UNIVERSITÄT ZU LÜBECK

Compiler Optimierung für strombasierte Sprachen

Compiler Optimization for stream-based languages

Bachelorarbeit

verfasst am
Institut für Software Engineering und Programmiersprachen

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Meiko Prilop

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Hannes Kallwies

Lübeck, den 1. Mai 2021

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Meiko Prilop

Zusammenfassung

Runtime Verifikation (RV) ist eine Möglichkeit Software auf ihre Korrektheit zu überprüfen. Ein zu beobachtendes System gibt in der RV Ausgaben in Form von Strömen aus, welche sich anhand einer vorher definierten Spezifikation verifizieren lassen. *Stream Runtime Verifikation* (SRV) erweitert dieses Prinzip um die Möglichkeit Funktionen auf diese Ströme anzuwenden, um neue Ströme zu erzeugen. In dieser Arbeit wird die SRV Sprache *TeSSLa* behandelt. TeSSLa ermöglicht es, Spezifikationen zu definieren und mithilfe eines Compilers *Monitore* auf Grundlage dieser Spezifikationen zu erzeugen. Da diese Systeme oft in einem zeitkritischen Kontext stehen, ist die zeitnahe Ausgabe von Ergebnissen des Monitors notwendig.

Innerhalb dieser Arbeit werden zwei Optimierungen des TeSSLa-Compilers untersucht. Dieser erzeugt aus einer TeSSLa-Spezifikation einen Monitor in der Programmiersprache Scala.
Die erste Optimierung besteht darin, Funktionen aus der TeSSLa Standart-Library direkt zu übersetzen, anstatt sie wie bisher in Verkettungen anderer TeSSLa-Funktionen, den sogenannten *TeSSLa-Core*-Funktionen, umzuwandeln. Die Implementierung einer *Domain Specific Language* (DSL) soll diese zielgerichtete Übersetzung ermöglichen.
Weiterhin verzichtete der TeSSLa-Compiler bisweilen auf die Verwendung von expliziten Konstanten im erzeugten Code, deren Verwendung im Vergleich zu der von Variablen ressourceneffizienter ist. Die zweite Optimierung soll mögliche Konstanten erkennen und dementsprechend übersetzen.

Es konnte gezeigt werden, dass die Implementierung weiterer Core-Funktionen die Laufzeit der Monitore um bis zu 90 % senkt. Die Analyse auf mögliche konstante Initialisierungen erbrachte hingegen keine Laufzeitverbesserungen. Beide Optimierungen zusammen konnten den Umfang der Monitore bis zu 72 % senken. Dadurch ließ sich auch ein Monitor ausführen, welcher vorher aufgrund seiner Länge in Programmzeilen nicht kompiliert werden konnte. Weiterhin resultieren die vorgenommenen Optimierungen allgemein in einer besseren Wart- und Lesbarkeit sowohl des TeSSLa-Compilers als auch der entstehenden Monitore.

Abstract

Runtime Verification (RV) is a way to verify software for correctness. A system to be observed outputs in RV in the form of streams, which can be verified against a previously defined specification. *Stream Runtime Verification* (SRV) extends this principle by allowing functions to be applied to these streams to generate new streams. In this work, we discuss the SRV language *TeSSLa*. TeSSLa allows specifications to be defined and *monitors* to be generated based on these specifications using a compiler. Since these systems are often in a time-critical context, the timely output of results from the monitor is necessary.

Within this thesis, two optimizations of the TeSSLa compiler are examined. This generates a monitor in the Scala programming language from a TeSSLa specification.
The first optimization consists of translating functions from the TeSSLa standard library directly, instead of converting them into concatenations of other TeSSLa functions, the so-called *TeSSLa-Core* functions, as was done before. The implementation of a *Domain Specific Language* (DSL) is supposed to enable this targeted translation. Furthermore, the TeSSLa compiler sometimes omitted the use of explicit constants in the generated code, whose use is more resource-efficient compared to that of variables. The second optimization is to detect possible constants and translate them accordingly.

It was shown that the implementation of additional core functions reduces the runtime of the monitors by up to 90 %. In contrast, the analysis for possible constant initializations did not yield any runtime improvements. Both optimizations together could reduce the size of the monitors by up to 72 %. Thus also a monitor could be executed, which could not be compiled before due to its length in program lines. Further the made optimizations result generally in a better maintainability and readability both of the TeSSLa compiler and the developing monitors.

Inhaltsverzeichnis

1

Einleitung

Software ist im Laufe des letzten Jahrhunderts zu einem festen Bestandteil des modernen Lebens geworden und aus dem Alltag nicht mehr wegzudenken. Der ständig fortschreitende Stand der Technik, besonders in sicherheitskritischen Bereichen wie beispielsweise der Luftfahrt oder der Medizin, weckt eine hohe Nachfrage an gut getesteter und verlässlicher Software. Aber auch in weniger sicherheitsrelevanten Softwareprojekten lassen sich hohe Kosten für die Wartung und Behebung auftretender Fehler feststellen [7]. Die Verwendung von Werkzeugen für das Testen und Verifizieren stellt ein probates Mittel zur effektiven Kosten- und Zeitersparnis dar.
Eine Möglichkeit, Systeme zu verifizieren, ist die *Runtime Verifikation* (RV) [10, 20]. Einem in Ausführung befindlichen Programm oder System werden Informationen extrahiert, um sein Verhalten zu analysieren und im Kontext einer Spezifikation auf Korrektheit zu überprüfen. Die Informationen dieses Systems liegen als sogenannte Ströme vor, was bedeutet, dass Informationen in zeitlichen Abständen von dem System ausgegeben werden und sich verändern können. Diese Ströme stellen die Eingaben einer Analyse der RV dar. Traditionell werden zu erfüllende Eigenschaften als logische Formeln spezifiziert, beispielsweise in linearer temporaler Logik. Zu einer gegebenen Spezifikation lässt sich ein Programm erzeugen, ein sogenannter *Monitor*, welcher den Lauf eines Systems überwacht und auf Verstöße dieser Spezifikation prüft. Die Analyse endet klassischerweise mit der Ausgabe eines Wahrheitswertes, je nachdem, ob ein Verstoß vorlag oder nicht.
Eine Erweiterung dieses Prinzips wird als *Stream Runtime Verifikation* (SRV) bezeichnet [9, 20]. SRV erweitert die RV um die Möglichkeit, auf Eingabeströme zu reagieren, diese zu verarbeiten und schließlich Ausgabeströme zu erzeugen. Anstatt eine Spezifikation nur auf erfüllt oder nicht erfüllt zu überprüfen, erlaubt es die SRV auch Messungen und weitergehende Berechnungen durchzuführen.
Notwendig für die Arbeit mit SRV sind Spezifikationssprachen wie *LOLA* [9]. LOLA ermöglicht das Definieren einer Spezifikation und das Erzeugen eines Monitors, um ein System anhand dieser Spezifikation zu prüfen. Sie ermöglicht jedoch nicht das Arbeiten mit asynchronen Strömen, deren Informationen in variablen Zeitabständen oder überhaupt nicht anliegen können. Im Gegensatz erlaubt es die 2017 am Institut für Programmiersprachen und Typsysteme entwickelte Spezifikationssprache *TeSSLa* [8, 20] auf Eingabeströme zu reagieren, auch wenn diese asynchron sind. Insbesondere lassen sich auf Grundlage vorausgegangener Events neue Events erzeugen. Mithilfe von TeSSLa lassen

sich Datenströme verarbeiten, live debuggen und unter Hinzunahme weiterer Werkzeuge visuell analysieren.

Folgende TeSSLa Spezifikation prüft beispielsweise, ob die vergangene Zeit zwischen zwei Lesezugriffen echt kleiner ist als drei. In diesem Fall wird ein Fehler ausgeben. Solch eine Spezifikation lässt sich in der SRV auf asynchronen Strömen definieren:

```
diff := time(read) - last(time(read), read)
error := filter(diff < 3, true)
```

In diesem Beispiel werden die Differenzen aufeinanderfolgender Zeitstempel des Eingabestroms *read* gebildet, um den Strom *diff* zu erstellen. *diff* wird daraufhin auf Differenzen echt kleiner drei gefiltert und sollte eine solche Differenz vorliegen, wird auf einem *error*-Strom ein *true* ausgegeben.

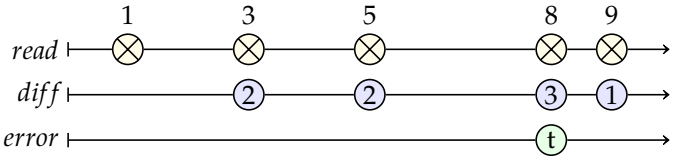

Abbildung 1.1: Visualisierung einer Spezifikation, bei der ein Fehler ausgegeben wird, sollte der nachfolgende Lesezugriff nach mehr als zwei Zeiteinheiten geschehen.

In der Praxis können Events sehr schnell hintereinander vom zu betrachtenden System ausgegeben werden. Durch das sich ständig verändernde Umfeld sind Ausgaben zeitkritisch. Veraltete Werte negieren die Aussagekraft der ausgegebenen Informationen und können eine erfolgreiche Verifikation des Systems verhindern.
Um Monitore auf Grundlage von TeSSLa-Spezifikationen erstellen und ausführen zu können, wurde ein Interpreter entworfen. Interpreter sind Systeme, welche Quelldateien zur Laufzeit in Maschinensprache übersetzen und ausführen. Sie sind die verbreitetste Art der Sprachenverarbeitung neben Compilern [2]. Da der Prozess des Interpretierens im Allgemeinen in längeren Laufzeiten resultiert, als dass es kompilierte Programme tun, wurde in [19] ein Compiler für die TeSSLa Programmiersprache vorgestellt.
Compiler übersetzen, ähnlich zu Interpretern, Quelldateien in Maschinensprache [2]. Der Unterschied ist, dass Compiler dies vor Ausführung des zu übersetzenden Programms tun, sodass ein Teil der Arbeit bereits vor Beginn der Laufzeit geleistet wird. Im Prozess des Kompilierens lassen sich weiterhin optional Optimierungen des entstehenden Programms durchführen.
Der Compiler von TeSSLa kann aktuell für eine TeSSLa-Spezifikation einen Monitor in der imperativen Programmiersprache Scala [25] erzeugen. Eine Spezifikation in TeSSLa wird dafür zunächst in eine gleich-mächtige Submenge aus neun Befehlen, dem sogenannte *TeSSLa-Core* oder *nativen* Funktionen reduziert und folgend in eine generische Zwischensprache übersetzt. Diese Zwischensprache, auch *Intermediate* Sprache genannt, ähnelt in ihrer Form einer imperativen Sprache, um so möglichst einfach in weitere imperative Sprachen übersetzt werden zu können. Nachdem die Spezifikation vollständig in diese Zwischensprache übersetzt wurde, wird der Monitor erstellt. Der Fokus der ent-

stehenden Monitore ist Effizienz, weshalb der Compiler einige Optimierungen auf der Intermediate Sprache durchführt.

Innerhalb dieser Arbeit werden zwei weitere mögliche Optimierungen dieses Compilers untersucht. Zum einen werden entstehende Scala-Programme optimiert, indem Konstanten (in Scala *val*) anstelle von Variablen (*var*) verwendet werden, sofern dies möglich ist. Die Verwendung von Konstanten ist im Allgemeinen effizienter und soll im Zuge dessen untersucht werden. Diese Analyse wird fortlaufend als *Var-Val-Analyse* bezeichnet.
Zum anderen soll die Übersetzung von häufig benutzten Funktionen aus der *Standart-Library*[32] zu einer Erweiterung des TeSSLa-Core führen. Bei der Reduzierung einer TeSSLa-Spezifikation auf die äquivalente Core-Spezifikation werden strombasierte, nicht-Core-Funktionen durch teils sehr lange Schachtelungen der neun Core-Funktionen ersetzt. Häufig verwendete, nicht nativ implementierte Funktionen sorgen daher für umfangreiche Monitore, obwohl ihre Funktionalität teils wesentlich kürzer implementiert werden könnte.
Für die Übersetzung der neun nativen Core-Funktionen in die Intermediate Sprache wird eine sogenannte *Domain Specific Language* (DSL) verwendet. Das bedeutet, dass innerhalb von Scala, der Sprache in der der TeSSLa-Compiler verfasst wurde, eine eigene Programmiersprache speziell für die Überführung von TeSSLa-Core-Funktionen in Intermediate Code erstellt wurde. Diese ermöglicht die native Implementierung der Core-Funktionen und deren direkte Übersetzung in die Intermediate Sprache.
Da diese DSL jedoch sehr umfangreich und damit schlecht zu warten ist, wird innerhalb dieser Arbeit eine gleich mächtige, aber kürzere und gebrauchstauglichere DSL vorgestellt. Die Überarbeitung soll dazu verwendet werden, die oben genannten Funktionen möglichst zeitsparend implementieren zu können. Die direkte Übersetzung dieser Funktionen ermöglicht es geschachtelte Anfrageketten aufzulösen. Gerade häufig verwendete, bisher nicht native Funktionen versprechen eine hohe Effizienzsteigerung.

Abschließend werden auf die durchgeführten Verbesserungen in Form von Laufzeitmessungen der Monitore eingegangen. Zu beachten ist hierbei, dass Scala-Programme durch den Scala-Compiler zu *Java-Bytecode* kompiliert werden. Dieser Bytecode wird innerhalb einer *Java-Virtual-Machine* (JVM) mithilfe eines *Just-In-Time* Compilers (JiT) in Maschinencode übersetzt. Da der JiT Compiler, der in dieser Arbeit verwendeten JVM *OpenJDK* Optimierungen auf dem Bytecode durchführt, wird weiterhin untersucht, ob die vorgenommenen Optimierungen gegebenenfalls auch auf dieser Ebene durchgeführt werden und den Zweck der Optimerungen unterlaufen. Außerdem bietet der Scala-Compiler für die Übersetzung in Java-Bytecode eigene optionale Optimierungen an. Ob und welche Verbesserungen diese Einstellungen ergeben, wird ebenso beleuchtet.

1.1 Aufbau dieser Arbeit

Diese Arbeit beginnt mit einer Einführung in die für diese Arbeit nötigen *Grundlagen*. Das Kapitel leitet mit einer Erläuterung der für die Thematik grundlegenden Runtime Verifikation ein. Anschließend wird die Spezifikationssprache TeSSLa eingeführt, indem die Syntax und die Eigenschaften der Sprache vorgestellt werden. Als nächstes wird der TeSSLa-Compiler beleuchtet. Hierfür wird zunächst auf die Struktur von Compilern eingegangen, bevor die exakte Funktionsweise des TeSSLa-Compilers beschrieben wird. Zudem wird die Scala-Programmiersprache vorgestellt, da sie die Grundlage für die Domain Specific Language bildet, die im Rahmen dieser Arbeit entwickelt wurde. Auch der Compiler wurde in Scala geschrieben. Eine Erläuterung von Domain Specific Languages schließt an, wofür sie benutzt werden und wie sie in Scala implementiert werden können. Es wird anhand eines Beispiels eine einfache DSL vorgestellt. Das Grundlagen-Kapitel schließt mit einer Einführung in die Just-In-Time Kompilierung.

Im Kapitel *Identifizierung möglicher Optimierungen* wird auf die im Vorfeld erkannten Möglichkeiten zur Optimierung des Systems eingegangen. Dabei wird an dieser Stelle lediglich die Idee beleuchtet. Anschließend wird im Kapitel *Implementierung* konkret auf die Umsetzung dieser in Form eines Vorher-Nachher-Vergleichs eingegangen.

Die Ergebnisse der Implementierung werden im Kapitel *Evaluation* vorgestellt, wobei auf erzielte Verbesserungen eingegangen wird und was der bereits erwähnte JiT-Compiler an Optimierungen selbst übernimmt.

Die Arbeit endet mit einer *Zusammenfassung* und einem abschließenden *Ausblick* darauf, welche Möglichkeiten sich nach der Überarbeitung des Compilers noch ergeben.

2

Grundlagen

2.1 Runtime Verifikation

Um ein Programm zu verifizieren, werden Eigenschaften definiert, welche auf korrektes beziehungsweise falsches Verhalten hinweisen. Diese Eigenschaften lassen sich als Spezifikationen zusammenfassen. Systeme und Programme können daraufhin auf Verstöße dieser formulierten Spezifikationen getestet werden.
Runtime Verifikation (RV) [10] ist eine Variante der Systemverifikation, bei der Informationen aus einem laufenden Programm extrahiert und benutzt werden, um dessen Verhalten zu beobachten und gegebenenfalls darauf zu reagieren. Beobachtungen werden in zwei grundsätzliche Arten unterteilt. Entweder werden Abbildungen des aktuellen Zustands des Laufes untersucht oder Aktionen und Zustandsübergänge aufgezeichnet. Diese Beobachtungen werden *Events* genannt.
Als *Lauf* eines Programms wird in der Software-Technik eine möglicherweise unendliche Sequenz der Systemzustände bezeichnet. Die Ausführung eines Programms bezeichnet einen endlichen Prefix eines Laufes, welche *Trace* genannt wird. Diese Traces können sich aus einzelnen Events oder aus Mengen von Events zusammensetzen.
Um ein Programm mittels RV verifizieren zu können, lässt sich aus einer Spezifikation ein sogenannter *Monitor* erzeugen. Nachdem festgelegt wurde, was in einem zu verifizierenden Programm ein Event ist, sodass sich Traces bilden lassen, und welche Spezifikation zu erfüllen ist, wird der Monitor erstellt. Ein Monitor ist ein Programm, welches den Trace des zu verifizierenden Systems auf Verstöße der Spezifikation kontrolliert. So lässt sich zu jedem Event ausgeben, ob ein Verstoß vorliegt oder die Spezifikation bis dahin erfüllt werden konnte.
Die RV ist keine vollständige Beweismethode. Nach der Prüfung eines einzelnen Laufes liegt folglich kein Beweis über die Korrektheit des Systems vor. Die RV ist dadurch jedoch weniger komplex und einfacher anzuwenden als andere formale Methoden wie das *Model Checking* oder das *Theorem Proving*.

Stream Runtime Verifikation (SRV) [20] erweitert die RV um die Möglichkeit, eine Menge von Eingabeströmen auf eine Menge von Ausgabe-Strömen abzubilden. Diese Erweiterung erlaubt es, mehrere einzelne Ströme zu komplexeren zu kombinieren, sodass diver-

sere Urteile entstehen können, welche über Wahr oder Falsch Aussagen, also das Erfüllen beziehungsweise das Verstoßen einer Spezifikation, hinausgehen.
In der SRV werden Abhängigkeiten zwischen den Werten von Eingabeströmen und selbst definierten Strömen beschrieben. Dies erlaubt es die "wohl erforschten Evaluationsalgorithmen" [20] der RV zu generalisieren um damit Statistiken über den Traces zu erfassen.

2.2 TeSSLa

Einführung

TeSSLa, Kurzform für Temporal Stream-based Specification Language, ist eine strombasierte Spezifikationssprache, welche seit 2017 am Institut für Software Engineering und Programmiersprachen der Universität zu Lübeck entwickelt wird [17]. TeSSLa ist, ähnlich wie *LOLA* [9], eine Sprache für die SRV cyber-physikalischer Systeme, welche Events mit Zeitstempeln nativ unterstützt.
Neben der TeSSLa-Spezifikationssprache existieren Implementierungen zur Nutzung von TeSSLa. Ein Compiler [19] existiert, um Monitore zu erstellen und die formulierten Spezifikationen prüfen zu können. TeSSLa als Spezifikationssprache der SRV folgt den Prinzipien von synchronen Programmiersprachen wie *Esterel* [11] oder *Lustre* [15]. Diese Sprachen dienen jedoch der kausalen Beschreibung von Systemen und nicht deren Beobachtung. Sie sind vielmehr als Programmiersprachen, anstatt als Spezifikationssprachen zu verstehen.

