel fest. Dadurch wird die Semantik stark überapproximiert und der Refinement-Checker kann die Data-Race-Warnungen nicht mehr ausschließen.

Eine Eigenschaft des Bauhaus Data Race Detectors könnte hier hilfreich sein. Die vom Bauhaus Data Race Detector durchgeführte Escape-Analyse stellt nämlich fest, dass der Zeiger auf die Variable b nicht aus dem Task h entkommt. In Verbindung mit der eingeschränkten Kontextsensitivität der Zeigeranalyse führt das dazu, dass in den Aufrufen aus dem Task low das Zeigerziel nach wie vor mit Sicherheit feststeht, da das Zeigerziel b durch die Escape-Analyse ausgeschlossen werden kann. Das reicht aber hier nicht aus, um die Data-Race-Warnungen auszuschließen, da nur im Task low, nicht aber im Task high das Ziel feststeht.

Das Beispiel ist geradezu prädestiniert dafür, eine andere Verbesserung der Zeigeranalyse zu motivieren: die Referenzparametererkennung. Eine kurze Erklärung zu Referenzparametern findet sich in Plödereder [Plö02].

In C-Programmen kommt es häufig vor, dass die Adresse eines Objektes nur ermittelt wird, um es mittels eines Referenzparameters zu übergeben. Es erscheint ausgesprochen sinnvoll, an dieser Stelle einer Zeigeranalyse eine begrenzte Kontextsensitivität einzubauen, die es erlaubt, das Zeigerziel in der aufgerufenen Methode zweifelsfrei zu benennen.

Im Bauhaus-Projekt wurde an solch einer Referenzparametererkennung bereits geforscht. In experimentellen Versionen des Bauhaus Data Race Detectors existiert eine (möglicherweise unvollständige) Implementierung. In der hier zugrunde gelegten stabilen Version ist diese jedoch nicht aktiviert.

Damit ist gezeigt, dass eine Verbesserung der Zeigeranalyse eine Verbesserung der Zustandsverwaltungserkennung bewirken kann. Dieser Umstand ist auch nicht weiter überraschend. Die Zustandserkennung ist, immer dann, wenn die Adresse von Zustandsvariablen ermittelt wird, begrenzt durch die Präzision der verwendeten Zeigeranalyse.

Andererseits lässt sich am verwendeten Beispiel erkennen, dass der Kontrollfluss im vorliegenden Ansatz auch interprozedural kaum abstrahiert wird. Die Ergebnisse sind innerhalb des Modells völlig pfad- und kontexsensitiv.

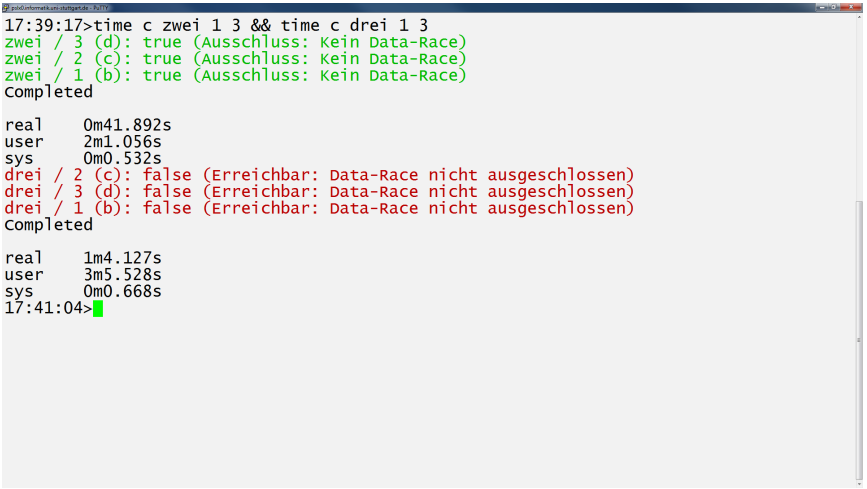
```
23  void high (void) {
24     // if (counter == 15) then Data-Race
25     if (a && b && c && d) x += 12;
26  }
27
28  void inc (void) {
29     if (d && c && b && a) d = false;
30     else if (!d && c && b && a) d = true;
31     if (c && b && a) c = false;
32     else if (!c && b && a) c = true;
33     if (b && a) b = false;
34     else if (!b && a) b = true;
35     if (a) a = false;
36     else a = true;
37  }
38
39  void low (void) {
40     // Reset counter
41     a = false; b = false;
42     c = false; d = false;
43
44     while (true) {
45        inc(); inc(); /* inc(); */
46
47        x = 5;
48     }
49  }
```

Abbildung 13.6: Beispiel eines zählenden Musters

13.3 Volle Pfad- und Kontextsensitivität

Die völlige Pfad- und Kontextsensitivität lässt sich an einem Beispiel noch deutlicher demonstrieren. Dazu wird das in Abbildung 13.6 abgebildete Programm betrachtet.

Kernstück des Beispiels ist ein Zähler, der mittels vier boolescher Zustandsvariablen `a`, `b`, `c` und `d` implementiert ist. Dabei steht in der Variablen `a` das Least-Significant-Bit und in der Variablen `d` das Most-Significant-Bit. Der



Konsekutiv wurden jeweils die drei Checks für die Beispiele »zwei« und »drei« parallel ausgeführt. Dafür wurden von FDR2 in diesem Ablauf etwa 42 Sekunden und 64 Sekunden benötigt.

Abbildung 13.7: Screenshot der Analyse des Beispiels aus Abbildung 13.6

Zähler hat die übliche Überlaufsemantik. Auf den Zählerstand 15 folgt der Zählerstand 0.

Die vielleicht etwas seltsam anmutende Prozedur `inc` inkrementiert diesen Zähler in einer Art und Weise, die vollständig vom Werkzeug Red erhalten wird. Die Funktion bildet einen Funktionalitätsrahmen für die beiden Tasks `low` und `high`.

Der Task `high` greift auf die möglicherweise Data-Race-behaftete Variable `x` nur dann zu, wenn der Zähler gerade auf 15 steht. Der Task `low` inkrementiert in einer Endlosschleife den Zähler im hier dargestellten Beispiel zweimal und greift dann auf die mögliche Data-Race-Variable zu. In dem abgewandelten Beispiel – »drei« genannt – ist der dritte, im Beispiel »zwei« noch auskommentierte Aufruf an die Inkrementierungsfunktion aktiv.

Für beide Varianten des Beispiels soll die Data-Race-Warnung auf der Variablen `x` geprüft werden. Abbildung 13.7 zeigt die Ausführung.

Festgestellt werden kann, dass im Beispiel »zwei« das Data-Race ausgeschlossen ist. Das liegt daran, dass der Wert des Zählers immer eine gerade Zahl ist, wenn der Task `low` auf die Variable `x` zugreift, also insbesondere nie 15 ist.

Anders verhält es sich im Beispiel »drei«. Hier ist der Wert des Zählers im fünften Durchlauf der Schleife 15. Das Data-Race ist damit erreichbar. Der Refinement-Checker erkennt dies auch erwartungsgemäß. Zusammengefasst belegen die Beispiele, dass der Ansatz innerhalb des Modells völlig pfad- und kontextsensitiv ist.

13.3.1 Laufzeit

Die Laufzeit ohne CSPC mit FDR2 ist recht lang (vergleiche Abbildung 13.7), obwohl das Programm nur recht kurz ist. Dies ist zum einen der Tatsache geschuldet, dass in diesem, sehr speziell geschriebenen Programm nur sehr wenig von dem Werkzeug Red abstrahiert wird – praktisch der gesamte Kontrollfluss ist für das Problem relevant. Zum Anderen besteht die Notwendigkeit, lange Pfade durch die Schleife zu betrachten.

Der Refinement-Checker FDR2 wendet kaum Kompressionen an, wenn er nicht durch den Benutzer auf Stellen hingewiesen wird, an denen dies möglicherweise sinnvoll ist. Gerade bei mehreren Zustandsvariablen scheint es ratsam, den Refinement-Checker darauf hinzuweisen, nach dem Verstecken von Variablensignalen eine Kompression anzuwenden, die von den durch das Verstecken entstehenden τ-Transitionen profitiert. Das gelingt durch eine kleine manuelle Anpassung des CSP_M-Modells. Als Kompression wurde hier die »Diamond«-Kompression (siehe Reference Manual von FDR2 [FO10]) ausgewählt.

Dies gibt Anlass zu einem kleinen vergleichenden Experiment. Eine der entstehenden CSP_M-Dateien wurde ausgewählt und es wurden vier Varianten erzeugt:

- »Original« bezeichnet die unveränderte Datei
- »CSPC« bezeichnet die mittels CSPC komprimierte Datei

Tabelle 13.2: Laufzeiten der unterschiedlichen Varianten mit unterschiedlichen Refinement-Checkern

\	FDR2	FDR3	FDR4
Original	34,99 s	5,73 s	5,69 s
CSPC	4,19 s	4,96 s	4,50 s
Diamond	0,72 s	0,43 s	0,43 s
beide	0,21 s	0,20 s	0,30 s

- »Diamond« bezeichnet die mit Komprimierungsanweisungen ergänzte Datei
- »beide« verwendet CSPC und ergänzt die Komprimierungsanweisungen

Alle vier Dateien wurden mit verschiedenen Versionen des Refinement-Checkers FDR untersucht.[1] Das Ergebnis war wie zu erwarten jedes mal identisch. Die Laufzeit unterscheidet sich jedoch deutlich. Alle Laufzeiten wurden auf dem Testsystem (siehe Tabelle 16.3 (Seite 271)) ermittelt. Die Zahlen wurden mittels des Kommandozeilen-Timers erhoben. Der Tabelle 13.2 sind die Laufzeiten der Ausführungen zu entnehmen.

In diesem Beispiel wirkt sich die Verwendung der Kompressionen ausgesprochen günstig auf die Laufzeit aus. Leider haben die Dateien, die realen Systemen entstammen, keine derartig offensichtlichen Stellen, an denen die Kompression angewandt werden kann.

13.4 Verzahnung von Zustandsvariablen

Der hier beschriebene Ansatz erlaubt es, mehrere Zustandsvariablen semantisch zu verzahnen. Ein Muster kann so aus mehreren Zustandsvariablen aufgebaut sein, von denen jede einzelne keine ausreichende Synchronisation

[1] Das Beispiel bereitet dem Refinement-Checker ProB insgesamt Schwierigkeiten bezüglich der Laufzeit. ProB in der Version 1.7 antwortet unter Verwendung beider Techniken nach 231,53 Sekunden.

```
1  volatile int a;
2  volatile int b;
3  int x;
4
5  void dr(void) {
6     x *= 37;
7  }
```

```
9  void task_low(void) {
10    if (a == 1 && b == 2)
11       dr();
12    if (a == 2 && b == 1)
13       dr();
14 }
15
16 void task_high(void) {
17    if (a == 1 && b == 1)
18       dr();
19    if (a == 2 && b == 2)
20       dr();
21 }
```

Interlock

Abbildung 13.8: Beispiel, das eine sehr einfache Verzahnung von Variablen erlaubt

bietet, die aber zusammengenommen ein sinnvolles Synchronisationsmuster bilden.

Auch dies lässt sich an einem Beispiel belegen, welches in Abbildung 13.8 dargestellt ist. In dem Beispiel sind die Zustandsvariablen `a` und `b` verzahnt. Die von Data-Race-Warnungen betroffenen Zugriffe auf die Variable `x` stehen in der Prozedur `dr`.

Recht offensichtlich kann sowohl der niederpriore als auch der hochpriore Task die fragliche Prozedur aufrufen, und zwar sowohl dann, wenn der Wert der Zustandsvariable `a` 1 ist, als auch, wenn der Wert 0 ist. Das Gleiche gilt für die Zustandsvariable `b`. Allerdings erfordert der niederpriore Task gegensätzliche Werte von `a` und `b`, während der hochpriore Task identische Werte einfordert.

Der verwendete Ansatz hat hier keine Schwierigkeiten, ein Data-Race auszuschließen. Analysen wie die von Schwarz et al., die Zustandsvariablen getrennt voneinander betrachten, können hier hingegen prinzipiell nicht funktionieren.

Kapitel 14

Abgrenzung zu Keul und Schwarz et al.

In Kapitel 6 (Seite 81) wurden die Ansätze zur Analyse von expliziter Zustandsverwaltung von Keul und Schwarz et al. vorgestellt. In diesem Kapitel wird demonstriert, dass der deutlich schwergewichtigerer Ansatz der vorliegenden Abhandlung eine tiefere Analyse von explizite Zustandsverwaltung ermöglicht und damit in vielen Fällen eine bessere Einschätzung von explizite Zustandsverwaltung liefert als die Verfahren von Keul und Schwarz et al., von denen er abgegrenzt wird. Dazu wird der Erfolg der Ansätze bei Anwendung an unterschiedlichen Beispielen verglichen. Zum Schluss werden die Eigenschaften aller drei Ansätze in einer Tabelle verglichen.

14.1 Vergleich mit Keuls Schema 1

In diesem Abschnitt wird der im Rahmen dieser Abhandlung entwickelte Ansatz mit der Zustandserkennung von Keul (siehe Abschnitt 6.3.1 (Seite 86)) nach Schema 1 verglichen. Da dazu eine lauffähige Implementierung

Tabelle 14.1: Ergebnisse unterschiedlicher Ansätze zur Analyse von expliziter Zustandsverwaltung

Beispiel	Auschluss durch Red, ...	Keul	Art
STATE–A	✓	✓	Falsch Positiv
STATE–B	✓	✗	Falsch Positiv
STATE–C	✗	✓	Richtig Positiv
STATE–D	✗	✓	Falsch Positiv

✓: Ausschluss, ✗: Kein Ausschluss

vorliegt, ist der Vergleich anhand von sorgfältig ausgewählten Beispielen einfach möglich.

Es werden vier Beispiele besprochen. Es wird dafür argumentiert, dass die Ergebnisse des neuen Ansatzes in allen vier Beispielen mindestens genauso sinnvoll sind wie die der Zustandserkennung von Keul. Das gilt auch für den Fall, in dem die Zustandserkennung von Keul ein falsch positives Ergebnis richtigerweise als falsch positiv bewertet.

In Tabelle 14.1 sind die Ergebnisse der vier Beispiele aufgeführt. Es ist jeweils angegeben, ob durch den neuen Ansatz und die Zustandserkennung von Keul die fraglichen Data-Races ausgeschlossen werden oder nicht und ob die Warnung falsch positiv oder richtig positiv ist.

Um die Daten zu erheben, wurden die in Abbildung 14.1 dargestellten Programmausschnitte mit passenden Deklarationen ergänzt und eine Nebenläufigkeitskonfiguration erstellt. Unter anderem wurde in allen vier Fällen die Variable `s` mit der Zeile »`volatile enum {GATHER = 1, SEND, STANDBY} s;`« deklariert.

Der Bauhaus Data Race Detector wurde dann so konfiguriert, dass er keine Zustandserkennung einsetzt, um die Ausgangssituation zu ermitteln. Wie erwartet hat sich dabei gezeigt, dass in allen vier Beispielen Data-Race-Warnungen für Zugriffe auf die Variable `dr` ausgegeben werden. Dann wurde mit dem neuen Ansatz geprüft, in welchen der Beispiele die Data-Race-Warnungen zurückgewiesen werden, und die Ergebnisse notiert. Zum Schluss wurde

die Zustandserkennung des Bauhaus Data Race Detector eingeschaltet, um zu prüfen, welche Data-Race-Warnungen sie ausschließt.

14.1.1 Beispiel State–A

Das Programm State–A passt in das Schema von Keul. Die Zustandsvariable wird nur an letzter Stelle innerhalb des Schutzes eines Prädikats verändert und alle drei Prädikate schließen sich gegenseitig aus.

Zusätzlich hält sich das Beispiel – wie die anderen drei auch – an die syntaktischen Vorgaben, die Keul macht. Die Zustandsvariable ist unmittelbar als Enum deklariert, die zugewiesenen Werte sind unmittelbar Enum-Konstanten und die Vergleiche beziehen sich ebenfalls auf Enum-Konstanten. Es wird also insbesondere auf Vergleiche mit Literalen und Ausdrücken mit statisch bekannten Werten verzichtet.

Dementsprechend hat die Zustandserkennung von Keul auch keine Schwierigkeiten, die Data-Races auf der Variablen `dr` auszuschließen. Der neue Ansatz kommt zum selben Ergebnis. Er schließt ebenfalls die Data-Races auf der Variablen `dr` aus.

14.1.2 Beispiel State–B

Auch das zweite Beispiel hält sich an das Schema 1 von Keul. Der Task mittlerer Priorität führt keinen relevanten Code aus. Im höchstprioren Task sind zwei Prädikate verschachtelt. Das führt dazu, dass die Zustandserkennung von Keul das Data-Race nicht ausschließt. Sie berücksichtigt nur das zuletzt durchlaufene Prädikat bei der Entscheidung. Dieses ist hier »`s != SEND`«, welches nicht widersprüchlich zum Prädikat des niederprioren Tasks ist. Es wird übersehen, dass das erste Prädikat, welches sehr wohl widersprüchlich zu dem des niederprioren Tasks ist, die Zuweisung genauso dominiert.

Der neue Ansatz hat hingegen keine Schwierigkeiten, die Data-Race-Warnung zurückzuweisen. Es zeigt sich hier klar der Vorteil einer tiefergehenden semantischen Analyse gegenüber der im Wesentlichen rein syntaktisch arbeitenden Zustandserkennung von Keul.

```
1  void t_low (void) {
2    if (s == STANDBY) {
3      dr = 0;
4      s = GATHER;
5    }
6  }
7
8  void t_mid (void) {
9    if (s == SEND) {
10     int l = g(dr);
11   }
12 }
13
14 void t_high (void) {
15   if (s == GATHER) {
16     dr = f();
17   }
18 }
```

STATE–A

```
1  void t_low (void) {
2    if (s == STANDBY) {
3      dr = 0;
4      s = SEND;
5    }
6  }
7
8  void t_mid (void) { }
9
10 void t_high (void) {
11   if (s != STANDBY) {
12     if (s != SEND) {
13       dr = f();
14     }
15   }
16 }
```

STATE–B

```
1  void t_low (void) {
2    if (s == STANDBY) {
3      dr = 0;
4    }
5  }
6
7  void t_mid (void) {
8    if (s == STANDBY) {
9      s = GATHER;
10   }
11 }
12
13 void t_high (void) {
14   if (s == GATHER) {
15     dr = f();
16   }
17 }
```

STATE–C

```
1  void t_low (void) {
2    if (s == STANDBY) {
3      dr = 0;
4    }
5  }
6
7  void t_mid (void) {
8    if (s == STANDBY) {
9      s = GATHER;
10   }
11 }
12
13 bool is_prime(int x);
14 void t_high (void) {
15   if (s == GATHER) {
16     if (is_prime(221))
17       dr = f();
18   }
19 }
```

STATE–D

Abbildung 14.1: Sourcecode von vier Beispielsystemen

14.1.3 Beispiel State–C

Im dritten Beispiel werden bewusst die Regeln von Keuls Schema 1 verletzt. Die Zustandsvariable wird vom mittelprioren Task im Schutze eines Prädikats verändert. Das ist nur erlaubt, wenn alle anderen Prädikate, in denen diese Zustandsvariable verwendet wird, dazu im Widerspruch stehen. Dies ist hier aber nicht der Fall. Im niederprioren Task wird ein identisches Prädikat verwendet.

