GI-Edition

Lecture Notes in Informatics

Stefan Jähnichen, Bernhard Rumpe,
Holger Schlingloff (Hrsg.)

Software Engineering 2012 Workshopband

27. Februar – 2. März 2012
in Berlin

Proceedings

199

[JRS12] S. Jähnichen, B. Rumpe, H. Schlingloff (Eds.)
Software Engineering 2012 Workshop Proceedings.
GI-Edition - Lecture Notes in Informatics (LNI). P-199. 2012.

Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.): Software Engineering 2012

Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.)

Software Engineering 2012
Workshopband

Fachtagung des GI-Fachbereichs Softwaretechnik

27. Februar - 2. März 2012
in Berlin

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-199
ISBN 978-3-88579-293-2
ISSN 1617-5468

Volume Editors
Stefan Jähnichen
Technische Universität Berlin / Softwaretechnik
Sekr. TEL 12-3, Ernst-Reuter-Platz 7, 10587 Berlin, Germany
Email: stefan.jaehnichen,@tu-berlin.de
Bernhard Rumpe
RWTH Aachen / Softwaretechnik
Department of Computer Science 3, Ahornstraße 55, 52074 Aachen, Germany
Email: rumpe@se-rwth.de
Holger Schlingloff
Humboldt-Universität zu Berlin / Softwaretechnik
Rudower Chaussee 25, 12489 Berlin, Germany
Email: hs@informatik.hu-berlin.de

printed by Köllen Druck+Verlag GmbH, Bonn

Software: the New Driving Force

so formuliert die Business Week bereits am 24. Februar 1983 auf ihrer Titelseite. In dem dazugehörigen Artikel werden der damalige Stand der Softwaretechnik und vor allem ihre Perspektiven dargestellt. Viele der damals aufgestellten Thesen sind noch heute richtig und belegen damals wie heute die Bedeutung der Softwaretechnik.

Software spielt eine zentrale Rolle bei der Nutzung innovativer Technologien und ihre Qualität ist für die Wertschöpfung der Produkte ein entscheidender Faktor. Sie muss höchsten Qualitätsanforderungen entsprechen und trotzdem kostengünstig hergestellt werden. Zuverlässigkeit, Sicherheit und Akzeptanz beim Nutzer sind Eigenschaften, die wir von moderner Software erwarten. Die Techniken zu ihrer Gewährleistung sind zwar zwischenzeitlich in hohem Maße professionalisiert und weiterentwickelt, bedürfen aber wegen der immer noch steigenden Komplexität der Systeme und der Integration mit anderen Technologien noch immer intensiver Forschung und weiterer Professionalisierung.

Die Software Engineering-Tagungsreihe wird vom Fachbereich Softwaretechnik der Gesellschaft für Informatik e.V. getragen. 2012 hat die Fakultät IV der TU Berlin die Gestaltung und Organisation der Tagung übernommen. Die SE 2012 bietet im Hauptprogramm sowohl eingeladene wissenschaftliche Vorträge als auch vom PC begutachtete detaillierte Berichte über laufende Forschungsarbeiten und -ergebnisse. Darüber hinaus werden in Praxisvorträgen am Industrietag aktuelle Problemstellungen, Lösungsansätze und gewonnene Erfahrungen präsentiert und zur Diskussion gestellt. Vor dem Hauptprogramm der Konferenz finden 4 Workshops sowie 3 Tutorials zu aktuellen, innovativen und praxisrelevanten Themen im Software Engineering statt. Abgerundet wird das Programm durch ein Doktorandensymposium, auf dem promovierende junge Wissenschaftlerinnen und Wissenschaftler ihre Arbeiten vorstellen und von erfahrenen Forscherinnen und Forschern konstruktive Rückkopplung zu ihren Dissertationsvorhaben erwarten. Fast schon traditionell ist ein Lehrertag in die SE 2012 integriert, auf dem sich Informatiklehrerinnen und -lehrer über neue Ansätze der Softwaretechnik für den Schulunterricht informieren.

Die Durchführung der Tagung Software Engineering 2012 wäre ohne die Mitwirkung der Sponsoren und vieler engagierter Personen nicht möglich gewesen. Ich bedanke mich daher bei allen Sponsoren, vor allem aber auch bei den vielen Helfern und Helferinnen der SE 2012:

- für die Gestaltung des Industrietags Axel Küpper, Ulrich Bareth und Sahin Albayrak

- für die Gestaltung des Doktorandensymposiums Petra Hofstedt und Claus Lewerentz

- für die Koordination der Workshops/Tutorials Holger Schlingloff und Bernhard Rumpe

Der Lehrertag wurde von der GI-Fachgruppe "Informatik-Bildung in Berlin und Brandenburg" organisiert. Dank gilt hier insbesondere Helmut Witten, der als Sprecher der Fachgruppe die Koordination übernommen hat.

Dank auch an alle Mitglieder des Programmkommittees, des Steering Committes, an den Fachbereich Softwaretechnik und die Organisatoren bei der Geschäftsstelle der GI und der bwo Marketing GmbH.

Und letztlich ganz besonderer Dank an alle meine Mitarbeiter, insbesondere Marc-Oliver Reiser, Doris Fähndrich, Marcus Mews, Andreas Mertgen und Steffen Helke.

Ohne diese viele Unterstützung wäre die SE 2012 nicht möglich.

Berlin, im Februar 2012

Stefan Jähnichen, Tagungsleiter

Tagungsleitung
Stefan Jähnichen, TU Berlin/FhG FIRST

Leitung Industrietag
Axel Küpper, TU Berlin/T-Labs
Sahin Albayrak, TU Berlin/DAI-Labor

Leitung Workshops und Tutorials
Bernhard Rumpe, RWTH Aachen
Holger Schlingloff, HU Berlin/FhG FIRST

Tagungsorganisation
Doris Fähndrich, TU Berlin
Steffen Helke, TU Berlin
Andreas Mertgen, TU Berlin
Marcus Mews, TU Berlin
Mark-Oliver Reiser, TU Berlin

Programmkomitee
Steffen Becker, Universität Paderborn
Klaus Beetz, Siemens AG
Manfred Broy, TU München
Bernd Brügge, TU München
Jürgen Ebert, Universität Koblenz-Landau
Gregor Engels, Universität Paderborn
Martin Glinz, Universität Zürich
Michael Goedicke, Universität Duisburg-Essen
Volker Gruhn, Universität Duisburg-Essen
Jens Happe, SAP
Wilhelm Hasselbring, Christian-Albrecht-Universität Kiel
Maritta Heisel, Universität Duisburg-Essen
Stefan Jähnichen, TU Berlin/FhG FIRST
Matthias Jarke, RWTH Aachen
Gerti Kappel, TU Wien
Udo Kelter, Universität Siegen
Jens Knoop, TU Wien
Heiko Koziolek, ABB
Claus Lewerentz, BTU Cottbus
Horst Lichter, RWTH Aachen
Peter Liggesmeyer, TU Kaiserslautern
Oliver Mäckel, Siemens AG
Florian Matthes, TU München
Oscar Nierstrasz, Universität Bern
Andreas Oberweis, KIT/FZI Karlsruhe
Barbara Paech, Universität Heidelberg
Peter Pepper, TU Berlin
Klaus Pohl, Universität Duisburg-Essen

Alexander Pretschner, KIT, Karlsruhe
Ralf Reussner, KIT/FZI Karlsruhe
Matthias Riebisch, TU Ilmenau
Andreas Roth, SAP
Bernhard Rumpe, RWTH Aachen
Thomas Santen, European Microsoft Innovation Center
Wilhelm Schäfer, Universität Paderborn
Klaus Schmid, Universität Hildesheim
Kurt Schneider, Leibniz Universität Hannover
Andy Schürr, TU Darmstadt
Rainer Singvogel, msg systems AG
Stefan Tai, KIT/FZI Karlsruhe
Andreas Winter, Universität Oldenburg
Mario Winter, Fachhochschule Köln
Uwe Zdun, Universität Wien
Andreas Zeller, Universität des Saarlandes
Heinz Züllighoven, Universität Hamburg
Albert Zündorf, Universität Kassel

Offizieller Veranstalter
Fachbereich Softwaretechnik der Gesellschaft für Informatik (GI)

Mitveranstalter
Technische Universität Berlin

Sponsoren

PLATIN

GOLD

SILBER

BRONZE

Inhaltsverzeichnis

5. Arbeitstagung Programmiersprachen (ATPS'12)

Zertifizierung und modellgetriebene Entwicklung sicherer Software (ZeMoSS)

5. Arbeitstagung Programmiersprachen (ATPS'12)

5. Arbeitstagung Programmiersprachen (ATPS'12)
—
Einleitung

Frank Huch (fhu@informatik.uni-kiel.de)
Janis Voigtländer (jv@informatik.uni-bonn.de)

Die Arbeitstagung Programmiersprachen soll dem Austausch zwischen Forschern, Entwicklern und Anwendern dienen, die sich mit Themen aus dem Bereich der Programmiersprachen beschäftigen. Alle Programmierparadigmen sind von Interesse: imperative, objektorientierte, funktionale, logische, parallele, graphische Programmiersprachen, auch verteilte und nebenläufige Programmierung in Intra- und Internet-Anwendungen, sowie Konzepte zur Integration dieser Paradigmen. Die ersten vier Arbeitstagungen Programmiersprachen fanden im Rahmen von GI-Jahrestagungen statt (Aachen 1997, Paderborn 1999, Ulm 2004, Lübeck 2009) statt. Erstmals in diesem Jahr findet sie zusammen mit der GI-Tagung Software Engineering statt.

Der Aufruf zur Einreichung von Beiträgen führte zu zehn Einreichungen. Jede Einreichung wurde von mindestens drei Programmkomiteemitgliedern, assistiert durch externe Gutachter, begutachtet. Nach ausführlicher Diskussion nahm das Programmkomitee acht Beiträge zur Präsentation auf der Tagung und Veröffentlichung im Tagungsband an.

Wir freuen uns, dass wir für das Programm außerdem folgenden eingeladenen Vortrag gewinnen konnten:

- Peter Pepper: The Beauty of Acausality
  Abstract: Computing usually means to compute something from something else — and this implicitly states a causal dependency. This is obvious in imperative programming, but holds for many variants of declarative programming as well. Yet, there are many aspects in software and programming which would benefit a lot by abstracting from a cause-effect based description. This can be found in a range of tools and languages, ranging from constraint programming to Modelica-style modeling. Major challenges remain the systematic presentation of the idea of acausality and above all the efficient implementation by compilers.

Die folgenden eingereichten Beiträge wurden angenommen und sind hier im Tagungsband enthalten:

1. Henning Thielemann:
   Live-Musikprogrammierung in Haskell
2. David Sabel:
   An Abstract Machine for Concurrent Haskell with Futures

3. Christian Heinlein:
   MOSTflexiPL — Modular, Statically Typed, Flexibly Extensible Programming Language

4. Michael Hanus, Björn Peemöller und Fabian Reck:
   Search Strategies for Functional Logic Programming

5. Dirk Tetzlaff und Sabine Glesner:
   Making MPI Intelligent

6. Dirk Richter und Roberto Hoffmann:
   Fehlalarmfreie Abstraktion in ISO-C konformer Semantik

7. Evgenij Belikov, Hans-Wolfgang Loidl, Greg Michaelson und Phil Trinder:
   Architecture-Aware Cost Modelling for Parallel Performance Portability

8. Tim Frey und Veit Köppen:
   Exploring Software Variance with Hypermodelling — An exemplary approach

Wir möchten uns bei allen Beteiligten für die Unterstützung zum Gelingen der ATPS'12 bedanken. Zunächst gilt unser Dank den Autoren und dem eingeladenen Vortragenden für ihre Beiträge zum Programm. Wir danken außerdem den Programmkomiteemitgliedern und externen Gutachtern. Sie alle haben in kurzer Zeit hart daran gearbeitet, die Einreichungen zu begutachten und den Autoren nützliches Feedback zukommen zu lassen. Das Programmkomitee, geleitet durch Frank Huch und Janis Voigtländer, bestand aus:

Robert Glück	Universität Kopenhagen
Clemens Grelck	Universität Amsterdam
Michael Hanus	Universität Kiel
Petra Hofstedt	Technische Universität Cottbus
Jens Knoop	Technische Universität Wien
Herbert Kuchen	Universität Münster
Rita Loogen	Universität Marburg
Martin Plümicke	Duale Hochschule Baden-Württemberg Stuttgart
Sven-Bodo Scholz	Heriot-Watt Universität Edinburgh

Schließlich möchten wir den lokalen Organisatoren der SE'12 für ihre Unterstützung im gesamten Prozess danken.

Wir hoffen, dass die ATPS'12 für alle Teilnehmenden ein interessantes und stimulierendes Ereignis wird und dass die Tagung wertvolle Gelegenheiten um Austausch von Ideen bietet.

Februar 2012

Frank Huch (Universität Kiel)
Janis Voigtländer (Universität Bonn)
PC chairs ATPS'12

Live-Musikprogrammierung in Haskell

Henning Thielemann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
06122 Halle
henning.thielemann@informatik.uni-halle.de

Abstract: Ziel unserer Arbeit ist es, algorithmische Musik interaktiv und mit mehreren Teilnehmern zu komponieren. Dazu entwickeln wir einen Interpreter für eine Teilsprache der nicht-strikten funktionalen Programmiersprache Haskell 98, der es erlaubt, das Programm noch während seiner Ausführung zu ändern. Unser System eignet sich sowohl für den Live-Einsatz zur Musikprogrammierung als auch als Demonstrations- und Lernumgebung für funktionale Programmierung.

1 Einführung

Unser Ziel ist es, Musik durch Algorithmen zu beschreiben. Wir wollen Musik nicht wie auf dem Notenblatt als mehr oder weniger zusammenhanglose Folge von Noten darstellen, sondern wir wollen Strukturen ausdrücken. Beispielsweise wollen wir nicht die einzelnen Noten einer Begleitung aufschreiben, sondern die Begleitung durch ein allgemeines Muster und die Folge von Harmonien ausdrücken. Als weiteres Beispiel mag ein Komponist dienen, der eine Folge von zufälligen Noten verwenden möchte. Er möchte die Noten aber nicht einzeln aufschreiben, sondern die Idee ausdrücken, dass es eine zufällige Folge von Noten ist. Dem Interpreten wäre es damit freigestellt, eine andere aber ebenso zufällige Folge von Noten zu spielen.

Der Programmierer soll den Grad der Strukturierung frei wählen können. Beispielsweise soll es möglich sein, „von Hand“ eine Melodie zu komponieren, diese mit einem Tonmuster zu begleiten, für das lediglich eine Folge von Harmonien vorgegeben wird, und das ganze mit einem vollständig automatisch berechneten Rhythmus zu unterlegen.

Mit dem bewussten Abstrahieren von der tatsächlichen Musik wird es aber schwierig, beim Programmieren das Ergebnis der Komposition abzuschätzen. Bei Musik, die nicht streng nach Takten und Stimmen organisiert ist, fällt es schwerer, einen bestimmten Zeitabschnitt oder eine bestimmte Auswahl an Stimmen zur Probe anzuhören. Auch der klassische Zyklus von „Programm editieren, Programm prüfen und übersetzen, Programm neu starten“ steht dem kreativen Ausprobieren entgegen. Selbst wenn Übersetzung und Neustart sehr schnell abgeschlossen sind, so muss doch das Musik erzeugende Programm und damit die

Musik abgebrochen und neu begonnen werden. Insbesondere beim gemeinsamen Spiel mit anderen Musikern ist das nicht akzeptabel.

In unserem Ansatz verwenden wir zur Musikprogrammierung eine rein funktionale Programmiersprache mit Bedarfsauswertung [Hug89], die nahezu eine Teilsprache von Haskell 98 [PJ+98] ist. Unsere Beiträge zur interaktiven Musikprogrammierung sind Konzepte und ein lauffähiges System, welches folgendes bieten:

- Algorithmische Musikkomposition, bei der das Programm verändert werden kann, während die Musik läuft (Abschnitt 2.1),
- gleichzeitige Arbeit mehrerer Programmierer an einem Musikstück unter Anleitung eines „Dirigenten“ (Abschnitt 2.2).

2 Funktionale Live-Programmierung

2.1 Live-Coding

Wir wollen Musik ausgeben als eine Liste von MIDI-Ereignissen [MMA96], also Ereignissen der Art „Klaviaturtaste gedrückt“, „Taste losgelassen“, „Instrument gewechselt“, „Klangregler verändert“ und Warte-Anweisungen. Ein Ton mit Tonhöhe C-5, einer Dauer von 100 Millisekunden und einer normalen Intensität soll geschrieben werden als:

```
main =
  [ Event (On c5 normalVelocity)
  , Wait 100
  , Event (Off c5 normalVelocity)
  ] ;

c5 = 60 ;
normalVelocity = 64 ;
.
```

Mit der Listenverkettung „++“ lässt sich damit bereits eine einfache Melodie beschreiben.

```
main =
   note qn c ++ note qn d ++ note qn e ++ note qn f ++
   note hn g ++ note hn g ;

note duration pitch =
  [ Event (On pitch normalVelocity)
  , Wait duration
  , Event (Off pitch normalVelocity)
  ] ;
```

```
qn = 200 ;  -- quarter note - Viertelnote
hn = 2*qn ; -- half note    - halbe Note

c = 60 ;
d = 62 ;
e = 64 ;
f = 65 ;
g = 67 ;
normalVelocity = 64 ;
```

Diese Melodie lässt sich endlos wiederholen, indem wir am Ende der Melodie wieder mit dem Anfang fortsetzen.

```
main =
   note qn c ++ note qn d ++ note qn e ++ note qn f ++
   note hn g ++ note hn g ++ main ;
```

Die so definierte Liste `main` ist unendlich lang, lässt sich aber mit der Bedarfsauswertung schrittweise berechnen und an einen MIDI-Synthesizer senden. Dank der Bedarfsauswertung kann man die Musik als reine Liste von Ereignissen beschreiben. Das Programm muss und kann selbst keine Ausgabebefehle ausführen. Der Versand der MIDI-Kommandos wird vom Interpreter übernommen.

In einem herkömmlichen interaktiven Interpreter[1] wie dem `GHCi` würde man die Musik etwa so wiedergeben:

```
Prelude> playMidi main        .
```

Will man die Melodie ändern, müsste man die Musik beenden und die neue Melodie von vorne beginnen. Wir wollen aber die Melodie ändern, während die alte Melodie weiterläuft, und dann die alte Melodie nahtlos in die neue übergehen lassen. Mit anderen Worten: Der aktuelle Zustand des Interpreters setzt sich zusammen aus dem Programm und dem Zustand der Ausführung. Wir wollen das Programm austauschen, aber den Zustand der Ausführung beibehalten. Das bedeutet, dass der Zustand in einer Form gespeichert sein muss, der auch nach Austausch des Programms einen Sinn ergibt.

Wir lösen dieses Problem wie folgt: Der Interpreter betrachtet das Programm als Menge von Termersetzungsregeln und die Ausführung des Programms besteht darin, die Ersetzungsregeln wiederholt anzuwenden, solange bis der Startterm `main` so weit reduziert ist, dass die Wurzel des Operatorbaums ein Terminalsymbol (hier: ein Datenkonstruktor) ist. Für die musikalische Verarbeitung testet der Interpreter weiterhin, ob die Wurzel ein Listenkonstruktor ist und falls es eine nichtleere Liste ist, reduziert er das führende Listenelement vollständig und prüft, ob es ein MIDI-Ereignis darstellt. Ausführungszustand des Interpreters ist der reduzierte Ausdruck. Während der Interpreter die vorletzte Note in der Schleife des obigen Programms wiedergibt, wäre dies beispielsweise:

[1] Der Interpreter wäre hier im wahrsten Sinne des Wortes der musikalische Interpret

```
Wait 200 :
    (Event (Off g normalVelocity) : (note hn g ++ main))
.
```

Der Ausdruck wird immer so wenig wie möglich reduziert, gerade so weit, dass das nächste MIDI-Ereignis bestimmt werden kann. Das erlaubt es zum einen, eine unendliche Liste wie `main` zu verarbeiten und zum anderen führt es dazu, dass in dem aktuellen Term wie oben angegeben, noch die Struktur des restlichen Musikstücks zu erkennen ist. Der abschließende Aufruf von `main` ist beispielsweise noch vorhanden. Wenn wir jetzt die Definition von `main` ändern, wird diese veränderte Definition verwendet, sobald `main` reduziert wird. Wir können auf diese Weise die Melodie innerhalb der Wiederholung ändern, beispielsweise so:

```
main =
    note qn c ++ note qn d ++ note qn e ++ note qn f ++
    note qn g ++ note qn e ++ note hn g ++ main ;
.
```

Wir können aber auch folgende Änderung vornehmen

```
main =
    note qn c ++ note qn d ++ note qn e ++ note qn f ++
    note hn g ++ note hn g ++ loopA ;
```

und damit erreichen, dass nach einer weiteren Wiederholung der Melodie die Musik mit einem Abschnitt namens `loopA` fortgesetzt wird.

Wir halten an dieser Stelle fest, dass sich die Bedeutung eines Ausdrucks während des Programmablaufs ändern kann. Damit geben wir eine wichtige Eigenschaft der rein funktionalen Programmierung auf. Wenn wir von Live-Änderungen Gebrauch machen, ist unser System also nicht mehr „referential transparent“. Beispielsweise hätten wir die ursprüngliche Schleife auch mit der Funktion `cycle` implementieren können

```
main =
  cycle
    ( note qn c ++ note qn d ++ note qn e ++ note qn f ++
      note hn g ++ note hn g ) ;
```

und wenn `cycle` definiert ist als

```
cycle xs = xs ++ cycle xs ;
```

dann würde dies reduziert werden zu

```
( note qn c ++ note qn d ++ note qn e ++ note qn f ++
  note hn g ++ note hn g )
```

```
++
cycle
   ( note qn c ++ note qn d ++ note qn e ++ note qn f ++
     note hn g ++ note hn g ) ;
.
```

Die Schleife könnte dann nur noch verlassen werden, wenn man die Definition von `cycle` ändert. Diese Änderung würde aber alle Aufrufe von `cycle` im aktuellen Term gleichermaßen betreffen. Zudem wäre es bei einem strengen Modulsystem ohne Importzyklen unmöglich, im Basis-Modul `List`, in dem `cycle` definiert ist, auf Funktionen im Hauptprogramm zuzugreifen. Dies wäre aber nötig, um die `cycle`-Schleife nicht nur verlassen, sondern auch im Hauptprogramm fortsetzen zu können.

Wir erkennen an diesem Beispiel, dass es vorausschauend besser sein kann, mit einer „von Hand“ programmierten Schleife der Form `main = ... ++ main` eine Sollbruchstelle zu schaffen, an der man später neuen Code einfügen kann.

Neben der seriellen Verkettung von musikalischen Ereignissen benötigen wir noch die parallele Komposition, also die simultane Wiedergabe von Melodien, Rhythmen usw. Auf der Ebene der MIDI-Kommandos bedeutet dies, dass die Kommandos zweier Listen geeignet miteinander verzahnt werden müssen. Wir wollen die Definition der entsprechenden Funktion „`=:=`“ der Vollständigkeit halber hier wiedergeben.

```
(Wait a : xs) =:= (Wait b : ys) =
  mergeWait (a<b) (a-b) a xs b ys ;
(Wait a : xs) =:= (y : ys) =
  y : ((Wait a : xs) =:= ys) ;
(x : xs) =:= ys  =  x : (xs =:= ys) ;
[] =:= ys  =  ys ;

mergeWait _eq 0 a xs _b ys =
  Wait a : (xs =:= ys) ;
mergeWait True d a xs _b ys =
  Wait a : (xs =:= (Wait (negate d) : ys)) ;
mergeWait False d _a xs b ys =
  Wait b : ((Wait d : xs) =:= ys) ;
```

Die grafische Bedienschnittstelle unseres Systems ist in Abbildung 1 zu sehen. Im linken oberen Teil kann der Benutzer den Programmtext eingeben. Mit einer bestimmten Tastenkombination kann er den Programmtext auf Syntaxfehler prüfen und in den Programmspeicher des Interpreters übernehmen. Das vom Interpreter ausgeführte Programm ist im rechten oberen Teil zu sehen. In diesem Teil hebt der Interpreter außerdem hervor, welche Teilausdrücke reduziert werden mussten, um den vorhergehenden in den aktuellen Ausdruck zu überführen. Auf diese Weise kann man den Verlauf der Melodie visuell verfolgen. Der aktuelle Term des Interpreters ist im unteren Teil der Bedienoberfläche dargestellt. Die in der Abbildung wiedergegebenen Texte entsprechen im Wesentlichen unserem

einführenden Beispiel, weisen aber unsere Melodie zusätzlich noch einem MIDI-Kanal zu und verwenden Definitionen von ++ und `map`, welche auf `foldr` aufbauen.

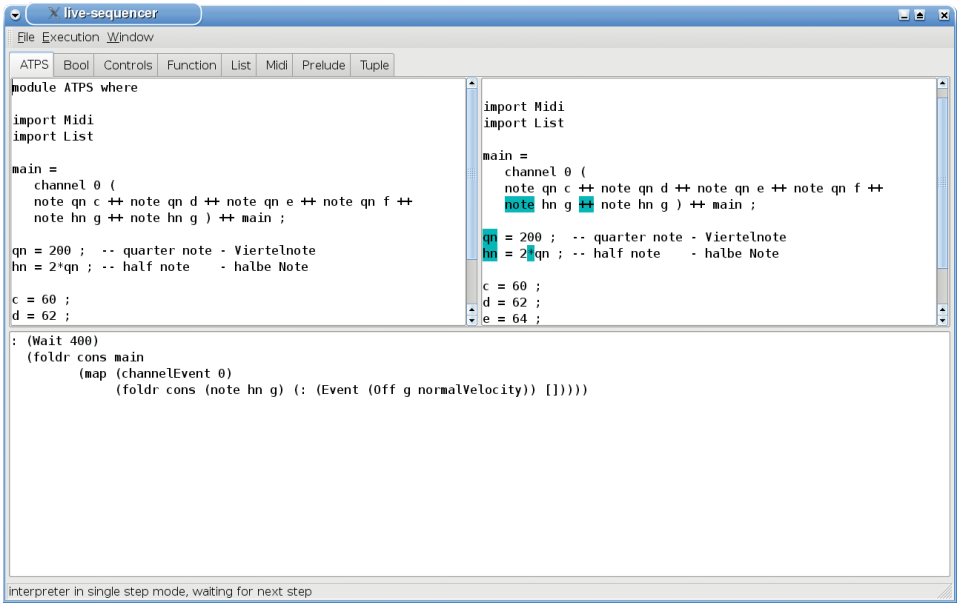


Abbildung 1: *Der Interpreter im Betrieb*

Das System verfügt über die Ausführungsmodi „Echtzeit“, „Zeitlupe“ und „Einzelschritt“. Der Echtzeit-Modus gibt die Musik wieder, so wie es die Notenlängen erfordern, während die anderen beiden Modi die Warte-Ereignisse ignorieren und stattdessen nach jedem Element der MIDI-Kommandoliste eine Pause machen. Die beiden letzten Modi sind zur Beobachtung der Ausführung und zur Fehlersuche gedacht. Sie eignen sich auch für den Einsatz in der Lehre, zur Erläuterung, wie ein Interpreter einer funktionalen Sprache mit Bedarfsauswertung im Prinzip funktioniert.

Der Interpreter ist in Haskell mit dem Glasgow Haskell Compiler GHC [PJ+12] implementiert und verwendet WxWidgets [SRZ+11] für die grafische Bedienoberfläche. Die verarbeitete Sprache unterstützt „Pattern matching“, vordefinierte Infix-Operatoren, Funktionen höherer Ordnung, unvollständige Funktionsanwendung. Aus Gründen der einfachen Implementierung gibt es bislang folgende Einschränkungen: Die interpretierte Sprache ist dynamisch typisiert, und kennt als Objekte ganze Zahlen, Texte und Konstruktoren. Sie ist formatfrei, weswegen nach Deklarationen stets Semikola gesetzt werden müssen. Viele syntaktische Besonderheiten werden nicht unterstützt, beispielsweise „List Comprehension“, „Operator Section“, Do-Notation, Let- und Case-Notation, frei definierbare Infix-Operatoren. Ein- und Ausgabeoperationen sind ebenfalls nicht verfügbar.

2.2 Verteilte Programmierung

Unser System soll es auch erlauben, das Publikum in eine Aufführung oder Studenten in die Vorlesung durch Programmieren einzubeziehen. Die typische Situation dafür ist, dass der Vortragende die Bedienoberfläche des Programms an die Wand projiziert, die Zuhörer die erzeugte Musik über eine Musikanlage hören und dass die Zuhörer über ein Funknetz und einen Browser Kontakt mit dem Vortragsrechner aufnehmen können.

Die implementierte funktionale Sprache verfügt über ein einfaches Modulsystem. Der Vortragende kann auf diese Weise ein Musikstück in mehrere Abschnitte oder Tonspuren zerlegen, und jeden dieser Teile in einem Modul ablegen. Die Module wiederum kann er Zuhörern zuweisen. Außerdem kann der Vortragende durch Einfügen eines bestimmten Kommentars festlegen, ab welcher Zeile der Zuhörer den Modulinhalt verändern darf. In den Zeilen davor stehen üblicherweise der Modulname, die Liste der exportierten Bezeichner, die Liste der importierten Module und grundlegende Definitionen. Auf diese Weise kann der Vortragende eine Schnittstelle für jedes Modul vorgeben.

Ein Zuhörer kann nun über einen WWW-Browser ein Modul abrufen und den veränderbaren Teil editieren. (Siehe Abbildung 2) Nach dem Editieren kann er den veränderten Inhalt an den Server schicken. Dieser ersetzt im Editor den Modultext unterhalb des Markierungskommentars mit dem neuen Inhalt. Dann wird der Text syntaktisch geprüft und im Erfolgsfall an den Interpreter weitergeleitet. Ist der Text syntaktisch nicht korrekt, so bleibt er im Editor, damit er notfalls vom Vortragenden überprüft und korrigiert werden kann.

Im Allgemeinen wird es nicht gelingen, ohne Vorbereitung auf diese Weise ein völlig neues Musikstück zu erschaffen. Der Vortragende kann aber seine Aufführung vorbereiten, indem er sich eine Aufteilung in Module überlegt und diese mit grundlegenden Definitionen füllt. Dies können Definitionen von Funktionen sein, die eine Liste von Nullen und Einsen in einen Rhythmus verwandeln, oder eine Liste von Zahlen in ein Akkordmuster oder eine Bassbegleitung. Der Vortragende kann dann durch Vorgabe von Takt und von Harmonien sicher stellen, dass die einzelnen Stücke zusammenpassen. Der Vortragende übernimmt in diesem Szenario nicht mehr die Rolle des Komponisten, sondern eher die Rolle des Dirigenten.

3 Verwandte Arbeiten

Algorithmische Komposition hat inzwischen eine lange Tradition. Als Beispiele seien hier nur Mozarts musikalische Würfelspiele und die Illiac-Suite [HI59] genannt. Auch gibt es mit Haskore [HMGW96] seit einiger Zeit die Möglichkeit, Musik in Haskell zu programmieren und damit verschiedene Klangerzeuger über MIDI zu steuern oder mit CSound, SuperCollider oder mit in Haskell geschriebenen Audiosynthesefunktionen Audiodateien zu erzeugen. Haskore baut ebenfalls auf der Bedarfsauswertung auf und erlaubt die elegante Definition von formal großen oder unendlichen Musikstücken bei geringem tatsächlichem Speicherverbrauch bei der Interpretation. Das kreative Komponieren wird allerdings da-

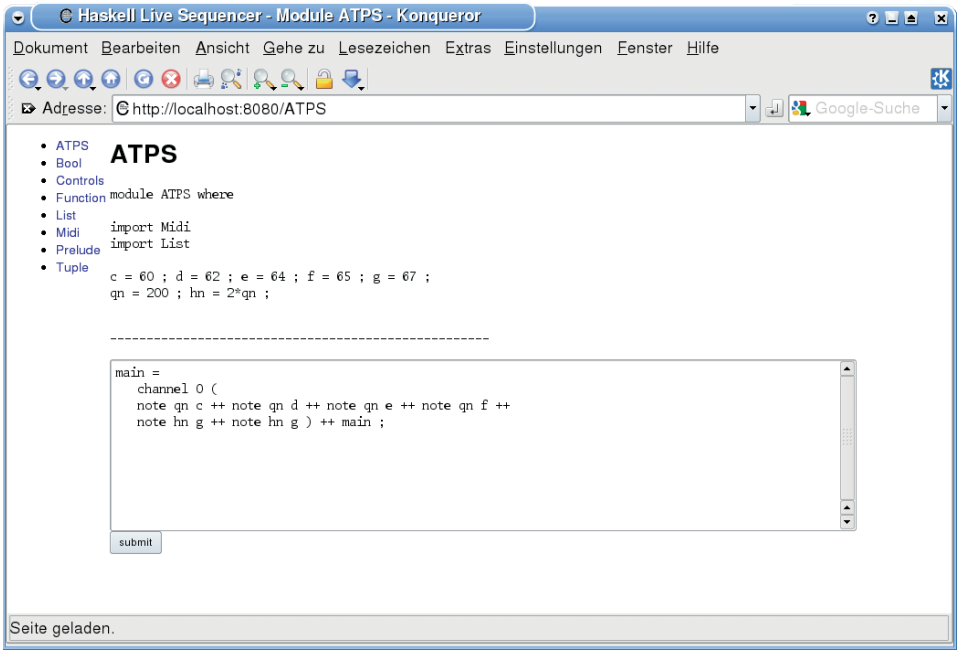


Abbildung 2: *Zugriff auf ein Modul über das Netz*

durch erschwert, dass man Änderungen erst nach Abbruch und Neustart des Programms hören kann.

Ein sehr populärer Ansatz zur Programmierung von Animationen, Robotersteuerungen, grafischen Bedienschnittstellen und Audiosignalverarbeitung in Haskell ist die funktionale reaktive Programmierung [EH97]. Wie für unsere Musikstücke wird auch hier der zeitliche Verlauf einer Animation, einer graphischen Bedienoberfläche oder ähnlichem durch eine Art unendliche Liste beschrieben. Darüber hinaus soll ein FRP-Programm auch auf äußere Ereignisse, wie zum Beispiel Bewegungen der Computermaus reagieren können. Im Gegensatz zu unserer Arbeit ist bisher allerdings nicht möglich, ein FRP-Programm während seiner Ausführung zu ändern.

Eine funktionale, aber nicht rein funktionale, Programmiersprache, die Änderungen des Programmtextes während der Programmausführung erlaubt, ist Erlang [Arm97]. Erlang folgt der strengen Semantik. Man könnte in Erlang eine Folge von MIDI-Kommandos nicht durch eine (lazy) Liste von Konstruktoren beschreiben, sondern bräuchte Iteratoren oder ähnliches. In ein laufendes Erlang-Programm kann neuer Programmcode auf zwei verschiedene Weisen eingebracht werden, entweder in dem das laufende Programm Funktionen (zum Beispiel Lambda-Ausdrücke) aufruft, die ihm als Nachrichten zugesandt werden oder indem ein Erlang-Modul durch ein neues Modul ersetzt wird. Wird ein Erlang-Modul nachgeladen, so behält das Laufzeitsystem die alte Version des Moduls im Speicher, um laufende Programmteile ausführen zu können. Lediglich modul-externe Aufrufe

springen in das neue Modul, wobei man einen externen Aufruf auch innerhalb desselben Moduls starten kann. Auch in Erlang müssen also Sollbruchstellen (externe Aufrufe oder Funktionsübernahme aus Nachrichten) zum späteren Einfügen von neuem Code geschaffen werden.

Unser Ansatz zur Änderung von Programmtext während der Programmausführung ist damit sehr ähnlich zum „Hot Code loading“ in Erlang. Die Bedarfsauswertung in unserem Interpreter führt jedoch dazu, dass beträchtliche Teile des Programmtextes im aktuellen Term enthalten sind. Diese werden von Änderungen an einem Modul nicht unmittelbar betroffen. Auf diese Weise ist es in unserem Ansatz nicht unbedingt nötig, zwei Versionen eines Moduls im Speicher zu halten, um einen glatten Übergang von altem zu neuem Programmtext zu erreichen.