TeSSLa definiert eine globale Uhr, sodass jedem Event des Stroms ein globaler Zeitstempel zugewiesen werden kann.
Die Ströme in TeSSLa lassen sich auf zwei verschiedene Arten interpretieren. Ein *Event-Strom* kann zu einem Zeitpunkt ein Event vorliegen haben, oder nicht. Ist an dieser Stelle kein Event vorliegend, so wird interpretiert, dass der Strom dort über keinen gültigen Wert verfügt. Die zweite mögliche Strom-Art ist der *Signal-Strom*, bei dem jeweils der Wert des letzten Events bis zum nächsten fortgilt. Diese Art des Stroms kann seinen Wert verändern, ist zwischen diesen Wertwechseln (den Signalen) jedoch konstant. Diese Interpretation ist sinnvoll, wenn beispielsweise ein kontinuierlicher Wert nur an bestimmten Stellen geliefert wird (z.B. Temperaturen).
Intern unterscheidet der Compiler diese Arten von Strömen nicht, TeSSLa bietet aber spezielle Operatoren um sinnvoll mit Strömen mit Signal-Semantik zu arbeiten.

Ein Beispiel zur Verständnis der Interpretation von Strömen sind ein Event-Strom *e* und ein Signal-Strom *s*. *s* wird mittels der Werte von *e* und eines *default*-Wertes *0* erzeugt. Anschließend lässt sich ein weiterer Event-Strom *e2* auf den Veränderungs-Signalen des Stroms *s* definieren:

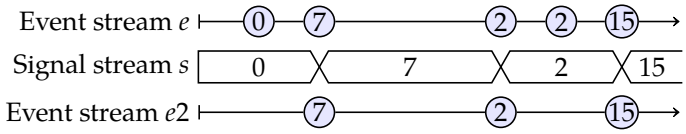

Abbildung 2.1: Darstellung der verschiedenen Stromarten.

Eigenschaften der Verwendung

In TeSSLa existieren die wohl bekannten Basistypen *Boolean*, *Integer*, *String*, *Float* und *Unit*. *Option* repräsentiert über *Some*() und *None* optionale Werte. Neben den *Collections Set*, *List* und *Map* existiert die Datenstruktur *Events*, welche einen Strom der Basistypen darstellt. TeSSLa unterscheidet zwischen einem *strombasierten* und einem *nicht-strombasierten* Teil. Nicht-strombasierte Funktionen sind Funktionen, die keine Ströme als Parameter erhalten, sondern einzelne Werte übergeben bekommen. Aspekt des nicht-strombasierten Teils ist weiterhin das Definieren von Konstanten. Der strombasierte Teil ist für die Verarbeitung von Strömen zuständig. Daher sind Eingaben strombasierter Funktionen vom Typ *Events*. Nicht-strombasierte Funktionen und Konstanten können im strombasierten Teil verwendet werden.

Eine konkrete TeSSLa-Spezifikation besteht aus drei Komponenten. Zunächst werden über das Schlüsselwort **in** die Eingabeströme der Spezifikation definiert. Als nächstes werden mittels **def** Strom-Variablen erzeugt. Zuletzt werden die mit **out** gekennzeichneten Funktionen von einem Monitor kontrolliert und ausgegeben.
Ein Beispiel für eine mögliche Spezifikation ist die Folgende:

```
in temperature: Events[Int]

def low  := x < 3
def high := x > 8

def unsafe := low || high

out *
```

Der Eingabestrom der Spezifikation mit dem Namen *temperature* ist vom Typ Integer. Zu jedem Zustand des Systems kann ein Integer-Wert vorliegen.
In diesem Beispiel wird in den Funktionen *low* und *high* der anliegende Wert für *temperature* ausgewertet, wobei *low* zu wahr auswertet, sollte der anliegende Wert echt kleiner drei sein, und *high* ein Wahr ergibt, falls der Wert echt größer acht ist. Evaluiert mindestens eine dieser beiden Funktionen, *low* oder *high*, zu wahr aus, wertet die Funktion

unsafe ebenfalls zu wahr aus. Schließlich werden alle diese Ströme mithilfe des Befehls **out** * von dem Monitor geprüft und ausgegeben. Mit einem Beispiel-Strom *temperature* von Integer-Werten kann dies wie folgt aussehen:

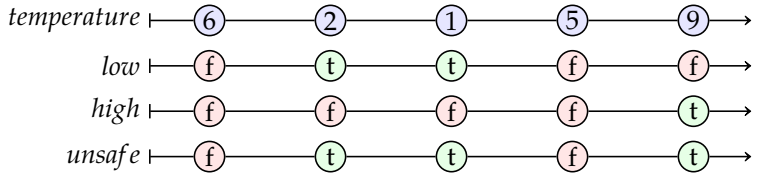

Abbildung 2.2: Beispiel Spezifikation über kritische Temperaturen

TeSSLa folgt in seiner Implementierung dem Paradigma einer funktionalen Programmiersprache, weshalb Funktionen als Werte und Parameter verwendet werden können. Typisch für funktionale Sprachen ist die Verwendung von rekursiven Definitionen. Auf den Gebrauch von den in der imperativen Programmierung typischen inneren Berechnungszustände, sowie auf die zugehörigen Seiteneffekte wird gänzlich verzichtet.

Der für die SRV relevante, strombasierte Teil von TeSSLa lässt sich auf neun Grundfunktionen reduzieren, die sogenannten *nativen* Funktionen oder *Core-Funktionen*. Mit diesen Funktionen lassen sich alle weiteren in TeSSLa enthaltenen Funktionen realisieren. Dieser Kern der TeSSLa Sprache wird als *TeSSLa-Core* bezeichnet.
Die neun nativen Funktionen sind folgend mit ihrer Signatur nach der offiziellen Dokumentation angegeben [32]. Die Signaturen beinhalten generische Typen, das heißt, dass sie einen beliebigen, aber festen Typen annehmen, je nachdem in welchem Kontext sie verwendet werden. Generische Typen werden mittels Verwendung von Großbuchstaben in eckigen Klammern hinter dem Namen der Funktion dargestellt. Unterschiedliche Großbuchstaben geben an, dass unterschiedliche Typen verwendet werden können, es sich jedoch auch um den selben Typ handeln kann. Argumente der Funktion können unter anderem Funktionen sein.
Zusätzlich zu den Signaturen der nativen Funktionen wird zu jeder eine Beschreibung und, falls möglich, ein Beispiel präsentiert.

nil[*T*]*: Events*[*T*] - erzeugt einen leeren Strom eines übergebenen Typs *T* ohne Events.

default[*T*](*x: Events*[*T*], *n: T*)*: Events*[*T*] - erzeugt einen Strom des Typs *T*, welcher alle Events eines übergebenen Stroms *x* enthält. Hat dieser übergebene Strom zum Zeitpunkt *0* keinen Wert, wird stattdessen der Wert *n* für diesen Zeitpunkt ausgegeben.

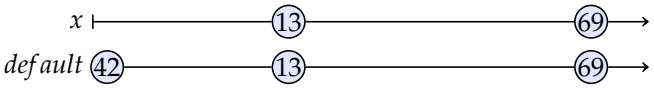

Abbildung 2.3: Beispiel eines *default*-Stroms

defaultFrom[*T*](*x: Events*[*T*], *y: Events*[*T*])*: Events*[*T*] - ähnlich zu *default* erzeugt die *defaultFrom*-Funktion einen Strom mit den Events des Stroms *x* des Typs *T*. Sollte der Strom *y* zu einem früheren Zeitpunkt als *x* ein erstes Event haben, so wird dieses und nur dieses Event mit ausgegeben.

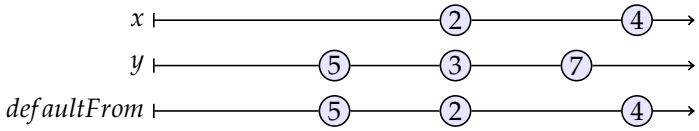

Abbildung 2.4: Beispiel eines *defaultFrom*-Stroms

time[*T*](*x: Events*[*T*])*: Events*[*Int*] - erzeugt einen Strom über die Zeitstempel der Events eines übergebenen Stroms *x*.

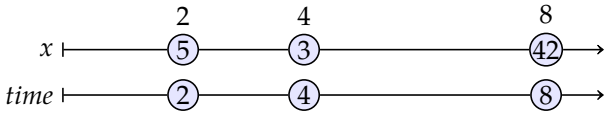

Abbildung 2.5: Beispiel eines *time*-Stroms

last[*T, U*](*x: Events*[*T*], *t: Events*[*Events*[*U*]])*: Events*[*T*] - nimmt zwei Ströme entgegen, wobei der Strom *t* als eine Art *trigger* fungiert; Liegt an *t* ein Event an, wird das zuletzt erfasste Event des Stroms *x* ausgegeben.

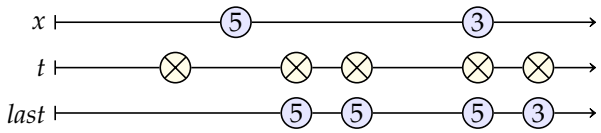

Abbildung 2.6: Beispiel eines *last*-Stroms

delay[*T*](*x: Events*[*Int*], *r: Events*[*T*]): *Events*[*Unit*] - nimmt einen Integer-Strom entgegen, dessen Events angeben, nach wie viel Zeiteinheiten ein *Unit*-Event ausgegeben wird. Es entsteht jedoch nur dann ein *Unit*-Event, solange zum selben Zeitpunkt des Integer-Events ein Event am Strom *r* (*reset*) anliegt, oder sobald zum selben Zeitpunkt des Integer-Events ein *Unit*-Event als Ergebnis eines vorherigen *delays* ausgegeben wurde.

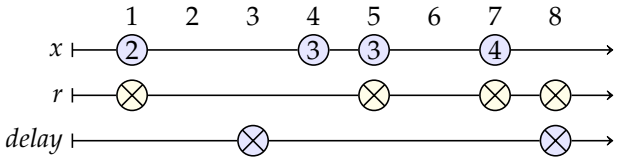

Abbildung 2.7: Beispiel eines *delay*-Stroms

lift[*T, U, V*](*x: Events*[*T*], *y: Events*[*U*], *f:* (*Option*[*T*], *Option*[*U*]) => *Option*[*V*]): *Events*[*V*] - wendet die Funktion *f* auf die Ströme *x* und *y* an und erzeugt einen Strom über dessen Ausgaben. *f* wird aufgerufen sobald ein Event auf einem der beiden Ströme *x* oder *y* vorliegt. Sollte zu einem Zeitpunkt auf beiden Strömen ein Event vorliegen, werden beide Events *f* übergeben. Ansonsten wird ein *None* für den Strom ohne Event übergeben. *lift* erzeugt kein Event, wenn nicht mindestens einer der zwei Parameter-Ströme ein Event aufweist. Der *Option*-Wert *None* wird dabei wie ein Event behandelt.
Wenn von einem *lift* gesprochen wird, handelt es sich zumeist um die hier gezeigte Funktion *lift2*. *lift* wird ebenfalls mit nur einem und bis zu fünf Eingabeströmen von TeSSLa unterstützt. Für diese Fälle muss der Funktion lediglich die gewünschte Anzahl an Eingabeströmen übergeben werden. Diese Funktionen heißen *lift1*, *lift3*, usw. Dieser Vorgang wird auch *liften* genannt.

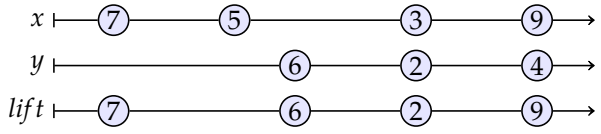

Abbildung 2.8: Beispiel eines *lift*-Stroms

slift[*T, U, V*](*x: Events*[*Int*], *y: Events*[*Int*], *f:* (*T, U*) => *V*): *Events*[*V*] - *slift* ermöglicht die Verwendung der *lift*-Funktion auf asynchronen Strömen. *f* wird jedes Mal ausgewertet, wenn ein neues Event auf einem der Ströme *x* oder *y* anliegt. Sollte zu einem Zeitpunkt einer der beiden Ströme kein neues Event haben, so wird *f* mit dem aktuellsten Event dieses Stroms ausgewertet. *f* wird nicht evaluiert bevor beide Ströme nicht mindestens ein Event hatten.

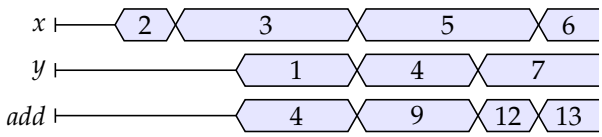

Abbildung 2.9: Beispiel eines *slift*-Stroms

merge[T](x: *Events*[T], y: *Events*[T]): *Events*[T] - kombiniert die Events der Eingabeströme zu einem einzelnen Strom. Ebenso wie bei *lift*, kann *merge* bis zu fünf Eingabeströme verarbeiten. Die Ströme werden dabei nach ihrer Position von links nach rechts priorisiert.

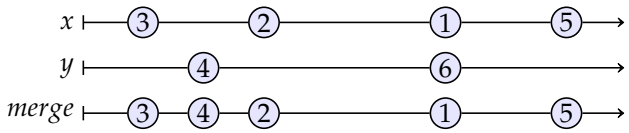

Abbildung 2.10: Beispiel eines *merge*-Stroms

Diese neun Funktionen sind ausreichend, um eine eigenständige Programmiersprache darzustellen (Turing-Vollständig). Weitere Funktionen werden auf Grundlage der neun Core-Funktionen definiert. Einige Funktionen wurden bereits erstellt und in TeSSLas Standart-Library [32] zusammengefasst. Beispielsweise inkrementiert eine *count*-Funktion bei jedem Eintreffen eines Events auf einem Strom eine Zählervariable. Diese Funktion lässt sich mithilfe einer *fold*-Funktion realisieren, welche wiederum aus einer Schachtelung eines *defaults* und eines *lifts* entsteht (Es folgen weitere Erläuterungen zu diesem Beispiel).

Die Anwendungsbereiche für TeSSLa sind neben der klassischen Verwendung für die RV, die Log-Analyse und das live Debugging. Für diese ist es unerlässlich, dass Ausgaben möglichst zeitnah zu ihrer Eingabe generiert werden, da sich der Zustand des Systems ständig verändern kann. Aussagen über Events, nachdem bereits n weitere folgten, sind hinderlich für das Nachvollziehen von Fehlern und könnten die Verifikation erschweren. Der Anspruch ist es daher möglichst effiziente Monitore zu generieren, um damit schnell Ausgaben für die zu beobachtenden Traces zu erhalten.

2.3 Compiler

Als *Compiler* werden Programme bezeichnet, die ein beliebiges Programm einer Quellsprache in ein äquivalentes Programm einer Zielsprache übersetzen [2]. Um ein Programm ausführen zu können, muss es in eine für den Computer verständliche Form überführt werden, weshalb Compiler häufig dazu verwendet werden, Programme einer Programmiersprache in Maschinencode zu übersetzen. Das entstandene Programm der Zielsprache lässt sich von einem Nutzer ausführen, um zu einer Eingabe eine Ausgabe zu generieren.
Interpreter sind eine weit verbreitete Alternative zu Compilern [2]. Anstatt ein Programm vor seiner Laufzeit zu übersetzen, führt ein Interpreter ein Programm der Quellsprache zusammen mit einer Eingabe direkt aus. Zumeist sind Compiler in der Erzeugung einer Ausgabe zu einer übergebenen Eingabe schneller als ein Interpreter. Oft sind Compiler

große, komplexe Systeme, welche in mehrere Sub-Systeme und Komponenten unterteilt werden. Diese Teile werden nach [2] in einen Durchlauf von sechs Phasen klassifiziert. Die Ausgabe des jeweils Vorherigen bildet die Eingabe des Nachfolgers. Die sechs Phasen werden weiterhin grob in eine *Analyse*- und eine *Synthesephase* unterteilt. Die Komponenten lauten:

Analysephase

1. Lexikalische Analyse/Scanning:

Liest einen Zeichen-Strom ein (das Eingabeprogramm) und gruppiert Zeichen in aussagekräftige Sequenzen, sogenannte *Lexeme*. Jedes Lexem produziert ein *Token*, unterteilt in einen Namen und gegebenenfalls einen Wert. Der Wert eines Tokens wird in der *Semantischen Analyse* und der *Codegenerierung* verwendet.

2. Syntaktische Analyse/Parsing:

Der Parser verwendet die vorab erstellten Token um eine baumartige Repräsentation der grammatikalischen Struktur des Token-Stroms zu erstellen. Typisch für eine derartige Struktur ist ein *Syntax-Tree*, dessen Knoten Operationen und deren Kinder deren Argumente abbilden. Falls Verstöße der Grammatik der Quellsprache vorliegen, werden sie in diesem Schritt erkannt.

3. Semantische Analyse:

Der Syntax-Tree und die Token-Werte werden verwendet um semantische Konsistenz auf Grundlage der Sprach-Definitionen zu prüfen. Ein typischer Bestandteil dieser Phase ist das *Type Checking*. Wird beispielsweise versucht einen Array mit einem *Float* statt eines *Integer* zu indizieren, wird der Fehler in dieser Phase erkannt.

Synthesephase

4. Zwischencodeerzeugung (Intermediate Code Generation):

Im Zuge der Übersetzung eines Quellsprachenprogramms in ein Ausgabeprogramm kann es sich anbieten, *Zwischenrepräsentationen* des Codes zu erzeugen. Eine solche Repräsentation sollte generisch sein, um einfach zu erzeugen und übersetzen sein zu können. Gerade bei Compilern, die verschiedene Quellsprachen oder mehrere Zielplattformen unterstützen, bietet sich dieses Verfahren an. Zwischenrepräsentationen werden auch *Intermediate Code*, *Zwischencode*, *Intermediate Sprache* oder *Zwischensprache* genannt.

5. Code Optimierung:

Die generische Darstellungsform des Zwischencodes bietet die Durchführung von Optimierungen an. Dabei soll optimierter Code entstehen, der infolgedessen schneller auszuführen ist. Es kann sich bei den Optimierungen um allgemein kompaktifizierende oder energieeffizientere Verbesserungen handeln.

6. Codegenerierung:
Der optimierte Zwischencode wird letztendlich auf ein Ausgabeprogramm *gemappt*. Sollte die Zielsprache Maschinencode sein, ist ein wesentlicher Bestandteil der Übersetzung die Allokierung von Registern. Hierfür werden die Token-Werte der *Lexikalischen Analyse* benötigt.

Eine umfangreichere Erläuterung der Phasen und Beispiele sind in [2] zu finden.

Die hohe Komplexität von Compilern entsteht durch die Vielzahl an verwendeten Praktiken. Beispielsweise werden *greedy* Algorithmen für Register Allokationen, heuristische Algorithmen für *List-Scheduling-Probleme* oder Graph-Algorithmen, um nicht-verwendete Codeabschnitte zu identifizieren, verwendet. Aber auch theoretische Konstrukte wie deterministische Automaten in Verbindung mit formalen Sprachen für das *Scannen* und *Parsen* der Eingaben und die dynamische Programmierung im Bezug auf die Reduzierung einer *High-Level* Sprache lassen sich in gewöhnlichen Compilern wiederfinden.