Die Zustandserkennung von Keul lehnt es auch ab, Data-Races zwischen diesen beiden mit identischen Prädikaten geschützten Bereichen auszuschließen. Das reicht aber nicht aus. Durch die Verletzung der Regeln wird in diesem Beispiel das gesamte Muster unbrauchbar.

Eine Data-Race-Situation ist leicht zu konstruieren: Nachdem der niederpriore Task das Prädikat »`s == STANDBY`« verifiziert hat, kann er vom mittelprioren Task unterbrochen werden. Dieser wird dann die Zustandsvariable auf den Wert `GATHER` setzen. Damit ermöglicht er dem höchstprioren Task, jederzeit den Schutzbereich zu unterbrechen und dennoch auf die Variable `dr` zuzugreifen.

Die Zustandserkennung von Keul schließt aber fälschlicherweise genau dieses Data-Race aus. Der neue Ansatz hingegen meldet richtigerweise, dass das Data-Race erreichbar ist.

Selbst wenn es nicht sehr schwer ist, die Verletzung der Regeln zu erkennen, ist das Beispiel dennoch ein Beleg für ein gefährliches Verhalten der Zustandserkennung von Keul. In einem System realistischer Größe ist es sehr leicht zu übersehen, dass sich zwei Prädikate nicht hinreichend gegenseitig widersprechen. Zwar mag technisch gesehen immer noch der Anwender, der die Zusicherungen an das Werkzeug nicht hinreichend geprüft hat, »schuld« an der Fehleinschätzung sein, befriedigend ist solches Verhalten allerdings nicht. Eine Stärke des neuen Ansatzes ist, dass auf Randbedingungen dieser Art verzichtet werden kann.

14.1.4 Beispiel State–D

Das vierte Beispiel ist eine Abwandlung des letzten. Hier ist einer der Zugriffe auf die Variable `dr` durch die Bedingung »`is_prime(221)`« geschützt. Unter der Annahme, dass die hier nicht gezeigte Funktion `is_prime` nur dann `true` zurückgibt, wenn ihr Argument eine Primzahl ist, handelt es sich bei dem Zugriff um toten Code. Schließlich ist $\mathbf{13 \cdot 17 = 221}$ keine Primzahl.

Der neue Ansatz ist bei Weitem nicht mächtig genug, um diesen Zugriff als tot zu erkennen. Im Modell ist er also nach wie vor zu erreichen und der neue Ansatz meldet konservativ abschätzend das Ergebnis »erreichbar«.

Keuls Zustandserkennung liefert hingegen das korrekte Ergebnis: Die Warnung kann zurückgewiesen werden. Es ist jedoch ersichtlich, dass dies demselben fehlerhaften Schluss wie beim Beispiel State–C geschuldet ist. Ein konservativ abschätzendes falsches Ergebnis ist hier einem auf fehlerhaften Schlüssen basierenden richtigen vorzuziehen.

14.2 Vergleich mit Keuls Schema 2

Zum Schema 2 von Keul liegt keine Implementierung vor. Daher lässt sich hier nur zeigen, dass der neue Ansatz prinzipiell mit Mustern zurechtkommt, die diesem Schema entsprechen.

Die Variationsmöglichkeiten sind bei diesem Schema begrenzt. Abbildung 14.2 zeigt eine der Möglichkeiten. In diesem Muster werden zwei Zustandsvariablen in der vom Schema 2 vorgesehenen Art und Weise verwendet.

Der Bauhaus Data Race Detector meldet sechs Data-Race-Warnungen auf den Variablen `dr1` und `dr2`. Erwartungsgemäß und richtigerweise werden alle sechs durch den neuen Ansatz ausgeschlossen.

```
1  /* State Variables */
2  volatile enum
3     {DO, SKIP} s1;
4  volatile enum
5     {OMIT, READ, WRITE} s2;
6
7  /* The DR Variables */
8  int dr1, dr2;
9
10 void t_low (void) {
11    s2 = READ;
12
13    s1 = SKIP;
14    dr1 = 0;
15    s1 = DO;
17    s2 = OMIT;
18    dr2 = f();;
19    s2 = WRITE;
20 }
21
22 void t_mid (void) {
23    int l;
24    if (s2 == READ)
25       l += dr2;
26 }
27
28 void t_high (void) {
29    if (s1 == DO)
30       dr1 *= 7;
31
32    if (s2 == WRITE)
33       dr2 *= 11;
34 }
```

Abbildung 14.2: Quelltext eines Beispielsystems

14.3 Vergleich mit Schwarz et al.

In diesem Abschnitt werden die Ergebnisse des neuen Ansatzes mit der von Schwarz et al. [SSVA14] beschriebenen Analyse verglichen. Eine ausführbare Implementierung der Analyse liegt nicht vor. Schwarz et al. haben aber einige Beispielprogramme dokumentiert und erklärt, wie ihre Analyse damit umgeht. Diese Beispiele sollen zunächst behandelt werden. Tabelle 14.2 zeigt die Ergebnisse beider Ansätze für die in Schwarz et al. [SSVA14] beschriebenen Beispielprogramme.

Für die Programme mit den Nummern 1 – 3 wurde der Quelltext direkt dem Beitrag entnommen. Die Programme mit den Nummern 5 und 6 werden in dem Beitrag so genau beschrieben, dass sie zwar nicht syntaktisch, aber zumindest semantisch äquivalent reproduziert werden konnten. Nur das Beispiel 4 (`resource_flag`) bleibt unklar.

Bei der Einschätzung, ob es sich bei den Zugriffen auf die fraglichen Varia-

Tabelle 14.2: Ergebnisse unterschiedlicher Ansätze zur Analyse von expliziter Zustandsverwaltung anhand der Beispiele von Schwarz et al.

#	Beispiel	Auschluss durch Red, ...	Auschluss durch Schwarz et al.	Art
1	`example_flag`	✓	✓	Falsch Positiv
2	`example_flag2`	✓	✓	Falsch Positiv
3	`inverse_flag`	✗	✗	Richtig Positiv
4	`resource_flag`	(?)	✓	(?)
5	`weak_flag`	✓/✗	✗	Teils Falsch Positiv
6	`arbiter_flag`	✓	✓	Falsch Positiv

✓: Ausschluss erfolgt ✗: Ausschluss erfolgt nicht
Die Beispiele `example_flag` und `example_flag2` werden in den Abbildungen 1 und 2 aus Schwarz et al. [SSVA14] dargestellt und in dieser Abhandlung in Abbildung 6.5 (Seite 93) und Abbildung 6.6 (Seite 94) reproduziert.

blen um Data-Races handelt, kann in allen Beispielen außer dem unklaren Beispiel 4 die Einschätzung von Schwarz et al. bestätigt werden.

Um die Beispiele mit dem neuen Ansatz analysieren zu können, wurde eine für alle Beispiele identische Nebenläufigkeitskonfiguration erzeugt und die Programme zu vollständigen C-Programmen ergänzt. An dem eigentlichen Quellcode mussten keine Anpassungen vorgenommen werden.

Bei den Programmen mit den Nummern 1 – 3 sowie 6 stimmen beide Werkzeuge im Ergebnis überein. Dies ist auch stets das richtige Ergebnis.

Im Beispiel 5 (`weak_flag`) zeigt sich ein Unterschied in der Arbeitsweise der Werkzeuge. Das Werkzeug von Schwarz et al. gibt Warnungen aus, die sich auf Variablen beziehen. Wenn eine Variable ein im Konflikt stehendes Zugriffspaar hat, wird vor dieser gewarnt. Die Bauhaus-Analyse zählt hingegen die einzelnen Konfliktpaare auf.

Das Beispiel 5 setzt das Muster richtig, aber nicht gründlich ein. (Siehe Schwarz et al. [SSVA14]: *»[...] employs the flag pattern correctly, but not thoroughly.«*) Während zwei Tasks auf eine geteilte Variable im Schutze eines richtig implementierten Musters in der Art von `example_flag` zugreifen,

greift ein dritter höchstpriorer Task ohne jeden Schutz auf dieselbe Variable zu.

Beide Werkzeuge tun, was von ihnen erwartet wird. Während das Werkzeug von Schwarz et al. einfach die Variable meldet, unterscheidet der neue Ansatz jeweils die Zugriffspaare. Enthält das Paar keinen Zugriff aus dem höchstprioren Task, kann für dieses Paar ein Data-Race ausgeschlossen werden.

14.3.1 Dynamische Prioritäten

Schwarz et al. stellen sich auch der Frage, wie sich ihr Ansatz bei Systemen verhält, die dynamische Prioritätenänderung von Tasks erlauben. Sie kommen nach sehr kurzer Erörterung zu dem Schluss, dass das Problem, dynamische Prioritäten zu berechnen und nachzuverfolgen, orthogonal zum Problem der Zustandserkennung ist[1] und sie sich daher auf Programme mit ausschließlich statischen Prioritäten beschränken können.

Im Ergebnisteil ihres Beitrags wenden Schwarz et al. ihr Werkzeug dennoch auf das Beispiel 4 (`resource_flag`) an, in dem Prioritäten dynamisch verändert werden. In diesem Beispiel verändern zwei Tasks unterschiedlicher Priorität die Zustandsvariable. Um Probleme auszuschließen, wird dabei sichergestellt, dass der jeweils andere Task eine mindestens ebenso hohe Priorität hat wie der gerade aktive Task. Nicht klar ist allerdings, wie ein nachträgliches Erhöhen der Priorität des Tasks, den man möglicherweise gerade dank ursprünglich höherer Priorität unterbrochen hat, die bereits geschehene Unterbrechung verhindern kann. Damit ist auch die Einschätzung von Schwarz et al., dass die Data-Race-Warnung falsch positiv ist, anzuzweifeln. In Tabelle 14.2 wurde daher ein Fragezeichen eingetragen. Es hat den Anschein, als sei das besprochene Beispiel ein Gegenbeispiel zur These der Orthogonalität. Ohne eine genaue Spezifikation der Semantik von Prioritätenänderung und dem eigentlichen Code-Beispiel lässt sich dies jedoch nicht mit Sicherheit sagen.

[1] Wörtlich schreiben Schwarz et al. [SSVA14]: *»The issue of computing and tracking dynamic priorities, however, is orthogonal to the problem of dealing with flag variables.«*

Der neue Ansatz kann in der aktuellen Implementierung jedenfalls nicht mit dynamischen Prioritäten umgehen. Das Ergebnis wäre in einer solchen Konstellation allenfalls zufällig richtig.

14.3.2 Grenzen

Die Grenzen der Analyse von Schwarz et al. hängen eng mit den von ihnen gewählten Kriterien für Zustandsvariablen ab. Diese sind so gewählt, dass sich zumindest die dem Autor vorliegenden Systeme realistischer Größe kaum mit der Analyse bearbeiten lassen. Dies wird anhand von an realen Mustern angelehnten Beispielen belegt.

Für die Beispiele kann das Ergebnis der Analyse von Schwarz et al. aufgrund ihrer eigenen Beschreibung hergeleitet werden. Ein tatsächliches Ausführen der ohnehin nicht vorliegenden Software ist damit nicht erforderlich.

14.3.2.1 Beispiel Write-Unknown

Das Beispiel auf der linken Seite von Abbildung 14.3 zeigt ein System, das die Zustandsvariable `s` in Abhängigkeit von über ein Netzwerk empfangenen Nachrichten setzt. Das Empfangen der Nachrichten wird im Quelltext durch die Funktion `receive` angedeutet.

Das Systemverhalten hängt zwar deutlich von dem genauen Wert ab, den die Zustandsvariable `s` erhält, nicht aber das Synchronisationsverhalten. Wird die Zustandsvariable zum Beispiel auf den Wert 5 gesetzt, ist sie daher weder gleich `CRUISE` noch gleich `STOP`, wird keine Zuweisung an die Variable `x` ausgeführt. Aber unabhängig davon, welchen Wert die Zustandsvariable `s` hat, bilden die beiden Zugriffe auf die Variablen `x` niemals eine Data-Race-Situation.

Der neue Ansatz kann das nachvollziehen. Er schließt ein Data-Race auf der Variable `x` aus. Mit dem Ansatz von Schwarz et al. lässt sich das Beispiel hingegen gar nicht erst sinnvoll analysieren. Der Variablen `s` wird schließlich ein Wert zugewiesen, den die Konstantenpropagierung mit Sicherheit nicht

```c
#define CRUISE 1
#define STOP 2

volatile int s;
int x;

void task_low(void) {
   /* Zustandsvariable
      auf via Netzwerk
      empfangenen Wert
      setzen */
   s = receive();
}

void task_medium(void) {
   if (s == CRUISE)
      x = 37;
}

void task_high(void) {
   if (s == STOP)
      x = 39;
}
```

WRITE-UNKNOWN

```c
volatile int s;
int x;

void prepare(int* p) {
   msg = *p & 0x0F;
   msg <<= 8;
}

void task_low(void) {
   if (s == 1) {
      x *= 6;
      s = 5;
   }
   /* Im Fehlerfall
      Datenpaket
      vorbereiten */
   if (IND)
      prepare(&s);
}

void task_high(void) {
   if (s == 5) {
      x <<= 1;
      s = 1;
   }
}
```

READ-VIA-POINTER

Abbildung 14.3: Beispielsysteme, die Grenzen des Ansatzes von Schwarz et al. aufzeigen

als eine Konstante erkennt, und somit kommt die Variable `s` auch nicht als Zustandsvariable in Betracht.

14.3.2.2 Beispiel READ-VIA-POINTER

Auch im Beispiel auf der rechten Seite von Abbildung 14.3 kommt eine für den Ansatz von Schwarz et al. unüberwindbare Schwierigkeit vor. Die Zustandsvariable `s` bildet hier wieder ein einfaches Muster, das Data-Races

auf der Variable `x` ausschließt. Der Zustand wechselt dabei zwischen »`s == 1`« und »`s == 5`« hin und her.

Zusätzlich, und auch dies ist ein völlig übliches Vorgehen, wird der Zustand im Falle eines Fehlers in Datenpaketen gespeichert, die später möglicherweise in permanenten Speicher geschrieben werden. Das Erstellen eines solchen Datenpakets wird angedeutet.

Hierzu wird die Adresse der Zustandsvariablen ermittelt und an eine Prozedur weitergereicht. An dieser Stelle lehnt es die Analyse von Schwarz et al. ab, die Variable `s` als Zustandsvariable zu akzeptieren.

Der neue Ansatz hat hingegen keine Schwierigkeiten zu erkennen, dass die ja nur lesenden Zugriffe auf die Variable `s` die Synchronisationseigenschaften nicht negativ beeinflussen.

Tatsächlich wäre an der Stelle, an der im Beispiel die Funktion `prepare` aufgerufen wird, auch ein schreibender Zugriff auf die Zustandsvariable für die Synchronisation unproblematisch. Wenn also die Zeigeranalyse ein weniger eindeutiges Ergebnis lieferte, was in einem realistischeren Beispiel durchaus vorkommen kann, und ein möglicher schreibender Zugriff auf die Variable somit nicht ausgeschlossen werden kann, wäre der neue Ansatz nicht überfordert. Dies wurde durch Veränderung des Beispiels und erneute Ausführung der Analyse überprüft.

14.3.2.3 Beispiel Non-Intact

Es ist nicht sofort offensichtlich, wie sich die Analyse von Schwarz et al. bei Anwendung auf das in Abbildung 14.4 dargestellte Programm verhält. Die Beschreibung des verwendeten Algorithmus ist aber klar genug, um nachzustellen, was passiert.

Eine wichtige Eigenschaft des Beispiels ist hier, dass der mittelpriore Task die Zustandsvariable `s` nicht intakt lässt. Das heißt, er stellt nicht sicher, dass sie nach Ablaufen des Tasks noch oder wieder denselben Wert hat, den sie zu Beginn hatte. An sich ist dieses Verhalten auch nicht untypisch.

In diesem Beispiel berechnet die Analyse von Schwarz et al. zunächst die möglichen Werte der Zustandsvariablen bei Zuweisung an die Variable `x` in

```
1  volatile int s;
2  int x;
3
4  void task_low(void) {
5     s = 1;
6     if (s != 1)
7        s = 3;
8     x = 31;
9  }
```

```
11 void task_medium(void) {
12    s = 2;
13 }
14
15 void task_high(void) {
16    if (s == 3)
17       x = 37;
18 }
```

Non-Intact

Abbildung 14.4: Beispielsystem, das Zustandsvariablen nicht intakt lässt

Zeile 8 unter vorläufiger Vernachlässigung der Nebenläufigkeit des Systems. Diese Zustandsmenge ist **{1}**. Genauso wird die Zustandsmenge bei dem Zugriff in Zeile 17 als **{3}** ermittelt.

Nun werden noch die Auswirkungen der parallelen Prozesse hinzugerechnet. Damit ergibt sich als Zustandsmenge der ersten Zuweisung die Menge **{1; 2}**, da im mittelprioren Task der Wert 2 zugewiesen werden kann. Da die Zustandsmengen der beiden Zugriffe (**{1; 2}** und **{3}**) disjunkt sind, kann ein Data-Race ausgeschlossen werden.

Ein Ausschluss des Data-Races ist aber problematisch: Tatsächlich kann sich dadurch, dass der mittelpriore Task die Zustandsvariable nicht intakt lässt, der Kontrollfluss im niederprioren Task ändern. Es zeigt sich, dass die Zuweisung des Wertes 3 an die Zustandsvariable nicht tot ist, wie eine Nebenläufigkeit ausblendende Analyse vermuten muss.

Da Schwarz et al. die Eigenschaft der Intaktheit nicht prüfen (und wohl auch nicht prüfen können), melden sie in diesem Beispiel einen Ausschluss, dem ein Analysefehler zugrunde liegt. Die Annahme, dass eine Variable, die die strengen, von Schwarz et al. geforderten syntaktischen Kriterien erfüllt, auch das komplexe semantische Kriterium der Intaktheit erfüllt, ist sehr optimistisch. Wenn die Intaktheit verletzt wird, kann sich der Kontrollfluss durch nebenläufiges Ändern von Zustandsvariablenwerten ändern. Diese Kontrollflussänderungen berücksichtigt die Analyse nicht.

Tabelle 14.3: Vergleich der Ergebnisse der Ansätze

Beispiel	Auschluss durch Red, …	Schwarz et al.	Art
WRITE-UNKNOWN	✓	—	Falsch Positiv
READ-VIA-POINTER	✓	—	Falsch Positiv
NON-INTACT	✗	✓	Richtig Positiv
INTERLOCK	✓	✗	Falsch Positiv

✓: Ausschluss erfolgt ✗: Ausschluss erfolgt nicht
Für Beispiel INTERLOCK siehe Abbildung 13.8 (Seite 236)

Der in dieser Abhandlung beschriebene Ansatz stößt hingegen zwangsläufig auf den Pfad zur Data-Race-Situation über den »Umweg« über den mittelprioren Task. Er schließt das Data-Race nicht aus.

Das Beispiel INTERLOCK wurde bereits im Abschnitt 13.4 (Seite 235) besprochen.

14.4 Zusammenfassung der Eigenschaften

Die Eigenschaften der drei untersuchten Ansätze sind in den Tabellen 14.4 und 14.5 zusammengefasst.