Sogenanntes Musik-Live-Coding, also das Programmieren eines musikerzeugenden Programms, während die Musik läuft, war bislang Spezialsprachen wie ChucK [WC04] und SuperCollider/SCLang [McC96] und ihren Implementierungen vorbehalten. Diese Sprachen sind beim Kontrollfluss an das imperative Programmierparadigma angelehnt und beim Typsystem an die objektorientierte Programmierung. Im wesentlichen funktionieren beide wie eine Client-Server-Lösung, wobei der Server die Klänge erzeugt und parallel zu einem Kommandozeileninterpreter läuft, von dem aus man Befehle an den Server schicken kann.

Auch in unserer Architektur läuft die Klangerzeugung parallel zur eigentlichen Programmierung und wird mit (MIDI-)Kommandos gesteuert. Jedoch wird in unserem Ansatz nicht programmiert, wie sich die Klangerzeugung ändern soll, sondern das erzeugende Programm wird direkt geändert.

4 Folgerungen und zukünftige Arbeiten

Unsere vorgestellte Technik zeigt einen neuen Weg der Live-Programmierung von Musik auf, der sich möglicherweise auch auf die Wartung anderer lange laufender funktionaler Programme übertragen lässt. Dennoch zeigt sich, dass man schon beim Programmieren einer ersten Version gewisse Sollbruchstellen vorsehen muss, an denen man später im laufenden Programm Änderungen einfügen kann. Auch mit automatischen Optimierungen des Programms müssen wir jetzt vorsichtig sein, denn eine Optimierung könnte eine solche Sollbruchstelle entfernen. Wenn ein Programm zur Laufzeit geändert wird, so sind Funktionen eben nicht mehr „referential transparent“, womit wir eine wichtige Eigenschaft der funktionalen Programmierung aufgeben.

Typsystem Um die Gefahr zu verringern, dass ein Musikprogramm nach einer Änderung wegen eines Programmfehlers abbricht, ist der nahe liegende nächste Schritt der Einsatz eines statischen Typprüfers. Dieser müsste nicht nur testen, ob das vollständige Programm nach Austausch eines Moduls noch typkorrekt ist, sondern er müsste zudem testen, dass der aktuelle Term im Interpreter bezüglich des neuen Programms typkorrekt ist.

Noch wichtiger wird ein Typprüfer im Mehrbenutzerbetrieb. Der Leiter einer Programmierveranstaltung mit mehreren Programmierern könnte jedem Teilnehmer Typsignaturen im nicht editierbaren Bereich seines Moduls vorgeben, die der Teilnehmer implementieren muss. Der Typprüfer würde dafür sorgen, dass Teilnehmer nur Änderungen einschicken können, die zum Rest des Musikstücks passen.

Auswertungsstrategie Derzeit ist unser Interpreter sehr einfach gehalten. Der aktuelle Term ist ein reiner Baum. In dieser Darstellung können wir nicht ausdrücken, dass der Wert eines Terms mehrmals verwendet wird. (Es gibt also kein „sharing“.) Wenn beispielsweise `f` definiert ist als `f x = x:x:[]`, dann wird der Aufruf `f (2+3)` reduziert zu `(2+3) : (2+3) : []`. Wenn weiterhin das erste Listenelement zu `5` reduziert wird, wird das zweite Listenelement nicht reduziert. Wir erhalten also `5 : (2+3) : []` und nicht `5 : 5 : []`. Da der Term nur ein Baum ist und kein Graph, brauchen wir keine eigene Speicherverwaltung, sondern können uns auf die automatische Speicherverwaltung des GHC-Laufzeitsystems verlassen, in dem der Interpreter läuft. Wenn ein Teilterm nicht mehr benötigt wird, so wird er aus dem Operatorbaum entfernt und früher oder später vom Laufzeitsystem des GHC freigegeben.

Selbst einfache korekursive Definitionen wie die der Fibonacci-Zahlen durch

```
fix (\fibs -> 0 : 1 : zipWith (+) fibs (tail fibs))
```

führen bei diesem Auswertungsverfahren zu einem unbegrenzten Anstieg der Termgröße. In Zukunft sollen daher weitere Auswertungsstrategien wie zum Beispiel die Graphreduktion mit der STG-Maschine [PJ92] hinzukommen, die dieses und weitere Probleme lösen. Anstelle eines Operatorbaums würde der aktuelle Term dann aus einem Operatorgraph bestehen, die Anwendung der Funktionsdefinitionen und damit die Möglichkeit der Live-Änderung einer Definition bliebe prinzipiell erhalten. Die Gefahr bei Live-Musikprogrammierung liegt natürlich darin, dass Programmänderungen abhängig von der Auswertungsstrategie verschiedene Auswirkungen auf den Programmablauf haben können. Das Verwenden des gleichen Objektes im Speicher an verschiedenen Stellen im aktuellen Term („sharing“) würde zwar im obigen Beispiel der Fibonacci-Zahlen den Speicherverbrauch begrenzen, könnte aber auch verhindern, dass eine geänderte Definition der aufgerufenen Funktionen noch berücksichtigt wird. Der Einzelschrittmodus würde es ermöglichen, in der Lehre verschiedene Auswertungsverfahren zu demonstrieren und Vor- und Nachteile miteinander zu vergleichen.

Offen ist, ob und wie wir unser System, das Änderungen des Programms während des Programmablaufs zulässt, direkt in eine existierende Sprache wie Haskell einbetten können. Dies würde es uns vereinfachen, die Wechselwirkung zwischen Programmänderungen, Optimierungen und Auswertungsstrategien zu untersuchen.

Hervorhebungen Es gibt noch ein weiteres interessantes offenes Problem: Wie kann man Textstellen im Programm passend zur erzeugten Musik hervorheben? Es liegt nahe, die jeweils gespielten Noten hervorzuheben. Dies wird zur Zeit dadurch erreicht, dass in

einer Wartephase alle die Symbole hervorgehoben werden, welche seit der letzten Wartephase durch den Interpreter reduziert wurden. Wenn aber eine langsame Melodie parallel zu einer schnellen Folge von Regleränderungen abgespielt wird, so führt das dazu, dass die Noten der Melodie nur kurz hervorgehoben werden, nämlich immer nur für die kurze Zeit, in der der Reglerwert konstant bleibt. Wir würden aber erwarten, dass die Hervorhebung eines Musikteils nicht von parallel laufenden Teilen beeinflusst wird. Formal könnten wir es so ausdrücken: Gegeben seien die serielle Komposition `++` und die parallele Komposition `=:=`, die sowohl für Terme als auch für Hervorhebungen definiert sein sollen. Gegeben sei weiterhin die Abbildung `highlight`, welche einen Term seiner Visualisierung zuordnet. Dann soll für zwei beliebige Musikobjekte `a` und `b` gelten:

```
highlight (a ++  b) = highlight a ++  highlight b
highlight (a =:= b) = highlight a =:= highlight b
```

Wenn man alle Symbole hervorhebt, die mittelbar an der Erzeugung eines `NoteOn`- oder `NoteOff`-MIDI-Kommandos beteiligt waren, dann erhält man eine Funktion `highlight` mit diesen Eigenschaften. Allerdings führt sie dazu, dass die Notenaufrufe kumulativ hervorgehoben werden. In

```
note qn c ++ note qn d ++ note qn e ++ note qn f
```

werden bei Wiedergabe von `note qn e` auch `note qn c` und `note qn d` hervorgehoben, denn diese erzeugen Listen und dass diese Listen endlich sind, ist ein Grund dafür, dass aktuell `note qn e` wiedergegeben werden kann. Die Aufrufe `note qn c` und `note qn d` sind also notwendigerweise daran beteiligt, dass `note qn e` reduziert werden kann.

Zeitsteuerung Eine weitere Schwierigkeit besteht im zeitlich präzisen Versand der MIDI-Kommandos. Bislang wartet der Interpreter bis zu dem Zeitpunkt, an dem eine MIDI-Nachricht verschickt werden soll, und beginnt dann erst mit der Berechnung des entsprechenden Listenelements. Wir vertrauen also darauf, dass die Berechnung schnell genug beendet wird und sich der Versand nicht allzu stark verzögert. Bei komplizierteren Berechnungen trifft diese Annahme natürlich nicht zu. Eine höhere Präzision könnten wir erreichen, indem wir die MIDI-Nachrichten mit einem Zeitstempel versehen und einige Zeit im Voraus verschicken. Das wirft neue Probleme der Synchronisation von Musik und grafischer Darstellung des Interpreterzustandes auf und es würde auch heißen, dass die Musik erst verzögert angehalten werden kann und man nur verzögert in einen anderen Ausführungsmodus wechseln kann.

5 Danksagungen

Dieses Projekt basiert auf einer Idee von Johannes Waldmann, die ich gemeinsam mit ihm entwickelt und in einen Prototypen umgesetzt habe. Ich möchte mich bei ihm, Heinrich

Apfelmus und den anonymen Gutachtern für das gründliche Lesen und für zahlreiche Verbesserungsvorschläge zu diesem Artikel bedanken.

Informationen über dieses Projekt zu Programmentwicklung, Demonstrationsvideos und Artikeln können Sie unter

```
http://www.haskell.org/haskellwiki/Live-Sequencer
```

abrufen.

Literatur

[Arm97] Joe Armstrong. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP 1997, Seiten 196–203, New York, NY, USA, 1997. ACM.

[EH97] Conal Elliott und Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Jgg. 32, Seiten 263–273, August 1997.

[HI59] Lejaren A. Hiller und Leonard M. Isaacson. *Experimental Music: Composition With an Electronic Computer*. McGraw-Hill, New York, 1959.

[HMGW96] Paul Hudak, T. Makucevich, S. Gadde und B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[McC96] James McCartney. Super Collider. `http://www.audiosynth.com/`, March 1996.

[MMA96] MMA. Midi 1.0 detailed specification: Document version 4.1.1. `http://www.midi.org/about-midi/specinfo.shtml`, February 1996.

[PJ92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PJ+98] Simon Peyton Jones et al. Haskell 98 Language and Libraries, The Revised Report. `http://www.haskell.org/definition/`, 1998.

[PJ+12] Simon Peyton Jones et al. GHC: The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2012.

[SRZ+11] Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn et al. wxWidgets 2.8.12. `http://docs.wxwidgets.org/stable/`, March 2011.

[WC04] Ge Wang und Perry Cook. ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, Seiten 812–815, New York, NY, USA, 2004. ACM.

An Abstract Machine for Concurrent Haskell with Futures

David Sabel

Computer Science Institute
Goethe-University Frankfurt am Main
sabel@ki.informatik.uni-frankfurt.de

Abstract: We show how Sestoft's abstract machine for lazy evaluation of purely functional programs can be extended to evaluate expressions of the calculus CHF – a process calculus that models Concurrent Haskell extended by imperative and implicit futures. The abstract machine is modularly constructed by first adding monadic IO-actions to the machine and then in a second step we add concurrency. Our main result is that the abstract machine coincides with the original operational semantics of CHF, w.r.t. may- and should-convergence.

1 Introduction

The process calculus *CHF* [SSS11a] is a model of the core language of Concurrent Haskell [PGF96, Pey01, PS09] but extended by implicit, concurrent futures which allow a declarative style of concurrent programming.

CHF is monomorphically typed and its syntax comprises (unlike the π-calculus [Mil99, SW01]) shared memory in form of Haskell's MVars, threads (i.e. futures) and heap bindings. Threads evaluate expressions which on the one hand may be monadic operations to create and access the MVars and to spawn new threads, and on the other hand are usual pure functional expressions extending the lambda calculus by data constructors, case-expressions, recursive let-expressions, as well as Haskell's `seq`-operator.

In [SSS11a] the operational semantics of *CHF* is defined by a small-step reduction as rewriting on processes. Program equivalence of processes and also expressions is given by a *contextual equivalence*: two programs are equal iff their observable behavior is indistinguishable even if the programs are used as a subprogram of any other program (i.e. if the programs are plugged into any arbitrary context). Besides observing whether a program *may* terminate (called *may-convergence*) contextual equivalence also tests whether a program never loses the ability to terminate (called *should-convergence*, or sometimes must-convergence, see e.g. [CHS05, NSSSS07, RV07, SSS08]). The classic notion of must-convergence additionally requires that all possible evaluations terminate. An advantage of using should-convergence is that it is invariant w.r.t. restricting the evaluator to fair scheduling (see e.g [SSS11a]), that contextual equivalence is closed w.r.t. a whole class of convergence predicates (see [SSS10]), and that inductive reasoning is possible.

In [SSS11a] contextual equivalence in *CHF* is deeply investigated and a lot of equiva-

lences are proved, and recently [SSS11b] shows that CHF is a conservative extension of its purely functional sublanguage, i.e. all equations that hold in the pure call-by-need lambda calculus also hold in the process calculus CHF. The obtained results show that the given operational semantics works well for (mathematically formal) reasoning. On the other hand the operational semantics is not easy to implement as an interpreter for CHF, since e.g. reduction contexts in [SSS11a] have a complex definition and reduction uses structural congruence of processes implicitly.

Hence, the motivation of this paper is to investigate an alternative operational semantics for CHF which can easily be implemented as an interpreter, i.e. we will develop an *abstract machine* to evaluate expressions and processes of CHF. As a starting point, we will use the abstract machine mark 1 introduced by Sestoft [Ses97] for call-by-need evaluation of pure functional programs (which implements the natural semantics given by [Lau93]).

Sestoft's machine mark 1 is a variant of the Krivine-machine which additionally implements sharing during evaluation (see [DF07]). The main components are a heap to model shared bindings, an expression which is evaluated, and a stack to efficiently store the current evaluation context. There are only few transition rules which perform the unwinding to find the next redex, perform reduction, or access and update shared bindings.

Variants of Sestoft's machine are well-used for several call-by-need lambda calculi to define the operational semantics, or to give an alternative description of the semantics, respectively. Some examples are [MSC99] for a call-by-need lambda calculus with erratic choice, [Mor98, Sab08] for call-by-need lambda calculi with McCarthy's amb-operator, [BFKT00] for specifying the semantics of Parallel Haskell, and [AHH$^+$05] for the semantics of functional-logic languages.

To construct an abstract machine for CHF we extend (a slightly modified variant of) Sestoft's machine (called $M1$) in two steps. The first extension (called $IOM1$) is to add the ability to perform monadic I/O-operations, i.e. we add storage (i.e. MVars), a further stack, and machine transitions to execute monadic actions to the machine $M1$. In a second step we extend the machine $IOM1$ by concurrency, i.e. we allow several threads and add transitions to spawn new threads. The concurrent machine is called $CIOM1$. A nice property of our construction is *modularity*, i.e. every extended machine reuses the already introduced transitions of the machine before. Thus $CIOM1$ is easy to implement, and indeed within a few hours we programmed a prototype of the machine in Haskell.

Albeit providing such a machine is an interesting result for itself, we also show that our machine is a *correct* implementation of the operational semantics of CHF: In Theorem 4.11 we show that may- and should-convergence defined by the rewriting semantics in [SSS11a] coincides with may- and should-convergence on the machine for every expression the machine starts with.

The structure of the paper is as follows: In Section 2 we briefly recall the calculus CHF together with some results on program equivalences in CHF which are required in later proofs. In Section 3 we introduce the abstract machine $CIOM1$ for CHF, where we develop the machine in three steps. In Section 4 we show that machine $CIOM1$ correctly implements the operational semantics of CHF. We conclude in Section 5. Not all proofs are included in the paper, but can be found in the technical report [Sab12].

2 The Process Calculus CHF

We recall the calculus CHF which models Concurrent Haskell extended by concurrent futures [SSS11a]. In Fig. 1(a) the syntax of *processes Proc* and expressions *Exp* is shown, where we assume that x, x_i, y, y_i are variables of some countably infinite set of variables.

Parallel composition $P_1 \mid P_2$ composes processes, and *name restriction* $\nu x.P$ restricts the scope of variable x to process P. A *concurrent thread* $x \Leftarrow e$ evaluates the expression e and then binds the result to the variable x. We also call variable x the *future* x. In a process there is usually one distinguished thread – the *main thread* – which is labeled with "main" (as notation we use $x \overset{\mathsf{main}}{\Longleftarrow} e$). *Bindings* $x = e$ represent global shared expressions. *MVars* are synchronizing variables, where $x\,\mathbf{m}\,e$ represents a *filled MVar* with content e, and $x\,\mathbf{m}-$ represents an *empty MVar*. In both cases we call x the *name of the MVar*. For a process P we say a variable x is an *introduced variable* if x is a future, a name of an MVar, or a left hand side of a binding. A process is *well-formed*, if there exists at most one main thread $x \overset{\mathsf{main}}{\Longleftarrow} e$ and the introduced variables are pairwise distinct.

We assume a finite set of *data constructors* c which is partitioned into sets, such that each set represents a type T. The constructors of a type T are ordered as $c_{T,1}, \ldots, c_{T,|T|}$, where $|T|$ is the number of constructors belonging to type T. We omit the index T, i in $c_{T,i}$ if it is clear from the context. Each constructor $c_{T,i}$ has a fixed arity $\mathrm{ar}(c_{T,i}) \geq 0$. We assume that there is a unit type $()$ with a single constant $()$ as constructor.

Besides the lambda calculus, expressions *Exp* (see Fig. 1(a)) comprise (fully-saturated) *constructor applications* $(c\ e_1 \ldots e_{\mathrm{ar}(c)})$, `case`*-expressions*, `seq`-expressions for sequential evaluation, `letrec`*-expressions* to express recursive shared bindings and monadic expressions *MExp* (described below). For `case`-expressions there is a $\texttt{case}_T$-construct for every type T which must have a `case`-alternative for every constructor of type T. We sometimes abbreviate the `case`-alternatives as $alts$. Variables in `case`-patterns $(c\ x_1 \ldots x_{\mathrm{ar}(c)})$ and bound variables in `letrec`-expressions must be pairwise distinct.

The monadic expression `return` e represents the monadic action which returns expression e, the binary operator $>\!\!>\!\!=$ combines monadic actions, the expression `future` e creates a concurrent thread evaluating the action e, the operation `newMVar` e creates an MVar filled with e, `takeMVar` x returns the content of MVar x, and `putMVar` $x\ e$ fills MVar x with e. `takeMVar` x blocks on an empty MVar, and `putMVar` $x\ e$ blocks on a filled MVar.

Example 2.1. *Futures allow a declarative programming style, since they allow implicit synchronization. Assume that* act_1, act_2 *perform computations resulting in numbers, and that we want to sum up both results when they are available, then we can use the action:*

$$\texttt{future}\ act_1 >\!\!>\!\!= \lambda res_1.\texttt{future}\ act_2 >\!\!>\!\!= \lambda res_2.\texttt{return}\ (res_1 + res_2)$$

Executing this action starts two concurrent futures for performing the actions act_1 *and* act_2*. The corresponding futures* res_1 *and* res_2 *are like pointers that will eventually point to the corresponding results. The futures are implicit, since there is no need to explicitly force the results of* res_1, res_2 *before computing the sum.*

Variables get bound by abstractions, `letrec`-expressions, `case`-alternatives, and by the

$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x.P \mid x \Leftarrow e \mid x = e \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}-$

$e, e_i \in \mathit{Exp} ::= x \mid m \mid \lambda x.e \mid (e_1\ e_2) \mid c\,e_1 \ldots e_{\mathrm{ar}(c)} \mid \mathtt{seq}\ e_1\ e_2$
$\mid \mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e$
$\mid \mathtt{case}_T\ e\ \mathtt{of}\ alt_{T,1} \ldots\ alt_{T,|T|}$ where $alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$

$m \in \mathit{MExp} ::= \mathtt{return}\,e \mid e_1 \gg\!= e_2 \mid \mathtt{future}\,e$
$\mid \mathtt{takeMVar}\,e \mid \mathtt{newMVar}\,e \mid \mathtt{putMVar}\,e_1\,e_2$

$\tau, \tau_i \in \mathit{Typ} ::= \mathtt{IO}\,\tau \mid (T\ \tau_1 \ldots \tau_n) \mid \mathtt{MVar}\,\tau \mid \tau_1 \to \tau_2$

(a) Syntax of Processes, Expressions, Monadic Expressions and Types

$\mathbb{D} \in \mathit{PC} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$ $\qquad$ $\mathbb{M} \in \mathit{MC} ::= [\cdot] \mid \mathbb{M} \gg\!= e$

$\mathbb{E} \in \mathit{EC} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\mathtt{case}\ \mathbb{E}\ \mathtt{of}\ alts) \mid (\mathtt{seq}\ \mathbb{E}\ e)$

$\mathbb{F} \in \mathit{FC} ::= \mathbb{E} \mid (\mathtt{takeMVar}\ \mathbb{E}) \mid (\mathtt{putMVar}\ \mathbb{E}\ e)$

$\mathbb{L} \in \mathit{LC} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
where $\mathbb{E}_2, \ldots \mathbb{E}_n$ are not the empty context.

$\widehat{\mathbb{L}} \in \widehat{\mathit{LC}} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
where $\mathbb{E}_1, \mathbb{E}_2, \ldots \mathbb{E}_n$ are not the empty context.

(b) Process-, Monadic-, Evaluation-, and Forcing-Contexts

Monadic Computations:

(lunit) $y \Leftarrow \mathbb{M}[\mathtt{return}\ e_1 \gg\!= e_2] \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[e_2\ e_1]$

(tmvar) $y \Leftarrow \mathbb{M}[\mathtt{takeMVar}\ x] \mid x\,\mathbf{m}\,e \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[\mathtt{return}\ e] \mid x\,\mathbf{m}-$

(pmvar) $y \Leftarrow \mathbb{M}[\mathtt{putMVar}\ x\ e] \mid x\,\mathbf{m}- \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mid x\,\mathbf{m}\,e$

(nmvar) $y \Leftarrow \mathbb{M}[\mathtt{newMVar}\ e] \xrightarrow{\mathit{CHF}} \nu x.(y \Leftarrow \mathbb{M}[\mathtt{return}\ x] \mid x\,\mathbf{m}\,e)$

(fork) $y \Leftarrow \mathbb{M}[\mathtt{future}\ e] \xrightarrow{\mathit{CHF}} \nu z.(y \Leftarrow \mathbb{M}[\mathtt{return}\ z] \mid z \Leftarrow e)$
where z is fresh and the created thread is not the main thread

(unIO) $y \Leftarrow \mathtt{return}\ e \xrightarrow{\mathit{CHF}} y = e$ if the thread is not the main-thread

Functional Evaluation:

(cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{\mathit{CHF}} \widehat{\mathbb{L}}[v] \mid x = v$ $\qquad$ if v is an abstraction or a variable

(cpcx) $\widehat{\mathbb{L}}[x] \mid x = c\,e_1 \ldots e_n$ $\qquad$ if c is a constructor, or a monadic operator
$\xrightarrow{\mathit{CHF}} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\,y_1 \ldots y_n] \mid x = c\,y_1 \ldots y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$

(mkbinds) $\mathbb{L}[\mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e]$
$\xrightarrow{\mathit{CHF}} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

(lbeta) $\mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{\mathit{CHF}} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

(case) $\mathbb{L}[\mathtt{case}_T\ (c\,e_1\ \ldots\ e_n)\ \mathtt{of}\ \ldots (c\,y_1\ \ldots\ y_n \to e)\ldots]$
$\xrightarrow{\mathit{CHF}} \nu y_1, \ldots, y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$

(seq) $\mathbb{L}[(\mathtt{seq}\ v\ e)] \xrightarrow{\mathit{CHF}} \mathbb{L}[e]$ $\qquad$ if v is a functional value

Closure: If $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$, and $P_1' \xrightarrow{\mathit{CHF}} P_2'$ then $P_1 \xrightarrow{\mathit{CHF}} P_2$

(c) Standard Reduction Rules

Figure 1: The Calculus *CHF*

restriction $\nu x.P$. This induces a notion of free and bound variables. With $FV(P)$ ($FV(e)$, resp) we denote the free variables of process P (expression e, resp.) and with $=_\alpha$ we denote α-equivalence. We assume that the *distinct variable convention* holds, i.e. all free variables are distinct from bound variables, all bound variables are pairwise distinct, and reductions implicitly perform α-renaming to obey this convention. For processes *structural congruence* $\equiv$ is defined as the least congruence satisfying the equations $P_1 \mid P_2 \equiv P_2 \mid P_1$; $\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P$; $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$; $P_1 \equiv P_2$, if $P_1 =_\alpha P_2$; and $(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2)$, if $x \notin FV(P_2)$.

For typing of processes and expressions *CHF* uses a monomorphic type system where data constructors and monadic operators are treated like "overloaded" polymorphic constants. The syntax of types *Typ* is shown in Fig. 1(a). $\texttt{IO}\,\tau$ means a monadic action with result type τ, $\texttt{MVar}\,\tau$ means an `MVar`-reference with content type τ, and $\tau_1 \to \tau_2$ is a function type. For a constructor c we let $\texttt{types}(c)$ be set of its monomorphic types. For simplicity we assume that every variable x has a fixed (built-in) type given by a global typing function Γ, i.e. $\Gamma(x)$ is the type of variable x. For space reasons we omit the typing rules of [SSS11a], but we use the notation $\Gamma \vdash P :: \texttt{wt}$ ($\Gamma \vdash e :: \tau$, resp.) meaning that (well-formed) process P can be well-typed (expression e can be well-typed with type τ, resp.) using the global typing function Γ. Special typing restrictions are that $x \Leftarrow e$ is well-typed, if $\Gamma \vdash e :: \texttt{IO}\,\tau$, and $\Gamma(x) = \tau$, and that the first argument of `seq` must not be an `IO`- or `MVar`-type, since otherwise the monad laws would not hold in *CHF* (and even not in Haskell, see [SSS11a]).

Operational Semantics and Program Equivalence The operational semantics of *CHF* (see [SSS11a]) is given by a small-step reduction which implements a call-by-need strategy. The definition requires several classes of contexts, which are shown in Fig. 1(b). For processes there are *process contexts PC*. For expressions, *monadic contexts MC* are used to find the first monadic action in a sequence of actions. For the evaluation of pure expressions, usual (call-by-name) *expression evaluation contexts EC* are used, and to enforce the evaluation of the (first) argument of the monadic operators `takeMVar` and `putMVar`, the class of *forcing contexts FC* is used. Since we follow a call-by-need strategy, we sometimes need to search a redex along a chain of bindings, which is expressed by the *LC*-contexts and as a special case by the $\widehat{LC}$-contexts.

Definition 2.2 (Call-by-Need Standard Reduction)**.** *A* functional value *is an abstraction or a constructor application, a* value *is a functional value or a monadic expression of MExp. The* call-by-need standard reduction $\xrightarrow{CHF}$ *is defined by the rules and the closure in Fig. 1(c). We assume that only well-formed processes are reducible. We also assume that successful processes (see below, Definition 2.3) are irreducible.*

The rules for functional evaluation include a sharing variant of β-reduction (rule (lbeta)), a rule for copying shared bindings into a needed position: For abstractions rule (cp) is used and for constructor applications rule (cpcx) shares the arguments before copying the constructor. The rules (case) and (seq) evaluate `case`- and `seq`-expressions, and the rule (mkbinds) moves `letrec`-bindings into the global set of shared bindings. For monadic computations the rule (lunit) applies the first monad law to proceed a sequence of monadic actions. The rules (nmvar), (tmvar), and (pmvar) handle the creation of and the access to

MVars where (tmvar) can only be performed on a filled MVar, and (pmvar) requires an empty MVar. The rule (fork) spawns a new concurrent thread, where the calling thread receives the name of the future as result. If a concurrent thread finishes its computation, then the result is shared as a global binding and the thread is removed (rule (unIO)).

For a reduction $\rightarrow$ (and also transitions and transformations) we denote with $\xrightarrow{+}$, $\xrightarrow{*}$ the transitive and the reflexive-transitive closure of $\rightarrow$, respectively. The notation $\xrightarrow{k}$ means a sequence of k $\rightarrow$-steps and $\xrightarrow{0\vee1}$ mean one or no reduction. We also sometimes attach a specific label to the arrow if we mean a specific reduction, and also write (CHF, a) for a CHF-standard reduction of kind a.

Contextual equivalence equates two processes P_1, P_2 if their observable behavior is indistinguishable if P_1 and P_2 are plugged into any process context. Thereby the usual observation is whether the evaluation of the process successfully terminates (called may-convergence). However, this observation is not sufficient in a concurrent setting, and thus we will observe may-convergence and a variant of must-convergence (called should-convergence, see also [RV07, SSS08, SSS11a]):

Definition 2.3. *A process P is* successful *iff it is well-formed and contains a main thread of the form $x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e$. A process P* may-converges *(written as $P{\downarrow}$), iff it is well-formed and reduces to a successful process, i.e. $\exists P' : P \xrightarrow{CHF,*} P' \wedge P'$ is successful. If $P{\downarrow}$ does not hold, then P* must-diverges *written as $P{\Uparrow}$. A process P* should-converges *(written as $P{\Downarrow}$), iff it is well-formed and remains may-convergent under reduction, i.e. $\forall P' : P \xrightarrow{CHF,*} P' \implies P'{\downarrow}$. If P is not should-convergent, then we say P* may-diverges*, written as $P{\uparrow}$. For an expression $e :: \texttt{IO}\ \tau$ we write $e\chi$ for any $\chi \in \{{\downarrow}, {\Downarrow}, {\uparrow}, {\Uparrow}\}$ iff $P\chi$ where $P := x \overset{\mathsf{main}}{\Longleftarrow} e$ and $x \notin FV(e)$.*

Note that $P{\uparrow}$ iff there is a finite reduction sequence $P \xrightarrow{CHF,*} P'$ such that $P'{\Uparrow}$.

Definition 2.4. Contextual approximation $\leq_{CHF}$ *and* contextual equivalence $\sim_{CHF}$ *on processes are defined as $\leq_{CHF} := \leq_{\downarrow} \cap \leq_{\Downarrow}$ and $\sim_{CHF} := \leq_{CHF} \cap \geq_{CHF}$ where*

$$\begin{array}{lll} P_1 \leq_{\downarrow} P_2 & \textit{iff} & \forall \mathbb{D} \in \textit{PC} : \mathbb{D}[P_1]{\downarrow} \implies \mathbb{D}[P_2]{\downarrow} \\ P_1 \leq_{\Downarrow} P_2 & \textit{iff} & \forall \mathbb{D} \in \textit{PC} : \mathbb{D}[P_1]{\Downarrow} \implies \mathbb{D}[P_2]{\Downarrow} \end{array}$$

Transformations and Reduction Lengths in CHF We recall some results of [SSS11a] on the correctness of several program transformations for CHF. Moreover, for some specific cases we prove that the reduction length of a standard reduction is not increased by a transformation. These results will be necessary later when we show that the abstract machine is a correct evaluator for CHF.

A *program transformation* γ is a binary relation on processes. It is *correct* iff $\gamma \subseteq \sim_{CHF}$.

In Fig. 2 some program transformations are defined, where $\mathbb{C}$ is a process context with an expression hole. The general copying rule (gcp) allows to copy a binding into an arbitrary position, the transformation (cpx) is the special case where the copied expression is a variable, and the transformation (cpcxxL) is the special case of (gcp) where the copied

(gcp) $\mathbb{C}[x] \mid x = e \to \mathbb{C}[e] \mid x = e$

(cpx) $\mathbb{C}[x] \mid x = y \to \mathbb{C}[y] \mid x = y$, where y is a variable

(cpcxxL) $\widehat{\mathbb{L}}[x] \mid x = c\, y_1\ \ldots\ y_n \to \widehat{\mathbb{L}}[c\, y_1\ \ldots\ y_n] \mid x = c\, y_1\ \ldots\ y_n$,
where c is a constructor or a monadic operator, $\widehat{\mathbb{L}} \in \widehat{LC}$, and all y_i are variables

(gc) $\nu x_1, \ldots, x_n.(P \mid \mathtt{Comp}(x_1) \mid \ldots \mid \mathtt{Comp}(x_n)) \to P$
if for all $i \in \{1, \ldots, n\}$: $\mathtt{Comp}(x_i)$ is a binding $x_i = e_i$, an MVar $x_i\, \mathbf{m}\, e_i$,
or an empty MVar $x_i\, \mathbf{m}\, -$, and $x_i \notin FV(P)$.

Figure 2: The Transformations (gcp), (cpx), (cpcxxL), and (gc)

expression is a constructor application or a monadic operator where all arguments are variables and the target must be inside an $\widehat{LC}$-context. The rule (gc) performs garbage collection and thus allows to remove unused parts of the process.

Theorem 2.5 ([SSS11a])**.** *The reductions* (CHF, lunit), (CHF, nmvar), (CHF, fork), *and* (CHF, unIO) *are correct transformations. The transformations* (cp), (cpcx), (lbeta), (case), (seq), (mkbinds) *are correct as transformation in any context (i.e. the reduction rules in Fig. 1(c) where the context* $\mathbb{L}$ *is replaced by an arbitrary process context* $\mathbb{C}$ *with an expression hole) such that the scoping is not violated by the transformation. The transformations (gcp), (cpx), (cpcxxL), and (gc) are also correct.*

We introduce a special notion for reduction lengths:

Definition 2.6. *If* $P_0 \xrightarrow{CHF} P_1 \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_n$ *where* P_n *is successful (*$P_n\Uparrow$*, resp.) and* $m \leq n$ *is the number of all reductions except for* (cp)*-reductions that copy a variable, then we write* $P_0{\downarrow^{[m,n]}}P_n$ *(*$P_0{\uparrow^{[m,n]}}P_n$*, resp.). We omit the process* P_n *if it is not of interest.*

In [Sab12] we show that the following properties on reduction lengths hold:

Proposition 2.7. *Let* P_1 *and* P_2 *be processes such that* $P_1 \xrightarrow{a} P_2$ *where* $a \in \{(CHF, cp), (gc), (cpx)\}$*. Then* $P_1{\downarrow^{[m,n]}} \implies P_2{\downarrow^{[m',n']}}$ *and* $P_1{\uparrow^{[m,n]}} \implies P_2{\uparrow^{[m',n']}}$ *where in both cases* $m' \leq m$ *and* $n' \leq n$*. If* $P_1 \xrightarrow{cpcxxL} P_2$*, then* $P_1{\downarrow^{[m,n]}} \implies P_2{\downarrow^{[m',n']}}$ *and* $P_1{\uparrow^{[m,n]}} \implies P_2{\uparrow^{[m',n']}}$ *where in both cases* $m' \leq m$*.*

3 Constructing an Abstract Machine for CHF

The goal of this section is to introduce an abstract machine for CHF. The construction of the machine is performed in three steps: first the machine $M1$ for evaluating pure functional expressions is introduced, then the machine is extended to handle monadic actions (called $IOM1$) and finally concurrency is added resulting in the machine $CIOM1$.

An Abstract Machine for Evaluating Pure Expressions The abstract machine $M1$ evaluates pure functional programs. It is analogous to Sestoft's machine mark 1 [Ses97]

but extended to operate also on `case`- and `seq`-expressions and to "functionally evaluate" monadic expressions, i.e. they are treated like ordinary constructor applications and *not* as actions. All of our abstract machines will only evaluate *simplified* expressions (analogous to *normalized expressions* in [Lau93, Ses97]):

Definition 3.1. *Simplified expressions Exp_S and simplified monadic expressions $MExp_S$ are built by the following grammar, where x, x_i are variables:*

$$
\begin{aligned}
e, e_i \in Exp_S \;&::= x \mid me \mid \lambda x.e \mid (e\ x) \mid c\ x_1 \ldots x_{\mathrm{ar}(c)} \mid \texttt{seq}\ e\ x \\
&\mid \texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e \\
&\mid \texttt{case}_T\ e\ \texttt{of}\ alt_{T,1} \ldots\ alt_{T,|T|} \ \textit{ where } alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i) \\
me \in MExp_S ::= \;&\texttt{return}\ x \mid x_1 \gg\!= x_2 \mid \texttt{future}\ x \\
&\mid \texttt{takeMVar}\ x \mid \texttt{newMVar}\ x \mid \texttt{putMVar}\ x_1\ x_2
\end{aligned}
$$

Simplified process $Proc_S$ are defined like processes Proc where all expressions are simplified expressions and additionally all MVars have only variables as content.