Compiler lassen sich in Varianten kategorisieren. *Native* Compiler [22] erzeugen Zielcode für eine Plattform, auf der sie selbst laufen. *Cross-Compiler* [13] hingegen erzeugen Zielcode für fremde Plattformen, häufig zu finden bei Zielcode für Hardware, welche nicht leistungsfähig genug sind, um einen Compiler zu hosten.
Weiterhin lassen sich Compiler in *Single-* [31] und *Multi-Pass* [30, 16] Compiler unterscheiden. Ein *Single-Pass* Compiler übersetzt eine Eingabe innerhalb eines einmaligen Lesens des Programmcodes. Das mehrfache Lesen der Programmanweisungen bei *Multi-Pass* Compilern wurde ursprünglich verwendet, da die Computer-Kapazitäten nicht ausreichten, um den vollständigen Compiler und den Programmcode im Hauptspeicher zu halten. Mittlerweile wird diese Art des Compilers zur Optimierung von Ausgaben verwendet. Dies kann sich auf die Ausführungszeit, den benötigten Speicherplatz oder den Energieverbrauch des Programms auswirken. Optimierungen erhöhen jedoch die Kompilierzeit und erschweren das Debugging, da die Reihenfolge der Anweisungen verändert werden könnte.
Während auf Server- oder Desktop-Systemen die Dateigröße und die Energieeffizienz eine untergeordnete Rolle spielen und die Performance im Fokus steht, werden in Mobile- und Embedded-Systemen (so auch in der RV) meist möglichst kleine, energiesparende Programme benötigt.

Sonderformen von Compilern sind unter anderem *Transcompiler* [34] und *Just-In-Time* Compiler [4]. Ersterer übersetzt Spezifikationen einer Programmiersprache in andere Programmiersprachen anstatt in Maschinencode. Ein Beispiel ist der im Folgenden beleuchtete TeSSLa-Compiler. In Abschnitt *4* des Grundlagenkapitels wird auf Just-In-Time Compiler näher eingegangen.

2.4 Der TeSSLa-Compiler

Der 2019 in [19] vorgestellte, in Scala implementierte Multipass-Transcompiler wurde entwickelt, um effizientere TeSSLa Monitore als die vom bereits existierenden Interpreter Generierten zu erzeugen. Der Compiler nimmt eine TeSSLa-Spezifikation entgegen und erzeugt auf Grundlage dieser ein Scala-Programm, den Monitor.
Hierfür wird die TeSSLa-Spezifikation nach Reduktion auf den TeSSLa-Core in eine Intermediate Sprache überführt, bevor sie in Scala übersetzt wird. Der zu generierende Scala-Code des Monitors besteht aus einem festen Code-Skelett, in welches Spezifikations-abhängige Codeabschnitte eingesetzt werden. Der Hauptbestandteil dieses Skeletts ist eine *while*-Schleife, welche in jedem Durchlauf Veränderungen mit Eintreffen des nächst größeren Zeitstempels verarbeitet.

Der Compilier-Vorgang ist somit in einen Durchlauf von drei Phasen unterteilt:

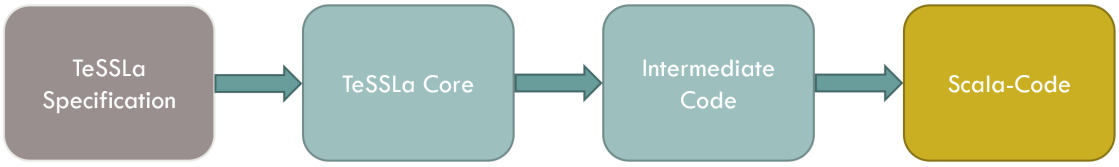

Abbildung 2.11: Die Phasenunterteilung des TeSSLa-Compiler

In der ersten Phase wird der strombasierte Teil einer Eingabe-Spezifikation in eine äquivalente, nur aus TeSSLa-Core-Funktionen bestehenden Spezifikation reduziert. Der Core besteht aus insgesamt neun Funktionen, den *nativ implementierten* Funktionen oder auch Core-Funktionen; *nil*, *default*, *defaultFrom*, *last*, *time*, *delay*, *lift*, *signalLift* und *merge* (siehe Kapitel 2.2). Jede weitere in TeSSLa enthaltene Strom-Funktion wird aus diesen neun Funktionen konstruiert. Einige Funktionen wurden so bereits erstellt und in der TeSSLa Standart-Library zusammengefasst [32]. Ein Beispiel, welches im Laufe dieses Sub-Kapitels weiter verwendet werden soll; Die Signatur der *filter*-Funktion ist wie folgt definiert:

```
def filter[A](events: Events[A], condition: Events[Bool]): Events[A]
```

Elemente eines Stroms *events* des Typs *A* (generisch, damit beliebig aber fest) werden mittels eines Signalstroms *condition* gefiltert. Ist der Wert des Stroms *condition true*, werden die Events des Stroms *events* zu einem neu generierten Strom hinzugefügt. Zeigt *condition* jedoch *false*, werden die eintreffenden Events auf *events* ignoriert.

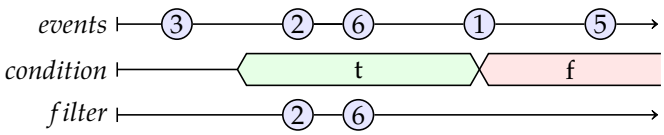

Abbildung 2.12: Beispiel eines *filter*-Stroms

Da *filter* keine nativ implementierte Funktion ist, wird sie aus einer Kombination mehrerer nativ implementierter Funktionen zusammengesetzt, in diesem Fall *lift*, *merge* und *last*:

```
lift(events, merge(condition, last(condition, events)),
     (e: Option[A], c: Option[Bool]) => if isSome(c) && getSome(c)
                                        then e
                                        else None[A])
```

Die Funktionen *getSome* und *isNone* sind Teil des nicht-strombasierten Aspekts TeSSLas und Teil der Standart-Library. Derartige Funktionen bleiben von der Reduktion unberührt. Wird *filter* in der ersten Phase der Kompilierung reduziert, bleibt also der Aufruf eines *lift*.
Den Abschluss der ersten Phase bildet ein sogenanntes *Flattening* der TeSSLa-Core-Spezifikation. Wie im Beispiel *filter* zu erkennen, entsteht durch die Reduktionen von nicht-Core-Funktionen geschachtelte Core-Funktionen. Zur einfacheren Verarbeitung bietet es sich an, das entstehende *Nesting* der Funktionen aufzulösen, um die Zuweisung der Variablen zu vereinfachen:

```
var_1 = last(condition, events)
var_2 = merge(condition, var_1)
var_3 = lift((events, var_2), (e: Option[A], c: Option[Bool]) =>
        if isSome(c) && getSome(c) then e else None[A])
```

Innerhalb der Reduktion des *filter*-Beispiels wäre dies beispielsweise der zweite Parameter des *lift*; *merge*. Wird das reduzierte *filter geflattet*, würde *lift* als zweiten Parameter eine Variable übergeben bekommen, welche den durch *merge* entstandenen Strom speichert.
Am Ende dieses *Flattening*-Vorgangs existiert eine äquivalente Version der noch kurz zuvor erzeugten Core-Spezifikation, in der Funktionen nur noch als Variablen gespeicherte Ströme (Strom-Variablen) verwenden.

Als nächstes wird in der zweiten Phase des Kompiliervorgangs die *flat* Core-Version der Eingabe-Spezifikation in einen Zwischencode überführt. Die Spezifikation in TeSSLa-Core wird Anweisung für Anweisung durchlaufen und zu den vorliegenden Funktionen werden bereits vor-implementierte Code-Blöcke erzeugt.
Dieser Vorgang spielt sich in den Klassen *TeSSLaCoreToIntermediate*, *NonStream-* und *StreamCodeGenerator* ab. In *TeSSLaCoreToIntermediate* wird über eine Core-Spezifikation iteriert, indem ein *Matching* auf die Zugehörigkeit einer Funktion durchgeführt wird - Die erkannten Funktionen werden jeweils in *StreamCodeGenerator* und *NonStreamCodeGenerator* übersetzt, je nachdem ob eine strombasierte, beziehungsweise eine nicht-strombasierte Funktion gematcht wurde.
Strombasierte Funktionen sorgen innerhalb des Matchings dafür, dass *TeSSLaCoreToIntermediate* die Funktion *translateExternSignalExpression* in *StreamCodeGenerator* aufruft. *translateExternSignalExpression* führt ein weiteres Matching durch und ruft je nachdem welche Funktion vorliegt, die korrespondierende Methode für die Erzeugung eines Code-Blocks auf. Zu jeder der neun nativen Funktionen existiert eine solche Methode. Diese Methoden beschreiben die Logik der Core-Funktionen jeweils als einen festen Code in Form einer

Domain Specific Language (siehe Kapitel *2.6*).
Bei Verwendung einer Methode wird ein fester Block Zwischencode erzeugt,der entsprechend der Spezifikation parametrisiert wird.

Teil dieser Blöcke und ausschlaggebend für die Komplexität der Sprache sind die intern verwendeten Variablen. Zu jedem Strom werden Informationen abseits des Wertes eines Events benötigt; Informationen über den vorherigen Wert (*_lastValue*), ob die Variable initialisiert wurde (*_init*), ob die Variable sich verändert hat (*_changed*), ob ein Fehler vorliegt oder ein Fehler zu vorherigem Wert vorlag (*_error* und *_lastError*), den Zeitstempel der letzten Schreiboperation der Variable (*_ts*) und falls nicht bekannt ist, ob der Strom aktuell ein Event vorliegen hat (*_unknown*) werden gespeichert. Diese Informationen sind für Eigenschaften von TeSSLa nötig, wie beispielsweise *_ts* für das Arbeiten mit asynchronen Strömen. Zu jeder Variable existiert ein Standard-Wert und ein Typ. Diese Variablen werden während der Entstehung des Zwischencodes erstellt (*variablenName_information*) und in der dritten Phase im Code-Skelett anhand ihrer Standard-Werte initialisiert.

Der entstehende Code ist, wie für Intermediate Sprachen üblich, möglichst generisch, um einfach in (in diesem Szenario) imperative Sprachen übersetzt werden zu können. Das Konzept eines Zwischencodes wurde, wie es für Compiler üblich ist, zur Erzeugung von Generizität verwendet und um infolgedessen besser optimiert werden zu können. Weiterhin lassen sich auf Grundlage der Zwischensprache und am Vorbild der bereits erfolgten Übersetzung zu Scala weitere imperative Ausgabesprachen hinzufügen.
Der Intermediate Code stellt innerhalb des TeSSLa-Compilers eine Abstraktionsebene zwischen dem Core und der Ausgabesprache dar. Die Optimierung und anschließende Übersetzung der Zwischensprache in eine Ausgabesprache bildet die dritte und damit letzte Phase des Kompiliervorgangs.

2.5 Scala

Scala ist eine objektorientierte, funktionale Allzweck-Programmiersprache, welche 2004 von Martin Odersky veröffentlicht wurde [25, 24]. Odersky arbeitete bereits zuvor an der minimalistischen Hybridsprache von funktionaler und objektorientierter Programmierung *Funnel*. 2001 begann er mit der Entwicklung von Scala, welche als vollwertige Sprache für reale Anwendungen geplant wurde. Scala wurde maßgeblich von der Programmiersprache Java beeinflusst und beschreibt sich selbst [25] als elegante Alternative für Java-Entwickler.

Java Integration

Scala-Programme kompilieren zu Java-Bytecode und laufen infolgedessen auf der *Java-Virtual-Machine* (JVM) [27]. Dies erlaubt es den Programmen Java-Archive (JAR's) anzusprechen und umgekehrt, sodass sich Java-Bibliotheken und -Frameworks in Scala-Projekte einbinden lassen.

Objekt-Orientierung

Scala ist im Gegensatz zu Java rein objekt-orientiert. Das heißt, dass jeder Wert als Objekt behandelt wird. Typen und Verhalten werden in *classes* und *traits* geschachtelt, wobei ein *trait* einer Mischung aus Interface und abstrakter Klasse in Java ähnelt (*mixins*). Klassen können über das Schlüsselwort *extends traits* einbinden, welche im Gegensatz zu Interfaces in Java bereits implementierte Funktionen enthalten können. Scala unterstützt weiterhin die Verwendung von abstrakten Klassen. Da *traits* jedoch von sich aus die Funktionalität einer abstrakten Klasse erfüllen, werden abstrakte Klassen in Scala nur verwendet, falls der Scala Code von Java Code aufgerufen wird oder eine Basisklasse mit festen Konstruktoren-Argumenten erstellt werden soll.
Klassen, von denen nur eine Instanz benötigt wird (*Singleton* Entwurfsmuster), werden in Scala als *object* definiert. Diese benötigen keine händische Initialisierung und werden an Stelle von statischen Feldern und Methoden verwendet.
Normale Klassen erlauben das Überschreiben ihrer Attribute, außer diese Attribute werden als konstant gekennzeichnet. Sogenannte *case classes* werden in Scala zur Darstellung unveränderlicher Daten verwendet, da sämtliche Konstruktoren-Parameter bereits als Konstanten deklariert werden. Der Vorteil gegenüber normalen Klassen ist, dass Instanzen von *case classes* nach ihrer Struktur verglichen werden und nicht nach Referenz, weshalb sie für das noch folgende *Pattern Matching* verwendet werden.

Funktionale Programmierung

Scala ist eine funktionale Programmiersprache, was bedeutet, dass Funktionen als Werte und Parameter verwendet werden können. Funktionen werden daher als *First-Class-Objects* bezeichnet.
Ein in der funktionalen Programmierung typisches und in dieser Arbeit viel gebrauchtes Konzept ist das *Pattern Matching*. Pattern Matching erlaubt es eine Eingabe auf definierte Muster zu vergleichen und entsprechend zu reagieren. In Scala funktioniert dies ähnlich zu den in Java verwendeten *switch-cases*, betrachtet jedoch nicht nur Werte, sondern auch die Strukturen und Typen von Objekten.

Syntax und Datenstrukturen

Scalas Syntax ist an die von Java angelehnt und übernimmt einige der Schlüsselwörter und Blocksyntax. Ein Semikolon am Ende einer Anweisung ist optional, außer Statements stehen in der selben Zeile hintereinander. In diesem Fall wird ein Semikolon als Trennsymbol verwendet.
Um Variablen in Scala zu deklarieren, verwendet man statt der Java-typischen Notation *Typ variablenName* ein *variablenName: Typ*. Mithilfe der Schlüsselwörter *val*, *var* und *def* lassen sich Konstanten, Variablen und Funktionen/Methoden deklarieren/initialisieren:

```
val konstant: Long = 42
var variable: Boolean = true
def methode(param1: Long, param2: Boolean): Unit = {...}
```

Neben den bekannten, primitiven Datentypen wie *Integer*, *Boolean*, *String* und ähnlichen werden in Scala die Datentypen *Unit*, *Nothing* und *Any* definiert. Weiterhin unterstützt Scala bekannte *Collections* wie *Sets*, *Listen* und *Maps*.

2.6 Domain Specific Languages

Eine *Domain Specific Language* (DSL) oder auch *anwendungsspezifische Sprache* ist eine formale Sprache, welche für die Interaktion innerhalb eines spezifischen Anwendungsgebietes verwendet wird [12]. Im Fall des TeSSLa-Compilers wird eine DSL zur Beschreibung der Übersetzung der TeSSLa-Core-Spezifikation in die Zwischensprache verwendet. In dieser Arbeit soll diese DSL durch eine kompaktere und gebrauchstauglichere DSL ersetzt werden.
Ein Beispiel für eine DSL ist *SQL*, welches ausschließlich in dem Kontext der Datenbanken-Bedienung verwendet wird. Der Entwurf einer DSL sollte einen möglichst hohen Grad an Problemspezifität anstreben. Die erzeugte Sprache soll alle Probleme seiner Domäne darstellen können und nichts darstellen, was außerhalb liegt. So lassen sich in SQL Datenbankenzugriffe tätigen, SQL ist jedoch nicht Turing-Vollständig und es lassen sich damit keine beliebigen Programme erstellen.
DSLs werden in *interne-* und *externe-* DSLs unterschieden [12]. Interne DSLs werden innerhalb anderer Programmiersprachen von Grund auf definiert. Für die Erstellung der DSL werden Komponenten der Sprachimplementierung dieser Programmiersprache verwendet, weshalb man sie als *Wirtssprache* der DSL bezeichnet. Am Beispiel Scala wird im folgenden Abschnitt aufgeführt, welche Komponenten eine Wirtssprache im Bezug auf DSLs bieten kann. Ein Beispiel für eine interne DSL ist *Rake* innerhalb seiner Wirtssprache Ruby [29].
Externe DSLs sind von Grund auf neu definierte Sprachen. Ein Beispiele für eine externe DSL ist das bereits erwähnte SQL.

Durch die Verwendung einer DSL soll grundsätzlich eine bessere Lesbarkeit erreicht werden. Jedoch soll gerade bei internen DSLs Redundanz verringert werden. Der Begriff *Boilerplate* stammt aus der Medienarbeit und dem Druckwesen und bezeichnet einen immer gleichbleibenden Textblock. Mittels DSL lassen sich Sprachkonstrukte definieren, welche den Umfang der Spezifikationen reduzieren können und damit *Boilerplate* teilweise einsparen. Der Gebrauch geeigneter DSLs kann in einfacher erkennbaren Optimierungen, leichterer Testbarkeit und reduzierten Wartungskosten resultieren.

DSLs im Bezug auf Scala

Die Scala-Programmiersprache als Wirtssprache für DSLs bietet einige Werkzeuge für effiziente interne DSLs und die damit einhergehende Reduzierung einer etwaigen Boilerplate.

Die Objekt-Orientierung erlaubt es Komponenten zu kapseln und wiederzuverwenden. Die funktionale Programmierung ermöglicht es präzisen Code zu verfassen.

Funktionen können weitestgehend (Ausnahmen sind Schlüsselworte der Sprache) freiwählbare Namen erhalten, was es ermöglicht Namen wie "+" und "*" zu vergeben, solange sie im Kontext eindeutig ist. Diese lassen sich statt der bekannten *dot*-Notation auch in *infix*-Notation schreiben. Beispielsweise lässt sich ein "*Kontostand.erhoehen*(*100.0*)" in *dot*-Notation mittels *infix* auch als *Kontostand erhoehen 100.00* schreiben, wobei zu erkennen ist, dass bei *Single-Parameter* Funktionen die Klammerung entfallen kann. Passend dazu lassen sich Standardparameter verwenden, sodass sich beispielsweise die Funktion *def erhoehen*(*wert: Double = 50.0*) definieren lässt, um *Kontostand erhoehen* mittels direkter Weitergabe des Wertes *50.0* zu vollziehen.
Das Schlüsselwort *implicit* ermöglicht es Funktionen implizit aufzurufen oder Parameter implizit zu übergeben. Wann immer eine übergebene Variable nicht den benötigten Typ der aufgerufenen Methode erfüllt, könnte eine im *Scope* verfügbare Methode diesen Wert in einen von der Methode angenommenen Typ *casten* oder sie direkt in diesen umwandeln. In vorherigem Beispiel ließe sich Folgendes definieren: Die Funktion *def erhoehen*(*wert: Euro*) erhält nicht mehr einen Double sondern ein erstelltes Objekt "Euro". Mittels einer impliziten Methode *implicit def doubleToEuro*(*wert: Double*) ließe sich dennoch *Kontostand erhoehen 100.0* schreiben, der übergebene Double-Wert wird implizit in ein Objekt Euro umgewandelt.
Um die Lesbarkeit weiter verbessern zu können, lassen sich die in Scala integrierten *case classes* verwenden. Diese lassen sich im Vergleich zu normalen Klassen ohne das Schlüsselwort "*new*" erzeugen und werden für die in dieser Arbeit entwickelten DSL eine wichtige Rolle spielen. Mehr Informationen bezüglich der Verwendung von Scala lassen sich in [23] finden.