Tabelle 14.4: Vergleichende Übersicht aller drei Ansätze, Teil I

\	**Red, ...**	**Keul**	**Schwarz et al.**
Konstanten	Enum-Werte, Literale, Ausdrücke mit statisch bekanntem Wert, Pseudo-Konstanten und artifizielle Zahlen-Konstanten	Enum-Werte	Werte, die die verwendete »off-the-shelf«-Konstantenpropagierung liefert.
Vergleichsoperationen	`==`, `!=`, andere werden konservativ approximiert	`==`, `!=`, andere werden konservativ approximiert	`==`, `!=`, wenn andere auftreten, wird die Variable abgelehnt.
Zuweisungen	Direkte Zuweisungen von Konstanten werden modelliert. Andere Änderungen des Wertes werden konservativ approximiert.	Direkte Zuweisungen von Konstanten werden modelliert. Die Abwesenheit von anderen Änderungen des Wertes muss der Nutzer sicherstellen.	Direkte Zuweisungen von Konstanten werden modelliert. Existieren andere Änderungen des Wertes, wird die Variable abgelehnt.
Pfad-/ Kontextsensitivität im Modell	Volle Pfadsensitivität, volle Kontextsensitivität, Kontrollfluss startet mit dem Einsprungspunkt des Tasks, der den ersten Zugriff der Data-Race-Warnung enthält.	Nur ein unmittelbar umschließendes Prädikat wird erkannt.	Volle Pfadsensitivität, volle Kontexsensitivität

Tabelle 14.5: Vergleichende Übersicht aller drei Ansätze, Teil II

\	**Red, ...**	**Keul**	**Schwarz et al.**
Zeiger auf Zustandsvariablen	Auflösung von Zeigerzielen durch Zeigeranalyse, in Zweifelsfällen konservative Approximierung	Nutzer muss sicherstellen, dass Zuweisungen an Zeigerdereferenzierungen keine Zustandsvariablen verändern. Werkzeug kann die dazu wesentlichen Informationen herleiten und anzeigen (Zeigeranalyse).	Variablen, deren Adresse ermittelt wird, werden abgelehnt.
Auswahl der Zustandsvariablen	Eine Heuristik empfiehlt, der Nutzer entscheidet, ohne mit der Entscheidung weitere Garantien abzugeben.	Automatische Erkennung. Nutzer muss Eigenschaften der Zustandsvariablen nachträglich garantieren oder Zustandserkennung abschalten. Siehe auch Abschnitt 6.3.1 (Seite 86).	Automatische Erkennung. Erfüllen die Variablen drei Bedingungen, geht die Analyse davon aus, dass eine weitere komplexe Bedingung (Intaktheit) erfüllt ist.
Zahl der Zustandsvariablen	Theoretisch beliebig, praktisch durch Laufzeit stark begrenzt.	Praktisch beliebig.	Praktisch beliebig?
Verzahnung der Zustandsvariablen	Wird vollständig nachvollzogen.	Bleibt unerkannt.	Wird ignoriert.

Kapitel 15

Mit CSPC zu grossen Systemen

Zur Herleitung der in den vorhergehenden Kapiteln dargestellten Ergebnisse wurde das Werkzeug CSPC nicht benötigt. Dort wurden kleine C-Quelltextdateien analysiert. Der in der vorliegenden Abhandlung verwendete Ansatz zur Analyse expliziter Zustandsverwaltung soll im Folgenden auch an großen Systemen erprobt werden. Dieses Kapitel stellt den Effekt dar, den CSPC auf den Einsatz an realen System hat.

15.1 Notwendigkeit

In Gibson-Robinson et al. [GABR14] werden Programme aufgelistet, die FDR2 und FDR3 als Backend verwenden. Eines davon ist CASPER, ein »Compiler for the Analysis of Security Protocols«. (Vergleiche Lowe et al. [LBDL09].) Der Autor dieser Abhandlung hat beim Einsatz des Werkzeuges von Lowe et al., das aus kurzen Spezifikationen von Kommunikationsprotokollen CSP_M-Dateien erzeugt, die Erfahrung gemacht, dass oftmals eine

manuelle Bearbeitung des erzeugten CSP_M-Codes erforderlich ist, um von FDR2 eine Antwort zu erhalten.

Ähnlich verhält es sich, wenn Refinement-Checker auf Dateien angesetzt werden, die mit dem neuen Ansatz der vorliegenden Abhandlung aus realen Systemen erzeugt werden. Bei kleinen, höchstens wenige Hundert Zeilen großen C-Programmen können die CSP_M-Modelle direkt dem Refinement-Checker übergeben werden und man erhält eine Antwort. Bei CSP_M-Modellen, die aus realen Systemen stammen, misslingt dieses Vorgehen allerdings meistens.

Beim neuen Ansatz ist es nicht vorgesehen, von Hand in die Dateien einzugreifen, daher wurde CSPC entwickelt, um die Dateien automatisiert vorverarbeiten zu können. CSPC soll manuelle Eingriffe unnötig machen, indem es die CSP_M-Modelle komprimiert.

Dass eine irgendwie geartete Vorverarbeitung bei den generierten CSP_M-Modellen notwendig ist, lässt sich experimentell leicht belegen. Es wurde FDR4 in der Version 4.2.2 eingesetzt, um Beispiele aus realen Systemen mit und ohne Vorverarbeitung zu untersuchen. Die Dateien stammen aus dem in Kapitel 16 beschriebenen Korpus. Dabei wurden sowohl die vorverarbeiteten als auch die nicht vorverarbeiteten Dateien an FDR4 übergeben. Um eine Antwort zu finden, wurde FDR4 jeweils ein bestimmtes Zeitbudget gegeben. Nach dessen Ablauf wurde der FDR4 gegebenenfalls beendet.

Tabelle 15.1 zeigt die Ergebnisse. Während es mit der Vorverarbeitung durch CSPC bei den meisten Beispielen eine Antwort gab, war FDR4 ohne sie erfolglos.

15.1.1 Diskussion

Die absolute Notwendigkeit eines Zwischenschrittes zur Komprimierung der Modelle lässt sich natürlich nicht beweisen. Es stellt sich weiterhin die Frage, ob durch die Vorverarbeitung nur das hausgemachte Problem behoben wird, dass schlicht zu verboser CSP_M-Code generiert wird. Eine offensichtliche Alternative zum hier angewendeten Vorgehen wäre, sich zu bemühen, kleinere Modelle zu erzeugen.

Tabelle 15.1: Antwortquote von FDR4 mit und ohne Vorverarbeitung

Zeitbudget	FDR4 (4.2.2)	CSPC + FDR4 (4.2.2)
2 Sekunden	0/261 0,0 %	193/261 73,9 %
20 Sekunden	0/261 0,0 %	208/261 79,7 %
200 Sekunden	0/261 0,0 %	227/261 87,0 %

Bei der Erzeugung des CSP_M-Modells stehen zusätzliche Informationen zur Verfügung, die CSPC nicht hat oder erst herleiten muss. So weiß man bei der Generierung, dass die immer wieder vorkommenden »Beliebige-Wiederholungs-Prozesse«, die dem Kleeneschen Sternoperator entsprechen, idempotent sind[1]. CSPC gelingt es im Moment meistens nicht, Vereinfachungen der Form »`P* ; P* = P*`« vorzunehmen.

Für die Komprimierung innerhalb der formalen Sprache spricht, dass die im Rahmen dieser Abhandlung entwickelten Optimierungen dadurch besser nachvollziehbar und verifizierbar sind. Für die formale Sprache CSP_M existieren mathematisch präzise Semantik-Modelle, sodass die Korrektheit der Transformation belegt werden kann. Die Auslagerung in ein eigenes Werkzeug ermöglicht auch erst den hier gewählten Testansatz. Würde bereits im Werkzeug Red die Zwischendarstellung optimiert werden, würde die Komplexität dieses Werkzeuges wohl stark anwachsen und damit das Vertrauen in seine Korrektheit sinken.

Ein weiteres vom Autor dieser Abhandlung entwickeltes, hier aber nicht besprochenes Werkzeug, das die Ausgabe des Werkzeuges Red noch vor der CSPM-Erzeugung komprimiert, hat keine praktischen Vorteile gezeigt.

[1] Der CSP-Operator `*` existiert zwar, hat aber nicht die hier gewünschte Semantik. Der Prozess `P*` hat eine Semantik, die »`P ; P ; P ; ...`« entspricht und vielleicht besser mit P^{ω} bezeichnet würde; benötigt wird hingegen »`SKIP |~| P |~| (P ; P) |~| (P ; P ; P) |~| ...`«.

Tabelle 15.2: Übersicht über die Systemtestfälle für CSPC

		Länge in Zeilen			Dateigröße		
	#	Min.	∅	Max.	Min.	∅	Max.
Hand-erstellte	108	4	25	127	38 B	442 B	2 kB
aufgesammelte	345	86	134	236	2 kB	4 kB	8 kB
zusammen	453	4	76	236	38 B	3 kB	8 kB

Die Werte sind teilweise gerundet.

15.2 Verifizierung und Validierung

Für den vorliegenden Ansatz ist das korrekte Funktionieren von CSPC von großer Bedeutung. Daher wurden große Anstrengungen unternommen, dies sicherzustellen. Wenn durch CSPC die Semantik der CSP_M-Modelle verändert würde, wären die Ergebnisse bei der Anwendung auf große Systeme in Zweifel zu ziehen.

Die Transformationen an sich sind theoretisch korrekt. Dies wurde bereits in Kapitel 11 ausführlich erläutert.

Systemtests wurden eingesetzt, um diese Korrektheit praktisch zu verifizieren und vor allem auch um die Fehlerfreiheit der Implementierung sicherzustellen. Ein Systemtestfall besteht aus einer CSP_M-Datei aus dem von hier verwendeten CSP_M-Fragment mit einer Prüfbedingung. Eine Übersicht über die Systemtestfälle zeigt Tabelle 15.2. Bei den Systemtestfällen handelt es sich zum einen um speziell von Hand erstellte CSP_M-Dateien und zum anderen um aus kleinen Beispielen entstandene CSP_M-Dateien.

- **Handerstellte Systemtestfälle:** Es wurden insgesamt 108 Systemtestfälle von Hand erstellt. Diese zielen überwiegend auf Randfälle der Teiltransformationen ab. Aufgrund ihrer Ausgestaltung als Eingabedateien handelt es sich dabei jedoch nicht um Unit-Tests, auch wenn sie eine ähnliche Zielrichtung haben.
- **Gesammelte Systemtestfälle:** Eine weitere Quelle für sinnvolle Systemtestfälle sind die aus kleinen C-Programmen erstellten CSP_M-Datei-

en, die beim Testen und Ausprobieren des Ansatzes anfallen. Es wurden 345 dieser Dateien gesammelt und der Systemtestfallbibliothek hinzugefügt.

Um einen Systemtestfall auszuführen, wird CSPC auf die Systemtestfalldatei angewendet. Anschließend wird (mit dem Werkzeug cspmchecker) sichergestellt, dass die Ausgabedatei typkorrekt ist. Außerdem wird der Refinement-Checker FDR2 auf der Ausgabedatei ausgeführt. Die Antwort von FDR2 wird dann mit der Antwort von FDR2 auf der Eingabedatei verglichen, wenn dies möglich ist – was bei den meisten der Eingabedateien der Fall ist –, oder wird mit einem vorher spezifizierten Ergebnis verglichen. Das zweite Vorgehen ist notwendig, um die Funktionalität zu testen, die dafür zuständig ist, die Dateien erst FDR2-tauglich zu machen. Dazu gehört das Auflösen leerer µ-Quantifizierungen.

Für den Systemtest wurden Dateien verwendet, die aus kleinen Beispielen stammen, weil sich bei diesen sowohl die Eingabe- als auch die Ausgabedatei von CSPC mit den Refinement-Checker erfolgreich bearbeiten lässt. Zur weiteren Validierung wurde das Werkzeug CSPC noch auf Beispiele aus realen Systemen angewandt, bei denen es eine wohlbegründete Erwartung bezüglich des Ergebnisses gab.

15.3 Laufzeit

Die Aufgabe von CSPC ist es, die CSP_M-Modelle so weit zu komprimieren, dass sie sich überwiegend in akzeptabler Zeit mit den Refinement-Checkern bearbeiten lassen. Dieses Vorgehen ist natürlich nur dann sinnvoll, wenn CSPC selbst mit moderater Laufzeit auskommt. Im Folgenden wird daher die Laufzeit von CSPC auf unterschiedlichen Eingabedateien genauer betrachtet.

15.3.1 Eingabedateien

Für die Laufzeitexperimente wurden drei unterschiedliche Arten von Eingabedateien verwendet, für die CSPC dann auch unterschiedliches Verhalten

zeigten. Die 801 verwendeten Eingabedateien lassen sich in diese Kategorien einteilen:

- **Dateien aus kleinen Beispielen:** Im Verlauf der Arbeit an dieser Abhandlung wurde der Ansatz immer wieder an kleinen Beispielsystemen erprobt. 345 dieser Dateien wurden gesammelt und für das Experiment verwendet. Es handelt sich dabei um die gleichen Dateien, die schon für die Systemtests verwendet wurden.
- **Dateien realen Systemen:** Am wichtigsten für diese Erhebung sind Dateien aus der Analyse von realen Systemen. Verwendet wurden die 261 Dateien, die aus dem in Kapitel 16 beschriebenen Korpus entstehen.
- **Übergroße Dateien:** Um die Grenzen von CSPC näher bestimmen zu können, wurden zusätzlich übergroße Dateien erzeugt. Diese sind entstanden, indem bei der Analyse realer Systeme unrealistisch viele (weitgehend unsinnige) Zustandsvariablen konfiguriert wurden. Die Zustandsvariablen wurden pseudo-zufällig ausgewählt – und zwar unabhängig davon, ob sie realistischerweise als Zustandsvariablen in Frage kommen. Es wurden 195 Dateien erstellt. Diese enthalten zwischen 357 und 3698 Zustandsvariablen (Mittelwert circa 2062, Median 2211).

In Tabelle 15.3 sind einige Daten zu den Eingabedateien dargestellt. Als Ausblick auf den Effekt, den CSPC auf die Dateien hat, ist die mittlere Größe der Ausgabedatei in der Tabelle enthalten. Abschnitt 15.4 bespricht die deutlich unterschiedlichen Kompressionsfaktoren der unterschiedlichen Gruppen von Eingabedateien.

15.3.2 Experiment

Im Experiment wurde das Werkzeug CSPC auf alle Eingabedateien angewendet und mit dem Kommandozeilenwerkzeug »time«[1] die Realzeit gemessen, die die Ausführung benötigt.

[1] Vergleiche hierzu `https://ss64.com/bash/time.html`.

Tabelle 15.3: Übersicht über die Eingabedateien des Experiments

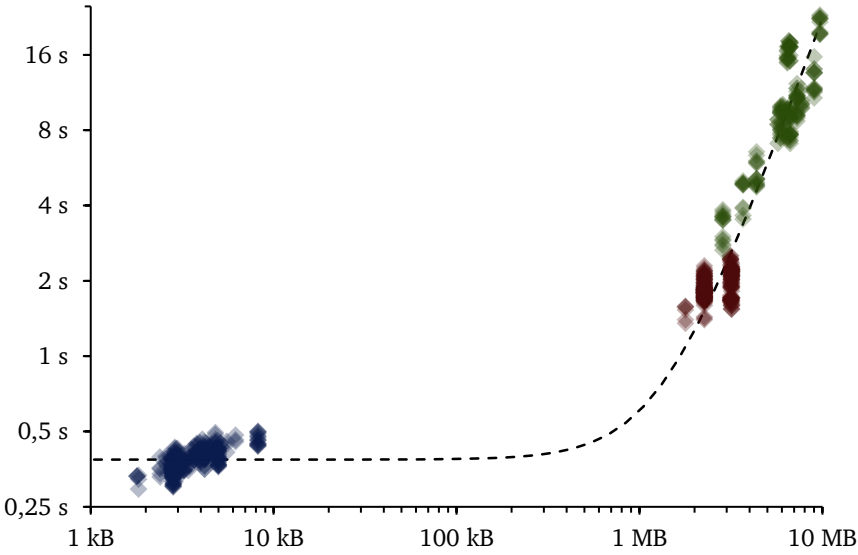
	#	Größe Eingabedatei Min.	∅	Max.	Mittlere Größe Ausgabedatei
kleine	345	2 kB	4 kB	8 kB	33,96 %
reale	261	1,7 MB	2,6 MB	3,1 MB	0,14 %
übergroße	195	2,8 MB	6,2 MB	9,5 MB	30,00 %
zusammen	801	2 kB	2,3 MB	9,5 MB	5,56 %

Die Werte sind gerundet.

Das Werkzeug CSPC ist in Java geschrieben und da die das Werkzeug ausführende virtuelle Maschine zwischen den Messungen nicht neu gestartet wurde, ist es durchaus möglich, dass durch Caching-Effekte die Ausführungszeit der späteren Durchläufe etwas reduziert wurde. Dies stellt die Ergebnisse aber nicht in Frage. Zum einen dürften ähnliche Caching-Effekte auch die Ausführung im realen Einsatz etwas beschleunigen, da auch dort oftmals mehrere Werkzeugdurchläufe kurz hintereinander stattfinden. Zum anderen ist es hier nicht das Ziel, genau zu zeigen, wie schnell sich die hier beschriebene Transformation implementieren lässt. Mit der (auch nicht sonderlich auf Performanz ausgelegten) Implementierung soll nur gezeigt werden, dass die Ausführungszeit in für die Anwendung realistischen Größenordnungen liegt.

Abbildung 15.1 zeigt die Ergebnisse der Messungen. Die hier wichtigste Gruppe ist die der Dateien aus realen Beispielen. Hier ergibt sich eine Ausführungszeit zwischen 1,4 und 2,5 Sekunden. Im Mittel sind es 1,9 Sekunden. Das bedeutet, dass das Werkzeug selbst für den interaktiven Einsatz schnell genug ist.

Dass CSPC auch bei etwas größeren Systemen nicht sofort an Leistungsgrenzen stößt, zeigt sich bei den übergroßen Systemen. Auch bei den extrem komplexen Dateien steigt die Laufzeit nur moderat. Im Mittel benötigen diese 10,3 Sekunden. Insgesamt scheinen die ermittelten Laufzeiten nicht gegen eine polynomiale Laufzeit zu sprechen.



Linke Gruppe: Dateien, die aus kleinen Beispielsystemen entstehen
Mittlere Gruppe: Dateien aus der Analyse realer Systeme
Rechte Gruppe: Übergroße Dateien
Die gestrichelte Linie stellt die Funktion $f(x) = 2,2 \cdot 10^{-13} \cdot x^2 + 0,39$ dar.

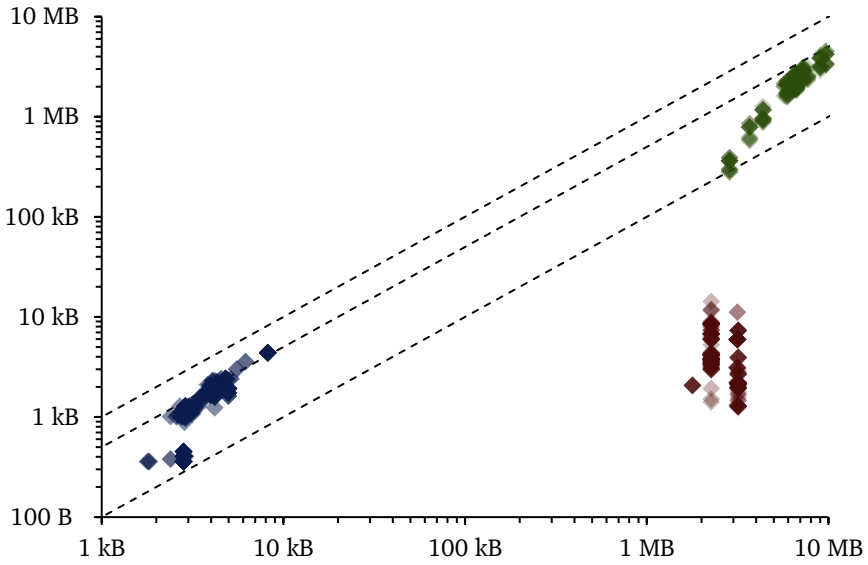
Abbildung 15.1: Diagramm, das die Laufzeit von CSPC in Relation zur Größe der Eingabedateien darstellt

15.4 Effekt

Es wurde gezeigt, dass die Vorverarbeitung von CSPC nötig und möglich ist; im Folgenden soll noch der Effekt, den CSPC auf die CSP_M-Modelle hat, näher beleuchtet werden.