We first define the state of $M1$:

Definition 3.2. *A* state *of machine $M1$ is a tuple $(\mathcal{H}, e, \mathcal{S})$ where: $\mathcal{H}$ is a* heap*, i.e. a mapping of (finitely many) variables to expressions. To make the mapping explicit, we use the notation $\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. We write $\mathcal{H}_1 \dot\cup \mathcal{H}_2$ for the disjoint union of the heaps $\mathcal{H}_1$ and $\mathcal{H}_2$. The second component, e, is a simplified expression. It is the currently evaluated expression. $\mathcal{S}$ is a stack, where allowed entries are $\#_{\mathsf{app}}(x)$, $\#_{\mathsf{seq}}(x)$, $\#_{\mathsf{case}}(alts)$, and $\#_{\mathsf{heap}}(x)$. We use list notation for stacks, i.e. $[]$ is the empty stack, and $a : \mathcal{S}$ is the stack with top entry a and tail $\mathcal{S}$.*

For a well-typed simplified expression e, the *initial state* of machine $M1$ is $(\emptyset, e, [])$. A state of $M1$ is a *final state* if it is of the form $(\mathcal{H}, v, [])$ where v is an abstraction, a constructor application, or a monadic expression. In Fig. 3(a) the *transition relation* $\xrightarrow{M1}$ of machine $M1$ is defined. The rules (pushApp), (pushSeq), and (pushAlts) perform unwinding to find the next redex. The corresponding contexts are stored on the stack. The rules (takeApp), (takeSeq), and (branch) perform beta-, seq-, and case-reduction. The rules (enter) and (update) are used to look up and restore (after a successful evaluation) bindings of the heap. The rule (mkBinds) moves local `letrec`-bindings into the (global) heap.

Compared to Sestoft's mark 1 we did some slight modifications (aside from handling `seq` and `case`): We did not include a rule (blackhole) for the case, that the redex is a variable which is not bound in the heap (e.g. this case may happen after trying to evaluate a recursive binding of the form $x \mapsto \texttt{seq}\ x\ x$). In our machine $M1$ there is simply no transition and the machine gets stuck. Another difference is in the (update) transition: While $M1$ allows to perform an update if the expression is a variable, Sestoft's mark 1 does not allow this transition. One reason for our modification is that later in the machine with IO-transitions ($IOM1$) we also must perform those updates, if the variables are names of MVars, e.g. for the process $y \Leftarrow \texttt{takeMVar}\ x \mid x = z \mid z\ \mathbf{m}\ v$ the name of the MVar z must be copied resulting in $y \Leftarrow \texttt{takeMVar}\ z \mid x = z \mid z\ \mathbf{m}\ v$. Finally, we do not explicitly perform α-renaming in our rules, but we assume that the distinct variable convention is always fulfilled and that necessary α-renamings are performed implicitly.

(pushApp) $(\mathcal{H}, (e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{app}}(x) : \mathcal{S})$

(pushSeq) $(\mathcal{H}, (\mathtt{seq}\ e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{seq}}(x) : \mathcal{S})$

(pushAlts) $(\mathcal{H}, (\mathtt{case}_T\ e\ \mathtt{of}\ alts), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{case}}(alts) : \mathcal{S})$

(takeApp) $(\mathcal{H}, \lambda x.e, \#_{\mathsf{app}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[y/x], \mathcal{S})$

(takeSeq) $(\mathcal{H}, v, \#_{\mathsf{seq}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, y, \mathcal{S})$, if v is an abstraction or a constructor app.

(branch) $(\mathcal{H}, (c\ x_1 \ldots x_n), \#_{\mathsf{case}}(\ldots (c\ y_1 \ldots y_n \to e) \ldots) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[x_i/y_i]_{i=1}^n, \mathcal{S})$

(enter) $(\mathcal{H} \cup \{y \mapsto e\}, y, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{heap}}(y) : \mathcal{S})$

(update) $(\mathcal{H}, v, \#_{\mathsf{heap}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \{y \mapsto v\}, v, \mathcal{S})$
if v is an abstraction, a constructor app., a monadic operator, or a variable with $v \neq y$

(mkBinds) $(\mathcal{H}, \mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \bigcup_{i=1}^n \{x_i \mapsto e_i\}, e, \mathcal{S})$

(a) Transition Relation $\xrightarrow{M1}$ of Machine $M1$

(M1) $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}')$
if $(\mathcal{H}, e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}', e', \mathcal{S}')$ on machine $M1$

(newMVar) $(\mathcal{H}, \mathcal{M}, \mathtt{newMVar}\ x, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{y\ \mathbf{m}\ x\}, \mathtt{return}\ y, [], \mathcal{I})$
where y is a fresh variable

(takeMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, x, [], \#_{\mathsf{take}} : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, \mathtt{return}\ y, [], \mathcal{I})$

(putMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, \mathtt{return}\ (), [], \mathcal{I})$

(pushTake) $(\mathcal{H}, \mathcal{M}, \mathtt{takeMVar}\ x, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{take}} : \mathcal{I})$

(pushPut) $(\mathcal{H}, \mathcal{M}, \mathtt{putMVar}\ x\ y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I})$

(pushBind) $(\mathcal{H}, \mathcal{M}, x \gg\!= y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\gg\!=}(y) : \mathcal{I})$

(lunit) $(\mathcal{H}, \mathcal{M}, \mathtt{return}\ x, [], \#_{\gg\!=}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (y\ x), [], \mathcal{I})$

(b) Transition Relation $\xrightarrow{IOM1}$ of Machine $IOM1$

(unIO) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, (\mathtt{return}\ y), [], [])\}) \xrightarrow{CIOM1} (\mathcal{H} \cup \{x \mapsto y\}, \mathcal{M}, \mathcal{T})$
if thread named x is not the main-thread

(fork) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, (\mathtt{future}\ y), [], \mathcal{I})\})$
$\xrightarrow{CIOM1} (\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, (\mathtt{return}\ z), [], \mathcal{I}), (z, y, [], [])\})$
where z is a fresh variable

(IOM1) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \cup \{(x, e, \mathcal{S}, \mathcal{I})\}) \xrightarrow{CIOM1} (\mathcal{H}', \mathcal{M}', \mathcal{T} \cup \{(x, e', \mathcal{S}', \mathcal{I}')\})$
if $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}')$ on machine $IOM1$.
The rule is only used if (fork) or (unIO) is not applicable for the thread named x.

(c) Transition Relation $\xrightarrow{CIOM1}$ of Machine $CIOM1$

Figure 3: Transition Relations of the Machines $M1$, $IOM1$, and $CIOM1$

Example 3.3. *We demonstrate the evaluation of machine $M1$:*

$$
\begin{array}{ll}
 & (\emptyset, \texttt{letrec}\ x_1 = (\lambda y.y)\ w,\ x_2 = \texttt{takeMVar}\ x_1, x_3 = x_2\ \texttt{in}\ (\lambda z.z)\ x_3, [])\\
\xrightarrow{M1,\text{mkBinds}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, (\lambda z.z)\ x_3, [])\\
\xrightarrow{M1,\text{pushApp}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, \lambda z.z, [\#_{\mathsf{app}}(x_3)])\\
\xrightarrow{M1,\text{takeApp}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, x_3, [])\\
\xrightarrow{M1,\text{enter}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1\}, x_2, [\#_{\mathsf{heap}}(x_3)])\\
\xrightarrow{M1,\text{update}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_2 \mapsto \texttt{takeMVar}\ x_1, x_3 \mapsto x_2\}, x_2, [])\\
\xrightarrow{M1,\text{enter}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2\}, \texttt{takeMVar}\ x_1, [\#_{\mathsf{heap}}(x_2)])\\
\xrightarrow{M1,\text{update}} & (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \texttt{takeMVar}\ x_1, [])
\end{array}
$$

The last state is a final state, since $M1$ treats monadic operators like values.

Extending $M1$ by Monadic I/O We will extend the machine $M1$, such that MVars and operations on MVars can be performed. We have to implement the operations of the monad, i.e. `return`, $>\!\!>\!\!=$ and the operations `takeMVar`, `putMVar`, `newMVar` to access and create MVars. The state of the machine is extended by two components: a set of MVars which models the memory and a further stack – called IO-stack – which allows a clean separation between monadic and functional evaluation. An *IO-stack* is a stack where the following entries are allowed: The symbol $\#_{\mathsf{take}}$ to store a `takeMVar` operation, entries of the form $\#_{\mathsf{put}}(x)$ to store a `putMVar`-operation, where x is the new (to-be-written) content of the MVar, and $\#_{>\!\!>\!\!=}(y)$ to store a $>\!\!>\!\!=$-operation, where y is the right argument of $>\!\!>\!\!=$.

Definition 3.4. *A state of the machine $IOM1$ is a tuple $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I})$ where heap $\mathcal{H}$, expression e, and stack $\mathcal{S}$ are as before (in machine $M1$). $\mathcal{M}$ is a set of MVars with variables as content: a filled MVar is written as $x\ \mathbf{m}\ y$, and an empty MVar is written as $x\ \mathbf{m}\ -$. $\mathcal{I}$ is an IO-stack.*

We only consider the evaluation of expressions of `IO`-type. For computing an expression $e :: \texttt{IO}\ \tau$ the machine $IOM1$ starts with state $(\emptyset, \emptyset, e, [], [])$. A state is a *final state* if both stacks are empty and the evaluated expression is of the form $(\texttt{return}\ x)$.

The transition relation $\xrightarrow{IOM1}$ of the machine $IOM1$ is defined in Figure 3(b). The first rule lifts all transitions of $M1$ to machine $IOM1$. The remaining rules have in common, that they require the (usual) stack $\mathcal{S}$ to be empty. That is how functional evaluation is separated from monadic computation: as long as the usual stack is filled, functional evaluation is performed and if the usual stack is empty, then monadic computations are performed. The rule (newMVar) creates a new MVar and returns its name. The rule (takeMVar) takes the content of a filled MVar. There is no rule for the case that the MVar is already empty. In this case the machine gets stuck. Performing the take-operation requires that the to-be-evaluated expression is already the name of the MVar. That is why first (pushTake) pushes the take-operation on the IO-stack and thus forces the argument to be evaluated first. The rules (putMVar) and (pushPut) are the corresponding rules for performing a `putMVar`-operation: First (pushPut) enforces the first argument to be evaluated (to get the name of the MVar),

then either (putMVar) is performed to fill the MVar (if it is empty) or the machine gets stuck, if the MVar is already filled. For implementing the monadic sequencing operator $\gg\!=$, the action on the left hand side is performed first. Hence the (pushBind)-operation stores the second argument on the IO-stack. When the execution of the first action ends successfully with $(\texttt{return}\ x)$, then rule (lunit) evaluates the $\gg\!=$-operator.

A single thread $y \Leftarrow \mathbb{M}[\mathbb{F}[e]]$ of *CHF* corresponds to a machine state of *IOM1* as follows: the IO-stack holds the corresponding $\mathbb{M}$-context of the expression and also the `takeMVar`- or `putMVar`-operation on the top-level of the $\mathbb{F}$-context. The call-by-name evaluation context $\mathbb{E}$ inside the $\mathbb{F}$-context is stored on the usual stack.

Since we only evaluate well-typed expressions, the following lemma holds:

Lemma 3.5. *For any machine state of IOM1 which is reachable from a start state for a well-typed expression $e :: \texttt{IO}\ \tau$, the IO-stack is of the following form: All entries are of the form $\#_{\gg}(x)$ except for the top-element which also may be $\#_{\mathsf{take}}$ or $\#_{\mathsf{put}}(x)$.*

Example 3.6. *We again consider the expression of Example 3.3 and its execution on machine IOM1, where we assume that the set $\mathcal{M}$ contains a filled MVar $w\,\mathbf{m}\,c$. The first eight transitions are as on machine M1, where the MVars and the IO-stack are irrelevant:*

$$(\emptyset, \{w\,\mathbf{m}\,c\}, \texttt{letrec}\ x_1 = (\lambda y.y)\ w,\ x_2 = \texttt{takeMVar}\ x_1, x_3 = x_2\ \texttt{in}\ (\lambda z.z)\ x_3, [], [])$$
$$\xrightarrow{\textit{IOM1},*} (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \texttt{takeMVar}\ x_1, [], [])$$

Now machine IOM1 proceeds as follows:

$$\xrightarrow{\textit{IOM1},\text{pushTake}} (\{x_1 \mapsto (\lambda y.y)\ w, x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, x_1, [], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{enter}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, (\lambda y.y)\ w, [\#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{pushApp}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, \lambda y.y, [\#_{\mathsf{app}}(w), \#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{takeApp}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1\}, \{w\,\mathbf{m}\,c\}, w, [\#_{\mathsf{heap}}(x_1)], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{update}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1, x_1 \mapsto w\}, \{w\,\mathbf{m}\,c\}, w, [], [\#_{\mathsf{take}}])$$
$$\xrightarrow{\textit{IOM1},\text{takeMVar}} (\{x_3 \mapsto x_2, x_2 \mapsto \texttt{takeMVar}\ x_1, x_1 \mapsto w\}, \{w\,\mathbf{m}\,-\}, \texttt{return}\ c, [], [])$$

Adding Concurrency Constructing the concurrent machine *CIOM1* from the sequential machine *IOM1* is easy, since most of the parts of the machine *IOM1* can be reused. Instead of evaluating a single expression, the machine *CIOM1* will evaluate several expressions in several threads. Any such thread consists of a to-be-evaluated expression, a stack, and an IO-stack. Moreover, since threads represent futures, every thread has a name (a variable). There is one unique distinguished thread, the *main thread*. If the main-thread is successfully evaluated, then the whole machine stops. Further components of the machine *CIOM1* are the heap $\mathcal{H}$ and the set of MVars $\mathcal{M}$ which are globally shared over all threads. For the transition relation of the machine *CIOM1* a single thread is non-deterministically selected and the (thread-local) transition is performed for the selected thread. For this thread-local transition we can reuse the transition relation of the machine *IOM1*. There are two exceptions: If the monadic operation `future` spawns a new thread, and if a thread finishes its evaluation such that its result can be shared in the heap.

Definition 3.7. *A* thread *(or* future*, alternatively) of the machine $CIOM1$ is a 4-tuple $(x, e, \mathcal{S}, \mathcal{I})$ where x is a variable, called the* name *of the future, e is a simplified expression which is evaluated by the thread, $\mathcal{S}$ is a stack, and $\mathcal{I}$ is an IO-stack. A future can be distinguished as a main-thread, which we sometimes write as $(x, e, \mathcal{S}, \mathcal{I})^{main}$.*

A state *of machine $CIOM1$ is a 3-tuple $(\mathcal{H}, \mathcal{M}, \mathcal{T})$ where $\mathcal{H}$ is a heap of shared bindings, $\mathcal{M}$ is a set of MVars, and $\mathcal{T}$ is a set of threads.*

Definition 3.8. *For a simplified expression $e :: \texttt{IO}\ \tau$ the* start state $\mathsf{Init}(e)$ *of machine $CIOM1$ is $(\emptyset, \emptyset, \mathcal{T})$ where $\mathcal{T} = \{(x, e, [], [])^{main}\}$ and x is a fresh variable ($x \notin FV(e)$).*

A state of the machine $CIOM1$ is a final state *if the main-thread is of the form $(y, \texttt{return}\ x, [], [])^{main}$ where y and x may be equal.*

Definition 3.9. *The transition relation $\xrightarrow{CIOM1}$ of machine $CIOM1$ is shown in Fig. 3(c). For one step a thread is selected which may proceed. This selection is performed nondeterministically over all threads. Note that threads which cannot proceed are not selected. Those threads are about to evaluate a variable which is not bound in the heap, or try to perform a* (takeMVar)- *or* (putMVar)*-transition on an empty or filled MVar. We also assume that transitions are not applicable to final states.*

When a thread successfully finishes its computation, the rule (unIO) removes the thread and stores the result in the heap by a new binding. Note that other threads which want to access the value of a future x will not be selected for transition until the result becomes available as a binding in the heap. The rule (fork) evaluates a `future`-operation and spawns a new thread. In all other cases the rule (IOM1) is used which lifts the transition relation $\xrightarrow{IOM1}$ of $IOM1$ to the concurrent machine $CIOM1$.

Note that for a real implementation one would require some kind of fairness and thus for instance organize the set of threads as a priority-queue of threads.

Definition 3.10. *A state S is* valid*, if there exists $e :: \texttt{IO}\ \tau$ such that $\mathsf{Init}(e) \xrightarrow{CIOM1,*} S$.*

We only consider valid states in the following. It is easy to verify that for any valid state of $CIOM1$ all introduced variables (names of MVars, left hand sides of heap bindings, and names of threads) are pairwise distinct, all $\#_{\mathsf{heap}}(x)$-entries in stacks are pairwise distinct, and all the variables x in such entries do not occur as a left hand side in the heap.

4 Correctness of the Abstract Machine

In this section we will show that the abstract machine $CIOM1$ is a correct evaluator for CHF, that is for all expressions $e :: \texttt{IO}\ \tau$ may- and should-convergence of CHF coincide with may- and should-convergence of the machine $CIOM1$ where e is simplified before the evaluation. Indeed we will not only consider expressions and will work with processes in most of our proofs. As a simplification we assume that in CHF for the evaluation of a process all ν-binders are dropped and that reduction does not introduce ν-binders. Instead corresponding α-renamings are performed implicitly to represent the according scopes.

We first show that it is correct to take into account simplified expressions and processes, only. The first translation shares all necessary parts to derive simplified processes, i.e. general processes can be transformed into simplified processes by creating new bindings.

Definition 4.1. *The function $\sigma :: Proc \to Proc_S$ translates processes into simplified processes. It is defined to be homomorphic over the term structure (e.g. $\sigma(P_1 \mid P_2) := \sigma(P_1) \mid \sigma(P_2)$, etc.) except for the following cases:*

$$
\begin{array}{lcl}
\sigma(e_1\ e_2) & := & \texttt{letrec}\ x = \sigma(e_2)\ \texttt{in}\ (\sigma(e_1)\ x) \\
\sigma(c\ e_1\ \ldots\ e_n) & := & \texttt{letrec}\ x_1 = \sigma(e_1), \ldots, x_n = \sigma(e_n)\ \texttt{in}\ c\ x_1\ \ldots\ x_n \\
 & & \textit{if } c \textit{ is a constructor, or a monadic operator} \\
\sigma(\texttt{seq}\ e_1\ e_2) & := & \texttt{letrec}\ x = \sigma(e_2)\ \texttt{in}\ \texttt{seq}\ \sigma(e_1)\ x \\
\sigma(x\ \mathbf{m}\ e) & := & x\ \mathbf{m}\ y \mid y = \sigma(e)
\end{array}
$$

The results in [SSS11a] imply that the translation σ preserves contextual equivalence:

Theorem 4.2. *For all processes $P \in Proc$: $P \sim_{CHF} \sigma(P)$.*

We define may- and should-convergence based on the machine transition of $CIOM1$:

Definition 4.3. *A valid state S* may-converges *($S{\downarrow_{CIOM1}}$) iff there exists a final state S' such that $S \xrightarrow{CIOM1,*} S'$; and S* should-converges *($S{\Downarrow_{CIOM1}}$) iff $\forall S' : S \xrightarrow{CIOM1,*} S' \implies S'{\downarrow_{CIOM1}}$. An expression $e :: \texttt{IO}\ \tau$* may-converges on *CIOM1 ($e{\downarrow_{CIOM1}}$) iff $\mathsf{Init}(\sigma(e)){\downarrow_{CIOM1}}$, and e* should-converges *on CIOM1 ($e{\Downarrow_{CIOM1}}$) iff $\mathsf{Init}(\sigma(e)){\Downarrow_{CIOM1}}$. We write $e{\Uparrow_{CIOM1}}$ iff $\neg(e{\downarrow_{CIOM1}})$ and $e{\uparrow_{CIOM1}}$ iff $\neg(e{\Downarrow_{CIOM1}})$.*

Note that if we would restrict evaluation to *fair* evaluations only, i.e. forbidding (infinite) reductions sequences where an executable thread is ignored infinitely long, then the induced predicates of may- and should-convergence are unchanged (see also e.g. [Sab08, SSS11a]). Thus for reasoning it is not necessary to explicitly treat fairness.

We will now define the translation ρ which translates valid machine states of $CIOM1$ into processes. Note that the resulting process is not necessarily simplified. In abuse of notation we allow also non-simplified expressions inside the machine state during the translation.

Definition 4.4. *Let $(\mathcal{H}, \mathcal{M}, \mathcal{T}) = (\bigcup_{i=1}^{n}\{x_i \mapsto e_1\}, \{m_1, \ldots, m_{n'}\}, \{T_1, \ldots, T_{n''}\})$ be a valid machine state of CIOM1 where m_i are MVars and T_i are threads.*
Then $\rho(\mathcal{H}, \mathcal{M}, \mathcal{T}) := x_1 = e_1 \mid \ldots \mid x_n = e_n \mid m_1 \mid \ldots \mid m_{n'} \mid \rho(T_1) \mid \ldots \mid \rho(T_{n''})$ where a single thread T_i is translated as follows:

$$
\begin{array}{ll}
\rho(y, e, \#_{\mathsf{app}}(x) : \mathcal{S}, \mathcal{I}) & := \rho(y, e\ x, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, \#_{\mathsf{seq}}(x) : \mathcal{S}, \mathcal{I}) & := \rho(y, \texttt{seq}\ e\ x, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, \#_{\mathsf{heap}}(x) : \mathcal{S}, \mathcal{I}) & := x = e \mid \rho(y, x, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, \#_{\mathsf{case}}(alts) : \mathcal{S}, \mathcal{I}) & := \rho(y, \texttt{case}\ e\ \texttt{of}\ alts, \mathcal{S}, \mathcal{I}) \\
\rho(y, e, [], \#_{\gg=}(x) : \mathcal{I}) & := \rho(y, e \gg= x, [], \mathcal{I}) \\
\rho(y, e, [], \#_{\mathsf{take}} : \mathcal{I}) & := \rho(y, \texttt{takeMVar}\ e, [], \mathcal{I}) \\
\rho(y, e, [], \#_{\mathsf{put}}(x) : \mathcal{I}) & := \rho(y, \texttt{putMVar}\ e\ x, [], \mathcal{I}) \\
\rho(y, e, [], []) & := y \overset{\mathsf{main}}{\Longleftarrow} e, \textit{ if } y \textit{ is a main-thread, and } y \Leftarrow e, \textit{ otherwise}
\end{array}
$$

Lemma 4.5. *Let S be a valid machine state with $S \xrightarrow{CIOM1} S'$. Then either $\rho(S) = \rho(S')$ or $\rho(S) \xrightarrow{CHF} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$.*

Proof. This follows by inspecting all cases (see [Sab12]). The (cpx) and (gc) transformations are necessary to remove variable-to-variable bindings which are introduced in CHF by (lbeta), (case), and (cpcx) but not by the corresponding transitions (takeApp), (branch), and (update). □

Proposition 4.6. *For every valid state S of CIOM1: $S{\downarrow_{CIOM1}} \implies \rho(S){\downarrow}$.*

Proof. Let $S_n{\downarrow_{CIOM1}}$, i.e. $S_n \xrightarrow{CIOM1} \ldots \xrightarrow{CIOM1} S_0$ where S_0 is a final state. We use induction on n: If $n = 0$, then S_n is a final state and $\rho(S_n)$ is successful. For the induction step assume that $\rho(S_{n-1}){\downarrow}$. The analysis in Lemma 4.5 shows that either $\rho(S_n) = \rho(S_{n-1})$, $\rho(S_n) \xrightarrow{CHF} \rho(S_{n-1})$, or $\rho(S_n) \xrightarrow{CHF} P \sim_{CHF} \rho(S_{n-1})$ (since (cpx) and (gc) are correct program transformations, see Theorem 2.5). For the first two cases obviously $\rho(S_n){\downarrow}$, for the third case $S_{n-1}{\downarrow}$ and contextual equivalence imply that $P{\downarrow}$ and thus also $\rho(S_n){\downarrow}$. □

Given a state S and a reduction of the corresponding process, say $\rho(S) \xrightarrow{CHF} P$, we now try to find a sequence of corresponding machine transitions for S.

Lemma 4.7. *Let S be a valid machine state, and let $\rho(S) \xrightarrow{CHF} P$. Then there exists a valid state S' with $S \xrightarrow{CIOM1,*} S'$ such that one of the following properties holds: (1) $\rho(S') = P$; or (2) $P \xrightarrow{CHF,cp} \rho(S')$; or (3) in case of a $(CHF, cpcx)$-reduction $P \xrightarrow{cpcxxL} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$; or (4) $P \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$.*

Proof. We give a brief description, details are in [Sab12]. Several transitions are necessary to find the corresponding redex using the transitions (pushBind), (pushApp), (pushSeq), (pushAlts), and (enter). For the first case a machine transition corresponds to standard reduction in CHF. The second and third case may occur if a (cp) or (cpcx) reduction is performed: then perhaps the corresponding heap binding in the machine is under evaluation of the wrong thread and the machine must perform two (update) transitions, where one corresponds to the (cp) (or (cpcx)) reduction, and the other one is also a (cp) standard reduction or a (cpcxxL)-transformation. If a constructor was shared by a $(CHF, cpcx)$-reduction, then the generated variable-to-variable bindings must be inlined and removed by performing a sequence of (cpx) and (gc) transformations. Case (4) describes a necessary removal of variable-to-variable bindings which are introduced by a (lbeta)- or (case)-reduction. □

Proposition 4.8. *For every valid machine state S of CIOM1: $\rho(S){\downarrow} \implies S{\downarrow_{CIOM1}}$.*

Proof. Let $P_n{\downarrow^{[m,n]}}P_0$, i.e. $P_n \xrightarrow{CHF} P_{n-1} \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_0$ where P_0 is successful, and m is the number of all reductions except of (cp)-reductions that copy a variable. We use induction on the pair (m, n), ordered lexicographically. For $n = 0$ the claim holds, since only final machine states are translated into successful processes. For the induction step assume that the claim holds for all $(m', n') < (m, n)$. We apply Lemma 4.7 to the reduction $\rho(S_n) = P_n \xrightarrow{CHF} P_{n-1}$ where $P_{n-1}{\downarrow^{[m',n-1]}}$ such that either $m' = m$ (if the

reduction is also a (cpx)-transformation), or $m' = m - 1$ (in all other cases). This shows $S_n \xrightarrow{CIOM1,*} S'$ by the following cases: (i) $\rho(S') = P_{n-1}$: Then $S_n{\downarrow_{CIOM1}}$ by the induction hypothesis. (ii) $P_{n-1} \xrightarrow{CHF,cp} \rho(S')$, or $P_{n-1} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$. Then Proposition 2.7 shows that $\rho(S'){\downarrow^{[m'',n'']}}$ where $(m'', n'') < (m, n)$. Applying the induction hypothesis to $\rho(S')$ yields $S'{\downarrow_{CIOM1}}$ and thus also $S_n{\downarrow_{CIOM1}}$. (iii) $P_{n-1} \xrightarrow{cpcxxL} \xrightarrow{cpx,*} \xrightarrow{gc,*} \rho(S')$. Then the equation $m' = m - 1$ must hold, since the standard reduction is $(CHF, cpcx)$. Proposition 2.7 shows that $\rho(S'){\downarrow^{[m'',n'']}}$ where $(m'', n'') < (m, n)$ and thus we can apply the induction hypothesis to $\rho(S')$ and have $S'{\downarrow_{CIOM1}}$ and thus also $S_n{\downarrow_{CIOM1}}$. □

Since $\neg{\downarrow} = {\Uparrow}$ and $\neg{\downarrow_{CIOM1}} = {\Uparrow_{CIOM1}}$, Propositions 4.6 and 4.8 also imply:

Lemma 4.9. *For every valid machine state S of CIOM1: $\rho(S){\Uparrow} \iff S{\Uparrow_{CIOM1}}$.*

Proposition 4.10. *For every valid machine state S of CIOM1: $\rho(S){\Downarrow} \iff S{\Downarrow_{CIOM1}}$.*

Proof. The claim is equivalent to $\rho(S){\uparrow} \iff S{\uparrow_{CIOM1}}$. Both directions can be proved by induction analogously to the proofs for may-convergence in Propositions 4.6 and 4.8 except for the base cases of the inductions which are covered by Lemma 4.9. □

Theorem 4.11. *For every expression $e :: \texttt{IO}\ \tau$ the equivalences $e{\downarrow} \iff e{\downarrow_{CIOM1}}$ and $e{\Downarrow} \iff e{\Downarrow_{CIOM1}}$ hold.*

Proof. This follows from Propositions 4.6,4.8, and 4.10 and since for any well-typed expression $e :: \texttt{IO}\ \tau$ we have $\rho(\mathsf{Init}(\sigma(e))) = x \overset{\mathsf{main}}{\Longleftarrow} \sigma(e) \sim_{CHF} x \overset{\mathsf{main}}{\Longleftarrow} e$ where the last equivalence holds by Theorem 4.2. □

5 Conclusion

We introduced the concurrent abstract machine *CIOM1* for evaluation of *CHF*-programs and showed that the machine is a correct evaluator w.r.t. the semantics of the process calculus *CHF*. Further work is to optimize the machine, e.g. by following the modifications presented in [Ses97] (avoiding substitutions by using closures, using a nameless representation by de Bruijn-indices, etc.) and showing correctness of them. Another direction is to analyze how to map the threads of *CIOM1* to a multicore architecture.

Acknowledgments I thank Manfred Schmidt-Schauß for reading this paper and for discussions on this paper. I also thank the anonymous reviewers for their valuable comments.

References

[AHH+05] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *J. Symb. Comput.*, 40:795–829, 2005.

[BFKT00] C. A. Baker-Finch, D. J. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *5th ICFP*, pp. 162–173. ACM, 2000.

[CHS05] A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the Pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.

[DF07] R. Douence and P. Fradet. The next 700 Krivine machines. *Higher Order Symbol. Comput.*, 20:237–255, 2007.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *20th POPL*, pp. 144–154. ACM, 1993.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus*. CUP, 1999.

[Mor98] A. Moran. *Call-by-name, call-by-need, and McCarthy's Amb*. PhD thesis, Dept. of Comp. Science, Chalmers university, Sweden, 1998.

[MSC99] A. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, *LNCS* 1594, pp. 85–102. 1999.

[NSSSS07] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.

[Pey01] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pp. 47–96. IOS-Press, 2001.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23th POPL*, pp. 295–308. ACM, 1996.

[PS09] S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *6th AFP*, pp. 267–305. Springer, 2009.

[RV07] A. Rensink and W. Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.

[Sab08] D. Sabel. *Semantics of a call-by-need lambda calculus with McCarthy's amb for program equivalence*. Dissertation, Goethe-Universität Frankfurt Germany, 2008.

[Sab12] D. Sabel. An abstract machine for Concurrent Haskell with futures. Frank report 48, Institut für Informatik, Goethe-Universität Frankfurt am Main, 2012. http://www.ki.informatik.uni-frankfurt.de/papers/frank/.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Progr.*, 7(3):231–264, 1997.

[SSS08] D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

[SSS10] M. Schmidt-Schauß and D. Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Inform. Process. Lett.*, 110(6):232 – 235, 2010.

[SSS11a] D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *13th PPDP*, pp. 101–112, ACM, 2011.

[SSS11b] D. Sabel and M. Schmidt-Schauß. On conservativity of Concurrent Haskell. Frank report 47, Institut für Informatik, Goethe-Universität Frankfurt am Main, 2011. http://www.ki.informatik.uni-frankfurt.de/papers/frank/.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes*. CUP, 2001.

MOSTflexiPL
Modular, Statically Typed, Flexibly Extensible Programming Language

Christian Heinlein

Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@htw-aalen.de

Abstract. MOSTflexiPL (oder kurz flexiPL) ist eine momentan in Entwicklung befindliche Programmiersprache, die vom Anwender nahezu beliebig syntaktisch erweitert und angepasst werden kann. Trotz dieser enormen Flexibilität besitzt die Sprache ein statisches Typsystem mit Ähnlichkeiten zu „dependent types“. Die Semantik neu definierter Sprachkonstrukte wird durch eine Abbildung auf bereits vorhandene Konstrukte in der Sprache selbst festgelegt, d. h. es sind weder Eingriffe in Compiler oder Laufzeitsystem noch ein „prozedurales“ Makrosystem wie in Lisp erforderlich. Der Sprachkern, d. h. die Menge der Grundkonstrukte, die sich nicht (sinnvoll) auf andere Konstrukte zurückführen lassen, folgt keinem bestimmten Programmierparadigma. Die meisten Grundkonstrukte sind funktionaler Natur, durch die Bereitstellung von Variablen (d. h. änderbarer Speicherzellen) wird aber auch imperatives Programmieren (im weitesten Sinne) unterstützt. Neben diesen Grundkonstrukten, gibt es eine Sammlung vordefinierter Standardkonstrukte zur Unterstützung unterschiedlicher Programmierstile, die bereits in der Sprache selbst geschrieben sind. MOSTflexiPL-Programme werden durch einen Compiler in assemblerartigen C++-Code übersetzt, der von jedem standardkonformen C++-Compiler in ausführbaren Code übersetzt werden kann. Wenn ein Programm aus mehreren Modulen besteht, können diese unabhängig voneinander übersetzt werden.

1 Einleitung und Motivation

Programmiersprachen werden in der Regel langsam, aber kontinuierlich weiterentwickelt. Beispielsweise wurde die ursprüngliche Sprache C von Kernighan und Ritchie aus den 1970er Jahren zu ANSI-C (1989) weiterentwickelt, aus dem wiederum in mehreren Schritten C99 (1999) hervorging. Ebenso wurde die ursprüngliche Sprache Java von 1995, nach zahlreichen kleineren Änderungen, im Jahr 2004 um wesentliche Sprachmittel erweitert, z. B. um „Generics“, Annotationen und Aufzählungstypen. Auch eine spezielle `for`-Schleife zur bequemeren Iteration über Arrays und Container wurde hinzugefügt. Auf manche dieser Verbesserungen haben Benutzer der Sprache jahrelang gewartet, weil sie die Programmierung z. T. erheblich erleichtern.

Aber selbst wenn man jeweils die aktuellste Version einer Sprache verwendet, kommt man als Programmierer immer wieder in Situationen, in denen ein maßgeschneidertes Sprachkonstrukt die tägliche Arbeit erleichtern und die Lesbarkeit des Codes verbessern könnte. Beispielsweise enthalten Deklarationen von Variablen mit sofortiger Initialisierung häufig einen hohen Grad von Redundanz (der sich durch Ver-

wendung von `import`-Anweisungen ein wenig reduzieren ließe), z. B.:

```
java.util.HashMap<String, java.io.InputStream> m =
  new java.util.HashMap<String, java.io.InputStream>();
```

Durch eine neue Deklarationsform (wie sie seit längerem für C++ geplant ist) könnte dies wie folgt verkürzt werden, indem eine Variable automatisch den Typ ihres Initialisierungsausdrucks erhält:

```
auto m =
  new java.util.HashMap<String, java.io.InputStream>();
```

Aber selbst wenn dieses Sprachkonstrukt eines Tages in die jeweiligen Sprachen integriert werden sollte, wird man als Programmierer früher oder später wieder in Situationen geraten, in denen man sich irgendein zusätzliches Sprachkonstrukt wünscht. Keine noch so gut entworfene Sprache wird jemals alle Anwender rundherum zufriedenstellen können.

Da man als normaler Programmierer mit gängigen Programmiersprachen nicht in der Lage ist, neue Sprachkonstrukte selbst zu definieren, muss man sich entweder mit dem Status quo arrangieren bzw. auf die nächste Sprachversion warten oder aber mit Hilfe eines Präcompilers seine eigene erweiterte Version der Sprache implementieren (wie z. B. in [He07]), was je nach Komplexität der Sprache und der gewünschten Erweiterungen mit erheblichem Aufwand verbunden sein kann.

Ein möglicher Ausweg aus diesem Dilemma ist die Verwendung einer Programmiersprache wie MOSTflexiPL[1], die von ihren Anwendern beliebig syntaktisch erweitert und angepasst werden kann. Sie bietet als Sprachkern eine möglichst kleine Menge von *Grundkonstrukten* (wie z. B. arithmetische und logische Operatoren sowie elementare Kontrollstrukturen), mit deren Hilfe sich beliebige weitere Sprachkonstrukte definieren lassen. Zusätzlich werden häufig benötigte *Standardkonstrukte* angeboten (z. B. weitere Kontrollstrukturen), die aber bereits als Erweiterungen des Sprachkerns in der Sprache selbst geschrieben sind.

Unter Verwendung dieser Grund- und Standardkonstrukte kann man in MOSTflexiPL bereits wie in jeder anderen Sprache programmieren, hat aber jederzeit die Möglichkeit, zusätzliche Sprachkonstrukte zu definieren, die z. B. die Programmierung einer bestimmten Anwendung erleichtern. Neue Sprachkonstrukte, die über eine bestimmte Anwendung hinaus nützlich erscheinen (wie z. B. die oben erwähnte `auto`-Deklaration), kann man – analog zu Funktions- oder Klassenbibliotheken in anderen Sprachen – zu einer Bibliothek zusammenfassen, um sie in anderen Anwendungen wiederverwenden oder anderen Anwendern zur Verfügung stellen zu können.