Beispiel DSL für einen DFA

Als Beispiel wird im Folgenden die Implementierung eines deterministischen endlichen Automaten (*DFA*) in Scala aufgeführt.
Eine Klasse *DFA* ermöglicht es, einen neuen Automaten zu erzeugen. Mithilfe der Funktionen *states* und *finalStates* werden die Zustände und die finalen Zustände des DFAs definiert. Eine Klasse *Transition* erlaubt es Transitionen anzulegen und bei Eingaben *on* mittels *from* und *to* Übergänge festzulegen. Eine Funktion *startFrom* gibt den Startzustand des DFA an und mittels Funktion *startWith* wird diesem ein Eingabewort übergeben. Folgender DFA, welcher Wörter beginnend und endend mit 1 aktzeptiert, lässt sich durch die spezifizierte DSL erzeugen:

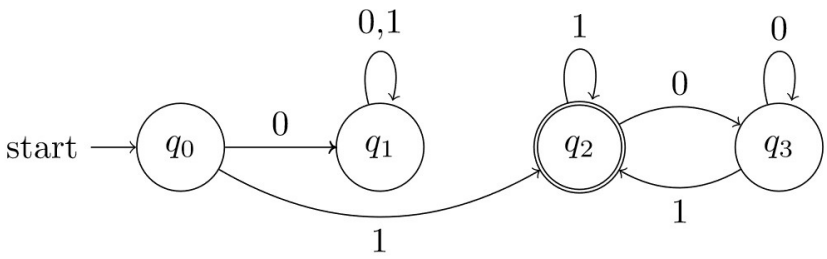

Abbildung 2.13: Ein DFA, welcher Wörter akzeptiert, die mit 1 beginnen und enden.

Code, der den vorherig beschriebenen DFA erstellt:

```
val dfa = newDfa()

dfa states Seq(S0, S1, S2, S3) finalStates Seq(S2)

dfa transitions { transition =>
    transition on '0' from S0 to S1
    transition on '1' from S0 to S2
    transition on '0' from S1 to S1
    transition on '1' from S1 to S1
    transition on '0' from S2 to S3
    transition on '1' from S2 to S2
    transition on '0' from S3 to S3
    transition on '1' from S3 to S2
}

dfa startFrom S0 withInput "11010101111011"

val hasInputAccepted = dfa.run
print(hasInputAccepted)
```

Die Ausgabe gibt abschließend an, ob das System zu gegebener Eingabe *11010101111011* einen akzeptierenden Zustand erreicht (Ausgabe *true*) oder nicht (Ausgabe *false*).

2.7 Just-In-Time Compilation

Der vom Java- oder Scala-Compiler erzeugte Bytecode wird von einer plattformunabhängigen JVM ausgeführt [27]. Die JVM ist Bestandteil der Laufzeitumgebung *Java Runtime Enviroment* (JRE), welches neben der JVM, für die Ausführung benötigte Software-Bibliotheken beinhaltet. Es existieren mehrere Implementierungen der JVM. Innerhalb dieser Arbeit wurde die JVM des Community-Projekts *OpenJDK* [6, 18, 26] verwendet, welche maßgeblich auf der JVM *HotSpot* [6, 28] von *Oracle* basiert. Eine JVM besteht grundlegend aus drei Komponenten. Der für diese Arbeit relevante Teil ist die *Execution Engine* (EE), welche für die Ausführung des Bytecodes zuständig ist. Die *Just-In-Time* (JiT) Kompilierung [4] ist eine gängige Methode, wie Bytecode von der EE ausgeführt werden kann.

Da Java *class*-Dateien nicht notwendigerweise vollständig zu Laufzeitbeginn vorliegen müssen [1], muss sich das sprachverarbeitende System der JVM dynamisch verhalten können. Da Interpreter jedoch im Vergleich zu Compilern in langsameren Laufzeiten resultieren, ist das Ziel, die Vorteile eines Compilers und eines Interpreters zu kombinieren, indem (Teil-)Programme erst bei Bedarf (*just in time*) in Maschinencode übersetzt werden. Kompilierte Programme haben generell eine kürzere Laufzeit, wohingegen Interpreter meist kompakter und damit portierbarer sind. Ein JiT Compiler soll eine Zwischenlösung anbieten und dabei möglichst portierbar sein.

Die Problematik eines JiT Compilers resultiert aus der Umsetzung einer Kombination beider Verfahren. Bei einem Compiler werden alle notwendigen Dateien vor Laufzeit kompiliert; bei einem Interpreter existiert solch eine Phase nicht. Um die Vorteile kombinieren zu können, muss eine JVM die Codeerzeugung in Form von Kompilierung und Optimierung zur Laufzeit durchführen [4, 18]. Die Idee ist es, auszuführenden Code zunächst zu interpretieren und einzelne Abschnitte gleichzeitig zu kompilieren. Dieser Prozess wird *Start-Up*- oder auch *Warm-Up*-Phase genannt [5, 6, 33]. [5, 14] konnten mit keiner Gewissheit feststellen, wann die *Start-Up*-Phase beendet und die *Steady*-Phase erreicht wird. Aufgrund von *Thread Scheduling*, *Garbage Collection* und Seiteneffekten des darunterliegenden Systems wird in [5] und [14] von nicht-Determinismus berichtet.

Aus diesem Grund ist die Anforderung an Optimierungsalgorithmen innerhalb eines JiT Compilers deren Effektivität und die Eigenschaft, *lightweight* [1] zu sein. Taktiken, die sich aus dieser Erkenntnis entwickelt haben, waren unter anderem das Identifizieren und gezielte Optimieren sogenannter *Hot Spots* [3]. Der Begriff *Hot Spot* bezieht sich meist auf häufig verwendete Methoden und Funktionen. Für jede Methode speichert die JVM einen Aufrufzähler [33]. Besonders häufig aufgerufene Methoden werden besonders stark optimiert. [33] kategorisiert in die Stufen *cold*, *warm*, *hot*, *veryHot* und *scorching*. [3] unterscheidet lediglich in die *Level 0*, *1* und *2*. Im Grundsatz sind die Kategorisierungen jedoch gleich - niedrige Stufen optimieren weniger und sind daher Laufzeit schonender. Funktionen oder Methoden hoher Stufen werden aufgrund ihres häufigen Auftretens stark optimiert, was sich in höheren Kompilierungskosten niederschlägt. Die Stufen innerhalb der *OpenJDK* JVM werden mit *0* bis *4* betitelt [6].
Neben solchen *on-the-fly* Optimierungen lassen sich Interpreter-typische Optimierungen durchführen. Da dem JiT Compiler wie bei einem Interpreter der gesamte Kontext eines Programms bekannt ist, kann der Eingabecode auf Grundlage dieser Information verändert werden. [1] beschreibt fünf Phasen des *Intel JiT Compilers*, in denen mehrfach über den Code iteriert wird, um Informationen zu sammeln, gleichbleibende Sub-Ausdrücke aufzulösen, Array-Grenzen zu überprüfen, *Peephole* Optimierungen durchzuführen und gegebenenfalls *Frame-Pointer* zu entfernen. [18] gibt eine umfangreiche Auflistung der Optimierungen innerhalb der *HotSpot* JVM an, darunter auch die Erkennung von Konstanten.

Aufgrund der Transformation einer TeSSLa-Spezifikation zu einem Scala-Programm, dem Monitor, ist der JiT Compiler für diese Arbeit relevant. Da ein Monitor zu Java-Bytecode kompiliert und dieser auf einer JVM ausgeführt wird, gilt es zu prüfen, ob die in dieser Arbeit vorgenommenen Optimierungen gegebenenfalls vom JiT Compiler übernommen werden und somit obsolet sind. In Hinsicht auf die Analyse auf konstante Initialisierungen dürfte der JiT Compiler dies durch die wiederholte Iteration und das Sammeln von Informationen des Codes bereits selbst umsetzen, wie es in der *HotSpot* JVM getan wird. Da *OpenJDK* direkt auf diesem System basiert, liegt die Vermutung nahe, dass diese die selben Optimierungen durchführt. Messungen und Interpretationen werden in Kapitel *5* vorgestellt.

3

Umgesetzte Optimierungen

Im Folgenden werden die zwei im Vorfeld erkannten und durchgeführten Optimierungen des TeSSLa-Compilers präsentiert. Die *Implementierung nativer Funktionen mit Hilfe einer neuen DSL* bezieht sich auf den Übergang der Eingabe-Spezifikation in die reduzierte Variante des TeSSLa-Core, der ersten Phase des Kompiliervorgangs. Die zweite Optimierung ist eine Var-Val-Analyse, welche sich auf die Initialisierung von Werten als Konstanten bezieht. Diese Analyse findet in Phase *3* statt, der Übersetzung der Intermediate Sprache in Scala.

3.1 Implementierung nativer Funktionen mit Hilfe einer neuen DSL

Aktuell wird für die Übersetzung einer TeSSLa Spezifikation in eine Ausgabesprache eine Reduktion der Spezifikation vorgenommen, welche alle strombasierten Funktionen der Standart-Library in die neun, nativen Core-Funktionen reduziert (Phase 1 des TeSSLa-Compilers, siehe Kapitel *2.4*). Dies kann in teils sehr umfangreichen Funktionsrümpfen resultieren, wie ein Beispiel zeigt:

Die in TeSSLa, jedoch nicht im TeSSLa-Core enthaltene Funktion *count* inkrementiert für jedes Auftreten eines Events eines übergebenen Stroms einen Zähler. Bei der Verwendung dieser Funktion in einer TeSSLa-Spezifikation wird folgende Umformung vorgenommen:

```
def count[T](x: Events[T]): Events[Int] =
    fold(x, 0, (y: Int, _: T) => y+1)
```

Die count-Funktion erhählt, wie zuvor beschrieben, einen Event-Strom eines beliebigen Typs und gibt einen Integer-Strom aus, dessen Events die Summe der gezählten Events sind. Die vorgenommene Reduktion zeigt, dass die Funktionalität der *count*-Funktion auch durch den Gebrauch eines *fold* ausgedrückt werden kann. *fold* erhält als Argument den Strom *x*, sodass zu jeder Veränderung dieses Stroms die übergebene Funktion (*y: Int*, *_: T*) => *y+1* angewandt wird. Bei erstmaligem Auftreten eines Events auf Strom *x* wird die Funktion mit dem übergebenem initialen Wert *0* für das *y* aufgerufen. Es wird *0+1* berechnet und das Ergebnis der Berechnung dient für den Fall eines weiteren Events auf

Strom x als das neue y. Da *fold* ebenfalls keine nativ implementierte Funktion ist, wird sie aus einer Kombination mehrerer nativer Funktionen zusammengesetzt:

```
fold[T, R](stream: Events[T], init: R, f: (R, T) => R): Events[R] =
    result where {
     def result: Events[R] =
       default(
         lift(last(result, stream), stream, (acc: Option[R],
              curr: Option[T]) =>
           if isNone(curr) then None[R]
           else if isNone(acc) then Some(f(init, getSome(curr)))
           else Some(f(getSome(acc), getSome(curr)))
         ),
         init
       )
}
```

Alle Elemente eines Stroms von Events des Typs T werden mittels einer Funktion f zu einem Element eines Typs R verarbeitet. Der erste am Strom anliegende Wert wird mit dem *init* Argument an f übergeben und verarbeitet. Das Ergebnis wird mit dem nächsten anliegenden Wert verarbeitet. Dabei ist es möglich, dass T und R der selbe Typ sind.
fold ist mit dieser Darstellung im Core-Format. Alle verwendeten Funktionen sind entweder Core-Funktionen oder nicht-strombasierte Funktionen, in diesem Fall *isNone* und *getSome*.

Wie sich erkennen lässt, kann die Implementierung einer vergleichsweise simplen Funktionalität in einem komplexen und umfangreichen Block Code resultieren, der anstelle der Durchführung einer einfachen Addition, mehrere verschiedene Funktionen aufruft. An dieser Stelle des Kompiliervorgangs wurde die erste Optimierung des Systems erkannt. Würden die meist verwendeten, nicht nativ implementierten Funktionen zu den Core-Funktionen hinzugefügt werden, würden umfangreiche Funktionskonstrukte aufgelöst werden. Die Laufzeit der Monitore könnte sich stark reduzieren lassen.

Aktuell wird für die Übersetzung der neun TeSSLa-Core-Funktionen in eine Zwischensprache eine DSL verwendet. Wie in Kapitel *2.4* beschrieben, existieren innerhalb dieser DSL feste Codeblöcke, welche die Funktionalität der Core-Funktionen abbilden. Wird eine, mittlerweile auf TeSSLa-Core reduzierte Spezifikation, in Intermediate Code übersetzt, werden diese festen Codeabschnitte in das Code-Skelett des entstehenden Ausgabeprogramms eingesetzt. Weitere Funktionen ließen sich an dieser Stelle als feste Codeblöcke erstellen.
Das grundlegende Problem der aktuell verwendeten DSL ist, dass sie nicht dafür ausgelegt war, einfach erweiterbar zu sein.

Ein typisches Beispiel ist die Core-Funktion *default*, welche in *StreamCodeGenerator* wie folgt implementiert worden ist:

```
private def produceDefaultStepCode(
    id: Identifier,
    ot: Type,
    stream: ExpressionArg,
    defVal: ExpressionArg,
    currSrc: SourceListing
  ): SourceListing = {
```

Die Signatur der Methode zur Erstellung eines *default*-Code-Blocks. *id* beinhaltet den Namen des Ausgabestroms der *default*-Funktion und seinen Typ *ot*. *stream* und *defVal* sind die beiden Argumente eines *defaults* und *currSrc* ist eine Bündelung aus fünf Listen (ein sogenanntes *SourceListing*), an welche die entstehenden Code-Blöcke der strombasierten, nicht-strombasierten und einiger nötiger Hilfs-Funktionen angefügt werden.

```
val (s, _) = streamNameAndTypeFromExpressionArg(stream)
val o = s"var_${id.fullName}"
val default = nonStreamCodeGenerator.translateExpressionArg(defVal,
                nonStreamCodeGenerator.TypeArgManagement.empty)
```

Im Rumpf der Methode werden zunächst die Konstanten *s* und *o* angelegt, um die Namen der Variablen des Eingabe- und des Ausgabestroms für den Zwischencode zu erzeugen. Anschließend wird *defVal* verwendet, um den Standardwert des Ausgabestroms zu ermitteln.

```
val newStmt = currSrc.stepSource
  .If(Seq(Seq(NotEqual("currTs", LongValue(0)))))
  .Assignment(s"${o}_changed", BoolValue(false),
              BoolValue(true), BoolType)
  .EndIf()
  .If(Seq(Seq(s"${s}_changed")))
  .Assignment(s"${o}_lastValue", s"${o}_value",
              defaultValueForStreamType(ot), ot)
  .Assignment(s"${o}_lastInit", s"${o}_init",
              BoolValue(false), BoolType)
  .Assignment(s"${o}_lastError", s"${o}_error", NoError, ErrorType)
  .Assignment(s"${o}_value", s"${s}_value", default, ot)
  .Assignment(s"${o}_init", BoolValue(true), BoolValue(true), BoolType)
  .Assignment(s"${o}_ts", "currTs", LongValue(0), LongType)
  .Assignment(s"${o}_error", s"${s}_error", NoError, ErrorType)
  .Assignment(s"${o}_changed", BoolValue(true),
              BoolValue(true), BoolType)
  .Assignment(s"${o}_unknown", s"${s}_unknown", BoolValue(false),
     BoolType)
  .EndIf()
```

Um den entstehenden Code an die dafür vorgesehene Liste (in diesem Fall *stepSource*, der ersten der fünf Listen) anfügen zu können, wird die Konstante *newStmt* angelegt. *newStmt* speichert die aktuelle *stepSource*-Liste und fügt neue Statements mittels der *dotted-Notation* an. Die Funktionalität dieses Code-Abschnitts kann in einer Pseudo-Schreibweise folgendermaßen zusammengefasst werden:

```
If(s_changed){
    o = s
}
```

Wobei *o* wie in der *default*-Methode der Ausgabestrom des *default* und *s* der Eingabestrom sind.
Der Grund für die, im direkten Vergleich zur Pseudo-Darstellung, enorme *Boilerplate* ist die Verwendung von internen Werten (*_value*, *_ts*, *_changed*, usw.), welche die Variable *o* betreffen. Diese wurden bereits im Kapitel *2.4* aufgeführt.

```
SourceListing(newStmt, currSrc.tailSource, currSrc.tsGenSource,
              currSrc.inputProcessing, currSrc.staticSource)
```

Abschließend wird der entstehende Code in Form eines neuen *SourceListing* zurückgegeben.

Wie sich der Umfang der genutzten DSL ergibt, wird anhand eines einzelnen *Assignment* deutlich: Die Zuweisung eines Wertes, gespeichert in der Variable *s*, an eine Variable *o* und die anschließende Aufnahme dieses Statements in den entstehenden Code resultiert in der folgenden Zeile DSL-Code:

```
.Assignment(s"${o}_value", s"${s}_value", defaultValueForStreamType(ot),
            ot)
```

Zu erkennen ist, dass einem Assignment der Typ *ot* der Variable *o* und ein Standardwert *defaultValueForStreamType*(*ot*) mit übergeben werden müssen. Dies führt bei Standardzuweisungen zu Redundanz. Beispielsweise wird der Timestamp-Wert einer Variable (*_ts*) oft durch *currTs*(den aktuellen Zeitstempel) ersetzt. Jede dieser Zuweisungen muss in der aktuellen DSL mit Angabe des Standardwertes und des Typs geschrieben werden.

Einen weiteren Aspekt des Umfangs stellen If-Statements dar. If-*Guards* der DSL akzeptieren nur aussagenlogische Formeln in *disjunktiver Normalform* (DNF). DNF bezeichnen in der boolschen Algebra die normierten Funktionsdarstellungen in Form einer Disjunktion aus Konjunkstionstermen. Einzelne Variablen dürfen dabei negiert werden.
Zu jeder Funktion existiert eine äquivalente Funktion in DNF-Schreibweise. Ein Beispiel:

$$A \Leftrightarrow B \wedge (\neg C \rightarrow A)$$

Lässt sich in DNF wie folgt darstellen:

$$(C \wedge B \wedge A) \vee (A \wedge B) \vee (\neg B \wedge \neg A) \vee (\neg C \wedge \neg A)$$

Um dies innerhalb der DSL darzustellen, müsste folgendes geschrieben werden:

```
.If(Seq(Seq(C,B,A), Seq(A, B), Seq(Negation(B), Negation(A)),
    Seq(Negation(C), Negation(A))))
```

Der Guard besteht aus einer Sequenz, welche wiederum aus Sequenzen von Variablen oder Ausdrücken wie *Negation*(*Variable*) besteht, welche als *Literale* bezeichnet werden. Die Literale innerhalb der innersten Sequenzen werden intern verundet und geklammert, sodass Konjunktionsterme, auch *Monome* genannt, entstehen. Die einzelnen Monome werden mit der Darstellung als Sequenz verodert und es entsteht ein einzelner Disjunktionsterm (*Klausel*). Der Schreibaufwand für Funktionen in DNF ist höher und der/die Programmierende muss jede boolsche Funktion vorher in eine äquivalente DNF umformen.

Falls eine Konstante angelegt werden soll, wird dies in gegebener DSL folgendermaßen realisiert:

```
.FinalAssign(s"${o}_changed", BoolValue(false), BoolType)
```

In diesem Beispiel wird sich die Variable *o* nicht verändern und somit ist der *changed*-Wert von *o* konstant *false*. Da jedoch im Vorfeld bekannt sein muss, ob es sich um eine Variable oder Konstante handelt, ist die Anwendung aufwendig.