15.4.1 Reduzierung der Dateigröße

Der wohl offensichtlichste Effekt des Werkzeuges CSPC ist die Reduzierung der Dateigröße. Um dies zu verdeutlichen, sind in Abbildung 15.2 die Größen der Ein- und Ausgabedateien des Experiments aus dem Abschnitt 15.3.2 gegeneinander aufgetragen.



X-Achse: Größe der Eingabedatei
Y-Achse: Größe der Ausgabedatei

Die Gruppen entsprechen denen aus der Abbildung 15.1. Die gestrichelten Linien entprechen einer unveränderten, halbierten und gezehntelten Dateigröße.

Abbildung 15.2: Diagramm, das die Größen von Ein- und Ausgabedateien von CSPC darstellt

Das Diagramm macht ersichtlich, dass ausnahmslos alle Dateien durch die Verarbeitung mit CSPC kleiner werden. Es zeigen sich aber Unterschiede im Grad der Reduzierung zwischen den drei Gruppen.

Wie Tabelle 15.3 (Seite 259) bereits vorweggenommen hat, sinken die Dateigrößen der kleinen und übergroßen Dateien im Mittel auf etwa ein Drittel der ursprünglichen Größe ab, während die realistisch großen Dateien im Mittel auf etwa ein Siebenhundertstel verkleinert werden. Dennoch sind die Ausgabedateien der realen Beispiele noch mehr als dreimal so groß wie die der kleinen Beispiele.

Der Unterschied lässt sich folgendermaßen erklären: Sowohl die kleinen als auch die übergroßen Dateien haben ein ausgewogenes Verhältnis zwi-

schen Kontrollfluss und Zugriffen auf Zugriffsvariablen. Die kleinen Beispiele enthalten wenig Kontrollfluss und wenige Zugriffe auf Zustandsvariablen; die übergroßen Beispiele enthalten viel Kontrollfluss und sehr viele Zugriffe auf Zustandsvariablen.

Bei den realen Beispielen ist das Verhältnis weniger ausgeglichen. Sie enthalten den Kontrollfluss eines realen Systems, aber nur etwas mehr Zugriffe auf Zustandsvariablen als die kleinen Beispiele. Schließlich sind viele der kleinen Beispiele erstellt worden, um in konzentrierter Form Muster zu analysieren, wie sie auch in realen Systemen vorkommen können.

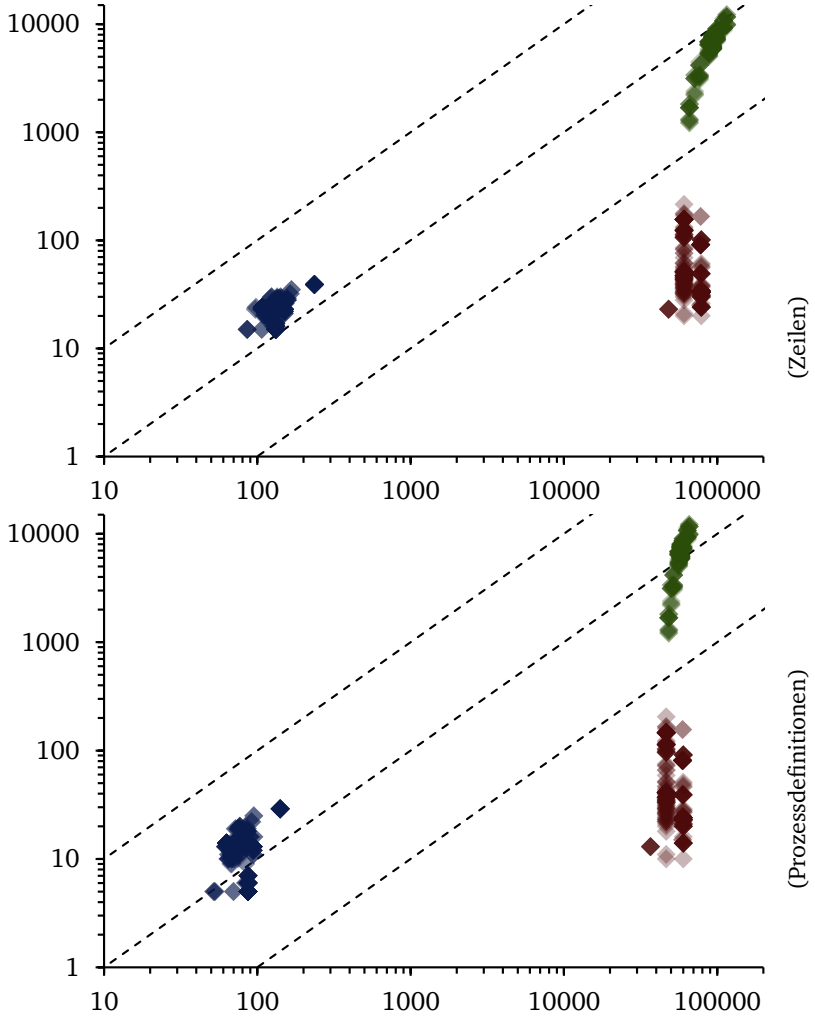
Die Daten belegen somit, dass es mit CSPC zumindest ein Stück weit gelingt, die Datenmenge auf das Wesentliche zu reduzieren. Ein Großteil des irrelevanten Kontrollflusses wird als solcher erkannt und aus der Datei entfernt. In geringerem Maße werden auch irrelevante Zuweisungen/Bedingungen durch CSPC entfernt.

15.4.2 Reduzierung der Prozessdefinitionen

Ein weiterer Effekt zeigt sich, wenn die Anzahl der Zeilen in den Ein- und Ausgabedateien betrachtet wird. Die Daten sind in Abbildung 15.3 (oben) dargestellt.

Es lässt sich im Diagramm erkennen, dass die Anzahl der Zeilen noch stärker reduziert wird als die Dateigröße. Da in den Dateien in den meisten Zeilen genau eine Prozessdefinition steht, ist dies auch ein Hinweis darauf, dass die Anzahl der Prozessdefinitionen deutlich abnimmt. Um dies zu validieren, ist die Zahl der Prozessdefinitionen im unteren Diagramm dargestellt.

Die Reduzierung der Prozessdefinitionen zeigt, dass CSPC auch einen deutlichen strukturellen Effekt hat. Würde die Verkleinerung hauptsächlich auf kosmetischen Änderungen wie dem Verkürzen von Bezeichnern basieren, würde die Anzahl der Prozessdefinitionen nicht reduziert werden. Unabhängig davon tragen die kosmetischen Änderungen, die CSPC durchführt, natürlich dennoch zur Reduzierung der Dateigröße bei.



X-Achse: Zahl der Zeilen/Prozessdefinitionen in der Eingabedatei
Y-Achse: Zahl der Zeilen/Prozessdefinitionen in der Ausgabedatei

Die Gruppen entsprechen denen aus den Abbildungen 15.1 und 15.2.
Die gestrichelten Linien entprechen einer unveränderten, gezehntelten und gehundertstelten Zeilen-/Prozessdefinitionenzahl.

Abbildung 15.3: Diagramm, das die Anzahl der Zeilen in Ein- und Ausgabedateien von CSPC darstellt

Kapitel 16

Anwendung auf reale Systeme

Im letzten Kapitel wurde die Rolle beschrieben, die CSPC beim Einsatz des hier vorgestellten Ansatzes zur Analyse von expliziter Zustandsverwaltung an realen Systemen einnehmen soll. Dieses Kapitel beschreibt und diskutiert die Ergebnisse der Analyse realer Systeme. Es wird beschrieben, wie ein Testkorpus aus passenden Eingaben erstellt wurde und geprüft, ob darin bestimmte Schwierigkeiten in angemessenem Umfang repräsentiert sind. Der Testkorpus dient dann als Grundlage einer Untersuchung der Ausführung auf realen Systemen.

16.1 Testkorpus

Um den Ansatz dieser Abhandlung an realen Systemen evaluieren zu können, wird eine Reihe von Applikationsmöglichkeiten des Ansatzes benötigt. Es wurde daher ein Testkorpus mit passenden Eingaben für den Ansatz zusammengestellt.

16.1.1 Untersuchte Systeme

Der Autor dieser Abhandlung hat durch die Kooperation seiner Forschungsabteilung mit verschiedenen Unternehmen der Automobilindustrie Zugriff auf Analyseergebnisse des Bauhaus Data Race Detector einer Zahl von eingebetteten Systemen aus Automobilen. Der eigentliche Quelltext steht dabei meist nicht zur Verfügung. Bei den hier besprochenen Systemen reichen die vorhandenen Informationen aber aus, um den beschriebenen Ansatz auszuführen.

Die Systeme sind in unterschiedlichem Maße sicherheitskritisch. Sie werden in einem späten Entwicklungszustand analysiert, der aber nicht als kundentauglich zertifiziert ist. Aufgrund von Verschwiegenheitsverpflichtungen des Autors benennt diese Abhandlung weder Hersteller noch genauen Einsatzzweck der Systeme. Keine der vom Autor dieser Abhandlung gemachten Aussagen soll ausdrücken, dass die Systeme in irgendeiner Weise unzulänglich sind.

Die untersuchten Systeme haben zwischen 35 000 und 60 000 ESLoCs. Der Begriff »ESLoC« (Efficient Source Lines of Code) bezeichnet Quelltextzeilen, aus denen mindestens ein Knoten im abtrakten Syntaxbaum hervorgeht. Damit zählen Leerzeilen und Quelltextzeilen, in denen ausschließlich Kommentare oder öffnende/schließende Klammern stehen, nicht dazu. Werden die Quelltextzeilen auf andere Art gezählt, ergeben sich teilweise deutlich größere Werte. Insbesondere Zählungen nach Makroexpansion können eine Größenordnung höher liegen.

16.1.2 Eingaben

Eine Eingabe für ein bestimmtes System benennt eine Menge zusammengehörender Zustandsvariablen und eine zu untersuchende Data-Race-Warnung. Die Menge der zusammengehörenden Zustandsvariablen identifiziert eine Instanz einer expliziten Zustandsverwaltung. Einem mit dem System vertrauten Entwicklungsingenieur dürfte es mit Quelltexteinsicht leicht fallen, Eingaben zu finden, deren Untersuchung interessiert. Ein solcher stand

aber nicht zur Verfügung, sodass es galt, mit der begrenzten Einsicht in die Systeme möglichst realistische Eingaben zu finden.

Das bedeutet, dass dem realen Einsatz entsprechend sowohl Instanzen expliziter Zustandsverwaltung auswählt wurden, bei denen die Erwartung bestand, dass diese synchronisieren, als auch solche, bei denen diese Erwartung nicht bestand. Wie in der realen Anwendung auch, sollte die angewendete Werkzeugkette die Synchronisationseigenschaften bestimmen. Bei den Instanzen expliziter Zustandsverwaltung, die nicht synchronisieren, handelt es sich nicht (oder zumindest nicht zwangsläufig) um Fehler, sondern um explizite Zustandsverwaltung, die nie zur Synchronisation gedacht war.

Das Erzeugen realistischer Eingaben bereitete einige Mühen. Bei Teilen der für untersuchenswert befundenen Instanzen expliziter Zustandsverwaltung wurden an den aus Synchronisationssicht interessanten Stellen keine Data-Race-Warnungen gefunden[1], sodass Data-Race-Warnungen künstlich eingefügt wurden, um die Instanzen expliziter Zustandsverwaltung dennoch angemessen evaluieren zu können. Manche der Systeme liegen uns in unterschiedlichen Konfigurationen vor, sodass Instanzen expliziter Zustandsverwaltung in unterschiedlichen Konfigurationen betrachtet werden konnten.

Um Instanzen expliziter Zustandsverwaltung auszuwählen, mussten geeignete Zustandsvariable gefunden werden. Die Heuristik des Werkzeuges Red lieferte für die Systeme jeweils etwa 100 Vorschläge für Zustandsvariablen. Viele der vorgeschlagenen Variablen erschienen aber offensichtlich zur Synchronisation ungeeignet. Um auch erfolgversprechende Eingaben zu finden wurde ein kleines Werkzeug implementiert, dass Schreib- und Lesezugriffe auf mögliche Zustandsvariablen übersichtlich darstellt. Dieses Vorgehen ermöglichte es vielversprechende und dennoch stark unterschiedliche behandelte Zustandsvariablen auszuwählen.

Schlussendlich wurden 261 Eingaben zu einem Testkorpus zusammengefasst. Diese stammen aus 15 Instanzen expliziter Zustandsverwaltung mit 6 bis 30 (durchschnittlich 17,4) Data-Race-Warnungen pro untersuchter

[1] Zum Beispiel, weil Data-Race-Warnungen bereits durch den Bauhaus Data Race Detector ausgeschlossen werden konnten.

Tabelle 16.1: Verteilung der Tiefe der Data-Race-Warnungen in den Eingaben des Testkorpus

Tiefe	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#	3	30	2	53	40	9	13	18	23	21	11	3	7	10	18

Instanz. Es ist nie mehr als eine Zustandsvariable gleichzeitig konfiguriert. Es wurden schlicht keine Instanzen expliziter Zustandsverwaltung gefunden, die mehrere Variablen in verzahnter Weise einsetzen.

16.1.2.1 Verteilung der Tiefe

Aufgrund des speziellen Prioritäten-basierten Scheduling-Verfahrens, das in den Systemen zum Einsatz kommt, ist es sinnvoll, die Tiefe einer Data-Race-Warnung wie folgt zu definieren: Die Tiefe einer Data-Race-Warnung ist die Anzahl der Tasks, die eine mindestens ebenso große Priorität haben wie der Task des niederprioren Zugriffs der Data-Race-Warnung.

Da der beschriebene Ansatz Tasks mit niedrigerer Priorität nicht betrachtet, entspricht die Tiefe der Anzahl der von dem beschriebenen Ansatz betrachteten Tasks. Für die Evaluation ist es daher wichtig, dass der Testkorpus Eingaben mit Data-Race-Warnung unterschiedlicher Tiefe enthält. Tabelle 16.1 zeigt die Tiefenverteilung des Testkorpus.

16.1.2.2 Repräsentation von zu `UNKNOWN` abstrahierten Zuweisungen

Die aus den 261 Eingaben resultierenden CSP-Dateien enthalten insgesamt 1226 zu `UNKNOWN` abstrahierte Zuweisungen. Nach der Vorverarbeitung bleiben 581 davon übrig. Dabei haben 95 der Dateien keine und die verbleibenden 166 Dateien durchschnittlich circa 3,5 zu `UNKNOWN` abstrahierte Zuweisungen.

Tabelle 16.2: Verwendete Versionen der Refinement-Checker

Name	Version	Datum
FDR2	FDR 2.94 Academic	12. Mai 2012
FDR3	FDR3 3.3.1	17. Juni 2015
FDR4	FDR4 4.2.2	10. September 2017
ProB 1.3	ProB CLI 1.3.6-final	4. März 2013
ProB 1.7	ProB CLI 1.7.1-final	5. Oktober 2017

16.2 Laufzeitbetrachtungen

In einer Reihe von Experimenten wurde ausprobiert, wie die Refinement-Checker mit den aus den Eingaben des Testkorpus entstehenden CSP_M-Modellen zurechtkommen. Dazu wurde für jede der Eingaben das CSP_M-Modell generiert und mit dem Werkzeug CSPC komprimiert.

16.2.1 Eingesetzte Refinement-Checker

Die entstandenen komprimierten CSP_M-Modelle wurden fünf unterschiedlichen Versionen der Refinement-Checker FDR und ProB zur Bearbeitung vorgelegt. Tabelle 16.2 zeigt die verwendeten Versionen.

Auf optimierende Konfigurationen der Refinement-Checker wurde weitgehend verzichtet. Die Dateien enthalten keine Hinweise auf Kompressionsmöglichkeiten, wie sie von FDR beachtet werden. Naheliegenderweise wird bei allen Refinement-Checkern die Kommandozeilenschnittstelle verwendet.

FDR2 wurde durch Setzen der Umgebungsvariablen `FDRPAGESIZE` auf »`128G`« erlaubt, mehr Speicher als in der Standardkonfiguration zu nutzen. In der Standardkonfiguration war er nicht in der Lage, die Dateien zu bearbeiten. Der Aufruf erfolgt mit »`fdr2 batch [FILENAME]`«.

FDR3 und FDR4 haben eine konfigurierbare Heuristik, die bestimmt, wie Teile der generalisierten kantenbeschrifteten Transitionssysteme komprimiert werden. Diese Heuristik wurde in der Standardkonfiguration belassen. Der Aufruf erfolgt mit »`refines [FILENAME]`«.

Auch ProB wurde in der Standardkonfiguration betrieben. Der Aufruf erfolgt mit »`probcli -assertions [FILENAME]`«.

16.2.2 Aufbau des Experiments

Um der realen Nutzung der Werkzeuge möglichst nahezukommen, wurde das Experiment folgendermaßen aufgebaut: Der jeweilige Refinement-Checker wurde gestartet und spätestens nach Ablauf eines vorgegebenen Zeitbudgets abgebrochen[1].

Bemerkenswert ist dabei, dass ein abgebrochener Prozess möglicherweise trotzdem ein Ergebnis meldet. Dieses Verhalten zeigen FDR3 und FDR4 wie auch ProB. Es kommt dadurch zustande, dass sich die Werkzeuge nach Ausgabe des Ergebnisses der Prüfbedingung nicht sofort beendeten, sondern noch (hier weitgehend nutzlose) Traces berechneten.

FDR3 hat die Konfigurationsoption (»`--brief`«), die dafür sorgen soll, dass nach Berechnung der Antwort der Refinement-Checker sofort beendet wird. Diese hatte aber nicht den gewünschten Effekt, sodass auf die Konfigurationsoption verzichtet wurde. Möglicherweise verbringen die Werkzeuge nach Ausgabe der Antwort auch noch Zeit mit eigentlich unnötiger feinteiliger Freigabe von Hauptspeicher. Alle gemeldeten Ergebnisse wurden gewertet.

Die beiden Versionen von ProB haben beide ab einem Zeitbudget von 2000 Sekunden bei einigen wenigen Eingabedateien nach circa 25 Minuten die Bearbeitung wegen unzureichenden Hauptspeichers abgebrochen. Dieses Verhalten wurde als Timeout gewertet.

Die Ausführung erfolgte skriptgesteuert auf einem dedizierten Testsystem. Es wurde organisatorisch sichergestellt, dass während der Ausführung keine anderen Anwendungen auf dem Testsystem ausgeführt wurden. Tabelle 16.3 zeigt einige Eigenschaften des Testsystems.

[1] Dazu wurde das Signal `SIGTERM` an den Prozess gesendet, das einen vergleichbaren Effekt hat wie das durch die Tastenkombination `Steuerung + C` gesendete Signal `SIGINT`. Hätte dies den Prozess nicht beendet, bestünde noch die Möglichkeit, das Signal `SIGKILL` zu senden. Das war hier jedoch nicht erforderlich, da die Refinement-Checker stets prompt auf das Signal `SIGTERM` reagierten.