Ein weiteres Anwendungsfeld ist die Definition problemorientierter Sprachen (domain-specific languages, DSLs) als Erweiterungen bzw. Modifikationen von MOSTflexiPL, d. h. die Verwendung von MOSTflexiPL als Wirtssprache für andere Sprachen. Da die syntaktischen Gestaltungsmöglichkeiten nahezu unbegrenzt sind, entstehen bei dieser Einbettung einer DSL in MOSTflexiPL kaum Nachteile gegenüber der

[1] Oder kurz „flexiPL“, das wie das englische „flexible“ mit hartem p statt weichem b ausgesprochen wird. Der ursprüngliche Name lautete STEEL – Statically Typed Extensible Expression Language, was ebenfalls wichtige Eigenschaften der Sprache zum Ausdruck bringt. Der neue Name sowie das im Titel verwendete Logo betonen aber noch mehr die flexible Erweiterbarkeit und Anpassbarkeit der Sprache.
Siehe auch `http://most.flexipl.info` bzw. `http://flexipl.info`.

wesentlich aufwendigeren Implementierung von Grund auf durch maßgeschneiderte Compiler, Interpreter o. ä. Prinzipiell sollte es möglich sein – obwohl das nicht das Ziel der Entwicklung ist –, nahezu beliebige andere Computersprachen in MOSTflexiPL einzubetten.

Schließlich kann MOSTflexiPL als Experimentierfeld für neue Sprachkonstrukte (z. B. im Bereich parallele oder natürlichsprachliche Programmierung) dienen. Da die Implementierung neuer Sprachkonstrukte vergleichsweise einfach ist, kann man sich als Entwickler vor allem auf ihren Entwurf konzentrieren. Außerdem lassen sich Spracherweiterungen leicht und schnell in Form von Bibliotheken an interessierte Nutzer bzw. „Tester“ weitergeben, deren Erfahrungen und Rückmeldungen anschließend zur Weiterentwicklung und Verbesserung verwendet werden können.

Neben diesen Vorteilen einer syntaktisch erweiterbaren Sprache, besteht natürlich auch die Gefahr des „Missbrauchs“, indem jeder Programmierer gedankenlos unsinnige Erweiterungen definiert, die dazu führen, dass sein Quellcode für andere Personen kaum noch verständlich ist. Diesem „Wildwuchs“ kann jedoch durch Richtlinien oder Absprachen begegnet werden, die ähnlich wie heutige Kodierrichtlinien für ein Projekt oder eine Firma vereinbart werden können.

Der vorliegende Artikel ist wie folgt gegliedert. Abschnitt 2 erläutert anhand einfacher Beispiele in aller Kürze die wichtigsten vordefinierten Sprachkonstrukte von MOSTflexiPL. Abschnitt 3 zeigt dann exemplarisch einige Beispiele nützlicher Spracherweiterungen. In Abschnitt 4 werden die wesentlichen Ideen zur Implementierung von MOSTflexiPL skizziert. Verwandte Arbeiten werden in Abschnitt 5 diskutiert. Abschnitt 6 gibt schließlich eine Zusammenfassung sowie einen Ausblick auf zukünftige Arbeiten. Um dem Leser einen Gesamteindruck von MOSTflexiPL vermitteln zu können, musste an vielen Stellen aus Platzgründen auf detailliertere Ausführungen verzichtet werden.

2 Vordefinierte Sprachkonstrukte

2.1 Operatoren, Ausdrücke und Programme

Ein *Operator* besteht aus einem oder mehreren *Namen* (Operatorsymbolen) und kann auf eine bestimmte Anzahl von *Operanden* angewandt werden. Abhängig von der Anzahl der Operanden unterscheidet man null-, ein-, zweistellige Operatoren etc., wie zum Beispiel (die Positionen, an denen die Operanden einzusetzen sind, sind durch Unterstriche gekennzeichnet):

- Der Subtraktionsoperator `_ - _` ist zweistellig und besitzt einen Namen.
- Der Vorzeichenoperator `- _` ist einstellig und besitzt ebenfalls einen Namen.
- Der Operator `_ ? _ : _` in C, C++ und Java ist dreistellig und besitzt zwei Namen.
- Der Klammerungsoperator `( _ )` ist einstellig und besitzt zwei Namen.
- Der Schleifenoperator `while _ do _ end` ist zweistellig und besitzt drei Namen.
- Der anonyme Operator `_ _`, mit dem zwei Operanden ohne Operatorsymbol dazwischen aneinandergereiht werden können, ist zweistellig und besitzt keinen Namen.

- Die Bezeichner `start`, `end`, `p-1` und `f'` sind alle nullstellig und besitzen jeweils einen Namen.
- Der mehrteilige Bezeichner `2nd last char` ist ebenfalls nullstellig und besitzt drei Namen.
- Literale wie z. B. `123`, `3.14` oder `'x'` sind formal ebenfalls nullstellige Operatoren.

Wie die Beispiele zeigen, können Operatornamen prinzipiell beliebige Zeichenfolgen sein (z. B. `p-1` und `f'`), und unterschiedliche Operatoren können durchaus gemeinsame Namen besitzen (z. B. `end` als Teil von `while _ do _end` und gleichzeitig als eigener Operator). Damit ist es z. B. auch möglich, den gleichen Operatornamen präfix, infix und postfix mit unterschiedlichen Bedeutungen zu verwenden. Allerdings können ungeschickt gewählte Namen zu Mehrdeutigkeiten bei der Verwendung der Operatoren und damit zu Übersetzungsfehlern führen (z. B. wenn es neben `p-1` einen weiteren nullstelligen Operator `p` gibt, weil dann `p-1` auch die Differenz von `p` und `1` bezeichnen kann, zumindest wenn `p` einen numerischen Typ besitzt).

Ein *Ausdruck* ist die Anwendung eines *n*-stelligen Operators auf *n* Operanden, bei denen es sich wiederum um Ausdrücke handelt. Als Grenzfall ergibt sich für $n = 0$ die Anwendung eines nullstelligen Operators auf null Operanden, d. h. einfach die Verwendung eines nullstelligen Operators, die auch als *atomarer Ausdruck* bezeichnet wird. Für $n > 0$ ergibt sich ein *zusammengesetzter Ausdruck*.

Ein MOSTflexiPL-*Programm* ist nichts anderes als ein Ausdruck.

2.2 Deklarationen

Eine *Deklaration* ist die Anwendung eines vordefinierten *Deklarationsoperators* auf entsprechende Operanden, zum Beispiel:

- Der Deklarationsoperator `_ : _` dient zur Deklaration von Konstanten, die automatisch einen eindeutigen Wert erhalten, wie z. B. `"Person" : type` (`Person` wird als Konstante des vordefinierten Typs `type` deklariert und stellt damit einen neuen Typ dar) oder `"p-1" : Person` (`p-1` wird als Konstante des soeben definierten Typs `Person` deklariert und stellt damit quasi ein Objekt dieses Typs dar).
  Formal sind Konstanten nichts anderes als nullstellige Operatoren, die bei jeder Anwendung den ihnen zugeordneten Wert als Ergebnis liefern.
  Anmerkung: Bei der Deklaration eines Operators muss sein Name (bzw. allgemein seine Signatur, siehe unten) in Anführungszeichen stehen (z. B. `"Person"` in der ersten Deklaration oben), bei seiner Verwendung jedoch nicht (z. B. `Person` in der zweiten Deklaration oben).
- Der Deklarationsoperator `_ : = _` dient zur Deklaration von Konstanten, die mit einem vorgegebenen Wert initialisiert werden, wie z. B. `"N" : = 10` (`N` wird als Konstante mit Wert `10` deklariert und besitzt damit automatisch den vordefinierten Typ `int`), `"M" : = 2 * N` oder `"p-2" : = p-1`.
- Der Deklarationsoperator `[ _ ] _ : _ { _ }` dient zur Deklaration von Operatoren, die eine *Parameterliste* (innerhalb der eckigen Klammern), eine *Signatur* (vor dem Doppelpunkt), einen *Resultattyp* (nach dem Doppelpunkt) und eine *Implementierung* (innerhalb der geschweiften Klammern) besitzen. Solche Operatoren entspre-

chen Prozeduren oder Funktionen in anderen Programmiersprachen, z. B.:

```
["x" : int; "y" : int]                  Parameterliste
"max x y" : int                         Signatur und Resultattyp
{ if x > y then x else y end }          Implementierung
```

Die Parameterliste besteht selbst wieder aus einer Folge von Deklarationen, die durch den zweistelligen sequentiellen Kompositionsoperator `_ ; _` getrennt sind.
Die Signatur (die, wie bereits erwähnt, in Anführungszeichen stehen muss) besteht aus beliebig vielen Wörtern, bei denen es sich entweder um Namen von Parametern (im Beispiel `x` und `y`) oder um Namen des zu deklarierenden Operators handelt (im Beispiel `max`, weil dies kein Name eines Parameters ist). Dadurch wird indirekt die *Syntax* des Operators festgelegt, d. h. wie er anzuwenden ist: Im Beispiel müssen dem Operatornamen `max` zwei Operanden (d. h. Ausdrücke) mit Typ `int` folgen, die den Parametern `x` und `y` entsprechen, z. B. `max 1 2` oder `max 1+2 max 3 4`. Hätte man als Signatur z. B. `"x max y"` angegeben, so müsste der Name `max` zwischen den beiden Operanden stehen.
Die Implementierung ist ein beliebiger Ausdruck, dessen Typ mit dem Resultattyp des Operators (im Beispiel `int`) übereinstimmen muss. Bei einer Anwendung des Operators wird dieser Ausdruck ausgewertet und sein Wert als Ergebnis der Operatoranwendung geliefert, nachdem zuvor die Operanden ausgewertet und die Parameter des Operators mit ihren Werten initialisiert wurden.
Die in der Implementierung verwendeten Operatoren `if _ then _ else _ end` und `_ > _` sind vordefinierte Standardoperatoren.

Da Deklarationen spezielle Ausdrücke sind und Ausdrücke beliebig verschachtelt werden können, können auch Deklarationen beliebig verschachtelt werden, zum Beispiel:

```
["n" : int]
"even n" : bool                                 globaler Operator
{
  ["n" : int]                                   lokaler Operator,
  "odd n" : bool                                der den globalen aufruft
  { if n == 0 then false else even n-1 end };

  if n == 0 then true else odd n-1 end          Aufruf des lokalen
}                                  Operators in der Implementierung des globalen
```

Das Beispiel zeigt nebenbei, dass Operatorimplementierungen beliebig (auch wechselseitig) rekursiv sein können. Außerdem sind Operatoren „Bürger erster Klasse“, d. h. sie können zum Beispiel als Parameter- und Resultatwerte anderer Operatoren auftreten, in Variablen gespeichert werden u. ä. Wenn ein lokaler Operator dadurch die Ausführung eines oder mehrerer umschließender Operatoren „überlebt“, werden deren Ausführungskontexte (die z. B. ihre Parameter und lokalen Variablen enthalten) so lange aufbewahrt, bis sie garantiert für keine lokale Operatorausführung mehr benötigt werden, d. h. lokale Operatoren stellen vollwertige Funktionsabschlüsse (closures) dar. Da jede Operatordeklaration letztlich die Syntax der Sprache erweitert, stellen lokale Deklarationen lokal begrenzte Syntaxerweiterungen dar (vgl. auch §2.5).

2.3 Explizite, implizite und deduzierte Parameter

Die Parameter `x` und `y` des oben definierten Operators `max _ _` heißen *explizite Parameter*, weil die zugehörigen Operanden bei einer Anwendung des Operators explizit angegeben werden.

Um einen *generischen* Maximumoperator definieren zu können, der auf zwei Operanden eines beliebigen Typs `T` angewandt werden kann, braucht man zusätzlich einen *deduzierten* Parameter:

```
["T" : type; "x" : T; "y" : T]
"max x y" : T
{ if x > y then x else y end }
```

Der Parameter `T` heißt deduziert, weil sein Wert bei einer Anwendung des Operators nicht explizit angegeben, sondern aus dem Typ anderer Operanden abgeleitet (deduziert) wird. Dementsprechend erkennt man einen deduzierten Parameter daran, dass er im Typ anderer Parameter (im Beispiel `x` und `y`) auftritt. Außerdem tritt sein Name typischerweise *nicht* in der Signatur des Operators auf. (Wenn er dort auftreten würde, wäre der Parameter sowohl explizit als auch deduziert, was ein relativ seltener Sonderfall ist.)

Typische Anwendungen des soeben definierten Maximumoperators sind `max 1 2` und `max 1.5 3.7`. Im ersten Fall besitzen die Operanden `1` und `2`, die den expliziten Parametern `x` und `y` entsprechen, beide den Typ `int`, sodass `T` gleich `int` deduziert wird. Im zweiten Fall ergibt sich entsprechend `T` gleich `float`, da die Operanden `1.5` und `3.7` beide diesen Typ besitzen. Ein Ausdruck wie z. B. `max 1 3.7` wird vom Compiler als fehlerhaft zurückgewiesen, weil es keine eindeutige Belegung für den deduzierten Parameter `T` gibt: Aus dem Operanden `1` mit Typ `int` ergäbe sich `T` gleich `int`, während sich aus dem Operanden `3.7` mit Typ `float` die Belegung `T` gleich `float` ergäbe. (In einer zukünftigen Version der Sprache soll `int` implizit in `float` umwandelbar sein. Dann wäre `T` gleich `float` in diesem Beispiel eine korrekte Belegung.)

Tatsächlich ist die obige Definition des Maximumoperators auch noch fehlerhaft: Da der Typ `T` der Parameter `x` und `y` beliebig sein kann, findet der Compiler für den Teilausdruck `x > y` keinen passenden Operator `_ > _` und weist daher die gesamte Operatordeklaration als fehlerhaft zurück. Er kennt nur die vordefinierten Vergleichsoperatoren für `int` und `float` sowie eventuelle benutzerdefinierte Operatoren für weitere Typen, aber keinen Größer-Operator für Werte eines beliebigen Typs.

Um dieses Problem zu beheben, könnte man den jeweiligen Vergleichsoperator explizit als zusätzlichen Operanden übergeben, was die Verwendung des Operators jedoch unnötig verkomplizieren würde. Alternativ kann man den Größer-Operator als *impliziten Parameter* übergeben:

```
[ "T" : type; "x" : T; "y" : T;
  ["u" : T; "v" : T] "u > v" : bool]
"max x y" : T
{ if x > y then x else y end }
```

Neben den bereits bekannten Parametern `T`, `x` und `y`, besitzt der jetzt definierte Maximumoperator einen weiteren Parameter `_ > _`, der selbst wiederum explizite Parame-

ter `u` und `v` mit Typ `T` besitzt. Demnach kann dieser Parameter jetzt als Operator im Teilausdruck `x > y` verwendet werden. Dieser Parameter ist weder explizit, weil sein Name nicht in der Signatur des Operators auftritt, noch deduziert, weil er nicht im Typ anderer Parameter auftritt. Diese dritte Art von Parametern heißt *implizit*.

Analog zu einem deduzierten Parameter, wird auch für einen impliziten Parameter bei einer Operatoranwendung kein expliziter Operand übergeben. Stattdessen wird implizit ein an der Anwendungsstelle sichtbarer Operator übergeben, der die gleiche Syntax (d. h. die gleiche Folge von Operatornamen und Operandenpositionen) wie der implizite Parameter und den gleichen (oder einen kompatiblen) Operatortyp besitzt. Daraus folgt, dass die Namen impliziter Parameter – im Gegensatz zu den Namen expliziter und deduzierter Parameter – für die Anwendung eines Operators relevant sind.

Für den Beispielausdruck `max 1 2` bedeutet das konkret: Die expliziten Parameter `x` und `y` werden mit den Werten der Operanden `1` bzw. `2` initialisiert. Da diese beide den Typ `int` besitzen, wird `T` gleich `int` deduziert. Für den impliziten Parameter `_ > _` wird ein Operator mit der gleichen Syntax `_ > _` übergeben, der zwei Operanden des Typs `T` (also `int`) akzeptiert und Resultattyp `bool` besitzt. Hierfür kommt nur der vordefinierte Größer-Operator für `int`-Werte in Frage.

Lautet der Beispielausdruck `max 1.5 3.7`, so ergibt sich `T` gleich `float`, d. h. für den impliziten Parameter `_ > _` wird jetzt ein Operator `_ > _` übergeben, der zwei Operanden des Typs `float` akzeptiert und wiederum Resultattyp `bool` besitzt.

Ein Ausdruck wie z. B. `max p-1 p-2` (mit `p-1` und `p-2` vom Typ `Person`) wird vom Compiler als fehlerhaft zurückgewiesen, weil es (standardmäßig) keinen Operator `_ > _` gibt, der zwei Operanden des Typs `Person` akzeptiert, und der Compiler somit keine Belegung für den impliziten Parameter `_ > _` finden kann.

Das letzte Beispiel zeigt, dass implizite Parameter nicht nur der Bequemlichkeit dienen – weil man ihre Werte nicht explizit übergeben muss –, sondern auch dazu verwendet werden können, Einschränkungen oder Nebenbedingungen für deduzierte Parameter auszudrücken: Der implizite Parameter `_ > _` des Maximumoperators zeigt an, dass dieser Operator nur auf Operanden angewandt werden kann, für deren Typ `T` es einen entsprechenden Vergleichsoperator gibt. Das ist in etwa vergleichbar mit „bounded polymorphism“, wie man ihn z. B. in Java findet, oder mit den Typklassen von Haskell.

2.4 Typen

Ein *Typ* ist ebenfalls nichts anderes als ein Ausdruck (in aller Regel ein *statischer* Ausdruck, siehe unten). Elementare Typen wie z. B. `int` oder `float`, aber auch einfache benutzerdefinierte Typen wie z. B. `Person`, entsprechen atomaren Ausdrücken. Strukturierte Typen, wie z. B. `List Person` (Liste von Personen), `int [10]` (Array von 10 `int`-Werten) oder `int # bool` (Paar von `int` und `bool`), erhält man durch die Anwendung „typwertiger“ Operatoren (d. h. Operatoren mit Resultattyp `type`, anderweitig auch als Typkonstruktoren bezeichnet) auf andere Typen und/oder Werte, zum Beispiel:[2]

[2] Da Typen somit von Werten abhängen können, handelt es sich um eine Form von „dependent types“ [Wi12].

```
["T" : type] "List T" : type;
["T" : type; "N" : int] "T [ N ]" : type;
["X" : type; "Y" : type] "X # Y" : type
```

Die hier definierten Operatoren sind *statisch*, weil sie keine benutzerdefinierte Implementierung (geschweifte Klammern) besitzen, die ihr Verhalten zur Laufzeit bestimmt, sondern ein vordefiniertes Verhalten mit folgenden Eigenschaften:

- Die mehrmalige Anwendung eines statischen Operators auf die gleichen Operandenwerte liefert immer den gleichen Resultatwert. Daraus folgt z. B., dass jede Verwendung von `List Person` denselben Typ liefert.
- Die Anwendung eines statischen Operators auf unterschiedliche Operandenwerte liefert unterschiedliche Resultatwerte. Daraus folgt z. B., dass `List Person` und `List int` garantiert unterschiedliche Typen sind.
- Anwendungen unterschiedlicher statischer Operatoren liefern unterschiedliche Resultatwerte. Daraus folgt z. B., dass `List Person` und `int # bool` garantiert unterschiedliche Typen sind (da `List _` und `_ # _` unterschiedliche statische Operatoren sind), ebenso aber auch `List Person` und `Person` (da `List _` und `Person` unterschiedliche statische Operatoren sind; tatsächlich sind die in §2.2 eingeführten Konstanten nichts anderes als parameterlose statische Operatoren).

Aus diesen Eigenschaften folgt, dass zwei statische Ausdrücke (d. h. Ausdrücke, in denen nur statische Operatoren auftreten) genau dann den gleichen Wert besitzen, wenn sie *strukturgleich* sind (d. h. wenn es sich um Anwendungen desselben Operators auf paarweise strukturgleiche Operanden handelt).

Wenn Typen statische Ausdrücke sind, bedeutet das, dass der Compiler ihre Gleichheit einfach durch einen Vergleich ihrer Struktur überprüfen kann. Wenn ein Typausdruck nicht statisch ist, d. h. Operatoren mit prinzipiell nicht vorhersagbarem dynamischem Verhalten enthält, gilt diese Äquivalenz von Wert- und Strukturgleichheit nicht mehr. Da der Compiler aber nur die Strukturgleichheit von Typausdrücken zur Übersetzungszeit und nicht ihre eventuelle Wertgleichheit zur Laufzeit überprüfen kann, betrachtet er Typen trotzdem nur dann als gleich, wenn sie strukturgleich sind. Daraus folgt zum Beispiel, dass `int [1+2]`, `int [2+1]` und `int [3]` aus Sicht des Compilers drei unterschiedliche Typen sind, weil die Ausdrücke nicht strukturgleich sind, obwohl ihre (i. d. R. uninteressanten) Werte zur Laufzeit natürlich gleich sein werden.[3]

Neben elementaren Typen wie `int` und `float`, gibt es zwei vordefinierte Typoperatoren `_ *` und `_ ?` zur Definition von *Sequenz-* bzw. *Variablentypen*. Für einen beliebigen Typ `T` bezeichnet der Typ `T*` Sequenzen von `T`-Werten beliebiger Länge, ähnlich wie `vector<T>` o. ä. in anderen Sprachen, während `T?` Variablen von `T`-Werten bezeichnet. Für eine Sequenz `s` vom Typ `T*` liefert `#s` ihre Länge, d. h. die Anzahl ihrer Elemente, während `s[i]` für `i` von 1 bis `#s` (jeweils einschließlich) ihr `i`-tes Element

[3] In einer zukünftigen Version der Sprache lassen sich vielleicht Regeln formulieren (etwa das Kommutativgesetz der Addition), die dem Compiler erlauben, durch entsprechende Strukturtransformationen unterschiedliche Ausdrücke als „logisch gleich“ zu erkennen und zu akzeptieren.

(vom Typ `T`) liefert. Für eine Variable `v` vom Typ `T?` liefert `?v` ihren aktuellen Wert (vom Typ `T`)[4], während `v << x` oder `x >> v` den Wert von `x` als neuen Wert zuweist.

Statische Operatoren, deren Resultattyp ein Variablentyp ist, können z. B. wie folgt zur (inkrementellen) Definition von „Attributen“ (vgl. [He07]) verwendet werden:

```
["p" : Person] "p name" : string?
```

Für jede Person `p` liefert der Ausdruck `p name` eine eindeutige `string`-Variable, deren Wert mittels `? p name` abgefragt und mittels `p name << ...` verändert werden kann. In gleicher Weise könnte man z. B. auch Attribute `_ head` und `_ tail` für verkettete Listen definieren und damit dem bis jetzt noch „nackten“ Typ `List T` eine konkrete Bedeutung geben.

2.5 Import-, Export- und Ausschlussdeklarationen

Mit *Import-* und *Exportdeklarationen* lässt sich detailliert spezifizieren, welche Operatoren in welchen Teilausdrücken eines Programms sichtbar und damit verwendbar sein sollen. Beispielsweise gibt es für den vordefinierten sequentiellen Kompositionsoperator `_ ; _` eine Exportdeklaration, die besagt, dass die Deklarationen beider Operanden exportiert werden, sodass sie über die jeweilige Anwendung des Operators hinaus sichtbar sind. Außerdem gibt es eine Importdeklaration, die besagt, dass die Deklarationen des ersten Operanden im zweiten Operanden sichtbar sind. Für einen benutzerdefinierten Operator `let _ in _ end` könnte man ebenfalls vereinbaren, dass die Deklarationen des ersten Operanden im zweiten Operanden sichtbar sind, aber nur die Deklarationen des zweiten Operanden exportiert werden, sodass die Deklarationen des ersten Operanden „privat“ bleiben und die durch sie definierten Syntaxerweiterungen nur lokal gültig sind.

Mit *Ausschlussdeklarationen* können Vorrang und Assoziativität von Operatoren sehr flexibel geregelt werden (insbesondere wesentlich flexibler als mit einer einfachen Tabelle ganzzahliger Operatorprioritäten). Beispielsweise wird die bekannte Punkt-vor-Strich-Regel dadurch ausgedrückt, dass additive Operatoren (`_ + _` und `_ - _`) als Hauptoperator (d. h. als „oberster“ Operator im zugehörigen Operatorbaum) der Operanden eines multiplikativen Operators (`_ * _` und `_ / _`) verboten sind. Damit ist von den zwei prinzipiell denkbaren Zerlegungen einer Zeichenfolge wie `2 + 3 * 4` nur diejenige korrekt, bei der sich die Multiplikation „unterhalb“ der Addition befindet. Der explizit geklammerte Ausdruck `(2 + 3) * 4` ist nichtsdestotrotz korrekt, da der Klammeroperator `( _ )` in MOSTflexiPL ein selbständiger Operator ist und der Plusoperator damit nicht mehr der Hauptoperator des linken Operanden der Multiplikation ist.

Die Links- bzw. Rechtsassoziativität eines binären Operators (oder einer Gruppe solcher Operatoren) kann ebenfalls durch eine Ausschlussdeklaration spezifiziert werden, die den Operator selbst als Hauptoperator seines rechten bzw. linken Operanden verbietet.

[4] Mit Hilfe impliziter Typumwandlungen kann man in einer zukünftigen Version der Sprache vereinbaren, dass eine Variable mit Typ `T?` bei Bedarf automatisch durch eine Anwendung des Operators `? _` in ihren Wert des Typs `T` umgewandelt wird. Das entspricht dann der aus anderen Sprachen bekannten automatischen Umwandlung von Variablen (L-Werte in C/C++) in ihre Werte (R-Werte).

3 Beispiele für Spracherweiterungen

3.1 Iteration über Sequenzen

Mit den in §2.4 erwähnten Operatoren für Sequenzen sowie dem ebenfalls vordefinierten Wiederholungsoperator `while _ do _ end` kann man wie folgt über eine Sequenz `s` mit Typ `T*` iterieren, um beispielsweise ihre Elemente auszugeben:

```
"i" : int?;                    int-Variable i
i << 1;                        Zuweisung i gleich 1
while ?i <= #s do              Solange Wert von i ≤ Länge von s:
  ... s[?i] ...                  Verwendung des Elements s[?i]
  i << ?i + 1                    Zuweisung i gleich Wert von i plus 1
end
```

Kompakter und bequemer wäre jedoch folgende Formulierung:

```
"v" : T?;                      T-Variable v
for v in s do                  Für jedes Element v von s:
  ... ?v ...                     Verwendung von ?v
end
```

Hierfür wird ein neuer Operator `for _ in _ do _ end` benötigt, der folgende explizite Parameter besitzt:

- eine Variable `v` des Typs `T?`, wobei `T` ein beliebiger Typ sein kann;
- eine Sequenz `s` des Typs `T*`;
- einen Schleifenrumpf `body` mit einem beliebigen Typ `B`.

Folglich benötigt der Operator zusätzlich deduzierte Parameter `T` und `B`, jeweils mit Typ `type`. Seine Deklaration kann daher wie folgt lauten:

```
["T" : type; "B" : type; "v" : T?; "s" : T*; "body" : B {}]
"for v in s do body end" : int
{ "i" : int?; i << 1;
  while ?i <= #s do
    v << s[?i]; body; i << ?i + 1
  end
}
```

Der Operator besitzt Resultattyp `int`, weil jeder Operator einen Resultattyp besitzen muss und Schleifen per Konvention die Anzahl der ausgeführten Iterationen als Resultatwert liefern. Demnach kann der Resultatwert der `while`-Schleife direkt als Resultatwert des `for`-Operators verwendet werden.

Die leeren geschweiften Klammern am Ende der Deklaration des Parameters `body` zeigen an, dass der entsprechende Operand nicht, wie sonst üblich, als Wert („call by value“), sondern als *unausgewerteter Ausdruck* übergeben wird, ähnlich zu „lazy evaluation“ in funktionalen Sprachen und „call by name“ in Algol. Seine Auswertung, deren Effekt ja typischerweise vom aktuellen Wert der Variablen `v` abhängt, erfolgt dann jedesmal, wenn der Parameter `body` in der Implementierung des `for`-Operators verwendet wird, d. h. konkret einmal pro Iteration.

3.2 Bildung von Teilsequenzen

Mit den in §2.4 erwähnten Operatoren und einem weiteren vordefinierten Operator _ + _ zur Verkettung zweier Sequenzen oder einer Sequenz und eines einzelnen Elements könnte man z. B. wie folgt einen Operator zur Bildung von Teilsequenzen definieren:

```
["T" : type; "s" : T*; "i" : int; "j" : int]
"subseq s i j" : T*
{ "t" : T*?; "k" : int?; k << max i 1;
  while ?k <= min j #s do
    t << ?t + s[?k]; k << ?k + 1
  end;
  ?t
}
```

Allerdings ist die Bedeutung einer Anwendung wie z. B. `subseq s 3 7` ohne Konsultation der Dokumentation nicht ohne weiteres klar. Wird eine Teilsequenz vom dritten bis zum siebten Element von `s` geliefert – und wenn ja, sind die Randelemente jeweils im Ergebnis enthalten oder nicht –, oder erhält man eine Teilsequenz mit sieben Elementen, die beim dritten Element von `s` beginnt? In unterschiedlichen Sprachen bzw. den zugehörigen Bibliotheken findet man in der Tat zahlreiche Variationen zu diesem Thema.

Wenn man sich gedanklich von der gewohnten Funktions- bzw. Methodensyntax anderer Sprachen löst und berücksichtigt, dass Operatoren in MOSTflexiPL eine beliebige Syntax besitzen können, kann man mit etwas Kreativität z. B. folgende Anwendungsmöglichkeiten konzipieren:

```
s[3 .. 7]      s[3 .. 7)      s(3 .. 7]      s(3 .. 7)
```

Hier dürfte aufgrund bekannter mathematischer Konventionen auch ohne viel Erklärung klar sein, in welchem Fall welche Randelemente zur gelieferten Teilsequenz gehören und welche nicht. Auch weitere Variationen wie z. B. `s[3 ..]` oder `s(.. 7]` dürften nahezu selbsterklärend sein. So liefert die Kombination `s[3 ..][.. 7]` z. B. eine Teilsequenz mit sieben Elementen, die beim dritten Element von `s` beginnt.

Der Operator `_ [ _ .. _ ]` besitzt dieselbe Implementierung wie der obige Operator `subseq _ _ _`, er bietet lediglich eine andere syntaktische „Verpackung“ an:

```
["T" : type; "s" : T*; "i" : int; "j" : int]
"s [ i .. j ]" : T*
{ subseq s i j }
```

Auch die übrigen Operatoren können auf diesen zurückgeführt werden, z. B.:

```
["T" : type; "s" : T*; "i" : int; "j" : int]
"s [ i .. j )" : T*
{ subseq s i j-1 }
```

```
["T" : type; "s" : T*; "i" : int]
"s [ i .. ]" : T*
{ subseq s i #s }
```

4 Implementierung

Abbildung 1 zeigt die funktionale Grobstruktur des Parsers, der zusammen mit der statischen Typprüfung das Herzstück des MOSTflexiPL-Compiler-Frontends darstellt.

Die meisten `parse`-Funktionen erhalten als Parameter die aktuelle Position im Eingabestrom sowie die Menge der an dieser Stelle sichtbaren Operatordeklarationen. Zu Beginn ist dies die Menge der vordefinierten Operatoren, zu der durch Aufrufe von `create_oper` sukzessive benutzerdefinierte Operatoren hinzukommen können. Durch operatorspezifische Import- und Exportdeklarationen (vgl. §2.5) kann die Menge der sichtbaren Deklarationen aber auch gezielt lokal eingeschränkt werden.

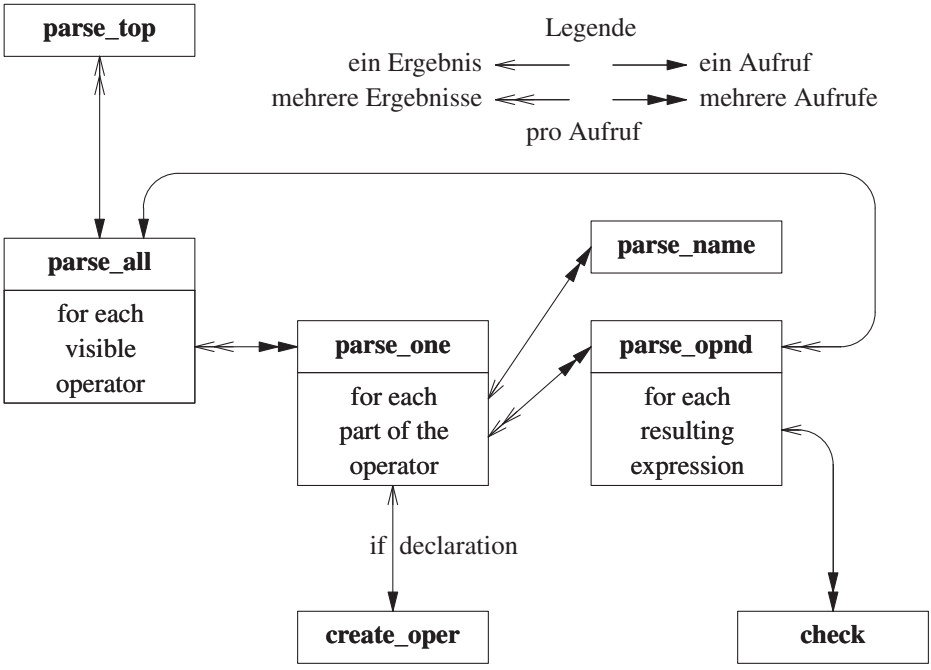


Abbildung 1: Funktionale Grobstruktur des Parsers

Da eine Eingabefolge prinzipiell immer mehrere Interpretationsmöglichkeiten besitzen kann, von denen manche später aufgrund von Ausschlussdeklarationen (vgl. §2.5) oder Typprüfungen eventuell wieder ausgesondert werden können, liefern die meisten Funktionen als Resultat eine Menge von Ausdrücken (d. h. Operatoranwendungen), die mögliche Zerlegungen der nachfolgenden Eingabe darstellen.

Die parameterlose Funktion `parse_top`, die vom Hauptprogramm des Compilers aufgerufen wird, ruft die Funktion `parse_all` mit Eingabeposition 1 und der Menge der vordefinierten Operatordeklarationen auf. `parse_all` hat die Aufgabe, *alle* möglichen Zerlegungen der nachfolgenden Eingabe zu ermitteln und zurückzuliefern. Sie ruft hierfür für jeden sichtbaren Operator die Funktion `parse_one` auf, die alle möglichen Anwendungen dieses *einen* Operators ermittelt und zurückliefert. `parse_one`

arbeitet hierfür die Syntax des Operators Schritt für Schritt ab, indem sie für jeden Teil der Syntax eine der Hilfsfunktionen `parse_name` (wenn es sich um einen Operatornamen handelt) oder `parse_opnd` (wenn es sich um einen Platzhalter für einen Operanden handelt), jeweils mit der entsprechenden Eingabeposition, aufruft. `parse_name` überprüft lediglich, ob der jeweilige Operatorname an der jeweiligen Position in der Eingabe steht, wenn zuvor ggf. Zwischenraum und Kommentar überlesen wird. (Das heißt, `parse_name` ist eigentlich ein degenerierter Scanner.) Wenn dies nicht der Fall ist, kann die Verarbeitung des Operators in `parse_one` sofort abgebrochen werden.

`parse_opnd` hat die Aufgabe, zu dem bis jetzt von `parse_one` konstruierten partiellen Ausdruck einen weiteren Operanden hinzuzufügen. Um die Menge der prinzipiell möglichen Operanden an der entsprechenden Eingabeposition zu ermitteln, wird zunächst rekursiv `parse_all` aufgerufen. Für jeden von dieser Funktion zurückgelieferten Ausdruck wird dann mit Hilfe der Funktion `check` überprüft, ob sein Typ auf den Typ des entsprechenden expliziten Parameters passt und ob keine Ausschlussdeklaration verletzt wird. Wenn dies der Fall ist, wird der Ausdruck als Operand zu einer Kopie des partiellen Ausdrucks hinzugefügt. Als Resultat liefert `parse_opnd` dann die Menge all dieser erweiterten Ausdrücke.

Auf diese Weise werden durch rekursives Backtracking alle möglichen Zerlegungen der Eingabe in typkorrekte Ausdrücke ermittelt. Wenn es genau eine solche Zerlegung gibt, wird die Eingabe als korrektes Programm akzeptiert und vom Compiler-Backend für diesen Ausdruck Zielcode generiert. Wenn es keine typkorrekte Zerlegung gibt, ist das Programm fehlerhaft. Gibt es mehr als eine korrekte Zerlegung, ist das Programm mehrdeutig (und damit ebenfalls fehlerhaft), was insbesondere dann auftreten kann, wenn der Vorrang zwischen Operatoren nicht ausreichend festgelegt wurde.