Für die Umsetzung der ersten Optimierung ist es dementsprechend sinnvoll, eine DSL mit Fokus auf Lesbarkeit und Umfangsreduzierung zu entwerfen. Funktionen lassen sich mit dieser Erweiterung zukünftig zeitsparend und in einem sprachlich natürlichen Stil einpflegen.
Die *fold*-Funktion ließe sich fortan innerhalb eines *extern*-Aufrufes formulieren.

```
def fold[T,R](stream: Events[T], init: R, f: (R, T) => R): Events[R] =
    extern("fold")
```

Dieser Ausdruck bedeutet, dass das Matching der Funktion *StreamCodeGenerator.translateExternSIgnalExpression* um den Funktionsnamen *fold* erweitert wird. Dadurch wird *fold* wie eine Core-Funktion behandelt.

Die Verwendung mehrerer geschachtelter Funktionsaufrufe ist ressourcenintensiver, sodass eine direkte Implementierung der Funktionen eine Zeitersparnis darstellen sollte.
Im Laufe dieser Arbeit werden neben *fold* die Funktionen *count*, *boolFilter*, *const*, *filter*, *defined*, *reduce*, *unitIf*, *average* und *pure* nativ implementiert, um besser auf entstandene Verbesserungen eingehen zu können. Hierzu mehr im Kapitel *Evaluation* und im Anhang *A*.

3.2 Var-Val-Analyse

Im generierten Scala-Code werden Variablen als veränderbar angelegt (*var*). Einige Variablen verändern sich während der Laufzeit eines Monitors eventuell nicht und können deshalb konstant sein.
Die *Var-Val-Analyse* ist eine Erweiterung, für die Erkennung von möglichen Konstanten. Da die Nutzung von Konstanten im Allgemeinen ressourceneffizienter ist als die von Variablen, soll untersucht werden, ob eine Ersetzung von Variablen durch Konstanten zu einer Leistungssteigerung der Monitore führen kann. Die Verwendung von Konstanten würde im Übrigen zu einer Steigerung der Lesbarkeit führen.

Im funktionalen Teil von TeSSLa können nur Konstanten definiert werden. Damit wäre es prinzipiell möglich, jede Variable als Konstante zu definieren.
Um einen Wert x in Scala initialisieren zu können, müssen alle anderen Werte, die x verwendet, bereits definiert sein. Wird folglich ein Scala-Programm aus einer TeSSLa-Spezifikation erzeugt, gilt folgende Aussage: Verwendet ein Wert x einen anderen Wert y, muss y vor x initialisiert werden, damit x eine Konstante sein kann. Aus dieser Tatsache lässt sich folgern, dass die Reihenfolge der Initialisierungen sortiert werden kann, um möglichst viele Werte als Konstanten zu deklarieren.
Die einzige Situation, in der ein Wert nicht konstant sein kann, ist im Fall einer rekursiven Abhängigkeit. Der Wert x verwendet, wie zuvor, den Wert y und dieser verwendet wiederum den Wert x. Da einer der beiden Werte vor dem anderen initialisiert werden muss, muss der jeweils andere für die Initialisierung verfügbar sein. Aus diesem Grund wird einer der beiden Werte mit einem Platzhalterwert vorinitialisiert (z.B. für Integer *0*).

Folgende TeSSLa-Spezifikation führt die zwei Funktionen *even* und *uneven* ein, mit deren Hilfe sich bestimmen lässt, ob eine Zahl gerade oder ungerade ist, abhängig von der zuerst aufgerufenen Funktion. Diese beiden Funktionen sind rekursiv voneinander abhängig. Weiterhin werden in diesem Beispiel die Werte *valueZero* und *valueOne* verwendet. Diese werden von den Funktionen verwendet, benötigen jedoch keine weiteren Werte innerhalb ihrer Initialisierung.

```
def valueZero = 0
def valueOne = 1
def even(i: Int): Bool = if i == valueZero
                         then true
                         else uneven(i-valueOne)
def uneven(i: Int): Bool = if i == valueZero
                           then false
                           else even(i-valueOne)
```

Sei *even* die Funktion, welche bei der Übersetzung in Scala-Code zuerst initialisiert wird. Für die Initialisierung von *even* wird die Funktion *uneven* benötigt, weshalb diese vor *even* initialisiert werden muss. Da bei der Initialisierung von *uneven* wiederum festgestellt wird, dass *even* verwendet wird und für diese Funktion bereits versucht wurde, sie zu initialisieren, wird *uneven* als *var* mit dem Platzhalterwert *null* initialisiert. Anschlie-

ßend lässt sich *even* als Konstante initialisieren und die Funktionalität von *uneven* wird nach dieser Initialisierung nachgeliefert. *valueOne* und *valueZero* verwenden keine weiteren Werte und können konstant sein.

```
val valueZero = 0
val valueOne  = 1
var uneven = null
val even = {if i == valueZero true else uneven(i-valueOne)}
uneven = {if i == valueZero false else even(i-valueOne)}
```

Die Var-Val-Analyse bezieht sich nur auf die Erzeugung von Monitoren als Scala-Programme. Sie könnte jedoch für jegliche Sprachen verwendet werden, welche zwischen Variablen und Konstanten unterscheiden. Sie wird aus diesem Grund Teil der Übersetzung des Intermediate Code in die Ausgabesprache sein (Phase *3* des Kompiliervorgangs, siehe Kapitel *2.4*).

4

Implementierung der Optimierungen

Dieses Kapitel gibt einen Überblick über die durchgeführten Techniken zur Implementierung der zwei Optimierungen. Der Abschnitt *Domain Specific Language* präsentiert die vorgenommenen Änderungen am TeSSLa-Compiler, um die Implementierung weiterer Core-Funktionen zu ermöglichen. In *Var-Val-Analyse* wird auf die Umsetzung der gleichnamigen Optimierung näher eingegangen.

4.1 Domain Specific Language

Die aktuelle Domain Specific Language befindet sich innerhalb des Objekts *IntermediateCode* und der Klasse *IntermediateCodeUtils*. Die Funktionalität der DSL soll für die Übersetzung in die Intermediate Sprache beibehalten werden. Daher wurden diese Klassen beibehalten und eine neue DSL wurde erstellt, welche auf ihre Funktionen zugreift. Auf diese Weise lassen sich einige Standards einführen, sodass sich die *Boilerplate* drastisch reduzieren lässt.
Die Klasse *StreamCodeGenerator* war für die Erstellung der nativ implementierten Funktionen zuständig. Anstatt die DSL aus *Intermediate Code* zu verwenden, wird innerhalb der erzeugenden Methoden fortan eine neue DSL verwendet. *StreamCodeGenerator* wurde daher durch die Klasse *DSLStreamCodeGen* ersetzt.
Um die Logik für die neue DSL zu speichern, wird ein Objekt *DSLHighLevelAST* angelegt. Da die DSL in jedem Kontext gleich bleibt und nur eine Instanz benötigt wird, kann ein Objekt angelegt werden. Dieses Objekt speichert die aufgerufenen Core-Funktionen als *Abstract Syntax Tree*(AST), wodurch eine neue Phase zwischen der zweiten und der dritten Phase eingefügt wird.

Zu Beginn werden grundlegende Befehle wie die Zuweisung eines Wertes an eine Variable definiert. Dafür lassen sich *case classes* in Kombination mit einem *implicit* verwenden.

```
final case class Assign(lhs: String) extends DSLfunc {
    def :=(r: Expressions): Assign
}
implicit def toExpression(s:String)
implicit def stringToAssign(s:String) = Assign(s)
```

Um beliebige Ausdrücke darstellen zu können, existiert die abstrakte Klasse *Expressions*, welche zur Repräsentation von Variablen, Boolscher-Logik und Lambdas als Ausdrücke einer Zuweisung verwendet wird. Werden konkrete Werte verwendet, werden diese mit den in *IntermediateCode* definierten *ImpLanVal*s dargestellt. Beispielsweise lässt sich ein *LongValue*(*0*) schreiben, um eine 0 vom Typ *Long* zu beschreiben. Mittels dieser Definitionen lässt sich nun ein

```
.Assignment(s"${o}_value", s"${s}_value", LongValue(0), LongType)
```

in ein äquivalentes

```
Define(o) ofType LongType
s"$o.value" := s"$s.value"
```

umwandeln. Der Typ der Variable *o* lässt sich auslagern, indem *o* vor Verwendung definiert wird. Der Typ der Variable wird an das Objekt *DSLHighLevelAST* übergeben und der passende Standardwert (in diesem Fall *LongValue*(*0*)) intern ermittelt.

Zu jeder angelegten Variable werden intern neun Werte gespeichert, um für Strom-Funktionen relevante Informationen zu speichern, wie bereits in Kapitel *2.4* erläutert. Sollen im selben Scope alle diese Werte gleichzeitig und standardmäßig gesetzt werden, wird folgender Block erstellt:

```
.Assignment(s"${o}_lastValue",s"${o}_value",defaultValueForStreamType(ot),ot)
.Assignment(s"${o}_lastInit", s"${o}_init", BoolValue(false), BoolType)
.Assignment(s"${o}_lastError", s"${o}_error", NoError, ErrorType)
.Assignment(s"${o}_value", s"${s}_value", default, ot)
.Assignment(s"${o}_init", BoolValue(true), BoolValue(true), BoolType)
.Assignment(s"${o}_ts", "currTs", LongValue(0), LongType)
.Assignment(s"${o}_error", NoError, NoError, ErrorType)
.Assignment(s"${o}_changed", BoolValue(true), BoolValue(true), BoolType)
.Assignment(s"${o}_unknown", s"${s}_unknown", BoolValue(false), BoolType)
```

Mit Hilfe der definierten *Assign case-class* lässt sich dies wie folgt schreiben:

```
Define(o) ofType ot
o := s"$s.value"
```

Durch das Weglassen eines *.internerWert* wird nur die Variable *o* ohne einen speziellen internen Wert angesprochen. Dies führt zu einer Zuweisung eines Standardwerts an jeden

internen Wert. Ein Beispiel wäre *BoolValue*(*true*) für *_changed*. Die Variable wurde verändert, wodurch *_changed* auf *true* gesetzt werden kann. Sollte es vorkommen, dass einer der Standardwerte bei einer solchen Blockzuweisung nicht der richtige ist, lässt sich der korrekte Wert noch nachträglich überschreiben:

```
Define(o) ofType ot
o := s"$s.value"
s"$o.error" := s"$s.error"
```

Für den Fall, dass der *_error*-Wert für *o* nicht auf *NoError* gesetzt werden soll, sondern explizit auf den *_error*-Wert von *s*, wurde eine Funktion entworfen, welche Mehrfachzuweisungen innerhalb eines Scopes entfernt.
Um eine Hilfsvariable zu verwenden, bei der nur der Wert und sonst die weiteren Informationen nicht von Interesse sind, lässt sich die Methode *asSingleVar* der *Define case class* nutzen:

```
Define(tmp) asSingleVar ofType LongType
tmp := 42
```

Die ursprüngliche Fassung ermöglichte das Anlegen von Konstanten mittels des Befehls

```
.FinalAssignment(Variable(s"${o}_ts"), LongValue(0), LongType)
```

unter der Prämisse, dass bekannt ist, dass diese an keiner anderen Stelle anders gesetzt werden. Statt einer direkten Implementierung eines solchen Befehls, wird nach Fertigstellung des AST ein Algorithmus angewandt. Dieser iteriert über den AST, markiert mögliche Konstanten (sie werden nie überschrieben) und ersetzt sie letztlich durch *.FinalAssignment*s.

Die Guards der If-Statements sind ein wesentlicher Faktor der Boilerplate der DSL. Der Grund ist die Darstellungsweise als DNF. Das Beispiel aus Kapitel *3.1*: Die aussagenlogische Formel

$$A \Leftrightarrow B \wedge (\neg C \rightarrow A)$$

wird in DNF umgeformt und schließlich in DSL implementiert:

```
.If(Seq(Seq(C,B,A), Seq(A, B), Seq(Negation(B), Negation(A)),
    Seq(Negation(C), Negation(A))))
```

Indem eine *case class* für ein If eingeführt wird und *Expressions* verwendet wird, lassen sich die Guards abkürzen:

```
 If((s"$v.ts" === "currTs") &&& s"$v.lastInit"
    ||| (s"$v.ts" !== "currTs") &&& s"$v.init")
```

Scala erlaubt es üblicherweise Funktionsnamen frei zu vergeben. Leider entstehen in der Implementierung Probleme, welche beispielsweise die Verwendung eines + unmöglich machen. Das bereits betrachtete Beispiel verdeutlicht dies:

```
o := s"$s.value"
```

Die Zuweisung der Stromausgabe *o* erhält den Wert eines Eingabestroms *s*. Ist dieser vom Typ Integer, lässt sich *o* auch der Wert von *s* um eins inkrementiert zuweisen:

```
"tmp" = 1
o := s"$s.value" + "tmp"
```

Da die Funktion der String-Konkatenation in Scala durch ein + repräsentiert, liegen dem Compiler an dieser Stelle zwei Interpretationsmöglichkeiten vor. Es lässt sich nicht entscheiden, ob die String-Konkatenation, oder das neu definierte + für die Addition zweier Werte zu einem neuen Event im Kontext einer TeSSLa-Spezifikation gemeint ist. Da das Problem hauptsächlich auf die Verwendung von Strings als Namen für Strom-Variablen zurückzuführen ist, sollte das Problem mittels OOP behoben werden können. Da dies jedoch bereits die Art der Implementierung in der ursprünglichen Version des Systems war, und das implizite Casten von Strings zu *Expressions* wesentlich einfacher war, wurde zunächst darauf verzichtet und auf eine drei-Symbol-lange Schreibweise (+++) zurückgegriffen. Dies signalisiert eindeutig, dass es sich um den Kontext der neuen DSL handelt.
Das *DSLHighLevelAST*-Objekt ermöglicht es nun die Klasse *StreamCodeGenerator* durch die Klasse *DSLStreamCodeGen* zu ersetzen. Die Methode *defaultStepSource* in *StreamCodeGenerator* ist die native Implementierung eines in TeSSLa-Core enthaltenen *default* (*3.1*). Durch die Verwendung der neuen DSL lässt sich *default* kürzer fassen:

```
val (s, _) = streamNameAndTypeFromExpressionArg(stream)
val o = s"var_${id.fullName}"
val default = nonStreamCodeGenerator.translateExpressionArg(defVal,
                nonStreamCodeGenerator.TypeArgManagement.empty)

Define(o) withDefault Some(default) ofType ot
If("currTs" !== LongValue(0))
s"$o.changed".:=(BoolValue(false)).standartTrue
EndIf
If(s"$s.changed")
  o.:=(s"$s.value").standartTrue
  s"$o.error" := s"$s.error"
  s"$o.unknown" := s"$s.unknown"
EndIf

currSrc.copy(stepSource = as(stepSource))
```

Die Klasse *DSLStreamCodeGen* wurde um die Funktionen *boolFilter*, *count*, *const*, *filter*, *defined*, *reduce*, *unitIf*, *average*, *pure* und *fold* erweitert.

4.2 Var-Val-Analyse

Um möglichst viele Werte als konstant deklarieren zu können, muss bekannt sein, welche weiteren Werte ein Wert verwendet. Darüber hinaus müssen die Ergebnisse gegebenenfalls sortiert werden. Folglich wird eine Möglichkeit benötigt, welche diese Abhängigkeiten darstellt.
Map ist, wie in *2.5* aufgeführt, eine Datenstruktur in Scala. *Maps* erlauben es zu einem Schlüssel einen Wert zu speichern. Diese Paare werden dann auch *Mappings* oder *Assoziationen* genannt. Eine *Map*, im Folgenden *Usage-Map* genannt, wird für die Werte einer TeSSLa-Spezifikation in Zwischensprache angelegt (Phase 3 des TeSSLa-Compilers vor Erweiterung um die DSL als AST). Diese *Map* speichert jeden Wert der Spezifikation einmal als Schlüssel. Jeder Schlüssel verweist auf die von diesem Wert verwendeten Werte. Es wird das aus *3.2* bekannte Beispiel wieder aufgegriffen:

```
def valueZero = 0
def valueOne = 1
def even(i: Int): Bool = if i == valueZero
                         then true
                         else uneven(i-valueOne)
def uneven(i: Int): Bool = if i == valueZero
                           then false
                           else even(i-valueOne)
```

Für dieses Beispiel wird folgende *Usage-Map*, zur Vereinfachung als Pseudocode, angelegt:

```
Map( "valueZero" -> (),
     "valueOne"  -> (),
     "even"      -> ("valueZero", "valueOne", "uneven"),
     "uneven"    -> ("valueZero", "valueOne", "even"))
```

Die *Usage-Map* ermöglicht es mittels einer $getValue(x)$ Funktion die Werte eines übergebenen Schlüssels x zu erhalten. Dies ermöglicht es, dessen verwendete Variablen rekursiv prüfen zu können. Sollte eine Schleife entstehen, bei der ein Wert schließlich ein zweites mal betrachtet wird, so muss dieser Wert variabel sein. Dieses Verfahren ist im folgenden Pseudocode dargestellt:

```
init listVars, listVals: List[Identifier] = List.empty()

checkVarVal(x: Identifier, stack: Set[Identifier])
        if( x in stack )
                listVars += stack
        end if
        if( x not in listVars && x not in listVals )
                for each value in usageMap.getValue(x)
                    checkVarVal(v, stack ++ x)
                if( x not in vars )
                        listVals += stack
                end if
        end if

for each key in usageMap
    if(key not in listVars && key not in listVals)
        checkVarVal(key)
    end if
```

Zwei Listen, *listVars* und *listVals*, speichern die später als *var*, bzw. *val* zugeordneten Werte. Ist ein Wert bisher weder *listVars*, noch *listVals* zugeordnet worden, wird er innerhalb des *foreach* der *usageMap* als Argument der Methode *checkVarVal* übergeben. Weiterhin wird dieser Funktion ein leeres Set bei Aufruf übergeben, welches der Rekursionserkennung dient. Ist das übergebene Argument nicht im Set enthalten, wird für jeden Wert, den das Argument verwendet, die Methode *checkVarVal* rekursiv aufgerufen. Das übergebene Set wird um den aktuell betrachteten Wert erweitert. Für das Beispiel aus *3.2* wird *checkVarVal*(*even*, ()) aufgerufen. *even* ist bisher weder *var* noch *val*. Jede, von *even* verwendete Variable, wird nacheinander an *checkVarVal* übergeben. Da jeweils *valueOne* und *valueTwo* keine weiteren Werte benötigen, werden sie in *listVals* gespeichert. Wird als drittes *uneven* an die Methode übergeben und werden dessen verwendete Werte rekursiv geprüft, bleiben die Werten *valueOne* und *valueTwo* unverändert. Diese sind bereits in *listVals* vorhanden. Wird jedoch das *even* in *uneven* erreicht und wird dieses wieder an *checkVarVal* mit *uneven* im Set übergeben, wird erkannt, dass eine Rekursion vorliegt. Dementsprechend wird *even* in *listVars* eingefügt und bei der Übersetzung schließlich als variabel definiert. Folgender Scala entsteht:

```
val valueZero = 0
val valueOne  = 1
var uneven = null
val even = {if i == valueZero true else uneven(i-valueOne)}
uneven = {if i == valueZero false else even(i-valueOne)}
```

5

Evaluation

Für die Evaluation des Performance-Gewinns der Optimierungen werden die Ausführungszeiten von sechs Monitoren mit und ohne Optimierungen gemessen und verglichen. Drei Spezifikationen beinhalten eine einzelne, dem Core neu hinzugefügten Funktion und sind damit triviale Beispiele. Die letzten drei Spezifikationen sind realitätsnahe Beispiele.

boolFilter - filtert einen Boolean-Strom, sodass ein Strom nur mit Events des Wertes *true* entsteht.

count - wie in vorherigen Kapiteln erläutert, zählt die *count*-Funktion Events eines übergebenen Stroms, indem ein Zähler inkrementiert wird. Die Events des entstehenden Stroms sind die Werte der Zählervariable.

average - bestimmt den Durchschnitt der Werte eines Integer-Stroms.

accSum - Beispiel aus [21]. Akkumuliert die Werte eines Stroms *values* ab des letzten *reset* in einem Strom *sum*. Verwendet unter anderem *const*, *pure* und *filter*.

filterByTime - Beispiel aus [21]. Filtert auftretende Events in einem bestimmten, zeitlichen Muster. Verwendet unter anderem *const*, *pure* und *filter*.

election - Ein Beispiel aus dem Kurs *Development of Safety-Critical Software*. Events stellen Wahlergebnisse eines Bundeslands dar. Ausgaben sind unter anderem Koalitionsbildungen und 5%-Hürden. Verwendet die Funktionen *fold* und *filter*.