Tabelle 16.3: Testsystem, auf dem die Experimente ausgeführt wurden

Prozessoren	4 × Intel Xeon E5–4640 v4 @ 2,1 GHz
Hauptspeicher	128 GB DDR4 (2400 MHz)
Festplatten	2 × Intel S3610 SSD (400 GB)
Betriebssystem	Debian Gnu/Linux 8 (jessie)

Tabelle 16.4: Anzahl der Antworten ungleich »Timeout« in Abhängigkeit vom Zeitbudget und vom Refinement-Checker

Zeitbudget	FDR2	FDR3	FDR4	ProB 1.3	ProB 1.7
1 s	169	176	176	74	1
2 s	183	193	193	111	86
5 s	193	194	194	118	122
10 s	194	207	207	131	137
20 s	197	208	208	144	148
50 s	210	210	210	171	188
100 s	210	211	211	189	196
200 s	211	213	213	194	196
500 s	212	220	219	198	198
1000 s	213	225	225	199	202
2000 s	213	227	227	199	203
5000 s	213	234	234	203	203

16.2.3 Ergebnisse

Tabelle 16.4 zeigt einige Ergebnisse des Experimentes. Es ist dargestellt, für wie viele der 261 Eingaben es vom jeweiligen Refinement-Checker mit dem jeweiligen Zeitbudget eine Antwort gab. Das Diagramm in Abbildung 16.1 zeigt dieselben Ergebnisse als Quote an.

Es ist festzustellen, dass FDR3 und FDR4 die besten Ergebnisse liefern. Ihre Erfolgsraten sind nahezu identisch – einzig bei einem Zeitbudget von

500 Sekunden FDR3 knapp besser – und bleiben bei jedem Zeitbudget unübertroffen.

FDR2 zeigt eine ähnliche Leistung, ist aber mit keinem Zeitbudget besser als FDR3 oder FDR4. Von Zeitbudgets, die größer als 50 Sekunden sind, kann er nur wenig profitieren. Die Ergebnisquoten der ProB-Refinement-Checker, insbesondere auch in der neueren Version 1.7, sind bei kleinem Zeitbudget deutlich geringer. Bei höherem Zeitbudget schließt die Ergebnisquote aber merklich auf. Sie erreichen jedoch nicht die Ergebnisquote von FDR3 und FDR4.

16.2.3.1 Innere Konsistenz

Die Ergebnisse, die die Refinement-Checker in dem durchgeführten Experiment liefern, sind in sich konsistent. Schafft es ein Refinement-Checker, in einem bestimmten Zeitbudget eine Antwort zu liefern, so schafft er dies auch mit größerem Zeitbudget. Alle Ergebnisse ungleich Timeout sind über Refinement-Checker- und Zeitbudget-Grenzen hinweg konsistent. Es kommt nicht vor, dass ein Refinement-Checker »Ausschluss« antwortet und ein anderer Refinement-Checker oder derselbe Refinement-Checker mit anderem Zeitbudget zum Ergebnis »Erreichbar« kommt.

Die Laufzeiten sind weitgehend deterministisch. Bei Wiederholung des Experiments ändern sich die Ergebnisse – auch bei niedrigen Zeitbudgets – kaum.

16.2.3.2 Implikationen für Anwendung

Für die praktische Anwendung der implementierten Werkzeugkette sind sowohl Antwortquote als auch Zeitbudget äußerst relevante Faktoren. Wie zu erwarten, sind beide positiv korreliert. So steigt im durchgeführten Experiment zum Beispiel die Erfolgsquote von FDR3 streng monoton mit dem Zeitbudget an.

Die für interaktive Verwendung relevanten Erfolgsquoten sind im Bereich von 5 oder 10 Sekunden Zeitbudget zu finden. Hier erreichen die drei

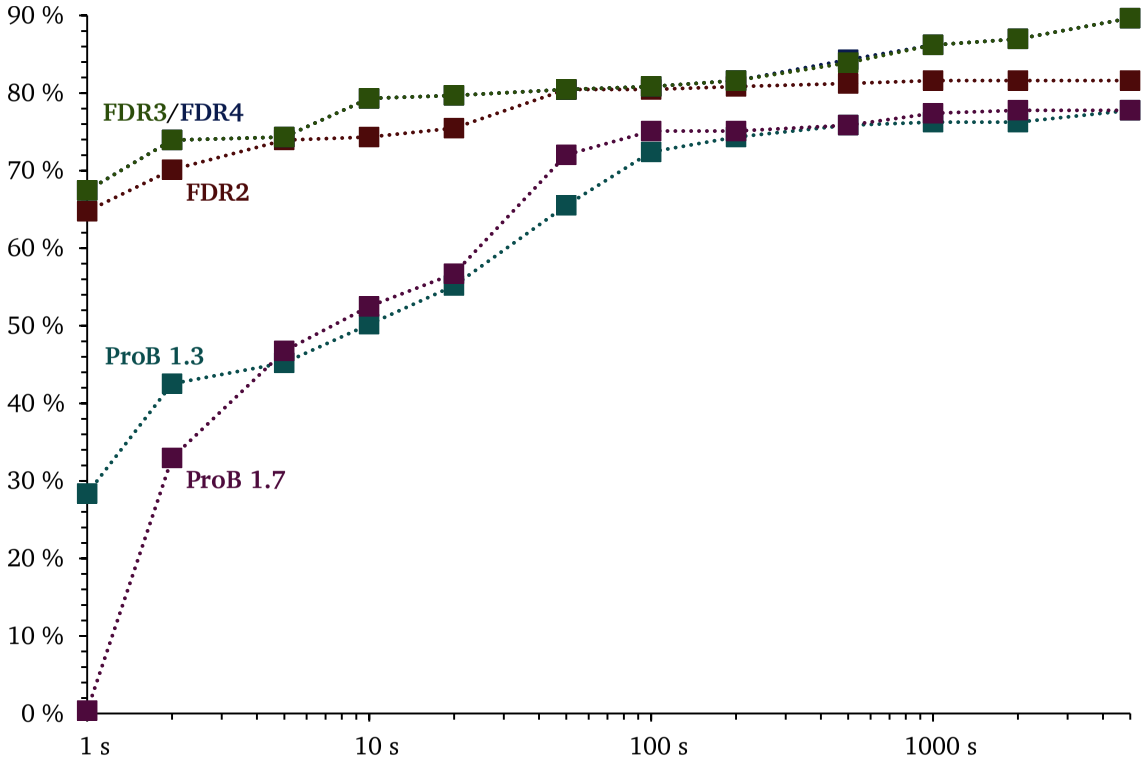


Abbildung 16.1: Antwortquote aufgeschlüsselt nach Zeitbudget und Refinement-Checker

getesteten Versionen des Refinement-Checkers FDR alle eine Erfolgsquote von etwa 75 %.

Ist man bereit, deutlich mehr Zeit zu investieren, lassen sich noch höhere Erfolgsquoten erzielen. FDR3 und FDR4 erreichen bei einem Zeitbudget von 5000 Sekunden (circa 83 Minuten) eine Quote von fast 90 %.

16.2.4 Ursachen langer Laufzeit

Aus den Ursachen der langen Laufzeit der Refinement-Checker bei Anwendung auf Modelle aus realen Systemen lassen sich möglicherweise Maßnahmen zur Verbesserung der Performanz ableiten. Daher wurden dazu die folgenden Untersuchungen angestellt.

16.2.4.1 Tiefe

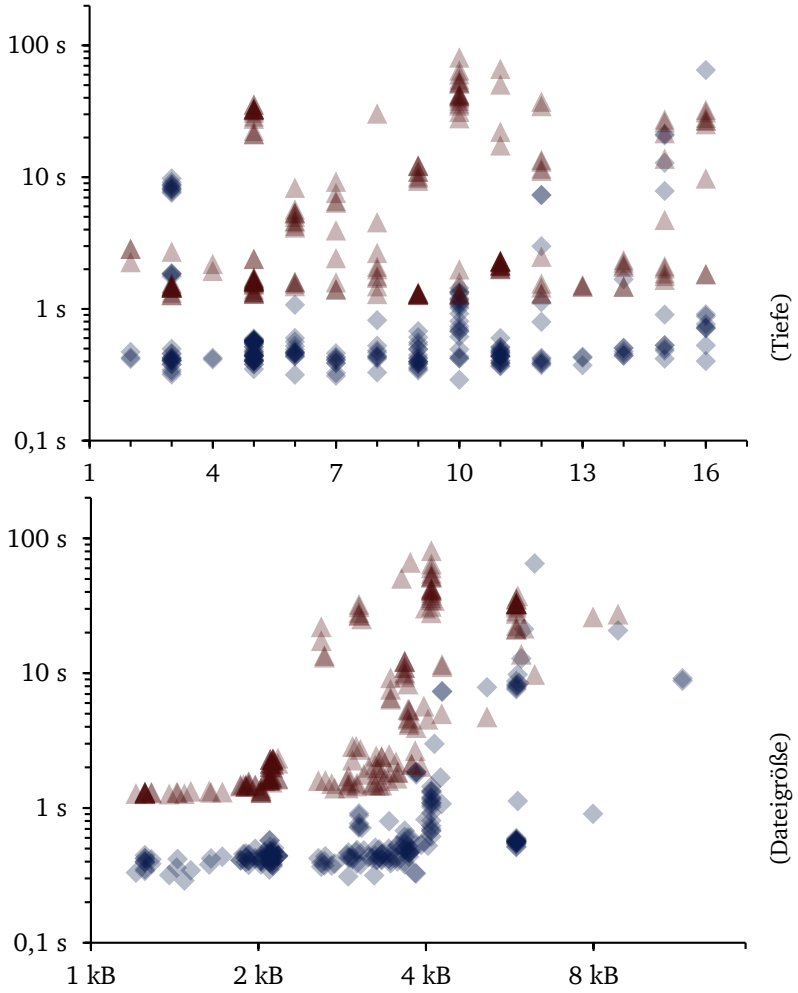
Es wurde untersucht, ob die Laufzeit wesentlich von der Tiefe der behandelten Data-Race-Warnung abhängt. Mit der Tiefe steigt schließlich der zu berücksichtigende Teil des Kontrollflusses. Es ist daher denkbar, dass sie maßgeblich die Komplexität bestimmt, mit der Refinement-Checker umgehen muss.

Die Laufzeit wurde daher mit der Tiefe der Data-Race-Warnungen ins Verhältnis gesetzt. Dies ist für die jeweils neuesten Versionen (also 4.2.2 beziehungweise 1.7) der beiden eingesetzten Refinement-Checker im Diagramm aus Abbildung 16.2 (oben) dargestellt. Die Werte stammen aus einem Durchlauf mit einem Zeitbudget von 100 Sekunden. Es werden nur die tatsächlich gelösten, also mit »Ausschluss« oder »Erreichbar« beantworteten Fragestellungen dargestellt.

Bei Betrachtung des Diagramms lässt sich erkennen, dass die Laufzeiten von ProB tendenziell länger sind als die von FDR. Ein klarer Trend, dass die Laufzeit mit der Tiefe steigt, ist jedoch nicht zu erkennen.

Es kann der Korrelationskoeffizient nach Pearson[1] berechnet werden. Da es keinen erkennbaren Grund gibt, einen linearen Zusammenhang zu

[1] Als Quelle kann hier der recht kurze Ergebnisbericht Pearson [Pea95] angegeben werden.



X-Achse: Tiefe des Data-Races/Dateigröße des CSP_M-Modells
Y-Achse: Refinement-Checker-Laufzeit
Dreiecke: ProB 1.7
Vierecke: FDR4

Abbildung 16.2: Relationen zwischen Tiefe/Dateigröße und Laufzeit

Tabelle 16.5: Erfolgsquote der Refinement-Checker in Abhängigkeit von Tiefe und Dateigröße

	Gruppe	Bereich	Gelöst von FDR	Gelöst von ProB	Quote FDR	Quote ProB
Tiefe	1	1 – 4	35/35	25/35	100,0 %	71,4 %
	2	5 – 8	73/115	73/115	63,5 %	63,5 %
	3	9 – 12	73/73	73/73	100,0 %	100,0 %
	4	13 – 16	29/38	28/38	76,3 %	73,7 %
Dateigröße	1	0 kB – 2 kB	32/32	32/32	100,0 %	100,0 %
	2	2 kB – 4 kB	121/121	119/121	100,0 %	98,3 %
	3	4 kB – 8 kB	54/80	47/80	67,5 %	58,8 %
	4	8 kB – 16 kB	3/28	1/28	10,7 %	3,6 %

vermuten, hat dieser nur geringe Aussagekraft. Es ergibt sich sowohl für FDR als auch für ProB ein schwacher positiver linearer Zusammenhang. Die Korrelationskoeffizienten nach Pearson betragen (gerundet) 0,13 und 0,14.

Bisher wurden nur die erfolgreich gelösten Eingaben betrachtet. Die Untersuchung kann auf alle Eingaben ausgeweitet werden. Dazu wurden die Eingaben in vier Gruppen mit aufsteigender Tiefe eingeteilt. Tabelle 16.5 zeigt die Lösungsquoten von FDR und ProB in den untersuchten Versionen.

Dabei ist keine klare Tendenz zu erkennen. Insgesamt scheint die Tiefe der Data-Race-Warnungen die Laufzeit nicht maßgeblich zu beeinflussen.

16.2.4.2 Dateigröße

Als weiterer möglicher bestimmender Parameter wurde die Dateigröße in Betracht gezogen. Mit Dateigröße ist dabei die Größe der Eingabe in die Refinement-Checker gemeint, das heißt die Modellgröße nach der Vorverarbeitung durch CSPC.

Im Diagramm aus Abbildung 16.2 (unten) lässt sich ein Zusammenhang zwischen Dateigröße und Laufzeit erkennen. Mit steigender Dateigröße scheint auch die Laufzeit anzusteigen.

Auch hier gibt es keinen erkennbaren Grund, einen linearen Zusammenhang zu vermuten. Bei der Berechnung der Korrelationskoeffizienten nach Pearson ergeben sich moderate lineare Zusammenhänge sowohl bei FDR als auch bei ProB. Die Korrelationskoeffizienten nach Pearson betragen (gerundet) 0,41 und 0,58.

Die Potenzregression[1] ergibt Potenzen von (gerundet) 1,2 und 2,1 für FDR und ProB.

Das Bild, das sich beim Betrachten der Tabelle 16.5 zeigt, ist hingegen recht eindeutig. Sowohl für FDR als auch für ProB sinkt die Lösungsquote der Gruppen monoton, und zwar von 100 % auf weniger als 15 %.

Insgesamt betrachtet scheint aber auch die Dateigröße die Laufzeit nicht alleine zu bestimmen. Es ist naheliegend, dass komplexitätsbestimmende Faktoren die Kompressionsmöglichkeiten von CSPC reduzieren.

16.2.4.3 Manuelle Untersuchung

Einige der Eingaben, die sich als besonders schwer für die Refinement-Checker herausgestellt haben, wurden manuell untersucht. Dabei zeigte sich in allen diesen Eingaben ein Muster: Die Eingaben enthielten Zugriffe auf Zustandsvariablen in Prozeduren, die vielfach aber nicht beliebig häufig aufgerufen werden.

Dies kann beispielsweise so aussehen: Die Zugriffe finden in einer Prozedur z statt. Diese wird von einer Betriebssystemprozedur p, vielleicht mit unterschiedlichen Parametern, viermal aufgerufen. Die Betriebssystemprozedur p wiederum wird von zahlreichen Prozeduren im Aufrufgraphen eines oder mehrerer Tasks aufgerufen.

Es liegt nahe zu vermuten, dass ein solches Muster zu einer Zustandsexplosion beitragen kann. Die Refinement-Checker werden hier zumindest zunächst gezwungen sein, alle Aufrufe der Prozedur z einzeln zu betrachten. Schließlich könnte der 27. Aufruf einen anderen Effekt haben als der 14. Ein ähnlicher Effekt wurde am »zählenden« Muster aus Abbildung 13.6

[1] Eine Regression nach der Methode der kleinsten Quadrate auf eine Funktionsgleichung der Form $f(x) = A \cdot x^B$.

(Seite 232) bereits besprochen. In den relativ leicht zu lösenden Eingaben wurde ein solches Muster hingegen nicht gefunden.

16.3 Reale Instanzen expliziter Zustandsverwaltung

Um eine Vorstellung von den ausgewählten Instanzen expliziter Zustandsverwaltung zu vermitteln, werden in diesem Abschnitt einige dem Korpus entnommene Eingaben beschrieben. Aus Gründen der besseren Verständlichkeit und wegen Vertraulichkeitsverpflichtungen wurden die Instanzen expliziter Zustandsverwaltung verfremdet und leicht vereinfacht.

16.3.1 Einfaches Muster

Eine der Instanzen expliziter Zustandsverwaltung aus dem Testkorpus folgt dem relativ einfachen Muster, das in Abbildung 16.3 dargestellt ist. Die Zustandsvariable ist hier quasi eine boolesche Variable. Sie hat entweder den Status `TIMEOUT` oder sie hat ihn nicht. Hier ist zu beachten, dass den Status `TIMEOUT` nicht zu haben nicht zwangsläufig bedeutet, den Status `INITIAL` zu haben, da eine Zuweisung existiert, die einen unbekannten Wert zuweist.

Dieses Muster funktioniert erwartungsgemäß. So gibt es beispielsweise weder Data-Races zwischen den Abschnitten `A` und `D` noch zwischen `A` und `E`. Die Data-Race-Warnungen zwischen `A` und `C` können hingegen nicht aufgrund des hier analysierten Musters ausgeschlossen werden, da die Prädikate, die sie schützen, identisch sind.

Die schreibenden Zugriffe, selbst der durch Zuweisung der Wertes `UNKNOWN` abstrahierte, stören das Muster nicht, da sie im niederprioren Task an geeigneter Stelle stattfinden. Auch durch die Zuweisung des Wertes `TIMEOUT` wird ein geschützter Bereich eröffnet.

Alle verwendeten Refinement-Checker bearbeiten alle zu diesem Muster gehörenden Dateien mit einem Zeitbudget von 2 Sekunden erfolgreich. FDR2, der gerade bei »einfachen« Aufgaben besonders schnell ist, benötigt für keine der Dateien mehr als 50 Millisekunden.

```
#define STATUS_TIMEOUT 3
#define STATUS_INITIAL 0

volatile int hw_loop_status = STATUS_INITIAL;

void task_high(void) {
   if (hw_loop_status != STATUS_TIMEOUT)
      ACC; /* A */
   else
      ACC; /* B */
}

void task_low(void) {
   if (hw_loop_status != STATUS_TIMEOUT)
      ACC; /* C */
   else
      ACC; /* D */

   if (IND)
      hw_loop_status = UNKNOWN;

   if (IND) {
      hw_loop_status = STATUS_TIMEOUT;
      ACC; /* E */
   }
}
```

Abbildung 16.3: Ein einfaches Muster aus dem Testkorpus

16.3.2 Ungewöhnliches Muster

Ein relativ ungewöhnliches Muster ist in Abbildung 16.4 dargestellt. Auch diesem Muster folgt eine der Instanzen expliziter Zustandsverwaltung aus dem Testkorpus. Über die Makros erhalten die Werte 1 und 25 die Namen `SHORT_PERIOD` und `LONG_PERIOD`. Diese werden in einem Hintergrundtask mit sehr niedriger Priorität der Zustandsvariablen zugewiesen.