Um die Performance des Backtracking-Algorithmus zu verbessern, werden fehlerhafte Zerlegungen durch die in den Parse-Vorgang integrierte Typprüfung so früh wie möglich erkannt und ausgesondert. Außerdem werden wiederholte Aufrufe von `parse_all` mit denselben Aufrufparametern (Eingabeposition und sichtbare Deklarationen), die aufgrund der Backtracking-Strategie sehr häufig auftreten, durch dynamisches Programmieren optimiert, indem die einmal ermittelten Zerlegungen in einer Tabelle gespeichert werden. Damit ist die Laufzeit des Compilers normalerweise etwa proportional zur Länge der Eingabe.

5 Verwandte Arbeiten

Eine der ältesten Programmiersprachen, die von Anfang an syntaktisch erweiterbar konzipiert wurde, dürfte Lisp sein [St90]. Ähnlich wie in MOSTflexiPL, besteht in Lisp weder ein Unterschied zwischen Operatoren (wie z. B. +) und Funktionen (wie z. B. `max`) noch ein Unterschied zwischen vordefinierten und benutzerdefinierten Operatoren bzw. Funktionen. Durch die Definition neuer Funktionen – oder Makros, die syntaktisch genauso verwendet werden wie Funktionen – erweitert man in gewisser Weise ständig die Syntax der Sprache.

Dies hatte zur Folge, dass zahlreiche problemorientierte Sprachen (domain-specific languages), beispielsweise im Bereich Wissensrepräsentation [Ma91], als Spracherweiterungen von Lisp konzipiert und implementiert wurden. Aber auch das Common

Lisp Object System (CLOS), das heute ein fester Bestandteil von Common Lisp ist, ist nichts anderes als eine Erweiterung der ursprünglichen funktionalen Sprache Lisp.

Ebenfalls analog zu MOSTflexiPL ist die Tatsache, dass Spracherweiterungen in der Sprache selbst formuliert werden und dass hierfür ein äußerst kleiner Sprachkern ausreichend ist. Wesentliche Unterschiede bestehen jedoch darin, dass Lisp kein statisches Typsystem besitzt und dass Ausdrücke immer geklammert werden müssen, wodurch der syntaktische Gestaltungsspielraum deutlich eingeschränkt ist. Außerdem kommt MOSTflexiPL bis jetzt ohne ein „prozedurales“ Makrosystem aus, d. h. es wird keinerlei Benutzercode zur Übersetzungszeit ausgeführt.[5]

Zusammengefasst bietet MOSTflexiPL im wesentlichen die gleiche Flexibilität wie Lisp, aber mit vollkommen frei gestaltbarer Syntax sowie statischer Typprüfung.

Eine modernere Sprache, die viele Ideen von Lisp übernommen hat, ist Dylan [Cr97]. Auch sie erlaubt dem Programmierer, die Syntax der Sprache in der Sprache selbst (eigentlich in einem Makrosystem, das Teil der Sprache ist) zu erweitern. Obwohl die Möglichkeiten weiter reichen als mit den einfachen Klammerausdrücken von Lisp, sind dem Anwender dennoch klare syntaktische Grenzen gesetzt, die er nicht überschreiten kann. Demgegenüber erlaubt das Operatorkonzept von MOSTflexiPL nahezu beliebige syntaktische Gestaltungsmöglichkeiten. Außerdem bietet auch Dylan kein statisches Typsystem.

Zahlreiche unterschiedliche Sprachen, wie z. B. Haskell [Ma10] und Prolog [CM94], bieten die Möglichkeit, neue Operatorsymbole zu definieren und damit die Syntax von Ausdrücken zu erweitern. Da in funktionalen Sprachen, ähnlich wie in MOSTflexiPL, kein Unterschied zwischen Ausdrücken und Anweisungen besteht, ist damit prinzipiell auch die Syntax von Anweisungen (z. B. Kontrollstrukturen) erweiterbar. Nicht erweiterbar ist jedoch die Syntax von Typen und Deklarationen.

Durch das einfache Prinzip „*Alles* ist ein Ausdruck“, unter anderem auch Typen und Deklarationen, lassen sich diese Teile der Sprache in MOSTflexiPL durch die Definition neuer Operatoren genauso erweitern wie beispielsweise Ausdrücke und Anweisungen.

Neben reinen Programmiersprachen, die innerhalb bestimmter Grenzen „in sich selbst“ erweiterbar sind, gibt es diverse „Frameworks“, bei denen syntaktische Erweiterungen – wiederum innerhalb bestimmter Grenzen – in einer separaten Metasprache definiert werden können. Beispiele hierfür sind XL bzw. XLR [Di12], Seed7 [Me12] und JSE [BP01].

Die Nachteile dieser Ansätze gegenüber MOSTflexiPL sind einerseits, dass Spracherweiterungen nicht direkt und einfach in der Sprache selbst formuliert werden können, und andererseits, dass wiederum bestimmte „syntaktische Grenzen“ nicht überschritten werden können.

Ein Ansatz, dessen Grundidee der von MOSTflexiPL am ähnlichsten ist, ist „π – a Pattern Language“ [KM09]. Das dort als Pattern bezeichnete Grundkonstrukt der Sprache – nach Aussage der Autoren das einzige, das es gibt – entspricht einem Operator in MOSTflexiPL: Es besitzt eine Syntax, die aus Namen bzw. Symbolen sowie Platzhaltern für Operanden besteht, und eine Bedeutung, die der Implementierung ei-

[5] Es gibt lediglich einen deklarativen (und typsicheren) Ersetzungsmechanismus (sog. „virtuelle Operatoren“), der u. a. zur Abkürzung von Typausdrücken verwendet werden kann, z. B. `IntSeq = int*`. Die Ersetzung von `IntSeq` durch `int*` ist wichtig, damit der Compiler beide als gleichen Typ erkennt (vgl. §2.4).

nes Operators in MOSTflexiPL entspricht. Damit bieten beide Ansätze dieselbe syntaktische Flexibilität, die letztlich daher rührt, dass beide Sprachen keinerlei vordefinierte Grammatik besitzen.

Ein entscheidender Pluspunkt von MOSTflexiPL gegenüber π ist wiederum das statische Typsystem – π ist vollständig dynamisch typisiert. Darüber hinaus bietet MOSTflexiPL eine Reihe weiterer Möglichkeiten, wie z. B. implizite und deduzierte Parameter (wobei letztere in einer dynamisch typisierten Sprache nicht benötigt werden) sowie Import-, Export- und Ausschlussdeklarationen, mit denen sich z. B. lokal begrenzte Syntaxerweiterungen realisieren lassen.

6 Zusammenfassung und Ausblick

MOSTflexiPL ist eine momentan in Entwicklung befindliche Programmiersprache, die vom Anwender nahezu beliebig syntaktisch erweitert und angepasst werden kann und damit u. a. als erweiterbare „general purpose programming language“ sowie zur Einbettung von „domain-specific languages“ verwendet werden kann. Sie besitzt ein statisches Typsystem und wird durch einen Compiler nach C++ übersetzt.

Seit der Verfügbarkeit eines ersten, stellenweise noch unvollständigen Compilers[6] konnten erste Erfahrungen mit der Verwendung der Sprache gesammelt werden. Dabei traten, wie nicht anders zu erwarten, noch einige Schwachstellen zutage, die im Zuge der Weiterentwicklung beseitigt werden sollen.

Das größte konzeptuelle Defizit ist momentan die Tatsache, dass benutzerdefinierte Deklarationsoperatoren (d. h. vom Anwender geschriebene Operatoren, die zur Deklaration anderer Operatoren verwendet werden sollen) nur sehr eingeschränkt funktionieren. Um diesen Mangel zu beseitigen, muss das Konzept von Deklarationen noch einmal grundlegend überarbeitet werden, insbesondere die Interpretation der Signatur eines Operators, die momentan eine Stringkonstante sein muss und daher z. B. nicht von den Parametern eines benutzerdefinierten Deklarationsoperators abhängen kann.

Weitere konzeptuelle Verbesserungen betreffen die Unterstützung für benutzerdefinierte Literale, Zwischenraum und Kommentare; momentan sind diese Sprachmerkmale fest vordefiniert. Zur Definition komplexerer Kontrollstrukturen wird noch ein allgemeiner Sprungmechanismus benötigt, mit dem sich dann spezielle Anweisungen wie `break`, `return` oder `throw` realisieren lassen.

Weitere Punkte auf dem konzeptuellen Wunschzettel sind implizite Typumwandlungen – die natürlich auch wieder beliebig benutzerdefinierbar sein sollen – sowie ein geeignetes Modulkonzept. Für beide Punkte sind die wesentlichen Ideen vorhanden und ausgearbeitet – daher findet sich das Stichwort „modular“ bereits im Namen der Sprache –, sie müssen aber noch im Compiler implementiert werden.

Schließlich gibt es momentan nur eine rudimentäre Fehlerdiagnose und -behandlung. Diese zu verbessern, dürfte aufgrund der Flexibilität der Sprache noch ein umfangreiches und schwieriges Forschungsunterfangen sein.

Neben diesen konzeptuellen Verbesserungen, muss auch der MOSTflexiPL-Compiler an diversen Stellen weiterentwickelt und verbessert werden. Beispielsweise gerät die

[6] Dieser wurde zum größten Teil von den Studierenden Marco Perazzo, Miriam Klement und Stefan Billet im Rahmen ihrer exzellenten Abschlussarbeiten implementiert.

Suche nach impliziten Parameterbelegungen bei „bösartigen“ Beispielprogrammen momentan noch in eine Endlosschleife, während die Suche nach deduzierten Parameterbelegungen bei komplexen Beispielen noch unvollständig ist. Außerdem kann sowohl die Performance des Compilers als auch die des generierten Codes noch deutlich verbessert werden. Schließlich sollen in Zukunft weitere Zielsprachen unterstützt werden, beispielsweise Java-Quell- oder Bytecode, wodurch MOSTflexiPL-Programme auf jeder Java Virtual Machine ausgeführt werden können.

Literaturverzeichnis

[BP01] J. Bachrach, K. Playford: “The Java Syntactic Extender (JSE).” In: *Proc. 2001 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA ’01)* (Tampa Bay, FL, October 2001), 31–42.

[CM94] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[Cr97] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[Di12] C. de Dinechin: *XL Extensible Language*. http://xlr.sourceforge.net (18.01.2012)

[He07] C. Heinlein: “Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance.” *Journal of Object Technology* 6 (3) March/April 2007, 101–151, http://www.jot.fm/issues/issue_2007_03/article3.

[KM09] R. Knöll, M. Mezini: “π – a Pattern Language.” In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009), 503–521.

[Ma10] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010. http://haskell.org/definition/haskell2010.pdf (18.01.2012)

[Ma91] R. MacGregor: “The Evolving Technology of Classification-Based Knowledge Representation Systems.” In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385–400.

[Me12] T. Mertes: *Seed7 – The Extensible Programming Language*. http://seed7.sourceforge.net (18.01.2012)

[St90] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[Wi12] Wikipedia contributors: *Dependent Type*. http://en.wikipedia.org/w/index.php?title=Dependent_type&oldid=470369653 (18.01.2012)

Search Strategies for Functional Logic Programming

Michael Hanus Björn Peemöller Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh|bjp|fre}@informatik.uni-kiel.de

Abstract: In this paper we discuss our practical experiences with the use of different search strategies in functional logic programs. In particular, we show that complete strategies, like breadth-first search or iterative deepening search, are a viable alternative to incomplete strategies, like depth-first search, that have been favored in the past for logic programming languages.

1 Introduction

Functional logic languages combine the most important features of functional and logic programming in a single language (see [AH10, Han07] for recent surveys). In particular, they provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination led to new design patterns [AH02, AH11] and better abstractions for application programming, e.g., as shown for programming with databases [BHM08, Fis05], GUI programming [Han00], web programming [Han01, Han06, HK10], or string parsing [CLF99]. Moreover, it is a good basis to teach the ideas of functional and logic programming, or declarative programming in general, with a single computation model and programming language [Han97]. The operational principles of functional logic languages have also been used for other computation tasks, like inverse computations [AGK06], partial evaluation [AFV98], or generation of test cases [FK07, RNL08].

An important feature of logic programming languages is non-deterministic search. In Prolog, which is still the standard language for logic programming, non-deterministic search is implemented via backtracking, which corresponds to a depth-first search traversal through the SLD proof tree [Llo87]. Due to this feature of Prolog, the idea of logic programming is often reduced to the combination of unification and backtracking, as shown by approaches to add logic programming features to existing functional languages (e.g., [CL00, Hin01]). This limited "backtracking" view of logic programming is also harmful to beginners, e.g., when newbies define their family relationships using a Prolog rule like

```
sibling(X,Y) :- sibling(Y,X).
```

In such cases, one has to explain from the beginning the pitfalls of backtracking which harms the understanding of declarative programming. From a declarative point of view,

a logic program defines a set of rules and a logic programming system tries to find a solution to a query w.r.t. this set of rules. In order to abstract from operational details, the search strategy has to be complete. Due to these considerations, the functional logic language Curry [He06] does not fix a particular search strategy so that different Curry implementations can support different (or also several) search strategies. Moreover, Curry implementations also support encapsulated search where non-deterministic computations are represented in a data structure so that different search strategies can be implemented as tree traversals [BHH04, HS98, Lux99].

In this paper, we present our practical results with different search strategies implemented in a new Curry system called KiCS2 [BHPR11]. KiCS2 compiles Curry programs into Haskell programs where non-deterministic values and computations are represented as tree structures so that flexible search strategies can be supported. Although the incomplete depth-first search strategy is the most efficient one (provided that it is able to find a result value), we show that complete strategies, like breadth-first search or iterative deepening search, are a reasonable alternative that does not force the programmer to consider the applied search strategy in his program.

In the next section, we briefly recall some principles of functional logic programming and the programming language Curry that are necessary to understand the remaining part of the paper. The encapsulation of search and the implementation of different search strategies are discussed in Section 3. These strategies are evaluated with a number of benchmarks in Section 4 before we conclude in Section 5.

2 Functional Logic Programming and Curry

Integrated functional logic programming languages combine features from functional programming and logic programming. Recent surveys are available in [AH10, Han07]. Curry [He06] is a functional logic language which extends lazy functional programming as to be found in Haskell [PJ03] and additionally supports logic programming features. Another functional logic language based on similar principles is $\mathcal{TOY}$ [LFSH99]. However, $\mathcal{TOY}$ does not offer flexible search strategies by a concept of encapsulating search (although it provides a concept of nested computation spaces in order to deal with failures in functional logic programming [LFSH04, SH06]). Therefore, we use Curry throughout this paper, although the concepts presented here could be also integrated in other functional logic languages.

A Curry program consists of the definition of data types and operations on these types. Since the syntax of Curry is close to Haskell, variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. Note that in a functional logic language operations might yield more than one result on the same input due to the logic programming features. For instance, Curry contains a *choice* operation defined by:

```
x ? _ = x
_ ? y = y
```

The choice operation can be used to define other non-deterministic operations like

```
coin = 0 ? 1
```

Thus, the expression "`coin`" has two values: `0` and `1`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form

$f\ t_1 \ldots t_n$ `|` c `=` e

where the (optional) condition c is a *constraint*, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is a value of type `Success` that denotes the always satisfiable constraint. Thus, we say that a constraint c is *satisfied* if it can be evaluated to `success`. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable values.

As a simple example, consider the following Curry program which defines a polymorphic data type for lists and operations to compute the concatenation of lists and the last element of a list:[1]

```
data List a = [] | a : List a    -- [a] denotes "List a"

-- "++" is a right-associative infix operator
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] → a
last xs | (ys ++ [z]) =:= xs = z
  where ys, z free
```

Logic programming is supported by admitting function calls with free variables (e.g., `(ys++[z])` in the rule defining `last`) and constraints in the condition of a defining rule. In contrast to Prolog, free variables need to be declared explicitly to make their scopes clear (e.g., "`where ys,z free`" in the example). A conditional rule is applicable if its condition is satisfiable. Thus, the rule defining `last` states in its condition that `z` is the last element of a given list `xs` if there exists a list `ys` such that the concatenation of `ys` and the one-element list `[z]` is equal to the given list `xs`.

Curry also offers standard features of functional languages, like modules or monadic I/O which is identical to Haskell's I/O concept [Wad97]. Thus, "`IO` α" denotes the type of an I/O action that returns values of type α.

The operational semantics of Curry is based on an optimal evaluation strategy [AEH00] which is a conservative extension of lazy functional programming and (concurrent) logic programming. A big-step and a small-step operational semantics of Curry can be found

[1] Note that lists are a built-in data type with a more convenient syntax, e.g., one can write `[x,y,z]` instead of `x:y:z:[]` and `[a]` instead of the list type "`List a`".

in [AHH+05]. Curry's semantics is sound in the sense of logic programming, i.e., each computed result is correct and for each correct result there is a more general computed one [AEH00]. In order to achieve completeness, one has to take *all* possible non-deterministic derivation paths into account. In contrast to Prolog, which fixes a (potentially incomplete) depth-first search strategy to find solutions, Curry does not fix a particular search strategy. Actually, descriptions of the model-theoretic [GMHGLFRA99] or operational [AHH+05] semantics of functional logic languages do not take a search strategy into account. Thus, different Curry implementations can support various search strategies. For instance, the Curry implementation PAKCS [HAB+10], which compiles Curry programs into Prolog programs, supports only a depth-first search strategy. MCC [Lux99] compiles Curry programs into C programs and uses also a depth-first search strategy to find the solutions of a given top-level goal. In addition, MCC offers the encapsulation of search (see below) so that other search strategies, like a complete breadth-first search strategy, can be used inside a Curry program. The Curry implementation KiCS [BH07, BH09], which compiles Curry programs into Haskell programs, offers depth-first and breadth-first search strategies for top-level goals as well as the encapsulation of search with user-definable strategies. In this paper we consider the Curry implementation KiCS2 [BHPR11], which is based on similar ideas than KiCS but uses a different compilation model avoiding side effects to enable better optimizations for target programs. KiCS2 also offers different search strategies for top-level goals and the encapsulation of search with user-definable strategies, which is described next.

3 Search Strategies

As mentioned above, Curry does not enforce a particular search strategy. A Curry implementation can provide various search strategies to find solutions or values for a given constraint or expression. The most advanced system in this respect is KiCS2 [BHPR11], which supports the evaluation of top-level expressions with depth-first search, breadth-first search, iterative deepening search, or parallel search strategies. Curry supports this flexibility since all operations with side effects are collected in monadic I/O operations [Wad97]. As a consequence of this computation model, all non-deterministic computations between I/O operations must be encapsulated since one can not apply two alternative I/O operations to an existing "world" and non-deterministically proceed with two alternative worlds ("one can not duplicate the world"). Therefore, Curry offers the encapsulation of search by representing non-deterministic computations in a data structure so that the computation of different solutions (one solution or all solutions) can conceptually be implemented as traversals on this data structure.

An early approach [HS98, Lux99] to encapsulating search in Curry is based on a primitive search operator

```
try :: (a → Success)  →  [a → Success]
```

that takes a constraint abstraction, e.g., `(\x->x=:=coin)`, as input, evaluates it until the first non-deterministic step occurs, and returns the result: the empty list in case of fail-

ure, a list with a single element in case of success, or a list with at least two elements representing a non-deterministic choice. For instance, `try (\x->x=:=coin)` evaluates to `[\x->x=:=0, \x->x=:=1]`. Based on this primitive, one can define various search strategies to explore the search space and return its solutions. [Lux99] shows an implementation of this primitive.

Although typical search operators of Prolog, like `findall`, `once`, or negation-as-failure, can be implemented using `try`, it became also clear that the combination of encapsulated search and demand-driven evaluation and sharing causes further complications [BHH04]. For instance, in an expression like

```
let y = coin in try (\x → x =:= y)
```

it is not obvious whether the non-determinism caused by the evaluation of `coin` (introduced outside but demanded inside the search operator) should be encapsulated or not. Hence, the result of this expression might depend on the evaluation order. For instance, if `coin` is evaluated before the `try` expression, it results in two computations where `y` is bound to `0` in one computation and to `1` in the other computation. Hence, `try` does not encapsulate the non-determinism of `coin` (this is the semantics of `try` implemented in [Lux99]). However, if `coin` is evaluated inside the capsule of `try` (because it is not demanded before), then the non-determinism of `coin` is encapsulated. These and more peculiarities are discussed in [BHH04]. Furthermore, the order of the solutions might depend on the textual order of program rules or the evaluation time (e.g., in parallel implementations). Hence, it is difficult to define a search operator as a pure function.

Due to these considerations, [BHH04] contains a proposal for another primitive search operator:

```
getSearchTree :: a  →  IO (SearchTree a)
```

It takes an expression and delivers a search tree representing the search space when evaluating the input:

```
data SearchTree a = Value a
                  | Fail
                  | Or (SearchTree a) (SearchTree a)
```

`(Value v)` and `Fail` represent a single value or a failure (i.e., no value), respectively, and `(Or t1 t2)` represents a choice (i.e., a non-deterministic value) between two search trees `t1` and `t2`. Since `getSearchTree` is an I/O action, its result (in particular, the order of subtrees) depends on the current environment, e.g., time of evaluation. To avoid the complications w.r.t. shared variables, `getSearchTree` implements a *strong encapsulation view*, i.e., conceptually, the argument of `getSearchTree` is cloned before the evaluation starts in order to cut any sharing with the environment. Furthermore, the structure of the search tree is computed lazily so that an expression with infinitely many values does not cause the nontermination of the search operator if one is interested in only one solution.

This primitive has been implemented for the first time in KiCS [BH07, BH09] and it is also provided in KiCS2 [BHPR11] considered in this paper. With this primitive, the programmer can define its own search strategy as `SearchTree` traversals in order to collect

all non-deterministic values into a list structure. For instance, a depth-first search strategy can be easily defined as follows:

```
allValuesDFS :: SearchTree a  → [a]
allValuesDFS Fail      = []
allValuesDFS (Value x) = [x]
allValuesDFS (Or  x y) = allValuesDFS x ++ allValuesDFS y
```

Note that the lazy evaluation of traversal operations like `allValuesDFS` has the advantage that the search strategy is decoupled from the control. For instance, we can define the following operation to print a single value of a non-deterministic expression:

```
printFirstValueDFS x =
  getSearchTree x >>= print . head . allValuesDFS
```

Thus, (`printFirstValueDFS` *exp*) can print some value even if the non-deterministic expression *exp* has infinitely many values. This is in contrast to Prolog's constructs for controlling search where different operators are necessary to compute one or all solutions.

It is well known that depth-first search lacks completeness, i.e., it might not be able to compute all existing values. For instance, consider the following operation that non-deterministically returns all increasing numbers from a given number `n`:

```
f n = f (n+1) ? n
```

Although `0` is a value of `(f 0)`, the evaluation of `(printFirstValueDFS (f 0))` does not terminate (provided that the primitive `getSearchTree` explores the non-determinism of "`?`" in left-to-right order). This problem can be avoided with complete search strategies, like breadth-first search strategy, which can be defined on `SearchTree` structures as follows:

```
allValuesBFS :: SearchTree a  → [a]
allValuesBFS t = collect [t]

collect []     = []
collect (t:ts) = values (t:ts) ++ collect (children (t:ts))

values []             = []
values (Fail    : ts) = values ts
values (Value x : ts) = x : values ts
values (Or  _ _ : ts) = values ts

children []             = []
children (Fail    : ts) = children ts
children (Value _ : ts) = children ts
children (Or  x y : ts) = x : y : children ts
```

The operation `values` extracts the values in each level of the tree and the operation `children` extracts all direct successors of a level in order to recursively collect their values.

Using `allValuesBFS`, we can print a value of the expression `(f 0)` by

```
getSearchTree (f 0) >>= print . head . allValuesBFS
```

In order to abstract from the details of the evaluation order, it would be preferable to use complete search strategies. However, complete strategies are often neglected due to performance reasons. For example, the breadth-first search strategy stores all child nodes of a tree level in a list to be explored later, which, as the search space potentially doubles on each level, may lead to an exponentially growing memory usage.

Another complete search strategy with a superior memory behavior is iterative-deepening search. Basically, it is a depth-first search strategy with a depth-bound which is incremented in each iteration. Thus, we compute in each iteration a list of values together with some information whether we have aborted (due to the depth-bound) the computation of further possible values. For this purpose, we define list structures that can also end with an `Abort`:

```
data AbortList a = Nil | Cons a (AbortList a) | Abort
```

and define the concatenation on such lists:

```
concA :: AbortList a  → AbortList a  → AbortList a
concA Abort       Abort       = Abort
concA Abort       Nil         = Abort
concA Abort       (Cons x xs) = Cons x (concA Abort xs)
concA Nil         ys          = ys
concA (Cons x xs) ys          = Cons x (concA xs ys)
```

Now we define an operation to collect values in a search tree within some level bounds (to avoid the repeated collection of values found in each iteration):

```
collectInBounds :: Int  → Int  → SearchTree a  → AbortList a
collectInBounds oldbound newbound st = collectLevel newbound st
 where
  collectLevel _ Fail      = Nil
  collectLevel d (Value x) = if d <= newbound - oldbound
                             then Cons x Nil
                             else Nil
  collectLevel d (Or  x y) =
    if d > 0
    then concA (collectLevel (d-1) x) (collectLevel (d-1) y)
    else Abort
```

Now, the entire search strategy consists of repeated calls to `collectInBounds` as long as the result list is aborted. In order to experiment with different parameters, the initial depth bound and the method to increase the depth bound in each iteration are passed as parameters to the main operation:

```
allValuesIDS :: Int  → (Int  → Int)  → SearchTree a  → [a]
allValuesIDS initdepth incrdepth st =
  iterIDS initdepth (collectInBounds 0 initdepth st)
 where
  iterIDS _ Nil = []
```

```
iterIDS n (Cons x xs) = x : iterIDS n xs
iterIDS n Abort =
  let newdepth = incrdepth n
   in iterIDS newdepth (collectInBounds n newdepth st)
```

The key advantage of depth-first search in comparison to breadth-first search is its memory behavior: whereas breadth-first search has to store all nodes in a level of the search tree in parallel, depth-first search only needs to store the nodes in the branch from the root to the current node under investigation. Since iterative deepening uses depth-first search in each iteration, it should have a memory behavior similarly to depth-first search and, in case of wide search trees, better than breadth-first search. The price for this behavior is the recomputation of the initial goal in each iteration. However, since we defined iterative deepening on a search tree, recomputation is not required. Instead, the already evaluated part of the search tree is kept in memory, sacrificing the better memory behaviour. Therefore, we also implemented the iterative deepening strategy for top-level goals without an explicit search tree but with a recomputation of the initial goal in each iteration (see Section 4).

The various operators to encapsulate search can also be used to implement an interactive top-level search to print all values of an expression as requested by the user. For instance, the following I/O operation interactively prints all elements of a given list:

```
printResults :: [a] → IO ()
printResults []     = putStrLn "No more values"
printResults (x:xs) = do print x
                         putStr "More values? "
                         inp <- getLine
                         if inp == "yes" then printResults xs
                                         else done
```

Hence, an interactive Prolog-like top-level behavior to show the values of an expression exp in depth-first order can be obtained by

```
getSearchTree exp >>= printResults . allValuesDFS
```

In addition, we can print the results in breadth-first order by using `allValuesBFS` instead of `allValuesDFS`. Based on these ideas, KiCS2 provides an interactive top-level search where the user can select various search strategies, e.g., depth-first, breadth-first, iterative deepening, or an experimental implementation of parallel search. However, the top-level search in KiCS2 is not implemented via the primitive encapsulation operators but in a monadic style (see also [BHPR11]) in order to avoid the explicit construction of the `SearchTree` structure. Thus, in the next section we both compare the different search strategies introduced above as well as the top-level search with the encapsulated search, in order to evaluate the potential overhead caused by abstracting and programming with search structures.

4 Benchmarks

In this section we evaluate the practical behavior of the various search strategies discussed so far. Since they are only supported by the Curry implementation KiCS2, we use this system for our evaluation. A general comparison of KiCS2 and other Curry implementations can be found in [BHPR11]. Since KiCS2 compiles Curry programs into Haskell programs, we used the Glasgow Haskell Compiler (GHC 7.0.4, option -O2) to compile and execute the generated target programs. All benchmarks were executed on a 32bit Linux machine running Ubuntu 11.10 with an Intel Core 2 Duo (2.13 GHz) processor and 4 GiB RAM. The timings were performed with the time command measuring the execution time (in seconds) of a compiled executable for each benchmark. Due to the lack of precise measurements of the space behavior, we measured only the execution times.

Program	DFS	IDS(+1)	IDS(*2)	eDFS	eBFS	eIDS(+1)	eIDS(*2)
PermSort	13.12	841.58	32.77	38.48	66.82	72.64	46.79
Last	0.10	660.08	0.26	0.34	0.19	6.59	0.39
Half	0.67	561.45	1.62	1.43	1.22	1.86	1.71
Graph	1.76	52.53	2.40	3.51	4.22	4.26	3.83
HorseMan	2.64	639.64	6.63	3.23	3.29	3.20	3.17
MAC	6.80	544.05	7.84	21.19	18.34	18.59	21.03
Queens	24.74	374.64	33.62	42.71	43.98	44.01	42.65

Figure 1: Benchmarks: computing all values

Figure 1 shows the benchmark results when all values of a given expression are computed with various search strategies. All benchmark programs are non-deterministic programs. "PermSort" sorts a list containing 15 elements by enumerating all permutations and selecting the sorted ones, "Last" computes the last element `x` of a list `xs` containing 10,000 elements by solving the equation "`ys++[x] =:= xs`" (see Section 2), "Half" computes the half `y` of a natural number `x` (in Peano representation) by solving the equation `y+y=:=x`, "Graph" computes some path in a graph where the edges are represented by a non-deterministic "successor" operation, "HorseMan" computes the numbers of horses and men from given numbers of heads and feet by searching for appropriate numbers in Peano representation, "MAC" solves the "Missionaries and Cannibals" puzzle (see [AH02, Sect. 3.1]) for reasonable numbers of missionaries and cannibals to obtain measurable run times, and "Queens" computes safe placements of queens on a chess board.

The columns in Figure 1 are the various search strategies considered in this paper. "DFS" denotes the top-level depth-first search strategy which is similar to `allValuesDFS` but implements this strategy in a monadic style without the explicit construction of a search tree. The encapsulated search strategies are prefixed by "e", i.e., "eDFS" and "eBFS" correspond to `allValuesDFS` and `allValuesBFS`, respectively. "eIDS(+1)" corresponds to `allValuesIDS` with an initial depth bound of `10` and the depth increment operation `(+1)`, whereas "eIDS(*2)" uses the same initial depth bound but the depth increment operation `(*2)`, i.e., to depth bound is doubled in each iteration. Finally, "IDS(+1)" and "IDS(*2)" are iterative deepening strategies with similar parameters but recompute the

Program	DFS	IDS(+1)	IDS(*2)	eDFS	eBFS	eIDS(+1)	eIDS(*2)
PermSort	12.57	804.82	32.09	34.75	67.02	70.47	46.66
Last	0.10	661.92	0.24	0.35	0.19	6.57	0.39
Half	0.34	72.64	0.58	0.69	0.59	0.78	0.78
Graph	0.00	0.10	0.04	0.00	0.04	0.04	0.03
HorseMan	2.06	636.62	6.64	2.52	2.60	2.61	2.57
MAC	0.14	38.75	0.94	0.94	2.81	2.75	2.79
Queens	1.40	370.26	33.65	2.44	8.95	8.86	3.04
NDNums	oom	47.89	0.14	oom	oom	oom	0.47

Figure 2: Benchmarks: computing a single value

initial expression in each iteration in order to trade space for run time, as discussed in Section 3.

The table entries in Figure 1 contain the run times in seconds to compute all values of the initial expression. As one can see, the top-level depth-first search is the most efficient search strategy. The encapsulated version of depth-first search (eDFS) with the explicit construction of the search tree causes some overhead, since the search strategy is encoded in the source program rather than in the run-time system as in the top-level search (DFS). The benchmarks also show that complete search strategies, like "eBFS" and "eIDS(*2)" are viable alternatives to the incomplete depth-first search strategy. Their overhead (sometimes they are even faster than "eDFS", but they are always slower than top-level search) is acceptable taking into account the fact that one need not to reason about the details of exploring the search space. The behavior of iterative deepening is largely influenced by its parameterization. A constant increment of the depth bound causes a big overhead compared to doubling the depth bound in each iteration, in particular, when the search tree is slim as in "Last". This is even worse for top-level iterative deepening which recomputes the initial expression in each iteration, see "IDS(+1)".

The benchmarks indicate that breadth-first search seems to be a good strategy taking into account the size of memory provided by modern computers. However, if the search tree is wide (i.e., the nodes in each level increases with the depth of the tree), then iterative deepening is superior to breadth-first search. To examine such a case, we added a benchmark "NDNums" which defines a non-deterministic operation with a high branching factor that returns all inreasing numbers

```
g n = g (n+1) ? n ? g (n+1)
```

and solve the equation "`g 0 =:= 29`". Obviously, the computation of all values will never terminate. Therefore, we executed the benchmarks to compute only a first value of an expression via different search strategies. Figure 2 contains the corresponding results where "oom" denotes a memory overflow in a computation. As one can see, iterative deepening is the only search strategy that is able to find a solution in all examples.

5 Conclusions

We discussed the use of different search strategies in functional logic programs. In order to enable user-programmable search strategies, modern functional logic languages like Curry provide primitives to represent non-deterministic computations or values as data structures that can be traversed like any term structure. The Curry implementation KiCS2, considered in this paper, provides a primitive `getSearchTree` that returns a tree representation of a non-deterministic computation. This representation can be used to define various search strategies, like depth-first search, breadth-first search, or iterative deepening search.

We have practically evaluated and compared the efficiency of these strategies. The benchmarks indicated that complete strategies are a viable alternative to incomplete strategies, that have been favored in the past due to limited memory requirements. Decoupling non-deterministic programs from their search strategy could lead to a more declarative programming style (instead of the use of predicates with side effects as in Prolog) and enable more potential for optimization, e.g., parallel search strategies. The explicit definition of search strategies has also been advocated for combining logic programs with different constraint solvers [FHPR06, Sch97].

Our results indicate that it could be reasonable to make complete strategies the default in functional logic languages. This would have a good impact on teaching declarative programming to beginners. Furthermore, if efficiency and memory limitations are important, one can still use an efficient strategy, like depth-first search, provided that it is able to find all solutions (e.g., in case of a finite search space). It is an interesting topic for future work to statically approximate situations where the use of theoretically incomplete strategies is sufficient.

References

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[AFV98] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[AGK06] S. Abramov, R. Glück, and Y. Klimov. An Universal Resolving Algorithm for Inverse Computation of Lazy Languages. In *Perspectives of Systems Informatics (PSI 2006)*, pages 27–40. Springer LNCS 4378, 2006.

[AH02] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.

[AH10] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, 53(4):74–85, 2010.

[AH11] S. Antoy and M. Hanus. New Functional Logic Design Patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.

[AHH+05] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[BH07] B. Braßel and F. Huch. On a Tighter Integration of Functional and Logic Programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.

[BH09] B. Braßel and F. Huch. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management*, pages 195–205. Springer LNAI 5437, 2009.

[BHH04] B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[BHM08] B. Braßel, M. Hanus, and M. Müller. High-Level Database Programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[BHPR11] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[CL00] K. Claessen and P. Ljunglöf. Typed Logical Variables in Haskell. In *Proc. ACM SIGPLAN Haskell Workshop*, Montreal, 2000.

[CLF99] R. Caballero and F.J. López-Fraguas. A Functional-Logic Perspective of Parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 85–99. Springer LNCS 1722, 1999.

[FHPR06] S. Frank, P. Hofstedt, P. Pepper, and D. Reckmann. Solution Strategies for Multi-domain Constraint Logic Programs. In *Perspectives of Systems Informatics (PSI 2006)*, pages 209–222. Springer LNCS 4378, 2006.

[Fis05] S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 54–59. ACM Press, 2005.

[FK07] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 63–74. ACM Press, 2007.

[GMHGLFRA99] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[HAB+10] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.

[Han97] M. Hanus. Teaching Functional and Logic Programming with a Single Computation Model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

[Han00] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[Han01] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[Han06] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

[Han07] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[He06] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[Hin01] R. Hinze. Prolog's control constructs in a functional setting - Axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.

[HK10] M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pages 201–216. Springer LNCS 5937, 2010.