Jeder erstellte Monitor wird sechsfach getestet:
1. Ohne jegliche Optimierungen oder zusätzliche Einstellungen. Um einen Vergleichswert zu schaffen, werden die Monitore auf Grundlage des ursprünglichen Systems getestet. Dieser Wert stellt den Vergleichswert der anderen Messungen dar.

2. Unter Hinzunahme der Var-Val-Analyse. Wie bereits im Vorfeld erörtert, wird davon ausgegangen, dass die Erweiterung des Systems um die Var-Val-Analyse keine Verbes-

serungen der Laufzeit mit sich bringt. Da die definierten Spezifikationen in Monitoren als Scala-Programme resultieren, wird ein Monitor zwangsläufig zu Java-Bytecode kompiliert und von einer JVM ausgeführt. Die verwendete JVM OpenJDK nutzt einen JiT-Compiler, welcher aufgrund seiner Funktionsweise selbst zwischen Konstanten und Variablen unterscheiden können sollte. Es wird erwartet, dass die Ergebnisse dieser Tests ähnlich zu denen aus Punkt 1. ausfallen.

3. Mit der neuen DSL und der einhergehenden Erweiterung des TeSSLa-Core um die Funktionen *fold*, *count*, *boolFilter*, *const*, *filter*, *defined*, *reduce*, *unitIf*, *average* und *pure*. Es wird davon ausgegangen, dass sich durch die Auflösung von Funktionsschachtelungen Laufzeitverbesserungen abzeichnen.

4. Unter Hinzunahme beider Optimierungen. Sollten beide Optimierungen individuelle Laufzeitverbesserungen ergeben, wird in dieser Testreihe geprüft, ob diese in Zusammenhang stehen.

5. Ohne Optimierungen, aber mit den Optimierungsmöglichkeiten des Scala-Compilers. Die Kompilierung durch den Befehl *scalac <Dateiname>* übersetzt ein Scala-Programm in Java-Bytecode. Optional kann dieser Vorgang mit weiteren Befehlen modifiziert werden. Mit Hilfe der Modifikation *-opt*, gefolgt von einer spezifischen Optimierungseinstellung, führt der Compiler bis zu 16 Optimierungen auf dem zu kompilierenden Code durch [23]. Innerhalb dieser und der noch folgenden Testreihe werden die Monitore mit allen im Umfang des Scala-Compilers enthaltenen Optimierungsmöglichkeiten getestet. Der Compiler könnte bereits Optimierungen vornehmen, welche durch die vorgenommenen Optimierungen ebenfalls entstanden sind. Dies wird in dieser Serie an Tests überprüft.

6. Mit beiden Optimierungen und den Optimierungsmöglichkeiten des Scala-Compilers. Falls sowohl die Optimierungen, als auch die Einstellungen des Scala-Compilers in Laufzeitverbesserungen resultieren, wird in diesem Schritt eine Abhängigkeit untersucht.

Die Monitore werden mit Eingaben von 1000, bis zu 50 Milliarden Events getestet. Wie in 2.7 erläutert, wird bei einem JiT Compiler zwischen einer *Start-Up-* und einer *Steady-*Phase unterschieden. Aus diesem Grund wird der JiT Compiler mit 1000 eintreffenden Events, welche nicht in die Messung hinein zählen, vorbelastet. Dies soll nötige Speicherallokationen und Initialisierungen vorwegnehmen, um den *steady state* zu erreichen und um keine Verzerrungen der Laufzeit entstehen zu lassen. Die Laufzeit wird anschließend ab Eintreffen des ersten, neuen Events bis zur Verarbeitung des letzten Event in Nanosekunden gemessen. Um die Monitore möglichst zufällig und damit realitätsnah zu testen, wurden Werte auf Grundlage des für die zu durchlaufende Schleife notwendigen Zählers erstellt.

Vergleich der Ergebnisse

Die folgenden Ergebnisse werden in Prozent im Vergleich zu den Ergebnissen der unoptimierten Monitore des original Compilers angegeben. Repräsentativ hierfür wurde der Durchschnitt über die Ergebnisse aller durchgeführten Tests gebildet. Blau steht für die Laufzeit der Testreihe ohne jegliche Optimierungen (1), und wird somit immer 100% erreichen. Rot repräsentiert die Laufzeit der Monitore unter Hinzunahme der Var-Val-Analyse (2), schwarz die Laufzeiten der Monitore mit der neuen DSL und dem erweiterten TeSSLa-Core (3). Gelb zeigt die Ergebnisse der Monitore mit Verwendung der Var-Val-Analyse und der neuen DSL (4). Grün gibt Auskunft über die Ergebnisse der Verwendung von *scalac -optimize* zu einem Monitor ohne die Optimierungen DSL oder Var-Val-Analyse (5), lila hingegen mit DSL und Var-Val-Analyse (6):

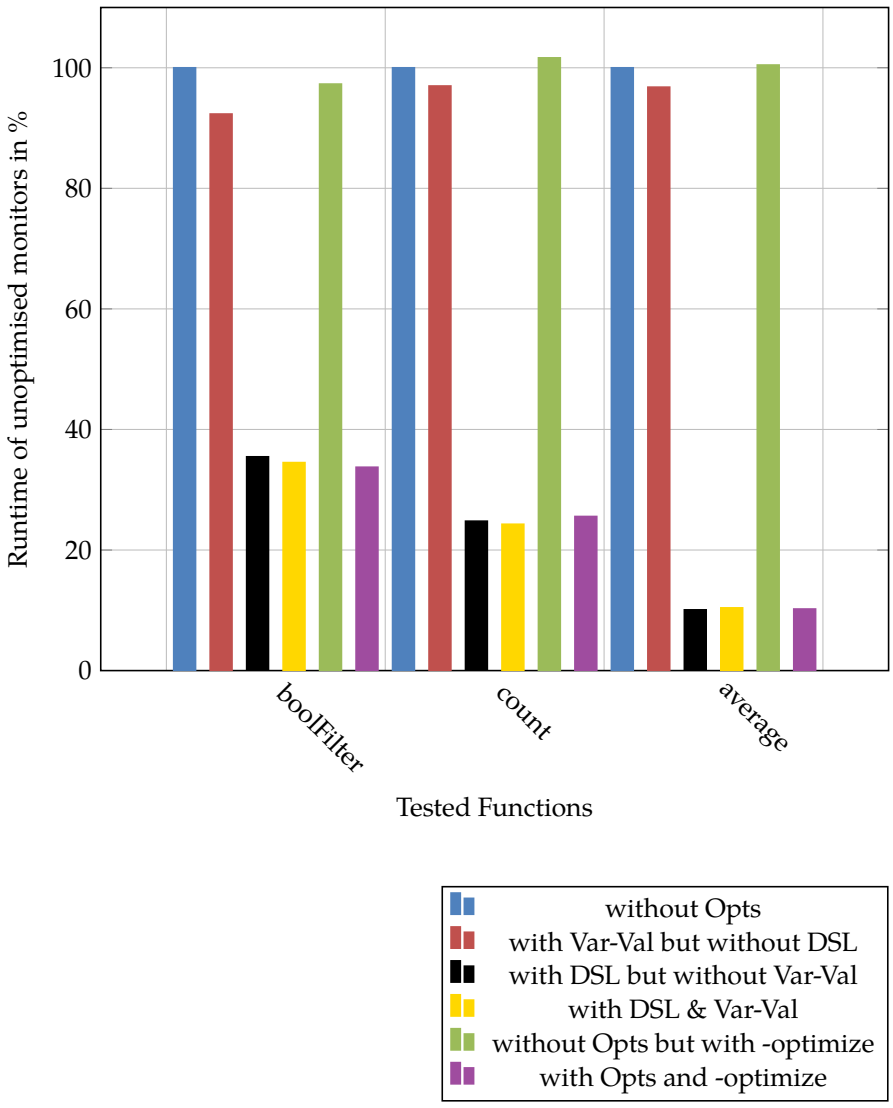

Abbildung 5.1: Messungen der Monitore trivialer Spezifikationen. Ergebnisse bilden den Durchschnitt über alle getätigten Tests eines Monitors (1000 - 50 Milliarden Events).

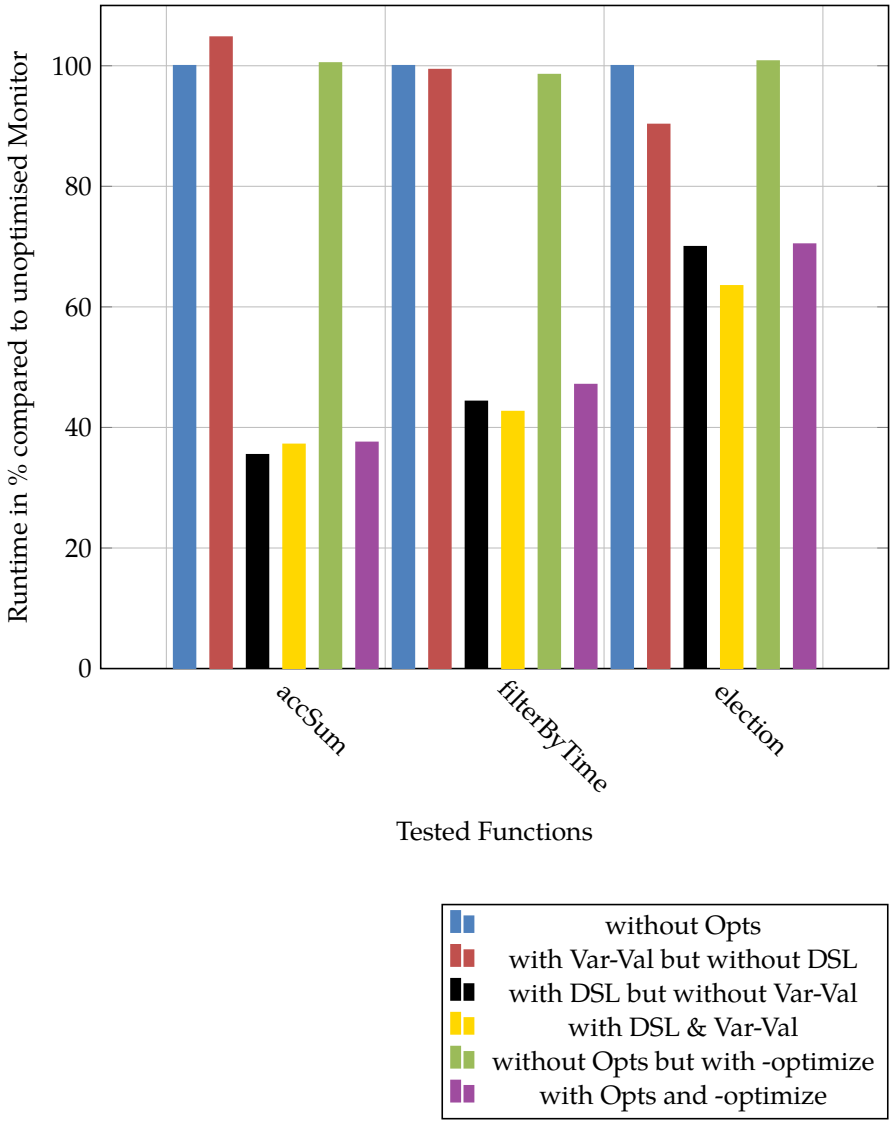

Abbildung 5.2: Messungen der Monitore realitätsnaher Spezifikationen. Ergebnisse bilden den Durchschnitt über alle getätigten Tests eines Monitors (1000 - 15 Millionen Events).

Genaue Angaben zu den Laufzeiten lassen sich im Anhang *A* finden. Die Var-Val-Analyse, einzeln dargestellt über die rote Säule, lässt keine Verbesserungen erkennen. Die Verwendung von Konstanten reduziert dennoch den Umfang der Monitore, da Vorinitialisierungen vermieden werden können. Die Erweiterung resultiert letztendlich in einer gesteigerten Lesbarkeit.

Die Optimierung um die hinzugefügte DSL schlägt sich in Laufzeitverbesserungen von bis zu 90% nieder. Da die neue DSL eine gesteigerte Lesbarkeit aufweist und wesentlich einfacher zu warten ist, resultiert sie folglich ebenfalls in einer gesteigerten Gebrauchstauglichkeit und Wartbarkeit des TeSSLa-Compilers. Die Änderungen erlauben es das

System auch in Zukunft um weitere Funktionen einfach erweitern zu können.

Die Einstellungen des Scala-Compilers zur Optimierung der Eingabeprogramme erzielten ebenfalls keine signifikanten Verbesserungen.

Die Länge der Programme konnte unter Verwendung beider Optimierungen um bis zu 72% reduziert werden. Dies ermöglichte es, einen für die Tests erzeugten Monitor zu kompilieren, bei welchem der Scala-Compiler vorher aufgrund seiner Länge den Fehler *class too long* ausgab.

Zusammenhang des JiT Compilers

Wie bereits im Vorfeld vermutet wurde, führt die Var-Val-Analyse zu keiner Verbesserung der Laufzeit der Monitore. Dies basiert auf der Tatsache, dass der JiT Compiler der verwendeten JVM OpenJDK aufgrund mehrfacher Iteration des Codes zwischen Konstanten und Variablen unterscheiden kann. Da der Monitor nun bereits vor der JiT Kompilierung zwischen Konstanten und Variablen unterscheiden kann, ließen sich die zu erkennenden Schwankungen der Laufzeiten (Abbildungen *5.1* und *5.2*) auf den nicht-Determinismus des JiT Compilers zurückführen. Die Warm-Up Phase des Systems wird wahrscheinlich von den Änderungen beeinflusst werden, sodass Schwankungen entstehen.

Die verschiedenen Optimierungsstufen des JiT Compilers lassen sich an den Messungen erkennen. Wie in Abbildung *5.3* dargestellt, steigt die Laufzeit der Monitore anfangs nicht linear mit der wachsenden Anzahl eintreffender Events. Ab einem gewissen *Threshold* scheint der JiT Compiler die letzte Stufe der Optimierung erreicht zu haben. Ab Testbereichen von 25 Millionen Events beginnt sich die Laufzeit zu stabilisieren und linear weiter zu wachsen.

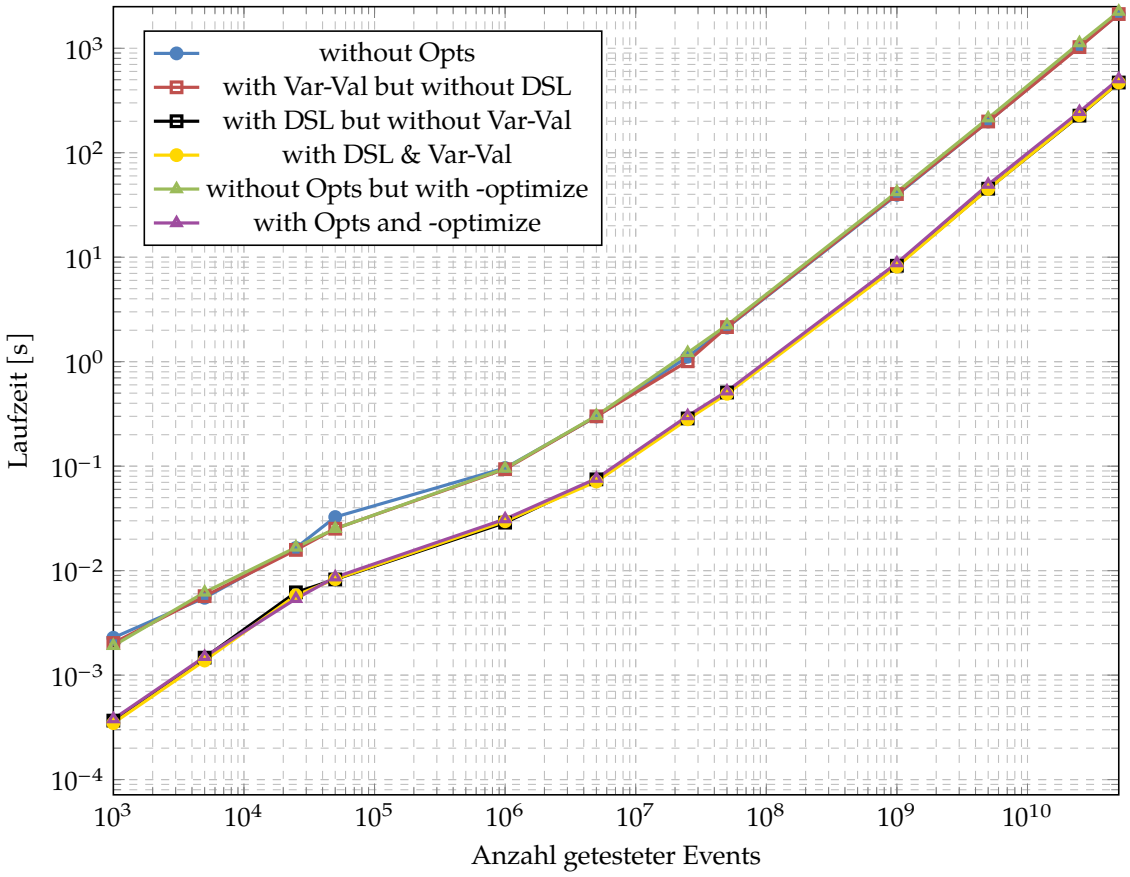

Abbildung 5.3: Darstellung der Ergebnisse von *count.tessla*. Es wurden bis zu 50 Milliarden Events getestet, mit einer maximalen Laufzeit von 2244,3 Sekunden.

Von besonderem Interesse ist der Testfall *election.tessla*. Ab einer Anzahl von fünf Millionen Events erbringt selbst die Optimierung um die neue DSL keine Verbesserungen mehr. Dies ist eine Anomalie, die bei keinem der anderen Testfälle auftrat.