Zusätzlich wird aber, wiederum in einem Task niedriger Priorität, die Variable dekrementiert. Der Wertebereich ist somit nicht auf `SHORT_PERIOD`

```
#define BASE_VALUE 100
#define DIVISOR_LONG 4
#define DIVISOR_SHORT 100
#define SHORT_PERIOD (BASE_VALUE / DIVISOR_SHORT)
#define LONG_PERIOD (BASE_VALUE / DIVISOR_LONG)

volatile unsigned int ignore_message = 0;

void irq_highest(void) {
   if (!ignore_message)
      ACC; /* A */
   else
      ACC; /* B */
}
void task_high(void) {
   if (!ignore_message)
      ACC; /* C */
   else
      ACC; /* D */
}
void task_medium(void) {
   if (!ignore_message)
      ACC; /* E */
   else
      ACC; /* F */
}
void task_low(void) {
   if (!ignore_message)
      ACC; /* G */
   else {
      ACC; /* H */
      ignore_message--;
      ACC; /* I */
   }
}
void task_background(void) {
   if (IND)
      ignore_message = SHORT_PERIOD;
   if (IND)
      ignore_message = LONG_PERIOD;
}
```

Abbildung 16.4: Ein ungewöhnliches Muster aus dem Testkorpus

und LONG_PERIOD beschränkt. Dazu passt auch, dass die Zustandsvariable mehrfach mit dem Wert 0 verglichen wird[1].

Mehrere relativ hochpriore Tasks erledigen nun unterschiedliche Arbeitspakete in Abhängigkeit vom Wert der Zustandsvariablen. Die möglicherweise unter gegenseitigem Ausschluss stattfindenden Quelltextabschnitte sind in der Abbildung als ACC; dargestellt.

Was ist wohl der Sinn dieses Musters? Eine recht anschauliche Erklärung ist, dass der Hintergrundtask die Zustandsvariable auf Werte größer 0 setzt, um zu bewirken, dass ein bestimmtes Signal in der näheren Zukunft ignoriert wird. Der periodische Task task_low wird die Zustandsvariable immer dann dekrementieren, wenn ihr Wert ungleich 0 ist. Das Setzen der Zustandsvariablen auf den Wert LONG_PERIOD bewirkt daher, dass die Zustandsvariable ungefähr für die 25-fache Periode des Tasks task_low einen Wert größer 0 hat (es sei denn, der Hintergrundtask setzt sie zwischenzeitlich auf den Wert SHORT_PERIOD).

Eine genaue Betrachtung ergibt, dass der Wert der Zustandsvariablen 0 nie unter- und 25 nie überschreitet. Alle Werte dazwischen sind hingegen erreichbar. Man kann das Muster somit als endlichen Automaten mit 26 Zuständen ansehen.

Das Erstaunliche an dem Muster ist, dass der im Rahmen dieser Arbeit erstellte Ansatz die Synchronisationseigenschaften erfasst, obwohl die Zustandsübergänge zum Teil durch approximierte Zuweisungen[2] erfolgen (gemeint ist die Dekrementierung). Sowohl das reale Muster als auch das hier dargestellte Modell wurden in den verwendeten Ansatz eingegeben. Tabelle 16.6 zeigt das in beiden Fällen identische Ergebnis.

Data-Race-Warnungen, an denen der Bereich I beteiligt ist, werden nicht ausgeschlossen. Die dem Bereich vorausgehende Zuweisung weist einen unbekannten Wert zu, sodass im Bereich I nichts über den Wert der Zustandsvariablen bekannt ist. Diese Tatsache ist auch richtig und keine Überapproximierung. Auch mit manueller Analyse ist festzustellen, dass beim

[1] Der Ausdruck »if (!ignore_message) [...]« ist in C gleichbedeutend mit »if (ignore_message == 0) [...]«.

[2] Die Zuweisung des unbestimmten Wertes UNKNOWN.

Tabelle 16.6: Ergebnisse des vorgestellten Ansatzes zu den Synchronisationseigenschaften des Musters aus Abbildung 16.4

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
A		—	—	—	—	—	—	—	—
B	=		—	—	—	—	—	—	—
C	✗	✓		—	—	—	—	—	—
D	✓	✗	=		—	—	—	—	—
E	✗	✓	✗	✓		—	—	—	—
F	✓	✗	✓	✗	=		—	—	—
G	✗	✓	✗	✓	✗	✓		—	—
H	✓	✗	✓	✗	✓	✗	=		—
I	✗	✗	✗	✗	✗	✗	=	=	

✓: Ausschluss
✗: Kein Ausschluss
=: Nicht geprüft, da Zugriffe aus demselbem Task stammen und daher keine Data-Race-Warnung erzeugen.

Erreichen des Bereichs `I` der Wert der Zustandsvariablen sowohl gleich als auch ungleich 0 sein kann.

Das Muster bereitet den Refinement-Checkern etwas mehr Mühe. Erst mit einem Zeitbudget von 50 Sekunden werden alle Data-Race-Warnungen von allen Refinement-Checkern erfolgreich bearbeitet.

16.3.3 Explizite Zustandsverwaltungen, die nicht synchronisieren

Dieser Abschnitt soll nicht suggerieren, dass alle Muster, die einen endlichen Automaten realisieren, zwangsläufig auch Synchronisationseffekte haben. Bei der Untersuchung der Systeme wurden auch Instanzen expliziter Zustandsverwaltung entdeckt, denen keine Synchronisationseigenschaften zugeschrieben werden können. Eigenschaften, die dies verursachen, sind zum Beispiel Schreibzugriffe in hochprioren Tasks, Prozeduren, die aus mehreren Tasks aufgerufen werden, und Zustandswechsel durch In- oder Dekrementieren.

16.3.3.1 Schreibzugriffe in hochprioren Tasks

Mehrfach wurde beobachtet, dass Tasks mit sehr hoher Priorität (zum Beispiel Interrupts), die fast alle oder sogar alle anderen Tasks unterbrechen können, auf Zustandsvariablen schreibend zugreifen, ohne dass dies durch Pfadprädikate stark beschränkt wird. Wenn ein solcher Schreibzugriff einen vom Ansatz dieser Abhandlung nicht erkannten Wert hat, also zu einem `UNKNOWN`-Write abstrahiert wird, nimmt dies der jeweiligen Instanz expliziter Zustandsverwaltung oftmals alle Synchronisationseigenschaften. Die Zustandsvariable kann dann fast immer einen beliebigen Wert haben, sich in fast jedem Zustand spontan ändern, und es ist nahezu unmöglich, noch einen Ausschluss zu beweisen.

Aber auch wenn der hochpriore Task nur statisch bekannte Werte zuweist, ist die Synchronisation oft schwer gestört. Ein typisches Verhalten ist, dass ein Zustandsautomat durch einen Interrupt in bestimmten Konstellationen auf den Startzustand zurückgesetzt wird – etwa um einen gerade ablaufenden Vorgang abzubrechen. Einen laufenden Vorgang abzubrechen mag zunächst harmlos erscheinen. Oft ermöglicht ein solches Abbrechen aber, sofort einen neuen Durchlauf im Automaten zu beginnen. Zwischen Tasks, die dann noch vom alten Wert ausgehen, und solchen, die schon im neuen Durchlauf angekommen sind, lassen sich dann oft Data-Race-Situationen finden. Pfade zu diesen Data-Races benötigen zwar teils zahlreiche Unterbrechungen zu genau ausgewählten Zeitpunkten; die Pfade werden aber von den Refinement-Checkern dennoch schnell gefunden. Schreibzugriffe in hochprioren Tasks erhöhen die Zahl der Kanten im endlichen Automaten teils merklich.

16.3.3.2 Prozeduren, die aus mehreren Tasks heraus aufgerufen werden

Ein weiteres Muster, das meistens keine Synchronisation liefert, besteht aus Prozeduren, die aus mehreren Tasks heraus aufgerufen werden. Steht in diesen ein Prädikat, das von einer Zustandsvariable abhängt, sieht es zunächst so aus, als seien hier viele Data-Race-Warnungen auszuschließen.

Dies ist aber aus einem einfachen Grund oft nicht so. Schließlich führen

```
1  #define STATE_OFF       0
2  #define STATE_PHASE_IN  1
3  #define STATE_ACTIVE    2
4
5  volatile unsigned int cmp_state;
6
7  void task_low(void) {
8     switch(cmp_state) {
9        case STATE_OFF:
10          /* ... */
11          break;
12       case STATE_PHASE_IN:
13          if (IND) cmp_state++;
14          break;
15       case STATE_ACTIVE:
16          break;
17       default:
18          break;
19    }
20 }
```

Abbildung 16.5: Zustandswechsel durch Inkrementieren

alle Tasks dieselben Operationen aus, wenn die Zustandsvariable einen bestimmten Wert hat. Wenn die Bedingung zum Beispiel »`if (zv == 3)`« lautet und im Then-Zweig auf eine globale Variable zugegriffen wird, greifen alle anderen Tasks ebenfalls im Zustand 3 auf diese Variable zu. Diese Zugriffe werden also zunächst nicht durch das Muster synchronisiert.

16.3.3.3 Zustandswechsel durch In- oder Dekrementieren

Vereinzelt werden Zustandsvariablen in realen Systemen durch Inkrementieren verändert. In Abbildung 16.5 ist ein solches Verhalten dargestellt.

Statt direkt einen neuen Wert zuzuweisen, wird die Zustandsvariable inkrementiert. Der Entwickler geht hier wohl davon aus, dass der Wert ohnehin feststeht und das der Ausdruck »`cmp_state++`« die gleiche Wirkung hat wie »`cmp_state = STATE_ACTIVE`«.

Dieses Vorgehen kann problematisch sein. Zum einen kann es zu unerwarteten Zustandsübergängen durch Race-Conditions kommen, wenn nebenläufig ausgeführte Tasks ebenfalls die Zustandsvariable verändern – dies ist im betrachteten Muster jedoch nicht der Fall. Zum anderen könnten Softwarefehler entstehen, wenn die Nummerierung der Zustände in der Annahme verändert wird, dass dies keine Auswirkungen hat.

Für den Ansatz dieser Abhandlung ist dieses Vorgehen auf jeden Fall problematisch – schließlich wird die Zuweisung durch eine `UNKNOWN`-Zuweisung abstrahiert. Je nachdem, wie das Muster also konkret ausgestaltet ist, kann es tatsächlich ungeeignet zur Synchronisation sein, dem Ansatz nur ungeeignet erscheinen oder auch erkennbar geeignet zur Synchronisation sein.

16.4 Erfolgreiche Synchronisation im Testkorpus

Es stellt sich die Frage, ob die untersuchten Instanzen expliziter Zustandsverwaltung aus dem Testkorpus nun tatsächlich synchronisieren. In Tabelle 16.7 sind dazu einige Zahlen aufgeführt.

Im oberen Teil der Tabelle zeigt sich, dass im Testkorpus grob ein Drittel

Tabelle 16.7: Übersicht über Ergebnisse, aufgeschlüsselt nach Antworten

		Zeitbudget 10 s		**Zeitbudget 5000 s**	
		#	**Quote**	**#**	**Quote**
Eingaben	**Lösbar**	207	79 %	234	90 %
	\ Ausschluss	65	25 %	70	27 %
	\ Erreichbar	142	54 %	164	63 %
	Timeout	54	21 %	27	10 %
Instanzen	**Alle lösbar**	9	60 %	11	73 %
	\ Alle ausgeschlossen	0	0 %	1	7 %
	\ Teils/teils	7	47 %	7	47 %
	\ Alle erreichbar	2	13 %	3	20 %
	Teilweise lösbar	6	40 %	4	27 %
	Keine lösbar	0	0 %	0	0 %

Die Prozentzahlen sind gerundet. Die Ergebnisse stammen von FDR4.

Tabelle 16.8: Ergebnisse, aufgeschlüsselt nach Instanzen und Antworten

Instanz	Zahl der Eingaben	Ergebnis Ausschluss	Ergebnis Erreichbar	Ergebnis Timeout
1	18	18	—	—
2	12	4	4	4
3	28	10	18	—
4	14	4	10	—
5	14	4	10	—
6	12	9	3	—
7	14	—	14	—
8	8	2	6	—
9	19	—	16	3
10	24	—	12	12
11	16	4	4	8
12	30	12	18	—
13	19	—	19	—
14	27	—	27	—
15	6	3	3	—
Summe	261	70	164	27

Die Ergebnisse stammen von FDR4 mit einem Zeitbudget von 5000 Sekunden.

der erfolgreich geprüften Data-Race-Warnungen ausgeschlossen werden kann. Dies gilt sowohl für ein Zeitbudget von 10 Sekunden als auch für ein Zeitbudget von 5000 Sekunden.

Für den unteren Teil der Tabelle wurden die einzelnen Eingaben nach ihren jeweiligen Instanzen expliziter Zustandsverwaltung zusammengefasst. Schon bei einem knappen Zeitbudget von 10 Sekunden gibt es keine Instanz expliziter Zustandsverwaltung, bei der alle Anfragen unbeantwortet bleiben.

Zum Teil zeigten sich dann – vor allem bei der zweiten Kategorie – auch überraschende Ergebnisse. Dass bei manueller Analyse leicht Fehler gemacht werden, ist in Wittiger und Felden [WF15] eindrücklich dargestellt. Bei den Mustern, von denen Synchronisation erwartet wurde, wurden die Grenzen in derselben Weise wie beim Muster aus Abbildung 16.4 ausführlich ausgetestet.

Dies erklärt die hohe Zahl an Mustern, bei denen nur ein Teil der Data-Race-Warnung zurückgewiesen wurde.

Insgesamt konnten bei 10 der 15 untersuchten Instanzen expliziter Zustandsverwaltung Synchronisationseffekte nachgewiesen werden. Dazu gehören die 8 Instanzen, die in Tabelle 16.7 bei einem Zeitbudget von 5000 Sekunden als »Alle ausgeschlossen« beziehungsweise als »Teils/teils« aufgeführt sind, als auch 2 der 4 Instanzen, die unter »Teilweise lösbar« aufgeführt sind. In Tabelle 16.8 sind Ergebnisse zu allen 15 untersuchten Instanzen dargestellt. Auch anhand dieser Tabelle lassen sich die in diesem Abschnitt genannten Zahlen nachvollziehen.

Auch aufgrund der begrenzten Sicht auf die Systeme ist nur schwer zu beantworten, ob die untersuchten Instanzen expliziter Zustandsverwaltung zur Synchronisation gedacht sind oder ob die Synchronisation nur ein nützlicher Nebeneffekt ist. Diese Frage ist aber weitgehend unabhängig von der Frage nach deren Funktionsweise.

Die Ergebnisse der Tabellen 16.7 und 16.8 lassen sich wie folgt interpretieren:

- Das Ergebnis Ausschluss bedeutet, dass die untersuchte Data-Race-Warnung falsch positiv ist. Es handelt sich bei dem Zugriffspaar also sicher um kein Data-Race. Außerdem wurde »die Richtige« Zustandsvariable ausgewählt.
- Aus Ergebnis Timeout lässt sich weder eine Aussage über die Data-Race-Warnung herleiten, noch über die Auswahl der Zustandsvariable.
- Das Ergebnis Erreichbar bedeutet, dass das Data-Race im Modell tatsächlich erreichbar ist. Dann trifft mindestens eine der drei folgenden Aussagen zu:
  - Die Warnung ist richtig positiv. Es handelt sich bei dem Zugriffspaar um ein tatsächliches Data-Race.
  - Es wurde eine »falsche« Zustandsvariable ausgewählt. Das heißt es wäre möglich, dass die Auswahl einer anderen Zustandsvariable zum Ergebnis Ausschluss geführt hätte.

– Das Data-Race ist im Modell durch eine konservative Approximation erreichbar geworden.

Kapitel 17

Conclusio und Ausblick

Dieses Kapitel fasst die Erkenntnisse und Ergebnisse dieser Abhandlung zusammen. Es zeigt Möglichkeiten und Grenzen der Übertragbarkeit der Ergebnisse auf. Im Ausblick wird dargestellt, welche weiterführenden Untersuchungen in der Zukunft durchgeführt werden könnten.

17.1 Erkenntnisse und Ergebnisse

Der Inhalt dieser Abhandlung lässt sich in drei Teile gliedern: die Entwicklung eines Ansatzes zur Analyse von expliziter Zustandsverwaltung, die daraus folgende Implementierung einer Werkzeugkette und die Evaluation bezüglich Mächtigkeit und Praktikabilität.

17.1.1 Ansatz

Es wurde ein schwergewichtiger Ansatz zur Analyse von expliziter Zustandsverwaltung als Mittel der Synchronisation vorgestellt. Der Ansatz reduziert eingebettete Systeme aus Automobilen mittels statischer Analyse zu relativ kleinen Modellen. Die Modelle werden in das formale Prozesskalkül CSP_M

übersetzt. Nach einer Kompression werden diese mit Refinement-Checkern bearbeitet, um die Synchronisationseigenschaften der Muster zu ermitteln. Dabei werden einzelne Data-Race-Warnungen als erreichbar oder ausgeschlossen bewertet. Der Ansatz erhält die Konservativität der zugrundeliegenden Data-Race-Analyse.

Eine Grundidee des Ansatzes ist, bei der Modellerzeugung und den statischen Analysen strenge Anforderungen an Laufzeit und Komplexität zu stellen, da ein Scheitern hier ein ganzes System unanalysierbar macht. Bei der Bearbeitung der einzelnen Data-Race-Warnungen hingegen werden die Anforderungen drastisch reduziert, um mächtige, aber komplexe Techniken wie das Refinement-Checking zu erlauben.

17.1.2 Implementierung

Der Ansatz der vorliegenden Abhandlung wurde implementiert. Dabei sind drei Werkzeuge entstanden:

- Das Werkzeug **Red**, das statische Analysen ausführt und Systeme zu Modellen reduziert. Das Werkzeug integriert dabei die Ergebnisse des Bauhaus Data Race Detector in das Modell.
- Das Werkzeug **Red2CSP**, das die Modelle in das formale Prozesskalkül CSP_M übersetzt und geeignete Prüfbedingungen erstellt.
- Das Werkzeug **CSPC**, das die Modelle komprimiert und somit den Refinement-Checkern erlaubt, diese besser oder überhaupt erst zu verarbeiten.

Die wichtigsten Erkenntnisse der Implementierung sind:

- Bekannte statische Analysen sind gut geeignet, um die für die Modellerstellung relevanten Informationen zu erheben. Die vorhandenen Zeigeranalysen lieferten hinreichend präzise Ergebnisse – auch wenn bei Zeigeranalysen zusätzliche Präzision stets hilfreich ist. Die Implementierung einer Konstantenpropagierung und -faltung nach bekannten Konzepten war allenfalls aufwendig und nicht konzeptionell

schwierig. Technische Schwierigkeiten bestanden hauptsächlich darin, eine Vielzahl an unterschiedlichen Schnittstellen von unterschiedlichen Entwicklern zu bedienen.

- Refinement-Checker haben Schwierigkeiten mit verbosen Modellen. Mit relativ einfachen Mitteln lassen sich die von Red2CSP erstellten CSP_M-Modelle jedoch stark komprimieren. Die Komprimierung verbessert das Verhalten der Refinement-Checker erheblich.

17.1.3 Evaluation der Praktikabilität

Um die Praktikabilität der Implementierung (und damit des Ansatzes der vorliegenden Abhandlung) zu evaluieren, wurde ein Testkorpus aus Eingaben erstellt. Die Eingaben stammen aus realen Systemen. Der Testkorpus wurde daraufhin untersucht, ob zu erwartende Schwierigkeiten in angemessener Weise repräsentiert sind.