[HS98] M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[LFSH99] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[LFSH04] F.J. López-Fraguas and J. Sánchez-Hernández. A Proof Theoretic Approach to Failure in Functional Logic Programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[Lux99] W. Lux. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[RNL08] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM Press, 2008.

[Sch97] C. Schulte. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 519–533. Springer LNCS 1330, 1997.

[SH06] J. Sánchez-Hernández. Constructive Failure in Functional-Logic Programming: From Theory to Implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.

[Wad97] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

Making MPI Intelligent

Dirk Tetzlaff, Sabine Glesner
Chair Software Engineering for Embedded Systems
Technische Universität Berlin, Germany
dirk.tetzlaff@tu-berlin.de
sabine.glesner@tu-berlin.de

Abstract: Mapping parallel applications to multi-processor architectures requires information about the execution times of the concurrent processes to find an optimal allocation and must take into account the interprocessor communication at runtime, whose overheads have emerged as the major performance limitation. However, both information cannot be statically known in advance. In this paper we present a sophisticated approach for mapping parallel MPI applications to concurrent architectures using *machine learning* techniques. This automatically generates heuristics that provide the compiler with knowledge of the considered runtime behavior, hence yielding more precise heuristics than those generated by pure static analyses. The heuristics can be used to direct the runtime environment of MPI, which enables the reallocation of processes to other processors at runtime and, furthermore, results in a better initial allocation of MPI processes.

1 Introduction

The allocation of processes to processing elements (*PEs*) is a vital part of mapping parallel applications to concurrent architectures. Finding an optimal solution requires information about execution times of processes during runtime of applications, which is not known in advance at compile time. Furthermore, it must take into account interprocessor communication at runtime, whose overheads have emerged as the major performance limitation in parallel applications. Hence, one has to predict loop iteration counts if interprocessor communication arises within loop bodies that do not have statically determinable loop bounds. However, static analyses cannot predict iteration counts precisely. Consequently, it is necessary to automatically incorporate knowledge of related dynamic behavior into the compiler but without the need for manual annotations or profiling to preserve an automatic, continuous and efficient compilation flow. Moreover, the technique should be highly scalable to facilitate the integration in industrial-strength compiler environments used for compilation of real-world applications.

We tackle the problem of incorporating statically unknown or imprecise information about the execution time and communication overhead during runtime of applications into the compilation flow with the use of *machine learning* (*ML*) techniques, especially *supervised classification learning*. Our approach thus automatically generates heuristics that efficiently provide the compiler with knowledge of runtime behavior, namely to predict loop

iteration counts and execution times of processes. Our experimental results for learning the loop iteration count can be found in [TG10].

In this paper, we focus on mapping MPI programs to concurrent architectures. We define how the runtime behavior predictions are used to allocate MPI processes to PEs and we present in detail the static code features used for learning the loop iteration count. Using our machine learned information for the mapping of MPI processes to concurrent architectures improves the initial allocation because processes that communicate most frequently with each other can be automatically allocated to PEs with the minimal communication latency in between. Furthermore, it enables the reallocation of processes at runtime whenever their communication behavior changes. Both features are great improvements to current MPI implementations because even the best MPI implementation cannot ensure that for instance frequently communicating processes are placed close to each other [Trä02].

While the learning phase entails a significant overhead, it is only executed once per architecture decoupled from the compilation, so the compile time for applications is not increased. To obtain the training data, we perform several profiling runs with different input for the programs. This data is condensed and abstracted with classification learning that relate static properties of an application to its dynamic behavior at runtime. This bridges the gap between static program analyses on the one hand and dynamic program behavior on the other. Because we use a concise representation (decision trees), the obtained predictors can be efficiently implemented (the resulting code consists of nested `if-else` statements). Due to the high scalability of the resulting heuristics, the additional compile-time overhead is negligible and lower than those of conservative program analyses. Thus, our approach preserves an efficient compilation flow. Since the heuristics are automatically generated and incorporated into the compiler without the need for user intervention, our approach also preserves an automatic, continuous compilation flow. Note that the generated heuristics can also be combined with static program analyses. When the analysis knows the results to be exact, that result can be used, otherwise, the heuristics is consulted. Thereby, our approach combines the benefits of static analyses and profiling without taking their disadvantages.

The paper is organized as follows: First, we outline the background of our approach in Section 2. Our approach to ML-based mapping is given in Section 3 and its implementation in Section 4. Then, we discuss related work in Section 5. Finally, conclusions are drawn in Section 6.

2 Background

In this paper, we contribute to a novel research domain, namely the prediction of runtime behavior to facilitate optimizations during compilation by applying machine learning techniques. In the following, we provide the necessary background knowledge of both areas, machine learning in Section 2.1 and compiler optimizations in Section 2.2. Because we aim at mapping MPI processes to concurrent architectures, we give an overview of MPI in Section 2.3.

2.1 Machine Learning

Machine learning can be used to automatically infer information from a series of observations. It has been central to artificial intelligence research from the beginning [Tur50]. Here, we consider classification learning for which several algorithms have been proposed. One of the most popular methods is classification tree learning [Ste09]. It has become popular because of its clear interpretation and its ability to provide a good fit in many cases [GLM04]. The advantage compared to other methods like *neural networks* (see [Wan03]) or *nearest neighbors* (see recent survey paper [Ind04]) is that the constructed predictors have concise representations (decision trees). Thus, once trained, making predictions takes a negligible amount of time. Moreover, knowledge extraction from decision trees for explanation about the classification process is provided. That is, the decision tree can be inspected to determine which features are important for classification.

For classification learning, each observation is described by its properties, or features, and is categorized concerning a set of given classes. The aim is to build a model that best explains the relationship from features to classes for the training data. The details of building the model via recursive partitioning can be found in [Alp10]. Once trained, the model can also be used to classify new observations, based on their features. Especially, it can be used to generate an executable heuristics. This provides the compiler with intelligence about the dynamic program behavior and can be used for a precise cost estimation. The idea of the heuristics is to focus on typical program behavior instead of considering all possibilities, as it is done by most static analyses.

To evaluate the precision of a predictor, it is applied to data for which the correct classes are known. Then, correct and predicted classes can be compared. For realistic results, predictors should be applied to unseen data for evaluation. As precision measure, the average deviation between predictions and correct classes can be used (mean absolute error). Additionally, considering the correlation between predictions and correct classes helps to decide whether a relationship was actually learned.

2.2 Code Generation

Compilers for high-level programming languages convert the source code to an *intermediate representation* (*IR*), which typically is transformed by subsequent optimizations whereby the semantics of the program has to be preserved. Finally, the IR is translated into machine code. For compilers, program analyses are vital to identify optimization opportunities and to ensure that the program behavior is not changed. Since these analyses are static, they can only make assumptions about how the program may behave at runtime. The obligation to consider every possible eventuality enforces the analyses to over-approximate, which can lead to highly imprecise results. As a consequence, the optimizations are limited and optimization potential is sacrificed.

However, in many cases, programs do not behave as bad as the compiler expects. Hence,

gathering information about the runtime behavior, called *profiling*, can be used to focus optimization efforts on realistic behavior. That is, the program has to be compiled first without runtime information. Then, the binary must be executed with typical input data, and afterwards gathered information is available to direct optimizations. As a result, profile-guided compilation is able to achieve noticeable performance improvements, though having the penalty to re-compile the program after its execution. In general, it works well only when the actual runtime tendencies of the program match those collected during profiling [Smi00]. That is, profiling is strongly dependent on the input data set of that profiling run.

Integrating a beforehand machine learned model into the compiler that holds observations from several profiling runs and relates them to static code features, as we do, eliminates the need for profiling at compile time and has the advantage to focus optimizations on more realistic behavior, not on one single behavior. Thus, we can make a more educated guess whether some transformation may be beneficial to motivate compiler optimizations.

2.3 Message-Passing Interface (MPI)

The Message-Passing Interface (*MPI*) standard is an interface specification of a message-passing library. An MPI program consists of autonomous processes which are identified with a unique ID, called *rank*. Each process is an instance of the same MPI program, executing its own code in an MIMD style. That is, the code executed by each process does not need to be identical. Which portion of this program will be executed by a process can be selected via the rank. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible [Mes09].

The processes communicate via calls to MPI communication primitives like `MPI_Send` and `MPI_Recv`, which are *blocking* operations. In other words, the operations do not return until the message data is stored either into the matching receive buffer or into a temporary system buffer. Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. There are also *synchronous* communication operations, for instance `MPI_Ssend` and `MPI_Srecv`. The synchronous send will complete only if the corresponding receive operation has started to receive the message. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. Other synchronization calls exist, where a number of processes perform a rendezvous, for instance `MPI_Barrier` or `MPI_Win_fence`.

To start an MPI program, the user has to specify how many processes shall execute the program. The *MPI runtime daemon* then starts this number of processes and is responsible

for the communication data transport and the coordination between the processes. The user can specify to which processing element (*PE*) a process will be allocated, but this allocation is fixed even if the runtime behavior indicates that reallocation to another PE would be beneficial. If no advice for the allocation is given, the MPI runtime daemon starts the processes in a Round Robin fashion on available PEs.

3 ML-based Mapping of MPI-Processes

Our approach for learning loop iteration counts and execution times of processes automatically generates classifiers, which relate the static code features to the dynamic behavior, thus providing the compiler with intelligence. At compile time, these classifiers can be used as precise and fast heuristics for communication-aware mapping of MPI processes to a concurrent architecture.

In the following, we outline the two phases of our approach for mapping MPI processes to concurrent architectures using ML-based runtime-behavior predictions, namely the training phase in Section 3.1 and the compilation phase in Section 3.2.

3.1 Training

In the one-off training phase per architecture, we collect code features by means of static analysis as well as dynamic runtime behavior through several profiling runs from a comprehensive suite of programs. Since each program is profiled with different input data, we collect realistic behavior which is less strongly input data dependent. Both information is fed into a learning algorithm, which is intended to find a statistical model (or more precisely, a collection of classifiers) that represents the relationship of static features to dynamic behavior with minimum error (see Figure 1). Since we are looking at supervised classification learning, the considered dynamic behavior has to be discretized to a set of classes.

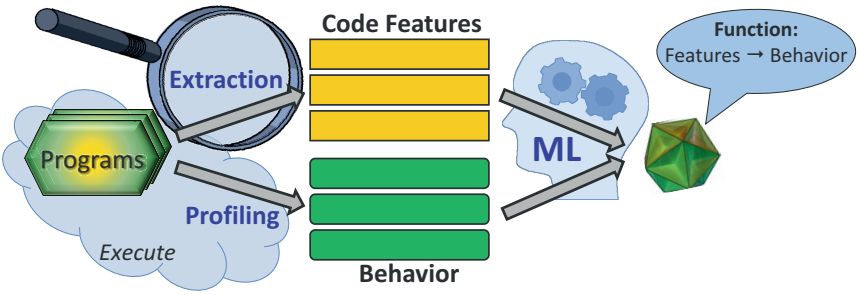


Figure 1: Training phase

The learning algorithm strives for two goals that are discussed in detail below. First, in Subsection 3.1.1, we consider learning loop iteration counts to determine the amount of interprocessor communication. Second, in Subsection 3.1.2, we employ ML techniques to predict precisely the execution time of a process on a particular processing element (*PE*) in order to map processes efficiently to concurrent architectures.

3.1.1 Learning Loop Iteration Counts

To learn statically undeterminable loop iteration counts, we consider static code features, which hold characteristics of the loop body, the loop bounds, and the function that contains the loop. In total, we use 114 code features (see Table 1 for a complete list of features).

For example, we consider code features such as the kind of the loop and the loop exit branches. We distinguish as kind of loops between `do-while` loops and `while-do` or `for` loops because the former ones iterate at least once. We consider the kind of exit branches to differentiate between exits that stem from the loop bound, exits that stem from `break` statements, and exits that arise from `goto` statements. We also consider the number of exit branches out of the loop because the more exit branches a loop contains (e.g., due to `continue` or `break` statements), the greater the likelihood that the loop iteration is actually terminated. If arrays are referenced within a loop body, the loop can be expected to iterate over all elements. We therefore consider the maximum size of the referenced arrays as a static feature.

We also examine characteristics of the structure of loop bounds like, for instance, the number of con- and disjunctions and (un-)equality comparisons between pointers or values. The more conjunctions a loop bound contains at the top level, the greater the likelihood that the loop will iterate less often. Parallel to this, the more disjunctions a loop bound contains at the top level, the greater the probability that the loop will iterate more often. Assuming a typical distribution of data values, the probability of two data values being equal is low. Hence, a comparison of values or pointers as to whether they are equal (or unequal) within a loop bound will probably result in more (or less) iterations. As characteristics of the function that contains the loop, we use the cyclomatic complexity [McC76] and the nesting depth [GS85]. Since the static prediction of execution frequency proposed by Wu and Larus [WL94] is a good estimation of how often a loop will execute, it is used to weight several features. Our experimental results for learning the loop iteration count can be found in [TG10].

3.1.2 Learning Execution Times

We propose learning the execution time of a process based on static code features such as the latency of the most probable path, the weighted sum of latencies of all instructions, the fraction of control instructions, loop-related features and the amount of interprocessor communication. The rationale is the following. The most probable path, which can be determined by static profile analysis [WL94], defines the likeliest runtime behavior. We use the predicted execution probabilities to also weight the sum of the latencies of instructions. Note that this latency is architecture-dependent, but every compiler has to know it

No.	Name	Description
1 – 2	kind-{struc,exit}	kind of loop structure and exit branches
3	cnt-exit	number of exit branches
4 – 9	{min,max}-sz-{arr,rec,un}	min. / max. size of referenced arrays, records, unions
10 – 15	{min,max}-bsz-{arr,rec,un}	ditto when referenced in loop bound
16 – 18	cnt-{arr,rec,un}	number of accesses to elements of arrays, records, unions
19 – 20	nest, max-nest	nesting of current loop, max. nesting of contained inner loops
21 – 23	cnt-{or,and,log}	number of disjunctions, conjunctions, and logical terms in loop bound
24 – 35	cnt-{i,f}-{l,le,g,ge,eq,ne}	number of comparisons of integer / floating points to be less / less or equal / greater / greater or equal / equal / not equal
36 – 47	cnt-{iz,fz}-{l,le,g,ge,eq,ne}	ditto when compared to zero
48 – 59	cnt-{im,fm}-{l,le,g,ge,eq,ne}	ditto when compared to minus one
60 – 71	cnt-{ic,fc}-{l,le,g,ge,eq,ne}	ditto when compared to another constant
72 – 77	cnt-ptr-{l,le,g,ge,eq,ne}	ditto when pointer are compared
78	cnt-ptr-nil	number of comparisons of pointer against `NULL`
79 – 86	cnt-{f}{scanf,printf,getc,putc}	number of calls to these functions
87 – 94	w-{f}{scanf,printf,getc,putc}	ditto weighted by static execution frequency
95 – 97	cnt-{fopen,fclose,fflush}	number of calls to these functions
98 – 100	w-{fopen,fclose,fflush}	ditto weighted by static execution frequency
101 – 103	cnt-{bb,ass,expr}	number of basic blocks, assignments, expressions within assignments
104 – 106	w-cnt-{bb,ass,expr}	ditto weighted by static execution frequency
107 – 109	frac-{call,ctrl,if}	fraction of function calls, statements altering the control flow, `if`-expressions
110 – 112	w-frac-{call,ctrl,if}	ditto weighted by static execution frequency
113	cc	cyclomatic complexity by McCabe
114	nd	nesting depth by Gong and Schmidt

Table 1: Static code features for learning loop iteration counts

for scheduling instructions. We can thus apply our ML-based mapping without the need to annotate latencies for new architectures. That is, we do not have to modify the feature extraction algorithm because it gets the latency information as an input parameter. The fraction of control instructions defines how much of the code will be executed conditionally. Loop-related features, such as the number of loop iterations, the loop nesting and the amount of interprocessor communication are relevant because they contribute most to the execution time. For example, the deeper the loop nesting, the longer the execution time can be expected to be.

Learning loop iteration counts enables us to predict precisely the amount of interprocessor communication, and the learned execution time enables us to map processes power-efficiently to PEs. Both learned information can be used to improve the mapping of MPI processes, as we show in the following section. Since different kinds of programs cannot be expected to behave similarly, we propose using program classification [AG09] to improve the preciseness of the resulting classifiers.

3.2 Compilation

At compile time, we aim at mapping MPI programs to concurrent architectures. Hence, we have to analyze the synchronization between the processes. That is, we analyze the MPI synchronization calls. On encountering such calls, we split each participating process into two tasks, the part of the process until the synchronization and the part of the process after the synchronization. The rationale is that the communication behavior of a process might change during execution. This modelling facilitates to identify points in the program where a reallocation of a process to another PE can be beneficial. To find an optimal mapping, we must take into account the interprocessor communication at runtime and the execution times of processes. To that end, we simply have to extract the code features of each task via highly scalable static analysis. From this, we obtain behavior predictions for execution times of tasks and the communication amount between tasks with our learned statistical model (see Figure 2). Because we also obtain the communication latency between PEs from profiling during the training phase, we have the *communication cost*.

From this, we build a *task graph* with vertices representing tasks and edges representing dependencies as shown in Figure 3 on the left. To reflect synchronization, we insert synchronization points into the graph. The tasks are labeled with the first number being the rank of the corresponding MPI process (i.e., the unique ID which is known by the MPI

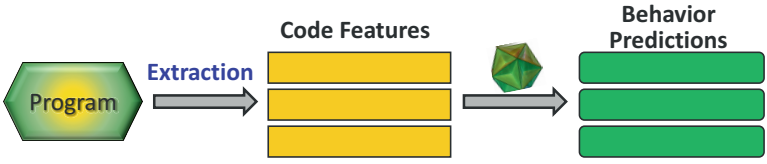


Figure 2: ML-based behavior prediction

runtime daemon) and the second number representing the part of the process. In Figure 3, tasks $T_{0.1}$ and $T_{1.1}$ as well as tasks $T_{1.3}$ and $T_{2.3}$ communicate with each other (indicated by a red arrow), and tasks that have dependencies (illustrated by gray arrows) execute in sequence while the other can execute in parallel. Next, we annotate the task graph with our predictions for the execution times and the communication amount. This information can then be used for communication-aware and power-efficient allocation of tasks to PEs.

The architecture we consider can be described by an undirected, weighted *host graph* $H = (V, E)$. The vertices in V correspond to PEs, edges in E represent the communication medium, and the weight of an edge models the communication cost between PEs. The goal of mapping parallel applications to concurrent architectures therefore can be seen as the problem to find an embedding of the corresponding annotated task graph $TG = (TV, TE)$ into H with respect to minimize the overall execution time and communication cost. To minimize overall the execution time we assign a priority ρ to each task T_i with predicted execution time t_i. We define this priority as

$$\rho(T_i) = t_i + \max_{T_j \in TV, (T_i, T_j) \in TE} \rho(T_j)$$

It can be interpreted as the length of the execution time of the critical path in TG from T_i until the end of the executions of all its descendants. Additionally, we partition the task graph into *regions* such that all tasks within a region can be executed in parallel. That is, there are no dependencies among the tasks. In Figure 3, we have three regions consisting of the tasks that are shown on the same horizontal level. Then we order the parallel executable tasks of each region with an priority-weighted list scheduling algorithm.

The final step is the allocation of tasks to PEs, which we perform per region. We do not constraint the characteristic of the PE. For instance, it can be a vector processor, a scalar processor with multiple cores, or an FPGA. Hence, we have a hierarchical communication structure. For example, cores of a processor can communicate via a fast cache, processors can communicate via shared memory or more expensive communication medium like a network connection. Träff [Trä02] shows that the embedding of the communication graph into the host graph H with a hierarchical communication structure can be solved by *weighted graph partitioning*, optimizing for *totalcut*. Since the problem is NP-complete, using a Kernighan-Lin like linear time heuristics is proposed [Trä06]. The heuristics can give better results than standard algorithms using recursive bipartitioning. We extend this

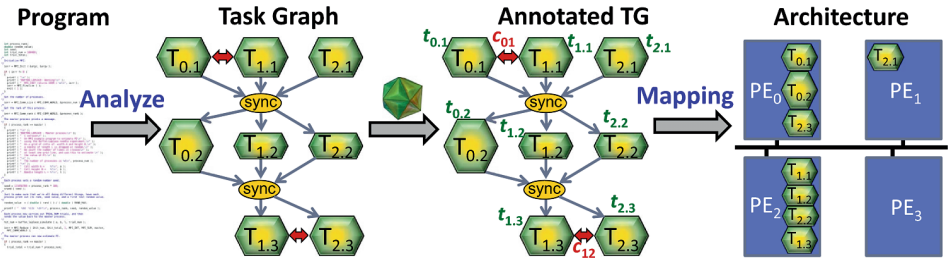


Figure 3: Mapping of MPI processes to concurrent architectures

heuristics by allowing tasks to be mapped to the same PE if the sum of its execution times do not exceed the execution time of other tasks in the same region. The resulting mapping of processes to PEs is automatically given to the MPI runtime daemon.

Consider the mapping of the annotated task graph to the architecture shown in Figure 3 on the right. The concurrent architecture is composed of PEs connected by a bus which results in different communication latencies. The latency between PE_0 and PE_2 is as low as the latency between PE_1 and PE_3, whereas all other latencies are much higher. At release time of the processes, the MPI runtime daemon without having predictions for the communication amount and the execution time typically performs workload balancing. This could result for example in allocating the process with rank 0 (i.e., tasks $T_{0.1}$ and $T_{0.2}$) to PE_0, the process with rank 1 (i.e., tasks $T_{1.1}$ to $T_{1.3}$) to PE_1, and the process with rank 2 (i.e., tasks $T_{2.1}$ to $T_{2.3}$) to PE_2.

Our ML-based predictions make possible to direct the MPI runtime daemon allocating $T_{1.1}$ to PE_2 (instead of to PE_1), which results in lower communication overhead. At each synchronization point, we analyze whether another allocation is beneficial and if so, we inject calls to signal the MPI runtime daemon that reallocation is necessary. Therefore, if the execution times of tasks $T_{1.2}$ and $T_{2.2}$ together are predicted to be no longer than the execution time of task $T_{0.2}$, the MPI runtime daemon will be directed to allocate both to the same PE. At this time, we can put PE_2 and PE_3 in a low power state. Assume that the last part of the processes with rank 1 and 2 have similar predicted execution times. At the second synchronization point the MPI runtime daemon has the information that both processes shall be allocated close to each other but not on the same PE because of their similar execution times and therefore reallocates task $T_{2.3}$ to PE_0. Thus, our approach is communication-aware and power-efficient.

4 Implementation

We have implemented the machine learning algorithm within the *R Project* [Dev08], together with the *rpart* package for the classification tree algorithm. *R Project* is a collection of statistical functions, which we have already successfully used in previous work [TG10]. For program classification [AG09], we use the *hclust* function for hierarchical clustering. We implement the compilation steps within the modular, industrial strength *Compiler Development System* (*CoSy*) of ACE [ACE11]. The steps during compilation like e.g., analyses, optimizations, and code generation are implemented in modules called *engines*. We have implemented the extraction of code features for the learning of the loop iteration counts and the static branch prediction algorithm proposed by Wu and Larus [WL94] for determining the execution probabilities and frequencies. To determine the actual loop iteration count for profiling during the training phase, we have used the *path profiling engine* of CoSy.

5 Related Work

The application of ML techniques in compiler frameworks has become a challenging research area. The key advantage of learning techniques is their ability to find relevant information in a high-dimensional space, thus helping us to understand and control a complex system. Heuristics automatically generated via learning algorithms often outperform hand-crafted models (cf. [MBQ02]). As an example application, this approach learns to decide if Loop Unrolling is beneficial or not. The approach of Fursin et al. [FMT+08] targets the selection of good optimization passes via *machine learning*. During the training phase, information about the program structure is gathered using 55 static program features and the behavior of programs when compiled under different optimization settings is recorded. After training, on encountering a new program the optimizations to be applied are selected.

Close to our approach for predicting execution times using ML techniques is the approach of Powell and Franke [PF09] for predicting cycle counts to speed up instruction set simulation. However, their method allows the training data to be updated, online feature selection, and switching between cycle- and instruction-accurate simulation to ensure the accuracy of the predictions being made. To the best of our knowledge, no approach targets task allocation in compilers using ML techniques.

The mapping and scheduling problems on multiprocessor systems are often modeled as *integer linear programming* (*ILP*) problems ([RGB+06, YH08]). The major drawback of ILP is its limitation to small problems due to its computational complexity. Heuristic approaches such as *genetic* or *evolutionary algorithms* are therefore being increasingly developed (e.g., [GTT09, PLL07, Yoo09, YH09]). Their main disadvantage is the extended compile time due to the iterative approach of finding good solutions. Common to all these approaches is the requirement of having information about communication overhead or execution time of the tasks. Thus, our technique supports these approaches by automatically generating predictions for the claimed information.

Several new approaches target communication- and power-aware task scheduling in compilers for concurrent architectures (though without using ML techniques). For example, the approach of Varatkar and Marculescu [VM03] takes an abstract application task graph and the architecture graph as inputs, assigns the tasks to processors and then schedules them. Nodes in the abstract task graph are annotated with deadlines and the number of cycles taken by that task for computation. Arrows, which indicate a control dependence between two tasks, are annotated with a number representing the quantity of communication traffic between the tasks. Since the amount of communication between the tasks and the processor cycles needed for each task has to be annotated, our technique can be adapted to this approach, thus eliminating the need for these annotations. However, voltage selection is modeled as an ILP problem, which suffers from computational complexity.

A task and data migration scheme that decides whether to migrate tasks or data to satisfy a given communication requirement is proposed by Ozturk et al. [OKSK06]. The choice is made at runtime based on statistics collected off-line through profiling. The major drawback of one profiling run is the strong dependence on the input data set of that run. That

is, selecting input sets that generate representative profiles is a difficult task. However, the authors' experience with several applications indicates that in some cases migrating the task itself – instead of data – can be more beneficial from both performance and power perspectives. Based on this observation, it is clear that task allocation and minimizing energy consumption should be jointly addressed during compilation.

To overcome the limitations of ILP due to computational complexity and the drastic over-approximation of static analyses due to unknown runtime behavior, we use supervised classification learning. This automatically generates classifiers, which relate static code features to the dynamic behavior, thus providing the compiler with intelligence. At compile time, these classifiers can be used as precise and fast heuristics for communication-aware task mapping and power minimization, which eliminates the need for manual annotations of forecast values.

6 Conclusions

Using ML techniques in compilers is a challenging research area. In this paper, we have presented a novel, highly scalable ML-based approach for predicting runtime communication overhead and execution time targeting the mapping of MPI processes to concurrent architectures. Our proposed learning algorithm automatically relates static code features to dynamic runtime behavior. This allows us to generate much more precise heuristics than those generated by static analyses, as our experimental results in [TG10] demonstrate.

Since the learning algorithm must be deployed only once per architecture decoupled from the compilation, the compile time itself is not increased, in contrast to other heuristic approaches such as *genetic* and *evolutionary algorithms* or profile-guided compilation. The latter approach is as well strongly dependent on the input data set, hence being too precise. Instead, we perform several profiling runs with different input data during the training phase. Thus, we focus on realistic behavior and not on one single behavior. Furthermore, our technique is applicable to all other approaches where user annotations or profiling are necessary, thus eliminating their requirements. Additionally, our solution is fully retargetable and is not limited to static mapping at compile time. Instead, the established heuristics can be used to make the MPI runtime scheduler more intelligent concerning the runtime behavior of the processes. With our predictions for the execution times and the communication amount, we can direct the MPI runtime scheduler to reallocate processes communication-aware and power-efficient instead of simply performing workload balancing. This is a great improvement to current MPI implementations because even the best MPI implementation cannot ensure that for instance frequently communicating processes are placed close to each other.

In the future, we will apply our approach for learning execution times. Because communication between processes can also arise within recursive functions, we plan to learn the recursion depth of these functions, thus giving us an estimate for the communication overhead in these cases. Finally, we will use all results to guide the mapping of parallel applications to concurrent architectures.

References

[ACE11] ACE. *Associated Compiler Experts bv., Amsterdam, The Netherlands*, 2011.

[AG09] Lars Alvincz and Sabine Glesner. Breaking the Curse of Static Analyses: Making Compiler Intelligent via Machine Learning. In *3rd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'09)*, January 2009.

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning, Second Edition*. Adaptive Computation and Machine Learning. The MIT Press, 2nd edition, 2010.

[Dev08] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[FMT+08] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, François Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. MILEPOST GCC: Machine Learning Based Research Compiler. In *Proceedings of the GCC Developers' Summit*, pages 7–19, June 2008.

[GLM04] Robert B. Gramacy, Herbert K. H. Lee, and William G. Macready. Parameter Space Exploration with Gaussian Process Trees. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, pages 45–53, New York, NY, USA, 2004. ACM.

[GS85] Huisheng Gong and Monika Schmidt. A complexity measure based on selection and nesting. *SIGMETRICS Perform. Eval. Rev.*, 13:14–19, June 1985.

[GTT09] C. Goh, E. Teoh, and K. Tan. A hybrid evolutionary approach for heterogeneous multiprocessor scheduling. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13:833–846, 2009.

[Ind04] Piotr Indyk. Nearest Neighbors In High-Dimensional Spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39, pages 877–892. CRC Press LLC, Boca Raton, FL, 2nd edition, April 2004.

[MBQ02] Antoine Monsifrot, François Bodin, and René Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proceedings of AIMSA'02*, pages 41–50, London, UK, 2002. Springer-Verlag.

[McC76] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[Mes09] Message Passing Interface Forum. *MPI: a Message-passing Interface Standard: Version 2.2*. University of Tennessee, 2009.

[OKSK06] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy. Selective code/data migration for reducing communication energy in embedded MpSoC architectures. In *Proceedings of GLSVLSI'06*, pages 386–391, New York, NY, USA, 2006. ACM.

[PF09] Daniel Christopher Powell and Björn Franke. Using Continuous Statistical Machine Learning to Enable High-Speed Performance Prediction in Hybrid Instruction-/Cycle-Accurate Instruction Set Simulators. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 315–324, New York, NY, USA, 2009. ACM.

[PLL07] Saeed Parsa, Shahriar Lotfi, and Naser Lotfi. An Evolutionary Approach to Task Graph Scheduling. In *International Conference on Adaptive and Natural Computing Algorithms (ICANNGA 1)*, pages 110–119, 2007.

[RGB+06] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Francesco Poletti, and Michela Milano. Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip. In *Proceedings of DATE'06*, pages 3–8, 3001 Leuven, Belgium, 2006. European Design and Automation Association.

[Smi00] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (Keynote Talk). In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, DYNAMO '00, pages 1–11, New York, NY, USA, 2000. ACM.

[Ste09] Dan Steinberg. CART – Classification and Regression Trees. In Xindong Wu and Vipin Kumar, editors, *The Top Ten Algorithms in Data Mining*, Chapman & Hall/CRC data mining and knowledge discovery series, chapter 10, pages 179–201. CRC Press, 2009.

[TG10] Dirk Tetzlaff and Sabine Glesner. Intelligent Task Mapping using Machine Learning. In *CiSE '10: Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4. IEEE Computer Society, dec. 2010.

[Trä02] Jesper Larsson Träff. Implementing the MPI Process Topology Mechanism. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Supercomputing '02, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[Trä06] Jesper Larsson Träff. Direct graph *k*-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

[Tur50] Alan M. Turing. COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 1950.

[VM03] Girish Varatkar and Radu Marculescu. Communication-Aware Task Scheduling and Voltage Selection for Total Systems Energy Minimization. In *Proceedings of ICCAD'03*, pages 510–517, Washington, DC, USA, 2003. IEEE Computer Society.

[Wan03] John Wang, editor. *Data Mining: Opportunities and Challenges*. IGI Publishing, Hershey, PA, USA, 2003.

[WL94] Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. In *Proceedings of MICRO 27*, pages 1–11, New York, NY, USA, 1994. ACM Press.

[YH08] Hoeseok Yang and Soonhoi Ha. ILP Based Data Parallel Multi-Task Mapping/Scheduling Technique for MPSoC. In *Proceedings of ISOCC'08*, volume 01, pages I–134–I–137, Nov. 2008.

[YH09] H. Yang and S. Ha. Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC. In *Proceedings of DATE'09*, pages 69–74, 2009.

[Yoo09] Myungryun Yoo. Real-time task scheduling by multiobjective genetic algorithm. *Journ. of System a. Software*, 82(4):619–628, 2009.

Fehlalarmfreie Abstraktion in ISO-C konformer Semantik

Dirk Richter, Roberto Hoffmann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
D-06099 Halle (Saale)
richterd@informatik.uni-halle.de
hoffmaro@informatik.uni-halle.de

Abstract: Viele Aussagen zum Programmverhalten sind in turingmächtigen Programmiersprachen unentscheidbar (Rice Theorem). Mittels Abstraktion vom Programmverhalten können nicht-turingmächtige Modelle automatisch erzeugt werden. Modelle in Form von symbolischen Kellersystemen (SPDS) erlauben eine so präzise Darstellung des Programmverhaltens, sodass eine ISO-C konforme Semantik möglich ist [RB12, KZR09]. Allerdings führen präzisere Darstellungen stets auch zu komplexeren und umfangreicheren Modellen. Dies erschwert Software-Modellprüfung, modellbasiertes Testen und Testdaten- und Codegenerierung. Mehr Abstraktion führt hingegen zu kleineren Modellen, allerdings auch zu *mehr* Fehlalarmen. Ziel dieser Arbeit ist es, bezüglich temporaler Aussagen unwichtige Teile eines SPDS zu identifizieren und von diesen Teilen so zu abstrahieren, dass *keine* Fehlalarme entstehen.

1 Einleitung

Im Rahmen von agilen Methoden (z.B. Extreme Programming) wird das Verhalten eines Softwaresystems durch viele Nebenbedingungen (Unit Tests, temporale Formeln oder JML) spezifiziert. Diese sind regelmäßig auf Einhaltung (Modellprüfung/Testen) zu überprüfen. Viele Modellprüfer beschränken dabei die Rekursionstiefe oder verbieten Methodenaufrufe. Durch diese Unter- bzw. Überapproximation von Methodenaufrufen entstehen Fehlalarme (False Negatives sowie Fehlabstraktionen), die durch korrekte Abbildung von Methodenaufrufen und Rekursion auf SPDS vermieden werden können. Im Gegensatz zu vergleichbaren Arbeiten zur Abstraktion von Modellen durch finite state Modellprüfer wie BLAST, SPIN, NuSMV/SMV, JavaPathFinder, F-Soft oder Bogor (Bandera Projekt) betrachten wir in dieser Arbeit die Modellabstraktion *unendlicher* symbolischer Modelle in Form von symbolischen Kellersystemen (SPDS). Eine Beschränkung auf eine maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung, welche z.B. durch Inlining verursacht wird. SPDS können mittels JMoped [SSE05] aus Java gewonnen und mittels des Modellprüfers Moped [ES01, Sch02, SSE05] überprüft werden und ermöglichen eine interprozedurale und kontextsensitive Weiterverarbeitung. Unter Verwendung des Cross-Compilers Grasshopper kann nicht nur Java 1.6 Code verwendet werden, sondern auch Microsoft Intermediate

Language. Es ist auch möglich, die Gültigkeit von Java Modeling Language (JML) Annotationen zu überprüfen, wenngleich dies in der Praxis derzeit noch unhandlich ist.

Mit SPDS kann eine ISO-C konforme Semantik definiert werden [RB12, KZR09], so dass C-Programme (z.B. für eingebettete Systeme) direkt als Modell genutzt werden können. Abstraktionen werden (u.a. wegen nicht ausreichender Vereinfachung) oft *manuell* erzeugt [NK00], was jedoch fehlerträchtig ist. Ziel dieser Arbeit ist die *automatische* Abstraktion für SPDS. Dazu wird die Wichtigkeit von Teilen der Modellbeschreibung mit einer Heuristik bewertet, welche einerseits Programmanalysen auf der Modellbeschreibung und andererseits gegebene temporale Formeln (die Spezifikation) nutzt. So ist es möglich, unwichtige Teile der Modellbeschreibung zu erkennen und davon zu abstrahieren.