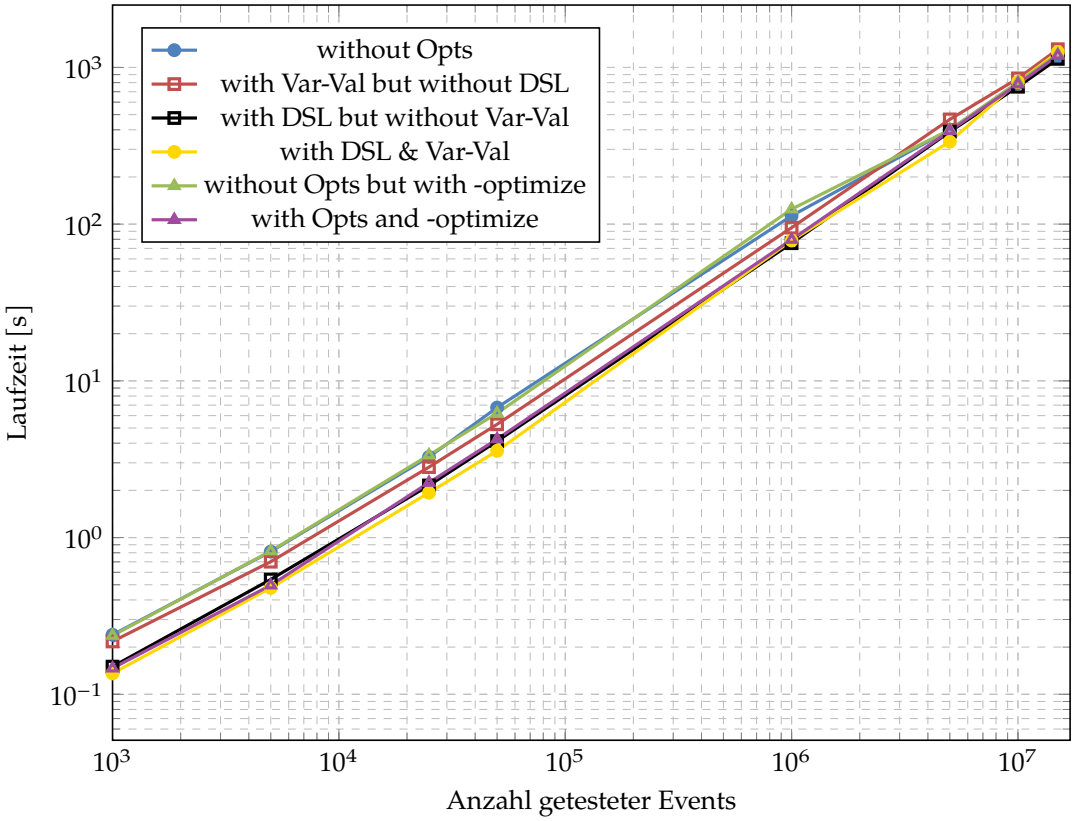

Abbildung 5.4: Darstellung der Ergebnisse von *election.tessla*. Es wurden bis zu 15 Millionen Events getestet; mit einer maximalen Laufzeit von 1306,044 Sekunden. Während der mit DSL optimierte *election.tessla*-Monitor zu Beginn nur 56 % der Laufzeit des unoptimierten Monitors benötigte, ist dessen Laufzeit in einem Bereich von 10 Millionen Tests gleich der Laufzeit des unoptimierten Monitors.

Genaue Angaben der Laufzeiten lassen sich dem Anhang *A* entnehmen. Die restlichen Laufzeiten der Monitore sind in Anhang *B* grafisch dargestellt.

6

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, den TeSSLa-Compiler um Optimierungen zu erweitern und damit die erzeugten Monitore zu beschleunigen. Zwei mögliche Optimierungen wurden umgesetzt.

Die erste durchgeführte Optimierung bezieht sich auf die Erweiterung des TeSSLa-Core um weitere Funktionen. Der TeSSLa-Core ist eine Sammlung aus neun Funktionen, aus denen jede weitere, in TeSSLa enthaltene, Funktion aufgebaut werden kann. Diese Sammlung wird für den Kompiliervorgang benötigt. Da nicht-Core-Funktionen durch Schachtelungen der Core-Funktionen aufgebaut werden, entstehen teils sehr umfangreiche Funktionsverkettungen. Durch die direkte Implementierung weiterer Funktionen wurde die Sammlung der Core-Funktionen erweitert, sodass sich diese Verkettungen auflösen lassen.
Dies wird durch den Einsatz einer DSL ermöglicht. Ursprünglich verwendete das System für die Übersetzung des TeSSLa-Core in eine Intermediate Sprache bereits eine DSL. Diese DSL war jedoch sehr komplex und produzierte einen gewaltigen Overhead bei der Implementierung von Funktionen. Diesbezüglich wurde eine neue DSL erstellt, welche die Funktionalität der alten beibehielt, jedoch im Umfang drastisch reduziert wurde. Mithilfe dieser DSL wurden zehn weitere Funktionen dem TeSSLa-Core hinzugefügt, wobei die Anpassungen zu einer Reduzierung der Laufzeit um bis zu 90% führten.

Die zweite durchgeführte Optimierung des Systems war die Erweiterung um eine Analyse auf mögliche konstante Initialisierungen. Ursprünglich konnte der TeSSLa-Compiler nur Variablen erzeugen. Eine Analyse auf rekursive Abhängigkeiten zwischen Werten ermöglichte es dem System Konstanten anzulegen, wenn dies möglich war. Messungen ergaben, dass diese Analyse keine signifikanten Verbesserungen der Laufzeit liefert. Die Analyse führt dennoch zu kürzeren Ausgabeprogrammen.

Abschließend wurde untersucht, ob der Scala-Compiler bei der Kompilierung der Monitore in Java-Bytecode und die JVM, speziell der JiT Compiler der JVM, bei Ausführung dieses Bytecodes selbst Optimierungen durchführen könnten. Die in dieser Arbeit betrachteten Optimierungen könnten somit obsolet sein.
Der Scala-Compiler führt mittels des Befehls *scalac -opt* bis zu 16 Optimierungen auf ei-

nem Quellprogramm durch. Unter Hinzunahme aller im Umfang des Compilers enthaltenen Optionen wurden die Monitore mit und ohne die, in dieser Arbeit vorgenommenen, Optimierungen getestet. Es konnten keine Veränderungen festgestellt werden.
Der kompilierte Bytecode wird von einer JVM mithilfe eines JiT Compilers ausgeführt. JiT Compiler bieten die Möglichkeit Optimierungen auf dem erhaltenen Bytecode auszuführen. Da der vollständige Kontext des zu übersetzenden Programms bekannt ist, lassen sich Optimierungen, wie die Initialisierung von Konstanten, durchführen. Dies ist der Grund, warum die *Var-Val-Analyse* keine Verbesserungen ergab.

Der Einsatz der neuen DSL wird es zukünftig ermöglichen, neue Funktionen einfacher dem Core hinzuzufügen und damit weiterhin die Laufzeiten entstehender Monitore zu senken. Die Auflösung von Funktionsverschachtelungen sorgt für einen allgemein besser lesbaren und damit besser wartbaren Scala-Code.
Außerdem ist die *Var-Val-Analyse* kein Misserfolg. Die mit der Analyse eintretende Sortierung und Verwendung von Konstanten reduziert, wenn auch geringfügiger als die DSL, den Umfang der Monitore. Dies resultiert ebenfalls in einer gesteigerten Gebrauchstauglichkeit.

Die Arbeit konnte weiterhin zeigen, dass Annomalien bei der JiT Kompilierung entstehen können, sodass die Optimierungen verblassen. Dies wird an den Messungen der *election.tessla*-Monitore deutlich. Obwohl die Verbesserungen der DSL in Tests mit einer geringen Anzahl an Events deutlich zu erkennen sind, ist ab einer Anzahl von 10 Millionen getesteter Events keine Verbesserung mehr vorhanden. Ähnlich wie es in [14] und [5] festgestellt wurde, lassen sich auch in dieser Arbeit die Optimierungsstufen und der Übergang in eine *Steady*-Phase der *OpenJDK* auf Grundlage enstandenen Messungen nicht genau festmachen.
Fortlaufend könnten umfangreichere Tests durchgeführt werden. Besonders die Verwendung von realitätsnahen Monitoren mit langen Schleifenrümpfen sind von Interesse. Da gängige JiT Compiler Schleifendurchläufe zählen, ist die Anzahl und Länge der Durchläufe relevant. Indem die Anzahl niedrig gehalten, die Länge hingegen maximiert wird, können signifikante Messungen genommen werden. Längere Durchläufe würden die Laufzeit stabilisieren und Seiteneffekte reduzieren. Schwankungen der Laufzeiten würden somit minimiert werden, sodass sich die eintretenden Optimierungsstufen des Compilers genauer identifizieren lassen sollten. Weiterhin ließe sich die JiT Kompilierung genauer untersuchen, indem unterschiedliche JVMs getestet werden.

Diese Arbeit zeigte, dass Optimierungen des TeSSLa-Compiler die Laufzeiten erzeugter Monitore reduzieren können. Selbst simple Compiler-Optimierungen sind sinnvoll und könnten dem Compiler zukünftig hinzugefügt werden.

Literatur

[1] Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V. M. und Stichnoth, J. M. Fast, Effective Code Generation in a Just-in-Time Java Compiler. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, S. 280–290. ISBN: 0897919874. DOI: 10.1145/277650.277740. URL: https://doi.org/10.1145/277650.277740.

[2] Aho, A. V., Sethi, R. und Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0201100886. DOI: 10.5555/6448.

[3] Arnold, M., Fink, S., Grove, D., Hind, M. und Sweeney, P. Adaptive optimization in the Jalapeno JVM. In: *SIGP*. 2011.

[4] Aycock, J. A Brief History of Just-in-Time. In: *ACM Comput. Surv.* 35(2):97–113, Juni 2003. ISSN: 0360-0300. DOI: 10.1145/857076.857077. URL: https://doi.org/10.1145/857076.857077.

[5] Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S. und Tratt, L. Virtual Machine Warmup Blows Hot and Cold. In: *Proc. ACM Program. Lang.* 1(OOPSLA), Okt. 2017. DOI: 10.1145/3133876. URL: https://doi.org/10.1145/3133876.

[6] Beckwith, M. *What the JIT!? Anatomy of the OpenJDK HotSpot VM*. https://www.infoq.com/articles/OpenJDK-HotSpot-What-the-JIT/. Besucht: 22.04.2021.

[7] Boehm, B. W. Software Engineering Economics. In: Hrsg. von M. Broy und E. Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 99–150. ISBN: 978-3-642-48354-7. DOI: 10.1007/978-3-642-48354-7_5. URL: https://doi.org/10.1007/978-3-642-48354-7_5.

[8] Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M. und Thoma, D. TeSSLa: Temporal Stream-Based Specification Language. In: *Formal Methods: Foundations and Applications*. Hrsg. von T. Massoni und M. R. Mousavi. Cham: Springer International Publishing, 2018, S. 144–162. ISBN: 978-3-030-03044-5.

[9] D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S. und Manna, Z. LOLA: runtime monitoring of synchronous systems. In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. 2005, S. 166–174. DOI: 10.1109/TIME.2005.26.

[10] E. Bartocci, Y. F. *Lectures on Runtime Verification*. 2018. DOI: 10.1007/978-3-319-75632-5.

[11] *Esterel: a Synchronous Reactive Programming Language*. https://www-sop.inria.fr/esterel.org/filesv5_92/. Besucht: 25.03.2021.

[12] Fowler, M. und Parsons, R. *Domain-specific Languages*. Addison-Wesley signature series. Addison-Wesley, 2011. ISBN: 9780321712943. URL: https://books.google.de/books?id=7lLHmAEACAAJ.

[13] *FreePascal Compiler*. https://freepascal.org/. Besucht: 20.04.2021.

[14] Georges, A., Buytaert, D. und Eeckhout, L. Statistically Rigorous Java Performance Evaluation. In: *SIGPLAN Not.* 42(10):57–76, Okt. 2007. ISSN: 0362-1340. DOI: 10.1145/1297105.1297033. URL: https://doi.org/10.1145/1297105.1297033.

[15] Halbwachs, N., Caspi, P., Raymond, P. und Pilaud, D. The synchronous data flow programming language LUSTRE. In: *Proceedings of the IEEE* 79(9):1305–1320, 1991. DOI: 10.1109/5.97300.

[16] Hansen, G. J., Shoults, G. A. und Cointment, J. D. Construction of a Transportable, Multi-Pass Compiler for Extended Pascal. In: *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*. SIGPLAN '79. Denver, Colorado, USA: Association for Computing Machinery, 1979, S. 117–126. ISBN: 0897910028. DOI: 10.1145/800229.806961. URL: https://doi.org/10.1145/800229.806961.

[17] ISP, U. z. L. *TeSSLa Language SpecificationVersion 1.0*. https://www.tessla.io/TeSSLaLanguageSpecification-1.0.pdf. Besucht: 27.04.2021.

[18] *ÏVM JIT-compiler overview¨, presentation by Vladimir Ivanov at Oracle*. http://cr.openjdk.java.net/~vlivanov/talks/2015_JIT_Overview.pdf. Besucht: 22.04.2021.

[19] Kallwies, H. Effiziente Code Generierung für strombasierte Spezifikationen. In: 2019.

[20] Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M. und Schramm, A. TeSSLa: Runtime Verification of Non-Synchronized Real-Time Streams. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: Association for Computing Machinery, 2018, S. 1925–1933. ISBN: 9781450351911. DOI: 10.1145/3167132.3167338. URL: https://doi.org/10.1145/3167132.3167338.

[21] Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M. und Thoma, D. Runtime Verification for Timed Event Streams with Partial Information. In: *Runtime Verification*. Hrsg. von B. Finkbeiner und L. Mariani. Cham: Springer International Publishing, 2019, S. 273–291. ISBN: 978-3-030-32079-9. DOI: 10.1007/978-3-030-32079-9_16.

[22] *Native Compiler*. https://www.techopedia.com/definition/8506/native-compiler. Besucht: 20.04.2021.

[23] Odersky, M., Scalacenter und Lightbend *Scala Documentation*. https://docs.scala-lang.org. Besucht: 19.04.2021.

[24] Odersky, M., Scalacenter und Lightbend *Scala GitHub Repo*. https://github.com/scala/scala. Besucht: 25.03.2021.

[25] Odersky, M., Scalacenter und Lightbend *Scala official website*. https://www.scala-lang.org. Besucht: 25.03.2021.

[26] *OpenJDK official Website*. https://openjdk.java.net/. Besucht: 18.04.2021.

[27] Oracle Corporation *Java Language and Virtual Machine Specifications*. https://docs.oracle.com/javase/specs. Besucht: 25.03.2021.

[28] Oracle Corporation *The Java HotSpot Performance Engine Architecture*. https://www.oracle.com/java/technologies/whitepaper.html. Besucht: 25.03.2021.

[29] *Rake Github*. https://github.com/ruby/rake. Besucht: 18.04.2021.

[30] Sassa, M., TOKUDA, J., SHINOGI, T. und INOUE, K. Design and Implementation of a Multipass-Compiler Generator (Mathematical Methods in Software Science and Engineering). In: Jan. 1979.

[31] *Single-Pass Compiler*. https://keleshev.com/one-pass-compiler-primer. Besucht: 20.04.2021.

[32] *TeSSLa Standard Library*. https://www.tessla.io/stdlib/1.2.2/root. Besucht: 25.03.2021.

[33] *The JIT compiler*. https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler. Besucht: 21.04.2021.

[34] *Transpiler*. https://devopedia.org/transpiler. Besucht: 20.04.2021.

A

Messungen

Alle Messungen werden in Sekunden angegeben. Jeder aufgeführte Wert bildet sich über den Mittelwert von zehn durchgeführten Tests. Insgesamt wurden über viertausend Tests genommen, unter anderem da unbrauchbare Tests verworfen werden mussten.
Die Tests wurden auf einem Windows Computer mit 8 GB RAM und einem Intel Core i5-1035G4 (1.5GHz) Prozessor durchgeführt.

boolFilter.tessla	without Opts	with Var-Val but without DSL	with DSL but without Var-Val	with DSL & Var-Val	without Opts but with -optimize	with Opts and -optimize
$n =$						
1,000	0.00154	0.00173	0.00069	0.00053	0.0015	0.00053
5,000	0.00446	0.0047	0.00187	0.00175	0.0044	0.00177
25,000	0.01307	0.01323	0.00586	0.00623	0.01455	0.0057
50,000	0.022	0.02154	0.0103	0.01045	0.02284	0.01004
1,000,000	0.07586	0.066	0,03656	0.03297	0.07222	0.0324
5,000,000	0.26462	0.21709	0.075	0.07872	0.23064	0.07463
25,000,000	0.9149	0.7769	0.28516	0.27785	0.8722	0.27851
50,000,000	1.72488	1.47759	0.49326	0.48994	1.64531	0.47783
1,000,000,000	32.2539	27.2038	7.97842	8.07329	30.0925	8.05971
5,000,000,000	157.831	137.324	45.7257	46.3645	152.083	45.8365
25,000,000,000	794.337	728.28	228.87	233.187	772.661	232.666
50,000,000,000	1629.233	1433	458.292	470.245	1567.67	481.158

Abbildung A.1: Messergebnisse des *boolFilter*-Monitors in Sekunden. Tests wurde für sechs Varianten durchgeführt; without Opts: Der Monitor ohne Optimierungen; with Var-Val but without DSL: Unter Hinzunahme der Analyse auf konstante Initialisierungen, jedoch ohne die neue DSL; with DSL but without Var-Val: Unter Hinzunahme der neuen DSL, jedoch ohne die Analyse auf konstante Initialisierungen; with DSL & Var-Val: Beide Optimierungen wurden verwendet; without Opts but with -optimize: Keine der erarbeiteten Optimierungen wurde verwendet, jedoch wurden die Optimierungsmöglichkeiten des Scala-Compilers aktiviert; with Opts and -optimize: Die erarbeiteten Optimierungen und die Optimierungsmöglichkeiten des Scala-Compilers wurden verwendet.

count.tessla	without Opts	with Var-Val but without DSL	with DSL but without Var-Val	with DSL & Var-Val	without Opts but with -optimize	with Opts and -optimize
$n =$						
1,000	0.00228	0,00203	0.00036	0.00035	0.00191	0.00038
5,000	0.00556	0.0057	0.00147	0.00139	0.00628	0.0015
25,000	0.01657	0.0158	0.00622	0.00587	0.01673	0.00539
50,000	0.03268	0.02522	0.00825	0.00834	0.02521	0.00866
1,000,000	0.09615	0.09365	0.02888	0.02978	0.09504	0.03129
5,000,000	0.29854	0.3006	0.07485	0.07136	0.30444	0.07612
25,000,000	1.1157	1.12646	0.28632	0.28116	1.21704	0.30269
50,000,000	2.12144	2.14412	0.50697	0.49306	2.23926	0.52192
1,000,000,000	39.55	40.2486	8.2858	8.13571	42.124	8.79384
5,000,000,000	199.122	199.064	45.2606	44.9933	214.316	49.5787
25,000,000,000	1042.309	1025.52	226.495	228.217	1121.64	246.587
50,000,000,000	2129.667	2125.087	469.896	469.665	2244.3	507.35

Abbildung A.2: Messergebnisse des *count*-Monitors in Sekunden. Tests wurde für sechs Varianten durchgeführt; without Opts: Der Monitor ohne Optimierungen; with Var-Val but without DSL: Unter Hinzunahme der Analyse auf konstante Initialisierungen, jedoch ohne die neue DSL; with DSL but without Var-Val: Unter Hinzunahme der neuen DSL, jedoch ohne die Analyse auf konstante Initialisierungen; with DSL & Var-Val: Beide Optimierungen wurden verwendet; without Opts but with -optimize: Keine der erarbeiteten Optimierungen wurde verwendet, jedoch wurden die Optimierungsmöglichkeiten des Scala-Compilers aktiviert; with Opts and -optimize: Die erarbeiteten Optimierungen und die Optimierungsmöglichkeiten des Scala-Compilers wurden verwendet.