Primäres Evaluationsmaß war die Laufzeit auf einem praktisch verfügbaren System, siehe Tabelle 16.3 (Seite 271). Die wichtigsten Erkenntnisse dieses Evaluationsteils sind:

- Die Prozessschritte, die für jedes System nur einmal ausgeführt werden müssen, vor allem also die statische Analyse, terminieren zuverlässig und haben auch bei Anwendung auf reale Systeme eine kurze Laufzeit. Dies entspricht der technischen Zielsetzung.

  Selbst die Prozessschritte, die für jede einzelne Data-Race-Warnung einmal ausgeführt werden müssen, terminieren – mit Ausnahme der Bearbeitung durch die Refinement-Checker – zuverlässig und schnell.

- Die besten getesteten Refinement-Checker erreichen mit einem Zeitbudget, das einem interaktiven Einsatz entspricht, bei Anwendung auf den Testkorpus in etwa drei von vier Fällen ein Ergebnis. Mit einem längeren Zeitbudget steigt die Erfolgsquote auf fast neun von zehn an. Damit ist gezeigt, dass der Ansatz trotz seines Einsatzes komplexer Technologien praxistauglich sein kann.

- Dies gilt aber nur nach Vorverarbeitung mit dem Werkzeug CSPC. Ohne diese terminieren die Refinement-Checker kaum. Dies steht im Kontrast zur Anwendung auf kleine Systeme, bei denen eine Vorverarbeitung nicht notwendig war.

17.1.4 Evaluation Mächtigkeit

Um die Mächtigkeit des Ansatzes dieser Abhandlung zu evaluieren, wurde die geschaffene Werkzeugkette an Mustern aus relativ kleinen selbst erstellten Programmen getestet. Außerdem wurde der Ansatz mit den Ansätzen von Keul und Schwarz et al. verglichen. Dabei wurde er auch an Beispielen dieser Autoren getestet. Weiterhin wurden kleine Beispiele erzeugt, die die Ansätze an ihre Grenzen bringen. Erkenntnisse der Untersuchungen sind:

- Es wurde gezeigt, dass der Ansatz dieser Abhandlung verzahnte Zustandsvariablen analysieren kann und dass er – in einem gewissen Sinne – vollständig pfad- und kontextsensitiv ist.
- Die Rolle, die die Präzision der Zeigeranalysen für die Analyseergebnisse der Synchronisationserkennung hat, wurde ergründet.
- Es ist gelungen zu zeigen, dass der Ansatz eine tiefere Analyse ermöglicht als die einfacheren Ansätze von Keul und Schwarz et al. Die bisherigen Ansätze erkennen bestimmte Muster, die hart eingebaut sind. Dabei prüfen sie die Bedingungen, die erfüllt sein müssen, nur teilweise und erlauben nur in begrenztem Maße syntaktische und semantische Abweichungen. Muster, die ihren festen Schemen widersprechen, können sie nicht sinnvoll analysieren. Der hier vorgestellte Ansatz liefert ein tieferes Verständnis der Synchronisationseigenschaften der untersuchten Muster. Er kann nötige Abstraktionen selbst herleiten und ist kaum auf Zusicherungen des Nutzers angewiesen. Wenn er auf Zugriffsmuster trifft, die nicht präzise modelliert werden können, kann die Analyse dennoch fortgesetzt werden – und durchaus auch mit Erfolg.

17.2 Übertragbarkeit der Ergebnisse

Die Ergebnisse dieser Abhandlung wurden an eingebetteten Systemen aus Automobilen erzielt. Sie sind daher zunächst spezifisch für die untersuchten Systeme. Die Implementierung geht fest von einem bestimmten Scheduling-Verfahren aus, das andere Systeme vielleicht nicht einsetzen, und die untersuchten realen Systeme stammen alle aus derselben Domäne. Die Ergebnisse entsprechen einem Existenzbeweis: Zumindest in einem bestimmten eng umrissenen Szenario ist die beschriebene Analyse möglich.

Jedenfalls ist der Ansatz der vorliegenden Abhandlung auf weitere Anwendungen übertragbar. Im Folgenden wird kurz skizziert, wie auf geänderte Eigenschaften der Systeme eingegangen werden kann.

17.2.1 Andere Schedulingverfahren

Wenn der Ansatz dieser Abhandlung auf Systeme aus anderen Domänen übertragen wird, zum Beispiel auf eingebettete Systeme aus dem Bereich Maschinenbau, muss er mit anderen Schedulingverfahren umgehen können. Auch in Automobilen dürften zum Teil andere Schedulingverfahren eingesetzt werden.

Das aktuell eingesetzte Schedulingverfahren ist auch nur auf Single-Core-Systemen sinnvoll anwendbar. Für Multi-Core-Systeme muss es erweitert werden.

In CSP_M stehen eine Vielzahl von Parallelitätsoperationen zur Verfügung. Insbesondere ist das Kalkül auch Turing-vollständig, sodass kaum zu bezweifeln ist, dass sich auch andere Schedulingverfahren modellieren lassen.

Wenn hierzu weitere Operatoren eingesetzt werden, müsste das Werkzeug CSPC angepasst werden. Andere Operatoren könnten andere Vereinfachungs- beziehungsweise Komprimierungsregeln benötigen.

Bei einem Wechsel der Schedulingverfahren wäre also eine Anpassung der Implementierung nötig. Der Ansatz an sich dürfte sich aber übertragen lassen. Es ist im Allgemeinen schwer zu beantworten, wie sich die Änderungen auf die Laufzeit der Refinement-Checker auswirken.

17.2.2 Andere verzahnte Synchronisationsmechanismen

Wenn der Ansatz dieser Abhandlung auf Systeme angewendet wird, die zustandsbasierte Synchronisation mit herkömmlichen Mechanismen wie globale Interruptdeaktivierung oder Mutexes verzahnen, müssten diese im Modell mit erfasst werden. Wie dies bei Mutexen gelingen kann, ist in Abschnitt 13.1 (Seite 221) beschrieben. Globale Interruptdeaktivierung lässt sich einfach mit den bestehenden Zustandsvariablen modellieren. Eine Unterbrechung dürfte dann nur noch nach Prüfung dieser Variablen erfolgen.

Um die Synchronisationsmechanismen zu erkennen, müsste die zugrunde gelegte Data-Race-Analyse angepasst werden. Ferner müsste die Modellgenerierung angepasst werden. Wenn aber keine neuen CSP_M-Konstrukte verwendet werden – wie bei der vorgeschlagenen Behandlung der Mutexe – müsste das Werkzeug CSPC nicht angepasst werden.

17.3 Ausblick

Die Ergebnisse dieser Abhandlung können Grundlage weiterer Forschung sein.

17.3.1 Rückkopplung in die statische Analyse

Ein mögliches Themenfeld ist die Rückkopplung in die statische Analyse. Ein denkbares Szenario ist eine Data-Race-Analyse, die an bestimmten neuralgischen Stellen einen Refinement-Checker danach fragt, ob nebenläufige Zugriffe ausschließbar sind. Lässt sich der Ausschluss zeigen, kann sie mit hoher Präzision weiterarbeiten. Ist er nicht nachzuweisen, muss sie, wie ohne Befragung der Refinement-Checker auch, konservativ approximieren. Durch Weiterverwendung der Ergebnisse könnte ihr Nutzen möglicherweise gesteigert werden.

17.3.2 Bessere Ausblendung irrelevanter Lesezugriffe

Die Implementierung der Analyse ließe sich auch technisch weiterentwickeln. Durch eine Data-Race-Warnungs-spezifische statische Analyse, die in der aktuellen Architektur nicht vorgesehen ist, könnten Lesezugriffe als irrelevant erkannt werden. Würden diese dann aus dem Modell entfernt, ließe sich vielleicht die Laufzeit verringern beziehungsweise die Erfolgsquote bei gleichbleibendem Zeitbudget erhöhen. Als irrelevant ließen sich Lesezugriffe einstufen, die weder auf einem Pfad zu einer Data-Race-Markierung noch zu einem Schreibzugriff liegen.

17.3.3 Kompressionsannotationen

Eine bessere Abstimmung der Modelle auf den verwendeten Refinement-Checker könnte die Performanz verbessern. Hier könnte insbesondere an einer automatisierten Annotation geforscht werden, die FDR zu Kompressionen an erfolgversprechenden Stellen bringt. Mit einfachen Lösungen dürfte hier aber kaum größerer Nutzen zu erzielen sein. Erste eigene Experimente des Autors an realen Systemen haben jedenfalls keine Performanzverbesserungen gezeigt.

17.3.4 Weitere Einsatzmöglichkeiten der Komprimierung

Der als Nebeneffekt im Rahmen der Abhandlung an dieser Abhandlung entstandene einfache Komprimierungsansatz zur Vorverarbeitung von CSP_M könnte weiterentwickelt werden. Manche seiner Strategien und Grundgedanken könnten vielleicht auch von anderen Werkzeugen genutzt werden, wenn deren Ausgaben ohne manuelle Eingriffe von den Refinement-Checkern nicht sinnvoll bearbeitet werden können. Es könnte auch untersucht werden, ob sich manche der Strategien, die das Werkzeug CSPC einsetzt, in Refinement-Checker integrieren lassen. Dabei wäre zu ergründen, ob sich auch bei nicht von dem Werkzeug Red2CSP generierten Eingaben ein Performanzgewinn zeigt.

Teil IV

Apparat

Literaturverzeichnis

[BG16] Ben Blum und Garth Gibson. „Stateless Model Checking with Data-Race Preemption Points“. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016, S. 477–493 (Zitiert auf der Seite 40).

[Boe11] Hans-Jürgen Boehm. „How to Miscompile Programs with "Benign" Data Races“. In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*. 2011 (Zitiert auf den Seiten 40, 48, 49, 75).

[BW01] Alan Burns und Andrew J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. 3rd edition. Addison-Wesley, 2001 (Zitiert auf den Seiten 65, 66, 69–71, 227).

[CEK+00] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang, Jean-François Girard und Martin Würthner. „Data Exchange in Bauhaus“. In: *7th Working Conference on Reverse Engineering*. Hrsg. von Bob Werner. IEEE Computer Society, 2000, S. 293–295 (Zitiert auf der Seite 52).

[CKL04] Edmund Clarke, Daniel Kroening und Flavio Lerda. „A Tool for Checking ANSI-C Programs“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Hrsg. von Kurt Jensen und Andreas Podelski. Bd. 2988. Lecture Notes in Computer Science. Springer, 2004, S. 168–176 (Zitiert auf der Seite 55).

[DIN05] DIN. *DIN EN ISO 9000/Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2005)*. 1.12.2005 (Zitiert auf der Seite 41).

[DKW08] Vijay D'Silva, Daniel Kroening und Georg Weissenbacher. „A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27.7 (2008), S. 1165–1178 (Zitiert auf den Seiten 59, 60, 118).

[DW13] Sandro Degiorgi und Martin Wittiger. „Ergebnisbewertung konservativer statischer Data-Race-Analysen". In: *15. Workshop Software-Reengineering (WSR)*. 2013 (Zitiert auf den Seiten 75, 76, 78, 306).

[EA03] Dawson Engler und Ken Ashcraft. „RacerX: Effective, Static Detection of Race Conditions and Deadlocks". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. 2003, S. 237–252 (Zitiert auf der Seite 51).

[Ess02] Robert Esser. *The Adelaide Refinement Checker*. 2002. URL: `http://cs.adelaide.edu.au/~esser/arc.html` (Zitiert auf der Seite 103).

[FG13] Timm Felden und Torsten Görg. „Werkzeugunterstützte Eliminierung von Data-Races in Eclipse". In: *15. Workshop Software-Reengineering (WSR)*. 2013 (Zitiert auf den Seiten 75, 78, 79).

[FO10] Formal Systems (Europe) und Oxford University, Hrsg. *Failures-Divergence Refinement: FDR2 User Manual*. 2010 (Zitiert auf den Seiten 102, 109, 178, 182, 187, 188, 234).

[FW07] Leo Freitas und Jim Woodcock. „FDR Explorer". In: *Electronic Notes in Theoretical Computer Science* 187 (2007), S. 19–34 (Zitiert auf der Seite 104).

[FW16] Timm Felden und Martin Wittiger. „Migrating Bauhaus from IML to SKilL". In: *18. Workshop Software-Reengineering und -Evolution (WSRE)*. 2016 (Zitiert auf den Seiten 123, 306).

[GABR13] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov und Andrew William Roscoe. *Failures Divergences Refinement (FDR) Version 3*. 2013. URL: `https://www.cs.ox.ac.uk/projects/fdr/` (Zitiert auf der Seite 102).

[GABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov und Andrew William Roscoe. „FDR3 – A Modern Refinement Checker for CSP“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Erika Ábrahám und Klaus Havelund. Bd. 8413. Lecture Notes in Computer Science. 2014, S. 187–201 (Zitiert auf den Seiten 102, 191, 253).

[GPR11] Ashutosh Gupta, Corneliu Popeea und Andrey Rybalchenko. „Threader: A Constraint-based Verifier for Multi-Threaded Programs“. In: *Computer Aided Verification*. Hrsg. von Ganesh Gopalakrishnan und Shaz Qadeer. Springer, 2011, S. 412–417 (Zitiert auf der Seite 58).

[Hoa04] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2004 (Zitiert auf den Seiten 96, 101, 104, 105, 107, 160, 165, 166, 182, 199, 200, 204, 208).

[Hoa78] Charles Antony Richard Hoare. „Communicating Sequential Processes“. In: *Communications of the ACM* 21.8 (1978), S. 666–677 (Zitiert auf der Seite 95).

[ISO05] ISO/IEC. *Standard for Programming Languages – C*. 6.05.2005 (Zitiert auf den Seiten 42, 70).

[ISO11] ISO/IEC. *Standard for Programming Languages – C++*. 28.02.2011 (Zitiert auf der Seite 25).

[JKS+13] Ali Jannesari, Nico Koprowski, Jochen Schimmel, Felix Wolf und Walter Franz Tichy. „Detecting Correlation Violations and Data Races by Inferring Non-deterministic Reads“. In: *Proceedings of the 19th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Seoul, Korea*. IEEE Computer Society, 2013, S. 1–9 (Zitiert auf den Seiten 50, 217).

[JSR04] JSR 133 Expert Group (William Pugh, Sarita Adve und Doug Lea), Hrsg. *JSR 133: Java Memory Model and Thread Specification*. 12.04.2004 (Zitiert auf der Seite 26).

[JT10] Ali Jannesari und Walter Franz Tichy. „Identifying Ad-hoc Synchronization for Enhanced Race Detection“. In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. 2010, S. 1–10 (Zitiert auf der Seite 50).

[Keu10] Steffen Keul. „Static Versioning of Global State for Race Condition Detection“. In: *Reliable Software Technologiey – Ada-Europe 2010*. Hrsg. von Jorge Real und Vardanega Tullio. Bd. 6106. Lecture Notes in Computer Science. Springer, 2010 (Zitiert auf der Seite 54).

[Keu11] Steffen Keul. „Tuning Static Data Race Analysis for Automotive Control Software“. In: *11th IEEE International Working Conference on Source Code Analysis and Manipulation*. 2011, S. 45–54 (Zitiert auf den Seiten 24, 47, 51, 76, 82, 84, 86, 115, 120).

[Keu12] Steffen Keul. *Analyse von Synchronisationsmustern in Steuersoftware (unveröffentlichter Projektbericht)*. Hrsg. von Abteilung Programmiersprachen und Übersetzerbau des Instituts für Softwaretechnologie der Universität Stuttgart. Stuttgart, 2. Oktober 2012 (Zitiert auf den Seiten 86, 91, 92, 217, 218).

[KNFW15] Nikolaos Koutsopoulos, Mandy Northover, Timm Felden und Martin Wittiger. „Advancing Data Race Investigation and Classification Through Visualization“. In: *IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. Hrsg. von Jürgen Döllner, Fabian Beck und Alexandre Bergel. 2015, S. 200–204 (Zitiert auf den Seiten 75, 77, 79, 80, 306).

[KPG+10] Steffen Keul, Mikhail Prokharau, Daniel Gerlach, Carola Jenke und Aoun Raza. „RaceVis: Ein Werkzeug zur Visualisierung von Data Races“. In: *12. Workshop Software-Reengineering (WSR)*. 2010 (Zitiert auf den Seiten 75–78).

[KYS09] KyungHee Kim, Tuba Yavuz-Kahveci und Beverly A. Sanders. „Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking“. In: *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2009, S. 495–499 (Zitiert auf der Seite 59).

[Lan92] William Landi. „Undecidability of Static Analysis“. In: *ACM Letters on Programming Languages and Systems* 1.4 (1992), S. 323–337 (Zitiert auf der Seite 47).

[LBDL09] Gavin Lowe, Philippa Broadfoot, Christopher Dilloway und Mei Lin Hui. *Casper – A Compiler for the Analysis of Security Protocols: User Manual and Tutorial*. Hrsg. von Oxford University Computing Laboratory. 2009 (Zitiert auf den Seiten 61, 253).

[LF08] Michael Leuschel und Marc Fontaine. „Probing the Depths of CSP-M: A new FDR-Compliant Validation Tool“. In: *Formal Methods and Software Engineering*. Hrsg. von Shaoying Liu, Tom Maibaum und Keijiro Araki. Bd. 5256. Lecture Notes in Computer Science. 2008, S. 278–297 (Zitiert auf den Seiten 100, 103).

[LL07] Jochen Ludewig und Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. 1. Aufl. Heidelberg: Dpunkt-Verlag, 2007 (Zitiert auf den Seiten 41, 53).

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, Jose Nelson Amaral, Bor-Yuh Evan Chang, Samuel Zev Guyer, Uday P. Khedker, Anders Møller und Dimitrios Vardoulakis. „In Defense of Soundiness: A Manifesto“. In: *Commun. ACM* 58.2 (2015), S. 44–46 (Zitiert auf der Seite 54).

[Mir] Horiba Mira. *Misra Web site > What is Misra*. `https://www.misra.org.uk/MISRAHome/WhatisMISRA/tabid/66/Default.aspx` (Zitiert auf der Seite 63).

[MIR08] MIRA. *MISRA-C: 2004: Guidelines for the use of the C language in critical systems*. 2. Ausgabe, Technical Corrigendum 1. 2008 (Zitiert auf den Seiten 32, 64, 70, 71, 132, 137, 140, 142).

[NT13] Dirk Nowotka und Johannes Frederik Jesper Traub. „Formal Verification of Concurrent Embedded Software“. In: *Design, Analysis and Verification: 4th International Embedded Systems Symposium*. Hrsg. von Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro C. Zanella und Franz-Josef Rammig. Springer, 2013, S. 218–227 (Zitiert auf den Seiten 58, 74).

[OC03] Robert O'Callahan und Jong-Deok Choi. „Hybrid Dynamic Data Race Detection“. In: *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* 38.10 (2003), S. 167–178 (Zitiert auf der Seite 50).

[Pea95] Karl Pearson. „Note on Regression and Inheritance in the Case of Two Parents“. In: *Proceedings of the Royal Society of London* 58 (1895), S. 240–242 (Zitiert auf der Seite 274).

[PGK11] Mikhail Prokharau, Daniel Gerlach und Steffen Keul. „Static Analysis of Predicate-based Synchronisation“. In: *13. Workshop Software-Reengineering (WSR)*. 2011 (Zitiert auf der Seite 86).

[Plö02] Erhard Plödereder. „Codeanalysen“. In: *Software für sicherheitskritische Systeme*. Hrsg. von Jürgen Winkler, Peter Dencker und Hubert B. Keller. Bd. 2002. Softwaretechnik. Shaker, 2002, S. 79–126 (Zitiert auf der Seite 231).