2 Grundlagen

2.1 Symbolische Kellersysteme

Die eingesetzten Modelle beschreiben sogenannte Kripkestrukturen. $M = (S, \rightarrow, L_A)$ heißt *Kripkestruktur*, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. Zur Beschreibung von (unendlich) großen Kripkestrukturen können Kellersysteme (Pushdown Systems) verwendet werden. $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt *Kellersystem*, falls P eine Menge von Zuständen, Γ eine endliche Menge (Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen ist. Informell ist ein Kellersystem ein Kellerautomat ohne Eingabe. Im Gegensatz zum Testen kann mittels Software-Modellprüfung die Abwesenheit von Fehlern in Kellersystemen formal nachgewiesen werden [Sch02, ES01, Ber06, EKS02, EHRS00, Wal00, BEM97]. (p, w) heißt *Konfiguration*, falls $p \in P$ und $w \in \Gamma^*$. (p, a) heißt *Kopf* der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Die Anzahl der Köpfe nutzen wir, um potentiell unendlich große Zustandsräume miteinander zu vergleichen. Die Menge aller möglichen Konfigurationen sei mit $konf(\mathcal{P}) := \{(p, w) \mid p \in P, w \in \Gamma^*\}$ bezeichnet. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Bei einem *Symbolischen Kellersystem* (SPDS) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben, was die Angabe des Kellersystems vereinfacht [Sch02]. Ein SPDS ist ein Tupel $S = (globs, prc)$ mit einer endlichen Menge globaler Variablen $globs$ und einer endlichen Menge an Prozeduren prc, wobei es eine ausgezeichnete Prozedur $main \in prc$ gibt. Eine Prozedur $q \in prc$ ist ein Tupel $q = (name, pars_q, loc_q, stats_q)$ mit eindeutigem Namen $name$, Parametervariablen $pars_q$, lokalen Variablen loc_q (mit $pars_q \subseteq loc_q$) und einer Liste an Anweisungen $stats_q$ (Rumpf). Jede Anweisung besitzt eine eindeutige Marke $l \in labels$. Die aktuelle Prozedur bzw. Marke einer Konfiguration s wird mit $prc(s)$ bzw. $s)$ bezeichnet. Belegungen lokaler Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (z.B. Heap) werden mit Hilfe der Zustände P des Kellersystems beschrieben. Ziel dieser Arbeit ist es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits vorher durch Abstraktion zu verringern.

Seien $Vars_q := globs \cup loc_q$ die Variablen im Gültigkeitsbereich einer Prozedur q und $Expr_{Vars_q}$ arithmetische Ausdrücke (Operatoren alle strikt, Assoziativitäten und Prioritäten wie in ISO-C) über diesen Variablen sowie $env^q : Vars_q \to \mathbb{Z}$ eine Variablenbelegung. Eine Variablenbelegung $env(s) \in ENV$ wird aus dem Kopf einer Konfiguration $s \in konf(S)$ erzeugt (q ist dann durch s bestimmt). Zur Veränderung eines Variablenwertes einer Variablen v an einer Konfiguration s auf den Wert c wird die Schreibweise $s[v \mapsto c]$ verwendet (dies ist ebenfalls eine Konfiguration). Sei weiter $[\![e]\!]_{env^q}$ die Auswertung eines Ausdrucks $e \in Expr_{Vars_q}$ mittels der Variablenbelegung env^q. Dann sind die SPDS-Anweisungen wie folgt definiert:

- **x = e;** mit $x \in Vars_q$ und $e \in Expr_{Vars_q}$ eine Zuweisung des Werts $[\![e]\!]_{env^q}$ an x.
- **p(e$_1$, e$_2$, ..., e$_n$);** mit e$_i \in Expr_{Vars_q}$ ein Aufruf der Prozedur p (Call-By-Value).
- **return e;** mit e $\in Expr_{Vars_q}$ ein Prozedurende mit Rückgabewert $[\![e]\!]_{env^q}$.
- **goto L;** ein unbedingter Sprung an die Marke L.
- **if (e) s;** eine bedingte Anweisung, welche s ausführt, falls $[\![e]\!]_{env^q} = 1$.

Zudem gibt es eine nichtdeterministische Funktion `rand`(e), welche fair einen zufälligen Wert c mit $0 \leq c \leq [\![e]\!]$ liefert[1]. Kommentare gelten bis zum Zeilenende und werden mit dem Symbol # eingeleitet. Weitere ISO-C Anweisungen lassen sich auf diese SPDS-Anweisungen zurückführen [RB12]. SPDS können mit Hilfe der Modellsprache Remopla [KSS06] beschrieben werden. Für Details zur Konstruktion von SPDS-Modellen aus C- und Java-Programmen sei auf [RB12, ES01, Obd02, RZ07] verwiesen.

Ein SPDS B *simuliert* ein SPDS A (in Zeichen $B \preceq A$), falls es eine Funktion $\Theta : konf(A) \to konf(B)$ gibt, so dass jeder Lauf $a1 \to a2 \to$ ... in A zu einem Lauf $\Theta(a1) \to \Theta(a2) \to$ ... in B wird. Zwei SPDS A und B heißen *bisimular* (in Zeichen $A \simeq B$), falls $A \preceq B$ und $B \preceq A$.

Beispiel 1 *(Beispiel eines SPDS)*
Abbildung 1 zeigt im linken Teil ein Beispiel eines SPDS. Die Prozedur `catch` *dient der Verarbeitung von Events. In der Schleife, welche mit* `NO_EVENT` *markiert ist, wartet* `catch`*, bis ein Event (mittels* `pick_event`*) zum Verarbeiten eintritt. Die Konstante* 0 *signalisiert dabei, dass kein Event aufgetreten ist. Die Konstante* 1 *hingegen signalisiert, dass ein Timeout stattgefunden hat, welcher gezählt wird, und ein Neuversuch (*`retry` $=$ `1`*) eingeleitet wird. In allen anderen Fällen handelt es sich um einen gültigen Event, welcher in der Reihung* `history` *als Verlauf (bis zu 100 Einträge) gespeichert wird. Ist ein Event eingetreten, welcher verschieden ist vom Parameter* e*, so wird weiter gewartet, bis der Event* e *eintritt. Erst wenn* e *eingetreten ist, dann wird rekursiv mit dem nächst kleineren Event fortgesetzt (Marke* `hit`*), wenn möglich. Der rechte Teil von Abbildung 1 stellt die zugehörige Abstraktion dar und wird später erläutert. Variablen haben darin einen deutlich kleineren Typ und deren abstrakte Variablenwerte stellen Äquivalenzklassen für viele mögliche konkrete Variablenwerte dar.*

[1] Die ISO-C Funnktion `rand`() kann als SPDS-Funktion `rand(RAND_MAX)` aufgefasst werden.

Abbildung 1: 2 SPDS-Beispiele (links: aus C-Code generiert, rechts: abstrahiert)

```
int offset(8);
int history(8)[100];
int retry(8);
int timeouts(8);

void catch(int e(8)) {
     int word(8);
     int event(8);
     event=0;
     word=1;
     retry=0;  # counts retries
pick:event = pick_event();
    if (event == 0) {   # NO_EVENT
        word=0;
        goto pick;  }
    word=1;
    if (event == 1) {   # timeout
        timeouts=timeouts+1;
        retry=1;    # enter retry mode
        goto pick; }
save:history[offset]=event;
     offset=offset+1;
     if (offset == 100) offset=0;
     if (event != e) goto pick;
hit: if (e > 2) catch(e-1);
     return; }
```

```
int offset(8)
int history(0)[100];
int retry(0);
int timeouts(0);

void catch(int e(4)) {
     int word(0);
     int event(4);
     event=0;
     word=0;
     retry=0;
pick:event = pick_event();
    if (event==0){
        word=0;
        goto pick;  }
    word=0;
    if (event == 1) {
        timeouts=0;
        retry=0;
        goto pick; }
save:history[offset]=0;
     offset=offset+1;
     if (offset == 100) offset=0;
     if (event != e) goto pick;
hit: if (e > 2) catch(e-1);
return; }
```

2.2 Syntax und Semantik temporaler Formeln

Die Logik CTL* besteht aus einer Menge von Zustandsformeln Z, welche Aussagen über einen Zustand eines Modells trifft. Innerhalb einer CTL* Formel selbst sind auch Pfadformeln (Menge P) erlaubt, welche wie folgt definiert sind. Die Quantoren A bzw. E werden verwendet, um Aussagen über *alle Pfade* bzw. *min. einen Pfad* zu treffen. Die Operatoren X, U, F sowie G repräsentieren die Operatoren *Nächster* (Next), *strenges Bis* (strong until), *Irgendwann* (future/eventually) bzw. *Immer* (always). Dabei sind E,F,G bzw. $\vee$ abkürzende Schreibweisen für $E\phi = \neg A \neg \phi, F\phi = true\ U\ \phi, G\phi = \neg F \neg \phi, \phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, wobei $\phi, \psi \in P$ sind. Sei $AExpr = Expr \cup labels \cup prc$ die Menge an möglichen Ausdrücken, Marken- und Prozedur-Namen. Induktiv sind dann CTL* Formeln wie folgt definiert:

- $AExpr \subset Z$
- $\forall \phi, \psi \in Z : \neg\phi \in Z,\ \phi \wedge \psi \in Z$
- $Z \subseteq P$
- $\forall \phi, \psi \in P : A\phi \in Z, \neg\phi \in P,\ \phi \wedge \psi \in P, X\phi \in P,\ \phi U \psi \in P$

Gilt eine Formel ϕ an einem Zustand s einer Kripkestruktur $M = (S, \rightarrow, L)$ bzw. an einer Konfiguration s eines SPDS $M = (R, \Gamma, \hookrightarrow)$, so schreiben wir dafür $s \models_M \phi$ (bzw. $s \models \phi$, wenn M aus dem Kontext klar ist). Sei $p = s_0 \rightarrow s_1 \rightarrow \ldots$ ein unendlicher Lauf, wobei $fst(p) = s_0$ der Anfangszustand bzw. die Anfangskonfiguration des Laufs p ist und $p_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$ der Suffix des Laufs beginnend bei s_i. Terminiert der Lauf p an einem Zustand bzw. einer Konfiguration s_n, so ist er endlich. Bei endlichen Läufen wird der letzte Zustand bzw. die letzte Konfiguration immer wieder wiederholt, so dass $p = \ldots \rightarrow s_{n-1} \rightarrow s_n \rightarrow s_n \rightarrow s_n \ldots$. Dann sei $\models$ neben Zuständen bzw. Konfigurationen s auch für Läufe p wie folgt erklärt:

$$
\begin{aligned}
s \models a &\quad :\Leftrightarrow \quad [\![a]\!]_{env^s} = 1 \vee a = s) \vee a = prc(s) \\
s \models \neg\phi &\quad :\Leftrightarrow \quad s \not\models \phi \\
s \models \phi \wedge \psi &\quad :\Leftrightarrow \quad s \models \phi \ \wedge \ s \models \psi \\
s \models A\phi &\quad :\Leftrightarrow \quad (\forall p = s \rightarrow s_1 \rightarrow \ldots : (p \models \phi)) \\
p \models \phi, \ mit\ \phi \in Z &\quad :\Leftrightarrow \quad fst(p) \models \phi \\
p \models \neg\phi, \ mit\ \phi \in Z &\quad :\Leftrightarrow \quad p \not\models \phi \\
p \models \phi \wedge \psi &\quad :\Leftrightarrow \quad p \models \phi \ \wedge \ p \models \psi \\
p \models \phi U \psi &\quad :\Leftrightarrow \quad \exists i \geq 0 : ((p_i \models \psi) \wedge (\forall j < i : p_j \models \phi)) \\
p \models X\phi &\quad :\Leftrightarrow \quad p_1 \models \phi
\end{aligned}
$$

Im Folgenden wird auch $K \models \phi$ für eine Menge an Konfigurationen K verwendet, falls $\forall s \in K : s \models \phi$.

Zum Beispiel 1 seien die temporalen Formeln aus Abbildung 2 gegeben. Sie beschreiben Eigenschaften des SPDS, die zu überprüfen sind. Hierbei besagt z.B. die Formel $G(0 \leq \texttt{offset} \leq 100)$, dass sich Variable `offset` stets größer oder gleich ist als 0 aber auch kleiner oder gleich 100.

Abbildung 2: Beispielhafte temporale Formeln zu Beispiel 1

Beschreibung	**temporale Formel**
Programm terminiert	$G(F\ \texttt{end})$
array out of bounds	$G(0 \leq \texttt{offset} \leq 100)$
nur gültige events	$G(0 \leq \texttt{event} < 16)$
nur gültige events	$G(2 \leq \texttt{e} < 16)$
jeder event wird geloggt	$G((\texttt{event} > 1) \Rightarrow !\texttt{pick}\ U\ \texttt{save})$
richtiger event wird behandelt	$G(\texttt{event} == \texttt{e} \Rightarrow !\texttt{pick}\ U\ \texttt{hit})$
Wiederholung bei timeout/nix	$G(\texttt{event} < 2 \Rightarrow !\texttt{save}\ U\ \texttt{pick})$
nur gültige events geloggt	$G(\texttt{save} \Rightarrow \texttt{event} > 1)$

2.3 SAT-Solving und SAT-Heuristiken

Der weit verbreitete Modelprüfer NuSMV bietet die Möglichkeit, temporale Formeln auf einem Modell nicht nur per BDD checking, sondern auch via SAT-Solving als Bounded Model Checking zu prüfen [CCG+02]. Dabei werden das Modell und die temporalen Formeln als SAT-Problem formuliert und einem SAT-Solver zur Entscheidung der Gültigkeit der Formeln auf dem gewählten Modell übergeben. In SAT-Solvern sind Heuristiken verfügbar, welche den eigentlich exponentiellen Aufwand für das Erfüllbarkeitsproblem in vielen praktischen Fällen deutlich zu reduzieren vermögen. Dies geschieht durch geeignete Variablenwahl und -belegung beim dabei angewandten Backtracking-Verfahren (Davis-Putnam-Logemann-Loveland-Algorithmus) [Hof05, Til05].

Ausgehend von dieser rein logisch-booleschen Formulierung des Modells und der temporalen Formeln in Konjunktiver Normalform (CNF) kann man weitere wichtige Informationen extrahieren: Man benutzt die Heuristiken des SAT-Solvers dazu, relevante Modellteile bezüglich gegebener temporaler Formeln zu bestimmen [MA03], analog zu Überdeckungsmaßen [HM09].

3 Abstraktionsprozess

Durch Abstraktion kann der Zustandsraum verkleinert werden. Beim Abstrahieren von einem Konfigurationenübergang (Eliminieren überflüssiger SPDS-Anweisungen) reduziert sich die Anzahl der Köpfe um $\frac{1}{n}$, wobei n die Anzahl der SPDS-Anweisungen darstellt. Andererseits führt das Abstrahieren von einer globalen 32-Bit Variable zu einer Reduktion um den Faktor 2^{32}. Wie Beispiel 1 zeigt, wird der SPDS-Konfigurationenraum bei Generierung aus höheren Programmiersprachen oft durch große Variablentypen bestimmt. Daher gelingt für solche SPDS eine Konfigurationenraumreduktion am stärksten durch Abstraktion von großen Variablentypen zu kleinen. Daher konzentrieren wir uns in dieser Arbeit auf die Abstraktion von Variablen.

3.1 Bestimmung Wichtigkeit γ von Variablenbits

Man kann mithilfe der Heuristiken eines SAT-Solvers für ein gegebenes Modell und dessen temporale Formeln unter Ausnutzung der logischen Zusammenhänge die "Wichtigkeit" einzelner Variablenbits ermitteln. Dazu seien $Vars^*$ die Variablenbits der Variablen $Vars$ des SPDS S. Dann werden als erstes das Modell und die temporalen Formeln ins NuSMV-Format überführt. Dabei wird via Def-Use-Analysen [Muc97] (als Programmanalysen bei höheren Programmiersprachen gut bekannt) der Datenfluss nachgebildet, um diesen unabhängig vom Kontrollfluss zu analysieren. In Abbildung 3 wurde Beispiel 1 als NuSMV-Modell umgesetzt, wobei wir aus Effizienzgründen die konkrete NuSMV-Datei symbolisch mittels Parameter beschreiben (Parameter p). Die Variablen wurden mit einem Präfix (`DR_`) versehen, damit eine Kollision mit reservierten Schlüsselworten vermieden

Abbildung 3: NuSMV-Sourcecode des Beispiels

```
PARAM p: 0..100;
VAR DR_retry: 0..255; DR_offset: 0..255; DR_history: array 0..100
  of 0..255; DR_event: 0..255; DR_timeouts: 0..255;
  DR_word: 0..255; DR_e: 0..255;
ASSIGN init(DR_e):= 15; init(DR_retry):= 0; init(DR_word):= 1;
  init(DR_offset):= 0;init(DR_history[p]):=0;init(DR_timeouts):=0;
  init(DR_i):=15;init(DR_event):={0..15};

next(DR_event)       := { 0 .. 15 };
next(DR_retry)       := case DR_event = 1: 1; TRUE: DR_retry; esac;
next(DR_word)        := case DR_event = 0: 0; TRUE: 1; esac;
next(DR_timeouts)    := case DR_event=1: DR_timeouts+1 mod 255;
                             TRUE: DR_timeouts; esac;
next(DR_history[p])  := case DR_offset = p & DR_event > 1: DR_event;
                             TRUE: DR_history[p]; esac;
next(DR_e)           := case DR_e > 2 & DR_event = DR_e: DR_e - 1;
                             DR_e <= 2 & DR_i > 2:DR_i; TRUE:DR_e; esac;
next(DR_offset)      := case DR_event>1 & DR_offset<100: DR_offset+1;
                        DR_event>1&DR_offset=100:0; TRUE:DR_offset; esac;
```

wird. Der in [HM09] vorgestellte modifizierte NuSMV wird dann dazu benutzt, einerseits das Modell und andererseits die temporalen Formeln in konjunktive Normalformrepräsentation zu überführen (CNF_S und CNF_ϕ), welche der charakteristischen Funktion der dem Modell bzw. der den temporalen Formeln entsprechenden Kripkestruktur entspricht. Aufgrund der Endlichkeit dieser Darstellung sind nur k Zeitschritte (frei wählbar) der auftretenden Systemübergänge enthalten. Wir haben für unser Beispiel $k = 5$ gewählt. Ausgehend von diesen Repräsentationen wird die "Wichtigkeit" der einzelnen Bits anhand deren logischer Komplexität und Vernetzung bestimmt. Dies geschieht analog wie beim SAT-Solving. Für das positive und das negative Literal einer Variablen berechnet die Heuristik anhand der in Form einer CNF-Datei vorliegenden Problemstruktur einen Zahlenwert als Maß zur Wichtigkeit $\gamma_S(v.i) \in \mathbb{R}$ (Modell CNF_S) bzw. $\gamma_\phi(v.i) \in \mathbb{R}$ (Formel CNF_ϕ) für alle Variablenbits $v.i \in Vars^*(S)$. Normalerweise wird dann im Laufe des SAT-Solving diejenige Variable zuerst belegt, welche über den höchsten Wert verfügt. An dieser Stelle bricht der modifizierte SAT-Solver ab und gibt stattdessen die ermittelten Werte für jedes Literal aus.

Die CNF-Darstellung des Modells beziehungsweise der temporalen Formeln enthält alle Bits aller Variablen für alle Zeitschritte $k = 0..5$. Hieraus wird schließlich das Maß zur Wichtigkeit $\gamma(v.i) \in \mathbb{R}$ für alle Variablenbits $v.i \in Vars^*$ bestimmt. Dabei werden Variablen entlang der Zeitachse sowie positive und negative Literale zusammengefasst. Aufgrund bisheriger Erfahrungen verwenden wir dafür statt Summenbildung das Maximum, um selten vorkommende, aber dafür wichtige Variablen angemessen zu berücksichtigen. Die Wichtigkeiten aus Modell und Formeln werden normiert (Skalierung auf 50%) und addiert. So sind Variablenbits, welche in beiden Teilen von Bedeutung sind, insgesamt umso wichtiger. Für Beispiel 1 ergeben sich die Wichtigkeiten aus Abbildung 4. Von den Variablen sind `DR_event0..3, DR_e0..3` und `DR_offset` am wichtigsten

($\gamma > 0$). Eher unwichtig sind hingegen `DR_word*`, `DR_event4..7`, `DR_e4..7`, `DR_history*`, `DR_retry*` und `DR_timeouts*`, weil sie unbedeutend im Modell und unnötig für die temporalen Formeln sind ($\gamma \approx 0$). Die ermittelte Wichtigkeit wird dann genutzt, um den Abstraktionsprozess zu leiten. Wichtige Teile werden beibehalten, unwichtige abstrahiert. Die Grenze kann dabei beliebig variiert werden, um sie dem gewünschten Abstraktionsgrad anzupassen.

Abbildung 4: Ermittelte Wichtigkeit der Variablenbits (Auszug), k=5

Variablenbit	γ_S	γ_ϕ	$\gamma \in [0,1]$
`DR_event.0`	1687.29	18.00	0.54
`DR_event.1`	1750.69	16.80	0.54
`DR_event.2`	1748.82	11.80	0.48
`DR_event.3`	1735.15	4.80	0.40
`DR_event.4`	0.00	0.00	0.00
...			
`DR_offset.0`	2434.98	43.85	0.98
`DR_offset.1`	2529.55	20.32	0.73
`DR_offset.2`	2161.85	18.98	0.65
`DR_offset.3`	1760.80	5.00	0.41
`DR_offset.4`	1558.04	5.00	0.37
`DR_offset.5`	1547.79	5.00	0.37
`DR_offset.6`	2201.81	5.00	0.49
`DR_offset.7`	2011.01	5.00	0.46
`DR_word.0`	113.62	0.00	0.03
`DR_word.1`	0.00	0.00	0.00
...			
`DR_history[0].0`	691.09	8.74	0.24
`DR_history[0].1`	20.44	8.00	0.10
`DR_history[0].2`	18.67	6.00	0.07
...			

3.2 α-Abstraktion von unwichtigen Variablenbits

Gegeben sei neben dem Quellmodell S der Abstraktionsgrad $\alpha \in [0,1]$. Dieser wird später automatisch bestimmt und quantifiziert, wie viel der Wichtigkeit in das Zielmodell $T_\alpha(S)$ einfließen soll. T_α sei dabei die Abstraktionsfunktion. Man wähle dann sukzessive die wichtigsten Variablenbits[2] $v.i \in W$ und konstruiere daraus minimal nötige Typen

[2]Z.B. indem die Variablenbits nach der Wichtigkeit γ sortiert werden.

$bits^{\alpha}(v) := \max\{i \mid v.i \in W\}$ für die Variablen $v \in Vars$ des Zielmodell, so dass

$$|\alpha - \sum_{v \in Vars} bits^{\alpha}(v) \,/\, \sum_{v \in Vars} bits(v)| \tag{1}$$

minimal ist. Alle Variablenbits $v.i \notin W$ mit $i > bits^{\alpha}(v)$ sind unwichtig und werden vom Quellmodell abstrahiert. Seien im folgenden abkürzend die Konstanten $r(v) := 2^{bits(v)}$ und $r_{\alpha}(v) := 2^{bits^{\alpha}(v)}$ verwendet. Ist $r_{\alpha}(v) = 1$ und damit $bits^{\alpha}(v) = 0$, so ist v unwichtig im Modell. v braucht dann nicht in einer Typdeklaration auftreten. Allerdings kann es dann noch lesende und schreibende Variablenverwendungen geben. Zum Zwecke der Veranschaulichung sei in dieser Arbeit eine solche Typdeklaration mit dem Typ 0 Bits erlaubt. Derartig definierte SPDS-Variablen mit dem Typ 0 Bits haben keinen Einfluss auf den Konfigurationenraum. Das abstrahierte Zielmodell $T_{\alpha}(S)$ besteht aus den verkleinerten Typen $bits^{\alpha}$ und bildet über die Variablenwerte aus S Äquivalenzklassen für $T_{\alpha}(S)$. Der konkrete Variablenwert $[\![v]\!]$ in S wird in $T_{\alpha}(S)$ abstrakt durch die Äquivalenzklasse $[\![v]\!] \% r_{\alpha}(v)$ repräsentiert. Ist z.B. $bits(v) = 5$ und $bits^{\alpha}(v) = 2$, so wird v von den höheren 3 Bits abstrahiert, so dass die abstrakten Werte 0 bis 3 jeweils die konkreten Werte aus Abbildung 5 darstellen.

Abbildung 5: Beispiel einer 3-Bit-Abstraktion von $bits(v) = 5$ auf $bits^{\alpha}(v) = 2$.

abstrakt	**konkrete mögliche Werte**
0	$0, 4, 8, 12, 16, 20, 24, 28$
1	$1, 5, 9, 13, 17, 21, 25, 29$
2	$2, 6, 10, 14, 18, 22, 26, 30$
3	$3, 7, 11, 15, 19, 23, 27, 31$

Schreibende Verwendungen einer SPDS-Variable v (also eine SPDS-Zuweisung) müssen von den konkreten Werten aus S in abstrakte Werte in $T_{\alpha}(S)$ konvertiert werden. Dazu werden SPDS-Zuweisungen der Form $v = e$ durch $v = e \% r_{\alpha}(v)$ aus gedrückt. Dabei fallen die Werte entsprechend ihrer Äquivalenzklasse zusammen. Lesende Verwendungen von v in S müssen für $T_{\alpha}(S)$ analog zu konkreten Werten der Äquivalenzklasse konvertiert werden. Dies wird erreicht, indem lesende Verwendungen von v in S für $T_{\alpha}(S)$ mittels $v + \texttt{rand}(\frac{r(v)}{r_{\alpha}(v)} - 1) * r_{\alpha}(v)$ ausgedrückt werden. Dadurch werden aus abstrakten Variablenwerten sämtliche mögliche konkrete Werte. Der Abstraktionsprozess wird durch eine Intervallanalyse [Muc97] ergänzt. Diese bestimmt statisch, dass manche Variablenwerte nie realisiert werden können. Dies wird genutzt, um nicht alle konkreten Belegungen eines abstrakten Wertes zu erzeugen, sondern nur diejenigen, welche nötig sind. Weiter werden Ausdrücke durch eine Konstantenfaltung ausgewertet und vereinfacht. Dadurch vereinfachen sich abstrakte Werte im Idealfall so stark, dass die Funktion `rand()` keiner Verwendung bedarf. Der Konfigurationenraum und die zu Grunde liegende Kripkestruktur verkleinern sich bei der Abstraktion erheblich.

Abbildung 6: Abstraktion temporaler Formeln

Für eine Zustandsformel p ist $\alpha_\tau^+(p) = \alpha^+(p)$ und $\alpha_\tau^-(p) = \alpha^-(p)$.
Für eine Formel $\phi \in \{\neg p, p \wedge q, Xp, pUq, Ap\}$ ist:

$$\begin{array}{lcllcl}
\alpha_\tau^-(\neg p) & = & \neg\alpha_\tau^+(p) & \alpha_\tau^+(\neg p) & = & \neg\alpha_\tau^-(p) \\
\alpha_\tau^-(p \wedge q) & = & \alpha_\tau^-(p) \wedge \alpha_\tau^-(q) & \alpha_\tau^+(p \wedge q) & = & \alpha_\tau^+(p) \wedge \alpha_\tau^+(q) \\
\alpha_\tau^-(Xp) & = & X\alpha_\tau^-(p) & \alpha_\tau^+(Xp) & = & X\alpha_\tau^+(p) \\
\alpha_\tau^-(pUq) & = & \alpha_\tau^-(p)U\alpha_\tau^-(q) & \alpha_\tau^+(pUq) & = & \alpha_\tau^+(p)U\alpha_\tau^+(q) \\
\alpha_\tau^-(Ap) & = & A\alpha_\tau^-(p) & \alpha_\tau^+(Ap) & = & A\alpha_\tau^+(p)
\end{array}$$

3.3 Temporale Abstraktion

Bei Abstraktionsgraden $\alpha < 1$ können ohne Abstraktion der temporalen Formel ϕ Fehlalarme entstehen, welche nicht erwünscht sind (false positives). In diesen Fällen erfüllt das abstrahierte Modell $T_\alpha(S)$ die Formel ϕ ($T_\alpha(S) \models \phi$), wo hingegen das ursprüngliche Modell S diese nicht erfüllt ($S \not\models \phi$). Daher wird für kleinere Abstraktionsgrade ($\alpha < 1$) ϕ zu ϕ^α abstrahiert, so dass aus $T_\alpha(S) \models \phi^\alpha$ stets auch $S \models \phi$ folgt. Dann kann es lediglich unechte Negativbeispiele (false negatives) geben, welche am Ausgangsmodell S auf Echtheit überprüft und für eine bessere Abstraktion genutzt werden können (CEGAR).

Der Abstraktionsgrad α induziert eine Abstraktionsrelation $R^\alpha \subseteq \mathit{konf}(S) \times \mathit{konf}(T_\alpha(S))$ zwischen den Konfigurationen von S und $T_\alpha(S)$. Nach Konstruktion ist $\mathit{konf}(T_\alpha(S)) \subseteq \mathit{konf}(S)$. Unwichtige Konfigurationen wurden vom Konfigurationenraum abstrahiert. Die Variablenbelegungen $[\![v]\!]$ von Konfigurationen $\mathit{konf}(S) \setminus \mathit{konf}(T_\alpha(S))$ werden mittels der Abstraktionsrelation R^α abgebildet auf deren Äquivalenzklassen $[\![v]\!]\%r_\alpha(v)$. Sei $p \in \mathit{AExpr}$ eine Bedingung (Prädikat) innerhalb einer temporalen Formel. Da Variablentypen verkleinert wurden, könnte es eine Konfiguration in S, aber nicht in $T_\alpha(S)$ mit Variablenbelegung $\mathit{env} \in \mathit{ENV}$ geben, so dass p erfüllt ist in S aber nicht in $T_\alpha(S)$. Wir wollen daher die sichere Abstraktion $p^\alpha \in \mathit{AExpr}$ so definieren, dass stets gilt $p^\alpha \Rightarrow p$. Damit kann der Operator α^- definiert werden als $\alpha^-(p) := \forall T_\alpha(p)$, wobei $\forall q$ bedeutet, dass q für jeden Funktionswert von `rand` erfüllt sein muss. $\alpha^-(p)$ ist damit für eine abstrakte Konfiguration s^α erfüllt, wenn p für alle zugehörigen konkreten Konfigurationen s erfüllt ist. Analog definiert man den Operator α^+ als $\alpha^+(p) := \exists T_\alpha(p)$. $\alpha^+(p)$ ist dann für eine abstrakte Konfiguration s^α erfüllt, wenn p für mindestens eine zugehörige konkrete Konfiguration s erfüllt ist. Ist $\alpha^+(p) = \alpha^-(p)$, so heißt α *präzise* bezüglich p. α^+ und α^- werden auf temporale Formeln erweitert zu α_τ^+ und α_τ^-. Diese sind induktiv gemäß Abbildung 6 definiert und werden als universelle bzw. existenzielle temporale Abstraktion bezeichnet. Als temporale Abstraktion für ϕ dient dann $\phi^\alpha := \alpha_\tau^-(\phi)$. Man beachte, dass äquivalente temporale Formeln verschiedene Abstraktionen besitzen können.

3.4 Eigenschaften

Bei einem Abstraktionsgrad $\alpha \approx 1$ werden nur sehr wenige unwichtige Variablen abstrahiert. In diesem Fall ist die Abstraktion typischerweise noch präzise. Mit sinkendem Abstraktionsgrad $\alpha \approx 0.01$ jedoch wird zunehmend auch mehr von den wichtigen Variablen abstrahiert, so dass dann die Abstraktion nicht mehr präzise ist. Wegen Abstraktion der gegebenen temporalen Formeln kommt es nicht zu Fehlalarmen (False Negatives, False Positives), wenn sich das Ergebnis der Modellprüfung für ϕ verändern würde.

Satz 1 *Für eine temporale Formel ϕ und ein SPDS S gilt:* $(T_\alpha(S) \models \phi^\alpha) \Rightarrow (S \models \phi)$.

Beweis (Skizze): Ergibt sich nach Konstruktion von T_α und ϕ^α analog [KP00].

□

Die Abstraktion T_α heißt *präzise* bezüglich S und α, falls auch die Umkehrung gilt. Dann ist $(T_\alpha(S) \models \phi^\alpha) \Leftrightarrow (S \models \phi)$. Der minimale Abstraktionsgrad, welcher noch präzise abstrahiert, heißt *optimal* und wird später näherungsweise bestimmt. Für präzise und optimale Abstraktionsgrade sind Fehlalarme ausgeschlossen.

Satz 2 *Es gilt $S \simeq T_\alpha(S)$, falls die in $T_\alpha(S)$ für lesende Variablenverwendungen aus den abstrakten Werten v generierten konkreten Werte „$v + \texttt{rand}(\frac{r(v)}{r_\alpha(v)} - 1) * r_\alpha(v)$“ sich mittels Programmanalysen/-Transformationen zu v vereinfachen lassen.*

Beweis (Skizze):
zu zeigen: $T_\alpha(S) \preceq S$
Hierzu konstruiere man eine entsprechende Funktion $\Theta : konf(S) \rightarrow konf(T_\alpha(S))$ gemäß obiger Abstraktion als $\Theta(s) := s[v \mapsto [\![v]\!]_{env(s)} \% r_\alpha(v)]$. Konkrete Werte in s werden dabei abgebildet auf ihre abstrakten Äquivalenzklassen. Seien nun lesende Verwendungen abstrakter Werte v in S betrachtet. Diese werden zu konkreten Werten „$v + \texttt{rand}(\frac{r(v)}{r_\alpha(v)} - 1) * r_\alpha(v)$“ in $T_\alpha(S)$ konvertiert. Durch die `rand`-Funktion entstehen potentiell neue Kontrollflüsse. Wird durch eine Programmanalysen (z.B. Intervallanalyse) in S die (konservative) Nebenbedingung $v \leq c$ für ein $c < r_\alpha(v)$ erkannt, so kann „$v + \texttt{rand}(\frac{r(v)}{r_\alpha(v)} - 1) * r_\alpha(v)$“ zu v vereinfacht werden, da größere Werte als c in S nicht realisierbar sind. Jede Äquivalenzklasse (abstrakter Wert) repräsentiert dann genau einen konkreten Wert. Ein Lauf $s1 \rightarrow s2...$ in S wird daher zu einem gültigen Lauf $\Theta(s1) \rightarrow \Theta(s2)...$ in $T_\alpha(S)$.
zu zeigen: $S \preceq T_\alpha(S)$
Der Kontrollfluss und damit auch der Lauf hängt nicht von toten Werten ab. Daher kann analog $\Theta(s) := s'$ gewählt werden, wobei s' aus s hervorgeht, indem Belegungen toter Variablen in S entsprechend passend gewählt werden. Nach obigen Betrachtungen ist dies stets möglich, da jede Äquivalenzklasse (abstrakter Wert) genau einen konkreten Wert repräsentiert.

Damit gilt $S \simeq T_\alpha(S)$.

□

4 Fehlalarmfreie Abstraktion

Der Abstraktionsprozess approximiert nun für gegebenes SPDS S, temporale Formel ϕ und maximalem Abstraktionsfehler $eps > 0$ den kleinsten Abstraktionsgrad, welcher noch präzise ist:

1 Berechne Wichtigkeit von Variablenbits und Intervallanalyse im Modell S.

2 Setze $\alpha_0 := 0$ und $\alpha_1 := 1$.

3 Setze $\alpha := \frac{\alpha_0+\alpha_1}{2}$.

4 Berechne α-Abstraktion $T_\alpha(S)$ von Modell S.

5 Setzte $\alpha_1 = \alpha$, falls $\phi^\alpha = \phi$ und $S \simeq T_\alpha(S)$. Andernfalls setzte $\alpha_0 = \alpha$.

6 Gehe zu 3, falls $\alpha_1 - \alpha_0 > eps$.

Schritt 5 wird dabei mittels der hinreichenden Bedingung aus Satz 2 konservativ bestimmt.