average.tessla	without Opts	with Var-Val but without DSL	with DSL but without Var-Val	with DSL & Var-Val	without Opts but with -optimize	with Opts and -optimize
$n =$						
1,000	0.00345	0.00339	0.0005	0.0005	0.00349	0.00047
5,000	0.00949	0.0085	0.00175	0.0017	0.00962	0.00159
25,000	0.03512	0.03424	0.00488	0.0055	0.03715	0.00642
50,000	0.06229	0.05952	0.00871	0.00897	0.05917	0.00806
1,000,000	0.28065	0.27233	0.02974	0.03061	0.27701	0.02856
5,000,000	0.83125	0.83317	0.07082	0.072	0.81736	0.07371
25,000,000	3.4674	3.44822	0.28137	0.29472	3.6993	0.28294
50,000,000	6.70632	6.77388	0.50162	0.52007	7.40376	0.50985
1,000,000,000	151.092	132.499	8.62418	9.21114	131.412	8.85002
5,000,000,000	687.568	681.0233	45.4635	48.5647	652.652	46.757
25,000,000,000	3398.108	3466.788	227.306	245.59	3782.022	249.536
50,000,000,000	7236.412	6808.37	454.946	498.484	6858.15	512.24

Abbildung A.3: Messergebnisse des *average*-Monitors in Sekunden. Tests wurde für sechs Varianten durchgeführt; without Opts: Der Monitor ohne Optimierungen; with Var-Val but without DSL: Unter Hinzunahme der Analyse auf konstante Initialisierungen, jedoch ohne die neue DSL; with DSL but without Var-Val: Unter Hinzunahme der neuen DSL, jedoch ohne die Analyse auf konstante Initialisierungen; with DSL & Var-Val: Beide Optimierungen wurden verwendet; without Opts but with -optimize: Keine der erarbeiteten Optimierungen wurde verwendet, jedoch wurden die Optimierungsmöglichkeiten des Scala-Compilers aktiviert; with Opts and -optimize: Die erarbeiteten Optimierungen und die Optimierungsmöglichkeiten des Scala-Compilers wurden verwendet.

accSum.tessla	without Opts	with Var-Val but without DSL	with DSL but without Var-Val	with DSL & Var-Val	without Opts but with -optimize	with Opts and -optimize
$n =$						
1,000	1.57	1.689	0.61935	0.59942	1.5415	0.615
5,000	7.73	8.055	2.9284	2.8696	7.6927	2.87635
25,000	33.288	34.441	12.33	12.1702	33.996	12.2
50,000	63.753	66.178	23.502	23.5569	65.183	23.384
1,000,000	1221.05	1215.412	444.757	455.818	1224.66	449.02
5,000,000	6191.924	-	-	2269.497	-	-

Abbildung A.4: Messergebnisse des *accSum*-Monitors in Sekunden. Tests wurde für sechs Varianten durchgeführt; without Opts: Der Monitor ohne Optimierungen; with Var-Val but without DSL: Unter Hinzunahme der Analyse auf konstante Initialisierungen, jedoch ohne die neue DSL; with DSL but without Var-Val: Unter Hinzunahme der neuen DSL, jedoch ohne die Analyse auf konstante Initialisierungen; with DSL & Var-Val: Beide Optimierungen wurden verwendet; without Opts but with -optimize: Keine der erarbeiteten Optimierungen wurde verwendet, jedoch wurden die Optimierungsmöglichkeiten des Scala-Compilers aktiviert; with Opts and -optimize: Die erarbeiteten Optimierungen und die Optimierungsmöglichkeiten des Scala-Compilers wurden verwendet.

filterByTime.tessla	without Opts	with Var-Val but without DSL	with DSL but without Var-Val	with DSL & Var-Val	without Opts but with -optimize	with Opts and -optimize
$n =$						
1,000	0.525	0.53	0.255	0.23174	0.5015	0.2875
5,000	2.44885	2.43275	1.0814	1.028	2.52493	1.084
25,000	10.55165	10.2318	4.437	4.63	10.166	4.6345
50,000	20.2379	19.7565	8.56	8.4014	19.47	9.0697
1,000,000	363.84	371.001	162.172	162.664	369.3222	173.96
5,000,000	1826.64	-	-	795.26	1761.174	835.6
10,000,000	3632.995	-	-	1612.81	3613.013	1672.32

Abbildung A.5: Messergebnisse des *filterByTime*-Monitors in Sekunden. Tests wurde für sechs Varianten durchgeführt; without Opts: Der Monitor ohne Optimierungen; with Var-Val but without DSL: Unter Hinzunahme der Analyse auf konstante Initialisierungen, jedoch ohne die neue DSL; with DSL but without Var-Val: Unter Hinzunahme der neuen DSL, jedoch ohne die Analyse auf konstante Initialisierungen; with DSL & Var-Val: Beide Optimierungen wurden verwendet; without Opts but with -optimize: Keine der erarbeiteten Optimierungen wurde verwendet, jedoch wurden die Optimierungsmöglichkeiten des Scala-Compilers aktiviert; with Opts and -optimize: Die erarbeiteten Optimierungen und die Optimierungsmöglichkeiten des Scala-Compilers wurden verwendet.

election.tessla	without Opts	with Var-Val but without DSL	with DSL but without Var-Val	with DSL & Var-Val	without Opts but with -optimize	with Opts and -optimize
$n =$						
1,000	0.23975	0.21731	0.15017	0.13596	0.23674	0.1465
5,000	0.81374	0.7015	0.5404	0.48	0.82	0.4967
25,000	3.27	2.8157	2.15006	1.932	3.36034	2.2423
50,000	6.753	5.2825	4.124	3.57323	6.216	4.2494
1,000,000	113.0223	94.953	75.6	78.6266	124.6053	79.66
5,000,000	400.33	465.864	388.9323	336.134	400.2004	393.6433
10,000,000	791.815	851.27	753.416	807.7726	805.47	782.7576
15,000,000	1155.803	1306.044	1134.2	1250.985	1209.8055	1190.448

Abbildung A.6: Messergebnisse des *election*-Monitors in Sekunden. Tests wurde für sechs Varianten durchgeführt; without Opts: Der Monitor ohne Optimierungen; with Var-Val but without DSL: Unter Hinzunahme der Analyse auf konstante Initialisierungen, jedoch ohne die neue DSL; with DSL but without Var-Val: Unter Hinzunahme der neuen DSL, jedoch ohne die Analyse auf konstante Initialisierungen; with DSL & Var-Val: Beide Optimierungen wurden verwendet; without Opts but with -optimize: Keine der erarbeiteten Optimierungen wurde verwendet, jedoch wurden die Optimierungsmöglichkeiten des Scala-Compilers aktiviert; with Opts and -optimize: Die erarbeiteten Optimierungen und die Optimierungsmöglichkeiten des Scala-Compilers wurden verwendet.

B

Grafischer Verlauf der Messungen

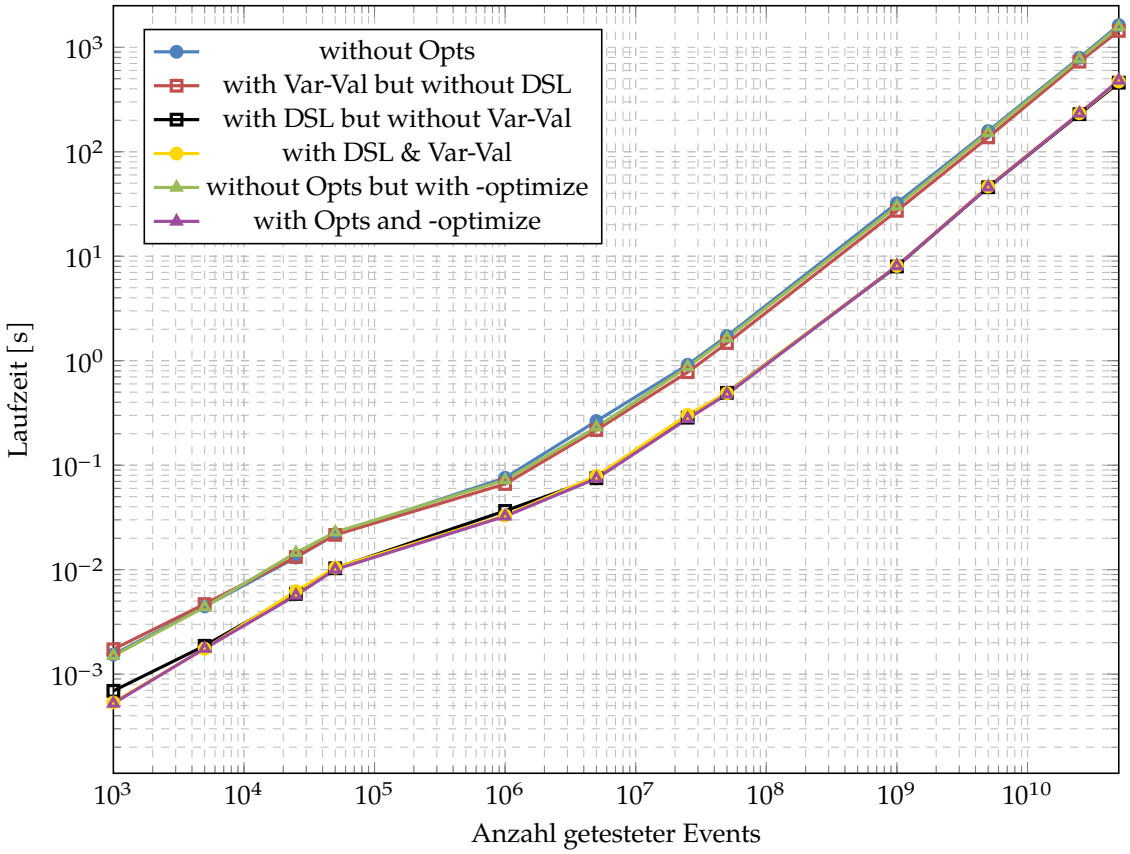

Abbildung B.1: Darstellung der Ergebnisse von *boolFilter.tessla*. Es wurden bis zu 50 Milliarden Events getestet, mit einer maximalen Laufzeit von 2244,3 Sekunden.

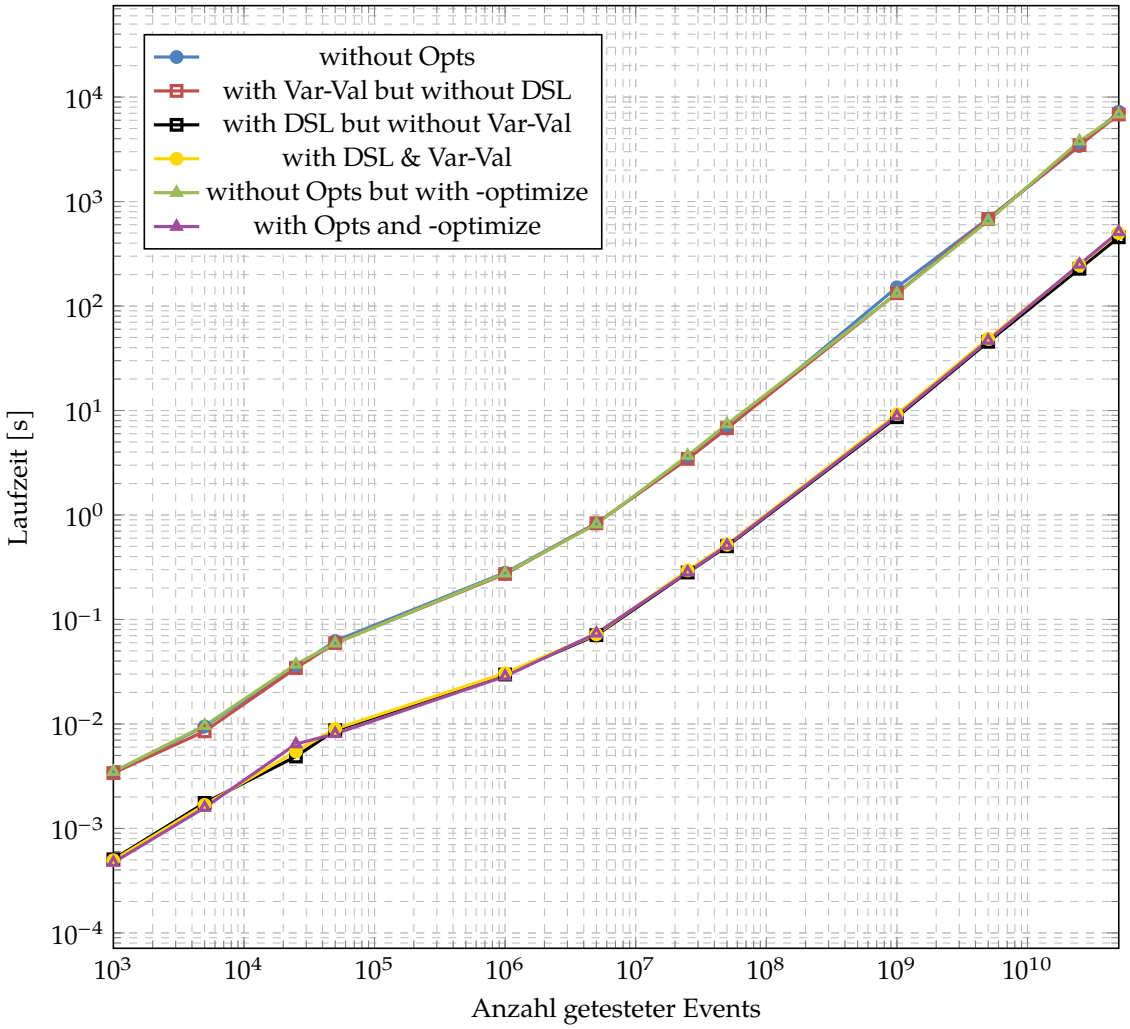

Abbildung B.2: Darstellung der Ergebnisse von *average.tessla*. Es wurden bis zu 50 Milliarden Events getestet, mit einer maximalen Laufzeit von 7236,4 Sekunden.

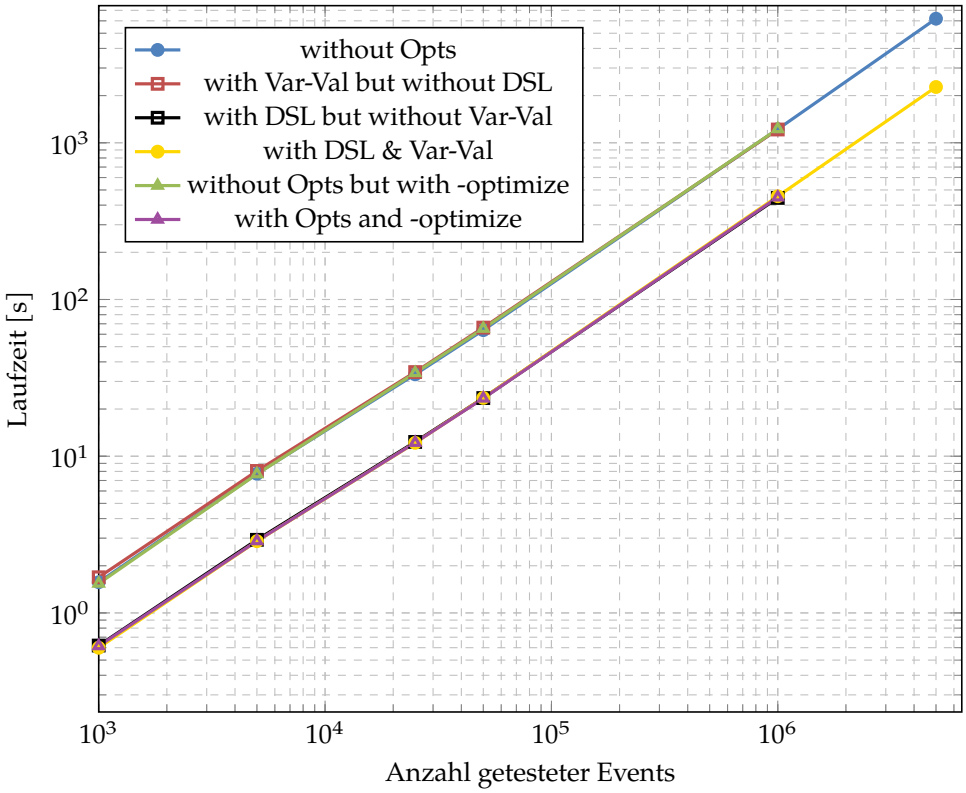

Abbildung B.3: Darstellung der Ergebnisse von *accSum.tessla*. Es wurden bis zu 5 Millionen Events getestet, mit einer maximalen Laufzeit von 6191,924 Sekunden.

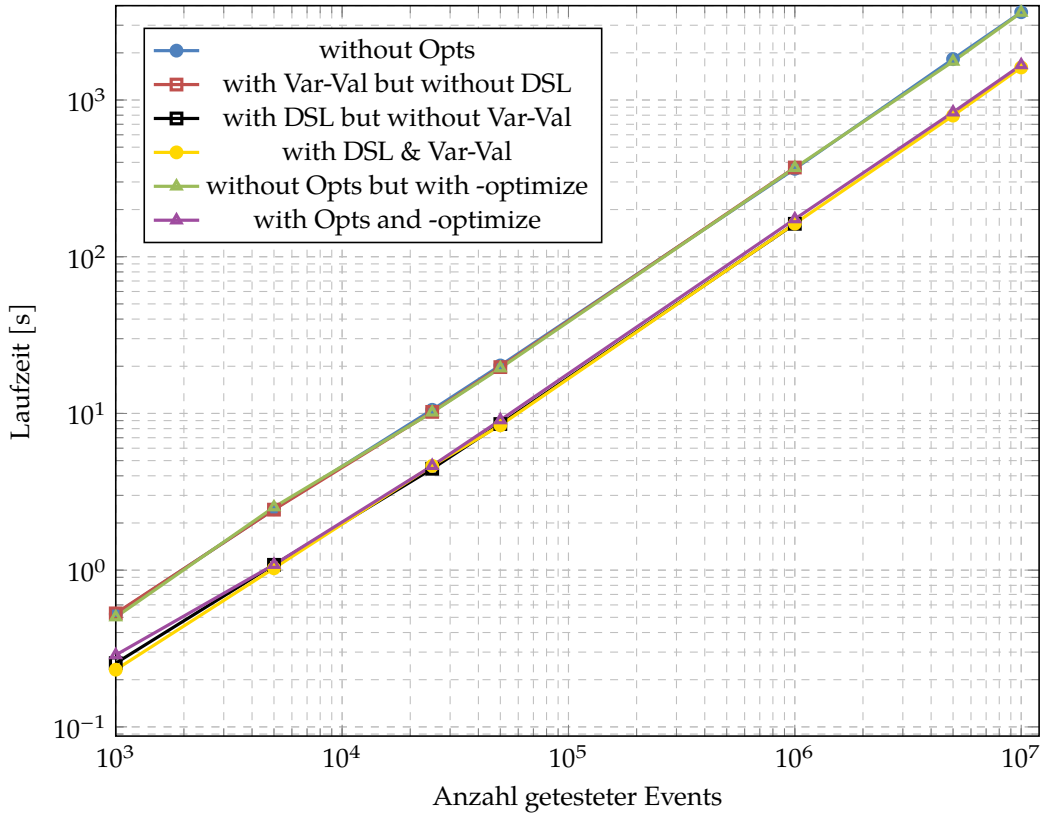

Abbildung B.4: Darstellung der Ergebnisse von *filterByTime.tessla*. Es wurden bis zu 10 Millionen Events getestet, mit einer maximalen Laufzeit von 3632,995 Sekunden.

C

Abbildungsverzeichnis

C Abbildungsverzeichnis