[RG05] Ishai Rabinovitz und Orna Grumberg. „Bounded Model Checking of Concurrent Programs“. In: *Computer Aided Verification*. Hrsg. von David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Kousha Etessami und Sriram K. Rajamani. Springer, 2005, S. 82–97 (Zitiert auf den Seiten 56, 57).

[RGG+95] Andrew William Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson und John Brian Scattergood. „Hierarchical compression for model-checking CSP or How to check $\mathbf{10^{20}}$ dining philosophers for deadlock“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1995, S. 133–152 (Zitiert auf der Seite 182).

[Ric53] Henry Gordon Rice. „Classes of Recursively Enumerable Sets and Their Decision Problems“. In: *Transactions of the American Mathematical Society* 74.2 (1953), S. 358–366 (Zitiert auf der Seite 45).

[Ros11a] Andrew William Roscoe. *CSP Tools (Website des Buches »Understanding Concurrent Systems«)*. 2011. URL: https://www.cs.ox.ac.uk/ucs/CSPtools.html (Zitiert auf der Seite 103).

[Ros11b] Andrew William Roscoe. *Understanding Concurrent Systems*. London: Springer, 2011 (Zitiert auf den Seiten 60, 96, 109).

[RVP06] Aoun Raza, Gunther Vogel und Erhard Plödereder. „Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering“. In: *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies*. Springer, 2006, S. 71–82 (Zitiert auf den Seiten 52, 53, 74).

[SA11] Bryan Scattergood und Philip Armstrong. *CSP_M: A Reference Manual*. 2011 (Zitiert auf der Seite 101).

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro und Thomas Anderson. „Eraser: A Dynamic Data Race Detector for Multithreaded Programs“. In: *ACM Transactions on Computer Systems* 15.4 (1997), S. 391–411 (Zitiert auf den Seiten 24–26, 49, 50).

[Sch09] Uwe Schöning. *Theoretische Informatik – kurz gefasst*. 5. Auflage [Nachdruck]. HochschulTaschenbuch. Spektrum Akademischer Verlag, 2009 (Zitiert auf der Seite 109).

[Sco72] Dana Scott. „Continuous Lattices“. In: *Toposes, Algebraic Geometry and Logic: Dalhousie University, Halifax, January 16–19, 1971*. Hrsg. von Francis William Lawvere. Springer, 1972, S. 97–136 (Zitiert auf der Seite 165).

[SSVA14] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani und Kalmer Apinis. „Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs“. In: *Verification, Model Checking, and Abstract Interpretation*. Bd. 8318. Lecture Notes in Computer Science. Springer, 2014, S. 21–38 (Zitiert auf den Seiten 82, 88–90, 92–95, 115, 217, 218, 226, 236, 237, 243–252, 292).

[Tra16] Johannes Frederik Jesper Traub. „Formal Verification of Concurrent Embedded Software“. Diss. 2016 (Zitiert auf den Seiten 58, 75).

[VTD06] Mandana Vaziri, Frank Tip und Julian Dolby. „Associating Synchronization Constraints with Data in an Object-Oriented Language“. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2006, S. 334–345 (Zitiert auf den Seiten 26, 51, 52).

[WF15] Martin Wittiger und Timm Felden. „Recognition of Real-World State-Based Synchronization“. In: *17. Workshop Software-Reengineering und -Evolution (WSRE)*. 2015 (Zitiert auf den Seiten 221, 286, 306).

[Wit16] Martin Wittiger. „Eliminating Data Race Warnings Using CSP“. In: *Proceedings of the 21st Ada-Europe International Conference on Reliable Software Technologies*. Hrsg. von Marko Bertogna, Luís Miguel Pinho und Eduardo Quinoñes. Bd. 9695. Lecture Notes in Computer Science. 2016, S. 30–43 (Zitiert auf der Seite 306).

[WK12] Martin Wittiger und Steffen Keul. „Extraktion von Interthread-Kommunikation in eingebetteten Systemen“. In: *Automotive – Safety & Security 2012*. Hrsg. von Erhard Plödereder, Peter Dencker, Herbert Klenk, Hubert B. Keller und Silke Spitzer. Bd. P-210. Lecture Notes in Informatics. 2012, S. 55–67 (Zitiert auf den Seiten 74, 306).

Von den hier aufgelisteten Publikationen sind einige im Rahmen des Promotionsstudiums des Autors entstanden. Diese haben die Tags [DW13], [FW16], [KNFW15], [WK12], [WF15] und [Wit16]. Teile dieser Abhandlung wurden bereits in diesen Publikationen veröffentlicht.

Die URLs wurden zuletzt am 19. November 2017 geprüft.

Abbildungsverzeichnis

Tabellenverzeichnis

Mathematische Konventionen

In dieser Übersicht werden einige in dieser Abhandlung geltenden mathematischen Konventionen erläutert.

Landau-Symbole

Landau-Symbole werden entsprechend den folgenden Definitionen verwendet:

Big-O Eine Funktion $f : \mathbb{N} \longrightarrow \mathbb{N}$ liegt in $\mathcal{O}(g(n))$, geschrieben $f(n) \in \mathcal{O}(g(n))$, genau dann, wenn $\exists_{k \in \mathbb{N}} \exists_{m \in \mathbb{N}} \forall_{n \in \mathbb{N}} (n \geq m) \rightarrow f(n) \leq k \cdot g(n)$.

Soft-O Eine Funktion $f : \mathbb{N} \longrightarrow \mathbb{N}$ liegt in $\widetilde{\mathcal{O}}(g(n))$, geschrieben $f(n) \in \widetilde{\mathcal{O}}(g(n))$, genau dann, wenn $\exists_{q \in \mathbb{N}} f(n) \in \mathcal{O}(g(n) \cdot log^q(g(n)))$.

Beispiele

Sei eine Funktion f bestimmt durch $f : \mathbb{N} \longrightarrow \mathbb{N},\ x \longmapsto \lfloor 3x^2 \cdot ln(x^3 + 1) \rfloor$, dann gilt $f(n) \in \mathcal{O}(n^2 \cdot log(n))$, $f(n) \in \mathcal{O}(n^3)$ und $f(n) \in \widetilde{\mathcal{O}}(n^2)$.

Sei eine Funktion g bestimmt durch $g : \mathbb{N} \longrightarrow \mathbb{N},\ x \longmapsto (x^2 + x) \cdot 2^x$, dann gilt $g(n) \in \mathcal{O}(n^2 \cdot 2^n)$, $g(n) \in \widetilde{\mathcal{O}}(2^n)$, aber nicht $g(n) \in \mathcal{O}(2^n)$. Mithin erlaubt es die Soft-O-Konvention, Terme, die mindestens logarithmisch kleiner sind als der Hauptterm, zu vernachlässigen.

Mengen

Der Begriff Menge wird gemäß den üblichen Definitionen verwendet. Insbesondere gilt Folgendes:

Teilmenge A ist eine Teilmenge von B, geschrieben $A \subseteq B$ oder $A \subset B$, genau dann, wenn $\forall_{x \in A}\ x \in B$. A ist eine echte Teilmenge von B, geschrieben $A \subsetneq B$, genau dann, wenn A eine Teilmenge von B ist und $A \neq B$.

Obermenge A ist eine Obermenge von B, geschrieben $A \supseteq B$ oder $A \supset B$, genau dann, wenn B eine Teilmenge von A ist.

Klasse Sind die Elemente einer Menge selbst Mengen oder können als Mengen betrachtet werden, kann der Begriff Klasse den Begriff Menge ersetzen, um auf diesen Umstand hinzuweisen. Alle Begriffsbedeutungen bleiben dabei erhalten.

Tupel und CSP-Traces

Die Begriffe Tupel und Traces (im Bezug auf CSP) werden gemäß den üblichen Definitionen verwendet. Hier werden Darstellungsaspekte beschrieben.

Tupel Tupel werden mit spitzen Klammern und durch Kommas getrennte Komponenten geschrieben: $\langle A, B \rangle$ ist ein Tupel aus A und B.

CSP-Traces Bestehen die in den Traces vorkommenden Signale ausschließlich aus einzelnen Buchstaben, also zum Beispiel a oder b, werden Traces schlicht durch Konkatenation gebildet.

So ist abaaa ein aus fünf Signalen bestehender Trace. Der griechische Buchstabe ϵ steht für den leeren Trace.

Bestehen die Traces hingegen aus Signalen mit längeren Bezeichnern, werden die Traces in spitze Klammern eingeschlossen und die einzelnen Signale durch Punkte getrennt. Zur weiteren Abgrenzung gegenüber Tupeln wird ein Strich hinzugefügt. Traces sehen somit folgendermaßen aus:

- $\langle \underline{\mathtt{up} \cdot \mathtt{down} \cdot \mathtt{up} \cdot \mathtt{up} \cdot \mathtt{open} \cdot \mathtt{close}} \rangle$ (Länge 6)
- $\langle \underline{\mathtt{open}} \rangle$ (Länge 1)

Präfix-Abschlüsse Die Traces eines CSP_M-Prozesses sind stets bezüglich Präfix-Bildung abgeschlossen. (Ist ein Trace Element der Traces eines CSP_M-Prozesses, so sind auch alle seine Präfixe, einschließlich des leeren Präfixes, Element dieser Menge.) Daher wird die Operation $\langle\!\langle \_ \rangle\!\rangle^{\mathrm{Pref}}$ definiert. Diese steht für die Menge aller Präfixe des dargestellten Traces.

Damit gilt

$$\langle\!\langle \underline{\mathtt{up} \cdot \mathtt{down} \cdot \mathtt{open}} \rangle\!\rangle^{\mathrm{Pref}} = \{\epsilon, \langle \underline{\mathtt{up}} \rangle, \langle \underline{\mathtt{up} \cdot \mathtt{down}} \rangle, \langle \underline{\mathtt{up} \cdot \mathtt{down} \cdot \mathtt{open}} \rangle\}.$$

Glossar

τ-Transition Eine stille, interne Transition in einem generalisierten kantenbeschrifteten Transitionssystem, ähnlich einem ϵ-Übergang in einem nicht-deterministischen endlichen Automaten. 109, 111, 173, 176, 179, 181, 182, 234

Abstrakter Syntaxbaum (AST für Abstract Syntax Tree) Baumförmige Darstellung der abstrakten Syntax eines Programms. In ihr wurden unnötige Details der konkreten Syntax entfernt. 54, 131, 132, 135, 142, 195, 198, 200, 201, 211, 214, 266, 313

Aufrufgraph (auch Call-Graph) Datenstruktur, die Informationen zu Prozeduraufrufen enthält – im einfachsten Fall ein gerichteter Graph aus Prozeduren. 31, 54, 78, 79, 122, 127–130, 138, 148, 155, 166, 167, 277, 309

Bauhaus Eine Sammlung von Werkzeugen zur statischen Software-Analyse von Programmcode, entwickelt an der Universität Stuttgart. 52–54, 88, 120, 131, 137, 152, 185, 224, 227, 244, 320

Bauhaus Data Race Detector Ein Bauhaus-Werkzeug zur statischen Data-Race-Analyse. 27, 31, 54, 65, 66, 86, 120, 122, 127, 131, 135, 145, 156, 224, 227, 230, 231, 238, 239, 242, 266, 267, 290

Bauhaus Intermediate Language (IML) Zwischensprache und Dateiformat der Bauhaus-Analysewerkzeugsammlung 120, 121, 123, 128, 156

CBMC (C Bounded Model Checker) Ein Werkzeug für (Bounded) Model-Checking zur formalen Verifikation von C-Programmen. 55–57

CSP (Communicating Sequential Processes) Mathematischer Kalkül zur präzisen Modellierung von Nebenläufigkeit. 60, 95–100, 104, 105, 108, 124, 182, 204, 255, 268, 316, 320–323

CSP$_M$ (machine-readable CSP) Formalisierte Eingabesprache der Refinement-Checker. 61, 95, 96, 99–104, 108–110, 124, 125, 127, 157, 158, 161, 163, 164, 167, 168, 170, 172, 173, 184, 187, 189, 191–195, 198, 202–205, 207, 209, 213, 219, 223, 230, 234, 253–257, 260, 269, 275, 308–310, 317, 322, *siehe auch* CSP & Refinement-Checker

Data-Race Unerwünschtes Speicherzugriffsmuster in nebenläufigen Programmen. 23–27, 29, 30, 33–43, 45–52, 55, 57, 63, 68, 73–76, 78, 84, 85, 88, 90, 92, 93, 115, 117, 120, 122, 124, 145, 146, 150, 182, 185–187, 189, 218, 222, 223, 225, 228, 230, 234, 236, 238, 239, 241, 244–247, 249, 250, 275, 278, 283, 287, 288, 307, 310, 320

Data-Race-Analyse Statische Software-Analyse, die Data-Races in Programmen sucht. 27, 30, 31, 34, 35, 39, 40, 51–54, 66, 72–74, 79, 116, 119, 120, 122, 127, 131, 140, 217, 223, 294, 319

Data-Race-Problem Das (unentscheidbare) Data-Race-Problem beschreibt die Frage, ob ein gegebenes Programm ein Data-Race enthält. 46, 47

Data-Race-Warnung Ein Paar abstrakter Zugriffe, bei dem es sich laut statischer Software-Analyse möglicherweise um ein Data-Race handelt. 35, 40, 54, 73–81, 84, 86, 90, 115–117, 119–122, 127, 128, 138, 145, 150, 155, 157, 185, 188, 193, 222–225, 227, 230, 231, 233, 236, 238, 239, 242, 245, 251, 266–268, 274, 276, 278, 281–283, 286, 287, 290, 308, 314, 322

Datenflussanalyse Statische Software-Analyse, die Eigenschaften über den Kontrollflussgraphen hinweg propagiert. 53, 54, 127, 131, 133

entscheidbar (Antonym: unentscheidbar, Synonym: rekursiv) Berechenbarkeitstheoretischer Begriff: Ein Problem ist entscheidbar, wenn es von einem stets haltenden Programm erkannt wird. 45–47, 100, 124, *siehe auch* semi-entscheidbar

explizite Zustandsverwaltung Programmiermuster, mit dem der Systemzustand in expliziter Weise mittels Zustandsvariable und Zustandswerten verwaltet wird. 82–86, 88, 89, 93–95, 115–117, 121, 124, 131, 133, 134, 217–219, 237, 238, 244, 266–268, 278, 279, 282, 283, 285–287, 308, 309, 314, 323

Failures-Divergences-Modell Semantisches Modell für CSP; häufig genutzt zum Nachweis von Live-Lock-Freiheit (schlußendlich wird immer sichtbarer Fortschritt erzielt). 108, 109, 167

Failures-Modell (Auch Stable-Failures-Modell) Semantisches Modell für CSP; häufig genutzt zum Nachweis von Dead-Lock-Freiheit. 106, 108, 109, 167

FDR2 (Failures-Divergences Refinement 2) Ein CSP-Refinement-Checker, entwickelt an der Universität Oxford. 102–104, 109, 165, 166, 172–174, 176, 178, 180–182, 186–189, 191, 205, 223, 225, 233, 234, 253, 254, 257, 269, 272, 278, 309, 310

FDR3 (Failures-Divergences Refinement 3) Neuauflage des FDR2 Refinement-Checker entwickelt an der Universität Oxford. 102, 104, 109, 110, 165, 166, 176, 179, 182, 188, 191, 205, 253, 269–272, 274

generalisiertes kantenbeschriftetes Transitionssystem (generalized labelled transition system, Abkürzung *GLTS*) Graph, bestehend aus Knoten und mit Signalen oder dem Buchstaben τ beschrifteten Kanten. Kann als Verallgemeinerung von nicht-deterministischen endlichen Automaten gesehen werden. 104, 108–111, 166, 172–174, 176–181, 188, 189, 191, 269, 309, 310, 319

Interrupt-Service-Routine (ISR) Speziell gekennzeichnete Prozedur, zu der der Kontrollfluss transferiert wird, wenn ein Interrupt ausgelöst wurde. 42, 64, 66

Klassifizierung Prozess der manuellen Untersuchung und Bewertung von Data-Race-Warnungen. 75, 76, 78, 79, 81, 84, 115

LTL (Linear Temporal Logic) Eine temporale Logik, typischerweise basierend auf den unären Operatoren **X** (next), **G** (globally) und **F** (eventually) sowie den binären Operatoren **U** (until) und **W** (weak until). Oftmals dargestellt im Kontrast zu CTL: Computational Tree Logic. 103

Nebenläufigkeitskonfiguration Beschreibung der Tasks eines nebenläufigen Systems. Enthält die Startprozeduren von Tasks und deren Priorität. 116, 120, 121, 223, 238, 244

ProB Ein Refinement-Checker, entwickelt an der Universität Düsseldorf, der sowohl CSP als auch B als Eingabesprache akzeptiert. 103, 104, 270, 272, 274, *siehe auch* CSP & Refinement-Checker

Prüfbedingung Die Aufforderung an einen Refinement-Checker, eine bestimmte Eigenschaft eines CSP_M-Prozesses zu prüfen. In dieser Abhandlung meist eine Prüfung auf Traces-Verfeinerung der Form: »`assert SPEC [T= PROG`«. 187–189, 192, 193, 198, 202–204, 208, 213, 223, 256, 310, *siehe auch* CSP & Refinement-Checker

Race-Condition Eigenschaft eines Programms, nicht-deterministisch, beispielsweise je nach Timing, für den Anwender unterscheidbares Verhalten zu zeigen beziehungsweise für den Anwender unterscheidbare Ausgaben zu erzeugen. 41, 42, 48, 50

Refinement-Checker Werkzeug, das Eigenschaften von CSP_M-Prozessen beweisen oder widerlegen kann. 101–105, 109, 124, 125, 165, 176, 178, 182, 187, 191, 193, 196, 202, 205, 219, 223, 225–227, 231, 234, 235, 254, 257, 269–278, 282, 283, 311, 314, 321, 322

semi-entscheidbar (Synonym: rekursiv aufzählbar) Berechenbarkeitstheoretischer Begriff: Ein Problem ist semi-entscheidbar, wenn es von einem (möglicherweise nicht stets haltenden) Programm erkannt wird. 46, *siehe auch* entscheidbar

Traces-Modell Semantisches Modell für CSP; häufig genutzt zum Nachweis von Safety-Aspekten (Fehlerfall ist unerreichbar). 105, 106, 108, 109, 141, 167, 187, 192, 199, 202, 207, 208

zustandsbasierte Synchronisation Programmiermuster (Untermenge der expliziten Zustandsverwaltung), das zusätzlich zur Synchronisation des Systems beiträgt. 85, 86, 90–92, 223, 308, *siehe auch* explizite Zustandsverwaltung

Zustandsmenge Menge der Zustandswerte. Bestandteil des Programmiermusters explizite Zustandsverwaltung. 82, 90

Zustandsvariable Bestandteil explizite Zustandsverwaltung. In dieser globalen Variable werden Zustandswerte gespeichert. 82–85, 87–93, 117, 119, 123, 127, 134, 135, 139, 140, 142–145, 150, 154, 155, 167, 169, 171, 172, 174, 175, 178, 179, 181, 185–187, 221–227, 229, 230, 232, 234, 236, 239, 241, 242, 245–249, 252, 258, 262, 266–268, 278, 279, 281–285, 287, 309–311, 321

Zustandswert Mit Mitteln der Programmiersprache benannter Wert. Bestandteil des Programmiermusters explizite Zustandsverwaltung. 82–84, 87, 321, 323