Wie man leicht sieht, stellt der Abstraktionsgrad α_1 eine Approximation des kleinsten präzisen Abstraktionsgrads dar (Intervallschachtelung):

Satz 3 *Sei α^* der optimale Abstraktionsgrad und α_1 wie oben bestimmt. Dann ist α_1 ein präziser Abstraktionsgrad für S und es gilt: $|\alpha_1 - \alpha^*| \leq eps$.*

Zur Veranschaulichung sei wieder Beispiel 1 aus Abbildung 1 betrachtet. Es ergibt sich eine Berechnungsfolge von $[0.5, 0.25, ..]$ für α_1, welche bei $\alpha_1 = 0.02$ terminiert. Dies ist eine Reduktion auf 2% der bisher verwendeten Variablenbits. Dabei ergeben sich die neuen minimal nötigen Typen aus Abbildung 7. Der Konfigurationenraum, gemessen in der Anzahl der möglichen Köpfe, wird dabei um den Faktor $2^{832} = 2.86 \cdot 10^{250}$ verkleinert. In Abbildung 1 rechts ist dazu das abstrahierte Modell zu sehen. Darin wurden

Abbildung 7: Berechnete Minimaltypen für Beispiel 1 bei Abstraktionsgrad $\alpha_1 = 0.02$.

	offset	history[*]	retry	timeouts	e	word	event	Gesamt
$bits$	8	800	8	8	8	8	8	848
$bits^{\alpha_1}$	8	0	0	0	4	0	4	16

Ausdrücke zu Konstanten wie z.B. $(0\%2\hat{}4) \equiv 0$ bzw. $(1\%2\hat{}0) \equiv 0$ sowie $(\texttt{timeouts}+1)\%1 \equiv 0$, welche in den Zuweisungen $\texttt{event} = 0$ bzw. $\texttt{word} = 1$ bzw. $\texttt{timeouts} = \texttt{timeouts} + 1$ usw verwendet wurden. Durch Konstantenfaltung werden die Ausdrücke $(\texttt{event} + \texttt{rand}(15) * 16) == 0$ bzw. $(\texttt{e} + \texttt{rand}(15) * 16) == 0$ im Beispiel 1 zu $\texttt{event} == 0$ bzw. $\texttt{e} == 0$ vereinfacht, da statisch festgestellt wird, dass stets $\texttt{event} < 16$ bzw. $\texttt{e} < 16$ im Modell gilt. Das entstandene SPDS-Modell in Abbildung 1 (rechts) enthält dann noch diverse Artefakte wie Zuweisungen an Variablen des Typs 0 Bits. Diese können später eliminiert werden mittels weiterer statischer Analysen (z.B. Slicing [RZ07]).

5 Verwandte Arbeiten

Das in dieser Arbeit vorgestellte Verfahren ist ähnlich zu [DHJ+01]. Auch dort werden Modell und temporale Eigenschaften derart abstrahiert, dass bei Modellprüfung der Abstraktionen Rückschlüsse auf das Ursprungsmodell möglich sind. Allerdings werden dort nur endliche Modelle sowie LTL betrachtet und es wird die Abstraktion manuell vom Nutzer durch die Bandera Abstraction Specification Language (BASL) bestimmt. Methoden dieser Arbeit sind auch für unendliche Strukturen (SPDS) nutzbar und abstrahieren vollautomatisch mittels eines frei wählbaren Abstraktionsgrads α. Anders als die Predikatabstraktion in [GS97] operieren wir direkt in der symbolischen Beschreibung und nicht auf dem zu Grunde liegendem ggf. sehr großen oder unendlichem Transitionssystem. In [MA03], [CCK+02] und [HJMS02] werden Beweise bzw. Gegenbeispiele für das Erfülltsein bzw. Unerfülltsein von SAT-Formeln verwendet, um relevante Modellteile zu identifizieren und zu abstrahieren. Wir hingegen lösen das aufgestellte SAT-Problem nicht (weil aufwändig) und untersuchen es lediglich mittels effizienter Heuristiken auf wichtige Modellteile. In [NK00] werden als Prädikatabstraktion wichtige Prädikate bezüglich der temporalen Spezifikation berechnet. Im Gegensatz zu unserem Verfahren terminiert deren Verfahren nicht immer, es wird nur Bisimularität betrachtet (daher Restmodell sehr groß und kein Einfluss auf Abstaktionsgrad), die Betrachtungen sind im wesentlichen auch mit Slicing realisierbar und es werden nur endliche Modelle (vom unendlichem Programm) betrachtet, weshalb viele unnötige Fehlalarme entstehen können. Letzteres ist wegen der ISO-C konformen Semantik in SPDS [KZR09] bei unserem Verfahren reduziert. Die in [SUM96] vorgestellte Deduktive Modellprüfung generiert wie unsere Methode Abstraktionen basierend auf gegebenen LTL Formeln. Diese ist allerdings auf LTL und endliche Modelle beschränkt und erfordert signifikanten manuellen Eingriff im Gegensatz zu unserer Methode. In [KP00] werden Techniken vorgestellt zur Abstraktion von Kontrollfluss und Daten. Auch diese ist beschränkt auf LTL, endliche Modelle und operieren auf Diskreten Kripkestrukturen statt in der symbolischen Beschreibung eines SPDS. Zudem muss im Gegensatz zu unserem Verfahren für die Abstraktion eine Zuordnung (mapping) von abstrakten auf konkrete Zustände manuell erfolgen.

6 Zusammenfassung und Ausblick

Es wurde ein Verfahren vorgestellt, das zu gegebenen temporalen Formeln ϕ_i und einem Modell S selektiv unwichtige Variablenbits identifiziert und davon abstrahiert. Dabei wurde die Wichtigkeit von Variablenbits im Modell heuristisch bestimmt, und schließlich das Modell von unwichtigen Variablenbits so abstrahiert, dass keine Fehlalarme entstehen (präzise Abstraktion). In früheren Betrachtungen wurde der Grad der Verkleinerung über einen Parameter (Abstraktionsgrad α) gesteuert [RH11]. Bei der Überführung des Quellmodells in ein weniger detailliertes Zielmodell können Fehlalarme (false negatives) entstehen, wenn der Abstraktionsgrad α zu klein ist. Der Abstraktionsgrad α wird in dieser Arbeit automatisch so bestimmt, dass die Abstraktion präzise ist und *keine Fehlalarme* entstehen.

Zwar ist es möglich auch eine optimale Abstraktion zu berechnen, jedoch übersteigt deren Komplexität die der Modellprüfung [RH11], was den vorgestellten heuristischen Ansatz rechtfertigt.

Positive und negative Literale einer Variablen x_i in der Repräsentation des Modells wurden in der vorgestellten Methode zusammengefasst. Eine feinkörnige Betrachtung der einzelnen Literale einer Variablen, z.B. wenn nur das positive als sehr wichtig identifiziert wurde, kann sicherlich weitere Informationen über das Modell enthüllen und zukünftig noch bessere Abstraktionen liefern, da dann z.B. alle Verbindungen mit Variablen, die nur mit dem negativen Literal von x_i verknüpft sind, auch reduziert werden können.

Literatur

[BEM97] Ahmed Bouajjani, Javier Esparza und Oded Maler. *Reachability Analysis of Pushdown Automata: Application to Model-Checking*. Proc. of the 8th International Conference on Concurrency Theory, LNCS 1243, 1997.

[Ber06] Felix Berger. *A test and verification environment for Java programs*. Diplomarbeit Nr. 2470, Universität Stuttgart, 2006.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani und Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer-Aided Verification*, Jgg. 2404 of *LNCS*. Springer, 2002.

[CCK+02] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith und Dong Wang. Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT Based Conflict Analysis. In Mark Aagaard und John OâLeary, Hrsg., *Formal Methods in Computer-Aided Design*, Jgg. 2517 of *Lecture Notes in Computer Science*, Seiten 33–51. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36126-X3.

[DHJ+01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Hongjun Zheng und Willem Visser. Tool-supported program abstraction for finite-state verification. In *Proc. of the 23rd International Conference on Software Engineering*, Seiten 177–187, Washington, DC, USA, 2001. IEEE Computer Society.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith und Stefan Schwoon. *Efficient algorithms for model checking pushdown systems*. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855, 2000.

[EKS02] Javier Esparza, Antonin Kucera und Stefan Schwoon. *Model-Checking LTL with Regular Valuations for Pushdown Systems*. Proc. of the 4th International Symposium on Theoretical Aspects of Computer Software, LNCS 2215, 2002.

[ES01] Javier Esparza und Stefan Schwoon. *A BDD-based model checker for recursive programs*. LNCS Volume 2102, 324-336, Springer, 2001.

[GS97] Susanne Graf und Hassen Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, Hrsg., *Computer Aided Verification*, Jgg. 1254 of *LNCS*, Seiten 72–83. Springer, 1997.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar und Grégoire Sutre. Lazy abstraction. In *Proc. of the 29th symposium on principles of programming languages*, Seiten 58–70, New York, USA, 2002. ACM.

[HM09] Roberto Hoffmann und Paul Molitor. Guiding property development with SAT-based coverage calculation. *Midwest Symposium on Circuits and Systems*, Seiten 1199–1202, 2009.

[Hof05] Roberto Hoffmann. A SAT Solving Framework: Conflict Analysis and Learning. Diplomarbeit, Institute of Computer Sciences, MLU Halle-Wittenberg, 2005.

[KP00] Yonit Kesten und Amir Pnueli. Control and data abstraction: cornerstones of practical formal verification. *Int. Journal Software Tools for Technology Transfer*, 2:328–342, 2000.

[KSS06] Stefan Kiefer, Stefan Schwoon und Dejvuth Suwimonteerabuth. *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.

[KZR09] Raimund Kirner, Wolf Zimmermann und Dirk Richter. On Undecidability Results of Real Programming Languages. In *15. colloquium prog. lang. (KPS)*, 2009.

[MA03] Kenneth L. McMillan und Nina Amla. Automatic Abstraction without Counterexamples. In Hubert Garavel und John Hatcliff, Hrsg., *Tools and Algorithms for the Construction and Analysis of Systems*, Jgg. 2619 of *LNCS*, Seiten 2–17. Springer Berlin / Heidelberg, 2003.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implem.* Morgan Kaufmann, 1997.

[NK00] Kedar S. Namjoshi und Robert P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In *Computer Aided Verification*, Jgg. 1855 of *LNCS*, Seiten 435–449. Springer, 2000.

[Obd02] Jan Obdrzalek. *Model Checking Java Using Pushdown Systems*. LFCS, University of Edinburgh, 2002.

[RB12] Dirk Richter und Christian Berg. Exact Gap Computation for Code Coverage Metrics in ISO-C. In *Proc. of 7th international Workshop on Model Based Testing*, 2012.

[RH11] Dirk Richter und Roberto Hoffmann. Spezifikationsgetriebene Abstraktion fuer Kellersysteme. In *16. colloquium prog. lang. (KPS)*, 2011.

[RZ07] Dirk Richter und Wolf Zimmermann. *Slicing zur Modellreduktion von symbolischen Kellersystemen*. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.

[Sch02] Stefan Schwoon. *Model-Checking Pushdown Systems*. TU München, 2002.

[SSE05] Dejvuth Suwimonteerabuth, Stefan Schwoon und Javier Esparza. *jMoped: A Java Bytecode Checker Based on Moped*. Tools and Alg. for Construction and Analysis of Systems, LNCS, Springer, 2005.

[SUM96] Henny Sipma, Tomas Uribe und Zohar Manna. Deductive model checking. In Alur und Henzinger, Hrsg., *Computer Aided Verification*, Jgg. 1102 of *LNCS*, Seiten 208–219. Springer, 1996.

[Til05] Daniel Tille. A SAT Solving Framework: Splitting Strategies. Diplomarbeit, Institute of Computer Sciences, MLU Halle-Wittenberg, 2005.

[Wal00] Igor Walukiewicz. *Model checking CTL Properties of Pushdown Systems*. In FSTTCS'00, LNCS 1974, 2000.

Architecture-Aware Cost Modelling for Parallel Performance Portability

Evgenij Belikov, Hans-Wolfgang Loidl, Greg Michaelson, Phil Trinder

School of Mathematical and Computer Sciences
Heriot-Watt University
EH14 4AS Edinburgh
{eb120,hwloidl,greg,trinder}@hw.ac.uk

Abstract: Languages for efficient parallel programming need to achieve high performance portability in order to harness the power offered by rapidly evolving parallel architectures. We use a combination of high-level architecture-aware cost modelling with a low-level, explicit control of coordination as a programming model to improve performance portability. We explore and quantify the impact of heterogeneity in modern parallel architectures on the performance of parallel programs on a range of clusters of multi-cores, varying in architectural parameters such as processor speed, memory size and interconnection speed. Additionally, we develop several formal cost models and automatically use these architectural characteristics to determine suitable granularity and work placement. The effectiveness of such cost-model-driven management of parallelism on common-place cluster hardware is demonstrated by measuring the performance of a parallel sparse matrix multiplication, implemented in C+MPI, on a range of heterogeneous architectures. On a cluster with 16 cores, the speedup increases from 6.2, without any cost model, to 9.1, indicating that even a simple, static cost model is effective in adapting the execution to the target architecture and in significantly improving parallel performance and scalability with negligible overhead.

1 Introduction

To fully harness the potential power offered by rapidly evolving and increasingly heterogeneous and hierarchical parallel architectures, parallel programming environments need to bridge the gap between expressiveness and portable performance. To achieve the latter, the dynamic behaviour of the application needs to adapt to the architectural characteristics of the machine, rather than tying it to one particular configuration.

Our long term goal is to automate this process of adaptation by using, on implementation level, architectural cost models and sophisticated run-time environments to coordinate the parallel computations, and, on language level, high-level abstractions such as algorithmic skeletons [Col89] or evaluation strategies [THLJ98]. In this paper, we focus on one concrete application, study the impact of heterogeneity on its performance, and demonstrate performance improvements due to the use of architectural cost models.

Parallel programming has proved to be significantly more difficult than sequential programming, since in addition to the need to specify the algorithmic solution to a problem the programmer also has to specify how the computation is coordinated, taking into account architectural specifics such as processor and interconnection speed. This is difficult, but manageable on homogeneous high-performance architectures. However, novel architectures are increasingly heterogeneous, being composed of different kinds of processors and using hierarchical interconnections. Therefore, current assumptions that all processors have roughly the same computational power or that communication between any two processors is equally fast, no longer hold.

The approaches to parallel programming range from high-level implicit, where the coordination is completely hidden from the programmer, over semi-explicit, to lower-level explicit models, where every aspect of coordination is exposed to the programmer [ST98]. While fully implicit approaches proved intractable for unrestricted parallelism, explicit approaches were historically favoured for optimisation capabilities. However, manual optimisations are prone to error and often result in non-portable code. Therefore, following the principle of separation of concerns, we use an architecture-aware cost model to automatically partition the work, adapting to the target architecture, hence improving load balancing on heterogeneous architectures and raising the level of abstraction, as well as increasing performance portability. Moreover, we use a message-passing programming model, since we cannot assume a heterogeneous and hierarchical architecture to be a shared-memory architecture and wish to avoid the complexities of implementing application-level virtual shared-memory abstraction. This eliminates the synchronisation overhead of shared-memory models, since there is no shared state, at the cost of packing and buffering. To avoid deadlocks we rely on a deadlock-free library implementation of collective operations. Furthermore, due to the simple communication pattern and data layout of the chosen application, it is possible to ensure that no race conditions can occur. By keeping the cost model encapsulated within a single function, minimal code changes are required to exchange the model without affecting the application code. Provided the necessary language and communication library support, the need for manual changes of code while migrating to another target architecture is eliminated altogether.

We contend that architecture-aware cost models can be beneficially applied to connect languages and architectures, narrowing the aforementioned gap, and in particular to facilitate performance portability, which is critical, since hardware architectures evolve significantly faster than software applications. We develop four simple cost models, characterising basic system costs on a heterogeneous network of multi-cores, and demonstrate how the use of these cost models improves parallel performance of a parallel sparse matrix multiplication algorithm, implemented in C and MPI, across a range of parallel architectures of varying heterogeneity with negligible overhead.

We continue by presenting general background in Section 2, followed by a discussion of our simple static architectural cost model and its integration within the application in Section 3. Subsequently, we present and evaluate experimental results of the application of cost-model-based approach to parallel sparse matrix multiplication in Section 4 and mention related research efforts in Section 5. Finally, we conclude in Section 6 and suggest further research to address the limitations of present work.

2 Background

An architecture defines a set of interconnected processing elements (PEs), memory units and peripherals. According to Flynn's taxonomy [Fly66], most of the currently available architectures fit into the MIMD category but also may have SIMD components. Moreover, new architectures are increasingly *hierarchical* including several memory and network levels and increasingly *heterogeneous* in respect to the computational power of the PEs and networking capabilities. To guarantee efficiency, it is required to integrate architectural parameters, due to their impact on performance, however, to provide expressive abstractions it is also required to hide the low-level details within a supporting parallel programming environment [Ski94]. We use the explicit low-level message-passing model that is commonly used to implement portable HPC applications on homogeneous and flat distributed-memory machines, and aim at enhancing performance portability for a variety of architectures by adding an architecture-aware cost model to guide the adaptation.

2.1 Parallel Computational Models and High-Level Parallelism

Ideally, a model of parallel computation would accomplish what the von-Neumann model does for sequential computation – it provides a way to design architecture-independent algorithms that would nevertheless efficiently execute on a wide range of uni-processor machines. On the one hand, such a model should hide the architecture-specific low-level details of parallel execution and provide high-level language constructs, thus facilitating parallel programming. On the other hand, the model should allow for cost estimation and efficient implementation on a broad range of target platforms, to support performance portability [Ski94]. Clearly, there is a tension between these two goals, and historically the parallel computing community has focused on exploiting architecture-specific characteristics to optimise run time.

One of the most prominent cost models for parallel execution is the PRAM model [FW78], an idealised analytical shared-memory model. The major drawback of PRAM is that, because of its simplicity, efficient PRAM algorithms may turn out to be inefficient on real hardware. This is due to the optimistic assumptions that all communication is for free and that an infinite number of PEs with unbounded amount of memory are available.

Another prominent cost model is the Bulk Synchronous Parallel model (BSP) [Val90], which restricts the computational structure of the parallel program to achieve fairly good predictability. A BSP program is composed of a sequence of super-steps, each consisting of three ordered sub-phases: a phase of local computation followed by the global communication phase and then by a barrier synchronisation phase. Although appropriate for uniform memory access shared-memory architectures, BSP assumes unified communication, thus renouncing locality as a performance optimisation, rendering BSP unsuitable for applications that rely on data locality for performance. The parameters used by the BSP model are the number of PEs p, the cost of global synchronisation l, global network bandwidth g, and the speed of processors that determines the cost of local processing.

MultiBSP is a recent update of the BSP model that aims primarily at modelling computations on multi-core architectures with cache hierarchies [Val11]. Benchmarks are used to determine the model parameters for each target architecture.

The LogP model [CKP+93] is another well-established cost model for distributed-memory architectures, based on the message-passing view of parallel computation and emphasising communication costs. LogP uses four parameters: L, an upper bound for the latency when transmitting a single word; o, the sending and receiving overhead; g, gap per byte for small messages, with $\frac{1}{g}$ representing the bandwidth; and the number of PEs P. The model has been subsequently extended to account for long messages (LogGP [AISS95]), for heterogeneity (HLogGP [BP06]) that uses vector and matrix parameters instead of scalar parameters in LogGP, and for hierarchy (Log-HMM [LMR95]).

These models provide a good selection of parameters that are relevant for performance on distributed-memory architectures. However, no sufficiently accurate unified parallel computation model that accounts for both heterogeneity and hierarchy exists to date.

2.2 Cost Modelling

In general, predicting computational costs proved intractable, and therefore some qualitative prediction is used in practice to successfully guide adaptation. Abstract models, developed for algorithm design, are usually architecture-independent by assigning abstract unit cost to the basic operations. For a more accurate prediction, the cost model needs to be parametrised with the experimentally determined characteristics of the target architecture. On the one hand, static cost models incur less overhead than the dynamic ones, however, they do not take dynamic parameters such as system load and network contention into account, which may severely affect performance. On the other hand, dynamic models suffer from additional run-time overhead that can cancel out the benefits of more accurate performance prediction.

Cost models can be used at design time to help choosing suitable algorithms and to avoid restricting the parallelism too early. At compile time, cost models are useful for performance debugging and for automatic optimising code transformations based on static analysis techniques such as type inference and cost semantics. Ultimately, at run-time, cost models can guide dynamic adaptation.

Higher-level cost models are commonly provided for algorithmic skeletons [Dar93], but they do not incorporate architecture information. Bischof et al. [BGK03] claim that cost models can help developing provably cost-optimal skeletons. Another conceivable use is for improving coordination or for determining granularity and mapping according to a custom goal function such as achieving highest utilisation of resources, maintaining highest throughput, lowest latency or achieving best performance-to-cost ratio [TCH+11].

Hardware cost models are most accurate and can be used to determine worst case execution time for safety-critical systems. Ultimately, cost models provide a way for a program, to a certain extent, to predict its own performance, which is a prerequisite for self-optimisation.

3 Architecture-Aware Cost Modelling

As opposed to characteristics of the application and dynamic run-time parameters, this work focuses solely on static architectural parameters. We devise and refine a simple cost model that is used to adapt the execution to a new target platform by determining suitable granularity and placement that affect static load-balancing for a chosen application. As mentioned above, we identify the number of PEs, the speed of PEs, the amount of L2 cache and RAM, as well as latency as the most relevant parameters. Rather than for run time prediction, these parameters are used by the cost model to estimate relative computational power of the available PEs which determines the partitioning of work.

3.1 Analytic, Static Cost Models

Typically, the only parameters used in the parallelisation of algorithms are the number of available PEs P and the problem size N. The naive approach, which serves as a baseline case, is to distribute chunks of work of equal size of $\frac{N}{P}$ among the PEs. This is sufficient for perfect load balancing for regular applications on regular data, if run on homogeneous and flat architectures. However, this simple strategy results in poor performance for applications that aim at exploiting irregular parallelism, operate on irregular data or run on hierarchical, heterogeneous systems, since it is necessary to distribute work in a way that accounts for the computational power of each PE and for the communication overhead. This intuition is captured by the following equation:

$$C_i = \frac{S_i}{S_{all}} \cdot N, \text{where } S_{all} = \sum_{i=1}^{P} S_i$$

with C_i being the chunk size that represents the work assigned to PE i depending on its computational power S_i in relation to the overall computational power of the system S_{all}. Our results in Section 4 underline this point. In the baseline case we set $S_i = 1$.

CPU-aware Cost Model (CM0): In the initial cost model we start with a single additional parameter — the *CPU speed* of a PE. At the first glance it is a primary attribute that characterises the computational power of a PE and can be used in addition to P and N to determine the chunk size. This results in the following cost model by changing the way relative computational power is estimated:

$$S_i = CPU_i$$

where CPU_i denotes the speed of PE i. However, there is consensus amongst practitioners on the high importance of increasing the cache hit ratio. A larger cache size results in fewer main memory accesses, thus improving the performance of an application [AMT11].

CPU-Cache-aware Cost Model (CM1): Thus the next step is to include the *L2 cache size* in our cost model, resulting in the following change to CM0 to obtain CM1:

$$S_i = aCPU_i \cdot bL2_i$$

where $L2_i$ denotes the cache size associated with a PE (in kilobytes) and a as well as b are scaling factors that can be tuned to prioritise one parameter over the other. However, the cache miss ratio for our application is fairly low[1], which explains why cache does not appear to be a good predictor in the untuned CM1 that performs worse than CM0. Due to time constraints, we have not tuned the scaling factors to further improve performance.

Based on probability theory, large estimated values for S_i suggest that both CPU speed and cache size are large, therefore a multiplicative rather than additive model is a natural choice in this case. This helps to avoid specific large values dominating all the other parameters, which would increase the amount of required tuning. Moreover, we display two distinct scaling factors for flexibility and to emphasise that each parameter can be separately tuned. However, all scaling factors can be combined into a single factor in the actual implementation.

CPU-Cache-RAM-aware Cost Model (CM2): Since very large problems may exceed *RAM size*, we include RAM as a binary threshold parameter in the next cost model – we require the problem size to fit into RAM if performance is critical. Moreover, we can incorporate the knowledge of the size of the RAM of each node in the cache-to-RAM ratio that further characterises the computational power of the node and facilitates adaptation to heterogeneous architectures. The ratio reflects the intuition that larger caches and larger RAM size generally positively affect performance. However, the strength of the respective effect depends on the particular application. The resulting change to CM1 to obtain CM2 is as follows:

$$S_i = \frac{aCPU_i \cdot bL2_i}{cRAM_i}$$

where RAM_i denotes the size of the main memory (in megabytes) and c is the corresponding scaling factor.

CPU-Cache-RAM-latency-aware Cost Model (CM3a): Finally, we refine our model to include the *latency* as an additional parameter that characterises the interconnection network, resulting in the following change to CM2 to obtain CM3:

$$S_i = \begin{cases} \frac{aCPU_i \cdot bL2_i \cdot dL_i}{cRAM_i}, \text{if } L_i < t \\ 0 \qquad\qquad , \text{otherwise} \end{cases}$$

On the one hand, latency can be used as a binary parameter to decide whether to send work to a remote PE. If the latency L_i exceeds a threshold t (both in ms), we do not send any work, otherwise we use latency with a corresponding scaling factor d in the cost model to determine the chunk size that is large enough to compensate for higher communication

[1] 0.03%, measured for the sequential run with cachegrind

overhead. Currently, we simply use latency to avoid slowdown by not sending any work to exceedingly far remote PEs. In ongoing work (CM3b), we aim to quantify the relationship among the benefit of offloading work and the communication overhead that can cancel out the benefits, as well as to automate the choice of a suitable threshold value.

3.2 Embedding within an Application

Although a cost-model is language-agnostic, it needs to be expressed in some language and embedded in the application code within a single function that determines the granularity of work packages for given problem size and is informed by the description of the target architecture. In our case, the cost model is implemented directly in the host language and it influences the behaviour of the parallel execution by determining the partitioning of work, which is distributed to the workers by using collective operations. Process management is handled implicitly by the communication library that places one process on each used PE. Though performance varies for different process placement policies that are described in Section 4.1, the use of cost models decreases this variation.

To introduce a static, analytic cost model, almost no changes are required, since partitioning is performed in any case. One refactoring step suffices to separate the partitioning from other coordination and computation concerns. In a well-designed modular application this step would be unnecessary. Instead, merely the partitioning function would be affected by introducing a cost model. Furthermore, depending on the host language, exchanging a cost model can be accomplished during the run-time of the application, facilitating the creation of adaptive components.

Moreover, it is conceivable to create an abstract partitioning function, that is then instantiated with the specifics of the data structure that represents work. We have studied such generic partitioning functions in the context of parallel Haskell before [LTB01] and demonstrated their advantages in terms of code re-usability. Hence, the cost model is neither tied to a specific language, nor to a data structure, nor to a specific algorithm.

Since we use a low-level coordination model, the well-known disadvantages remain: a careless programmer may introduce deadlocks and non-portable optimisations. At this point we rely on a highly tuned and deadlock-free communication library and ensure the absence of race conditions by using non-overlapping buffers. Ideally, the library would avoid the packing and buffering overhead if the work is sent to a PE on the same node.

4 Experimental Results and Evaluation

The experiments include running versions of the sparse matrix multiplication (discussed in Section 4.2) that differ in the used cost model and thus in the way the work is distributed on different target platforms.

4.1 Experimental Design

We measure the run times for each architecture, using PE numbers in the range $1..16$ for a fixed data size of 8192×8192 with sparsity of 1% for both matrices to be multiplied. To minimise the impact of interactions with the operating system and other applications, the median of the run times of five runs is used for each combination.

In our experiments, we vary heterogeneity by adding different machines from different subnets and by using different numbers of the available cores. We also study a separate experiment, adding a slow machine (`linux01`) to the network. We explore application behaviour under two process allocation policies: in the `unshuffled` setup we send chunks to a PE as many times as there are cores to be used and then continue with the next host, whereas in the `shuffled` setup we distribute the work in a round robin fashion among all multi-core machines. Thus, a 4 processor execution on 4 quad-cores uses all 4 cores on one quad-core in an `unshuffled` setup but uses 1 core of each quad-core in a `shuffled` setup. This variation aims at investigating the predictability of performance, i.e. the change of scalability as new nodes are introduced and sending data to hosts on other subnets is required. To assess the influence of the interconnection network, we study configurations with varying communication latencies: an all-local setup with fast connections is compared to a setup with $\frac{1}{8}$ remote machines.

4.2 Parallel Sparse Matrix Multiplication

We have chosen sparse matrix multiplication as an application that is representative for a wide range of high-performance applications that operate on irregular data [Asa06]. A matrix is termed sparse if it has a high number of zero-valued elements. Computationally, sparseness is important because it allows to use a more space-efficient representation. Algorithms operating on such a representation will automatically ignore zero-entries, at the expense of some administrative overhead, and will therefore be more time-efficient. In practice, we consider large matrices with less than 30% of the elements being non-zero.

Matrix multiplication for two given matrices $A \in \mathbb{Z}^{m \times n}$ and $B \in \mathbb{Z}^{n \times l}$, written $C = A * B$, is defined as $C_{i,j} = \sum_{k=0}^{n} A_{i,k} * B_{k,j}$. Here, the focus is on how sparse matrices are represented in memory and how they can be distributed across the available PEs efficiently allowing for parallel calculations. Our cost model is used to improve parallel performance and performance portability by determining the size of the chunk of the result matrix to be calculated by each processor, thus matching the granularity of the computation to the processor's computational power.

Although low-level and rather difficult to apply, we have decided to use C and MPI for parallel programming, since this combination proved to facilitate development of efficient yet portable software. The focus in this paper is on system level implementation of strategies to achieve performance portability, rather than on language level abstractions or automated support for parallelisation. In particular, we use the MPICH1, since only this implementation of the MPI standard supports mixed-mode $32/64$-bit operation.

We follow the *Partitioning - Communication - Agglomeration - Mapping* (PCAM) design methodology [Fos95] and employ the load balancing scheme for static irregular problems [Qui04, p. 72]. In our case mapping is trivial, since we partition the work statically in exactly as many chunks as there are available PEs. To minimise communication overhead we broadcast the whole matrix A and scatter disjunct chunks of columns of the matrix B to the workers. Note that by using an architecture-aware cost model, better static load balancing is achieved, since more powerful PEs receive proportionally more work. We use highly tuned collective operations (`MPI_Scatterv`, `MPI_Gatherv`, and `MPI_Bcast`) for potentially higher performance [Gor04].

A matrix is stored as an array of $(row, column, value)$ triplets to maintain a high level of efficiency, since the elements are sorted in consecutive memory and allow for fast access. We randomly generate two sparse matrices with a specified sparseness, for flexibility, and a seed parameter, for repeatability. Thus, this first phase of the algorithm is local and performed only by the master and is not included in the run time.

The structure of the matrix multiplication algorithm is fairly standard, and we focus here on the architecture-specific aspects, in particular on gathering the architecture descriptions and determining granularity and placement based on these descriptions. After the two matrices are generated, the master gathers the information about the available architecture, in particular processor speed, RAM size, cache size and latency. All of our cost models are static, and we therefore do not monitor dynamic aspects of the computation, thus avoiding additional overhead. In the next step the *sizes of the chunks* are determined using a given cost model, and the data is distributed across the available PEs. After receiving the work, each PE, including the master, performs matrix-vector multiplications locally and sends the results back to the master process.

4.3 Target Architectures

The available machines run under CentOS 5.5 and are local[2], hence remote latency is emulated using the `netem`[3] utility that adds an additional queueing discipline to the operating system kernel and allows to delay packets at the interface. In different setups some of the available machines (as summarised in Table 1) are combined to architectures that range from homogeneous flat architectures to heterogeneous and hierarchical ones.

Table 1: Machines available for combination to different architectural configurations

alias	arch (bit)	PEs	speed (GHz)	L2 cache (MB)	RAM (GB)	cache/RAM
linux_lab	32	2	3.40	2x2	3	1.33
linux01	32	2	2.40	1x2	2	1.00
lxpara	32	8	2.33	2x6	8	1.50
beowulf	64	8	2.00	2x4	6	1.33

[2]using Gigabit Ethernet, except for `linux01` that has a 100Mbit interface
[3]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

Different setups are represented by a list of integers denoting the number of participating nodes and the number of corresponding cores followed by additional information such as the type of the machines, and concrete setup, i.e. with `linux01` vs without `linux01`, all-local or partly remote. For example, `1x8 1x4 1x2 2x1 (beowulf, lxpara, linux01, linux lab)` describes a configuration with five nodes of which one is a `beowulf` machine using all of its cores, another is a `lxpara` multi-core machine using four of its cores, another node is `linux01` using both cores, and the remaining two nodes are the `linux lab` machines using one core each. By default we refer to the all-local `unshuffled` experiments without `linux01`, where the `x8` and `x4` are `beowulf` or `lxpara` machines, and the `x2` and `x1` are `linux lab` machines.

4.4 Results and Evaluation

First, we present results for different unshuffled all-local configurations and for shuffled vs unshuffled, all-local configurations. Next, we compare the cost models for the most heterogeneous configuration: shuffled, all-local, `2x4 2x2 4x1` and investigate the impact of latency for a partly remote configuration.

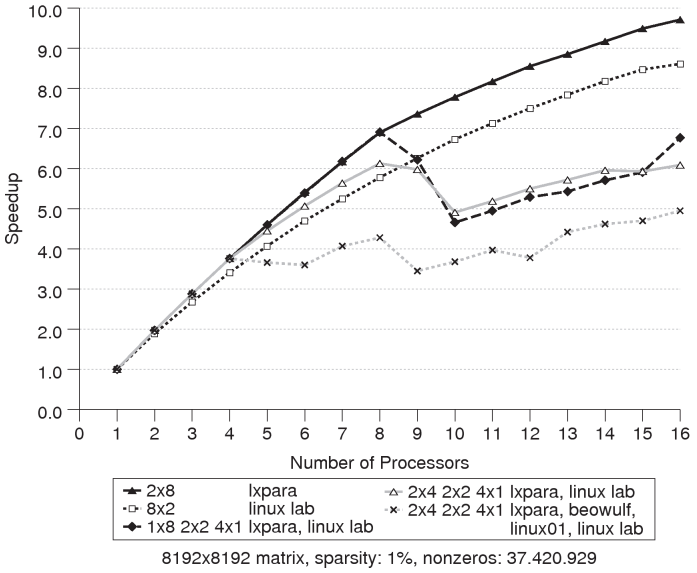


Figure 1: Speedups for unshuffled all-local setups with *varying heterogeneity*

Different All-Local Configurations: We begin with a homogeneous (all nodes have the same computational power) and flat (all nodes are on the same subnet) architecture. Figure 1 shows the speedups for up to 16 PEs, using configurations of varying heterogeneity.

The speedup is almost linear on one multi-core machine and decreases due to communication overhead once we leave the node (e.g. more than 8 PEs for the `2x8 lxpara` setup). Further decrease in performance can be observed for configurations that include machines from a different subnet (`linux lab`), due to the increased communication overhead. In summary, the graphs in Figure 1 depict the decrease in performance with increasing heterogeneity of the setup. This observation is the starting point for our work on using architectural cost models to drive the parallel execution.

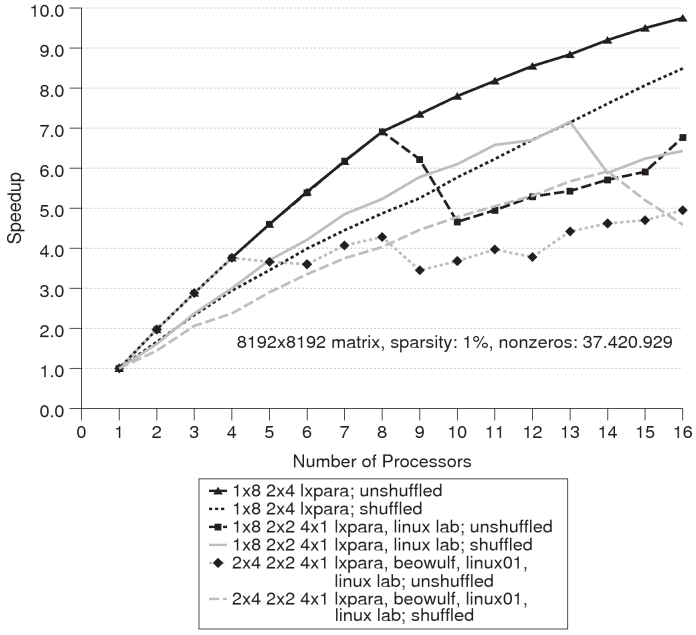


Figure 2: Speedups for selected *unshuffled versus shuffled* all-local setups

All-Local, Heterogeneous and Hierarchical Case: The `2x4 2x2 4x1` configuration in Figure 1 represents the most heterogeneous case, where instances of each available architecture are used (`beowulf`, `lxpara`, `linux lab` and the slow `linux01`). This configuration exhibits significant performance degradation: compared to the homogeneous case, with a speedup of almost 10.0 on 16 processors, the speedup drops to 5.0. While some of this drop is inevitable, due to the hardware characteristics, we show below that this result can be significantly improved by exploiting information from an architecture-aware cost model.

Figure 2 depicts the effect of the placement strategy on predictability by showing the graphs for different `unshuffled` (ticked) and `shuffled` (unticked) placement on different configurations. For a small number of PEs it is beneficial to keep the execution local by fully exploiting one multi-core before placing computations on another multi-core.

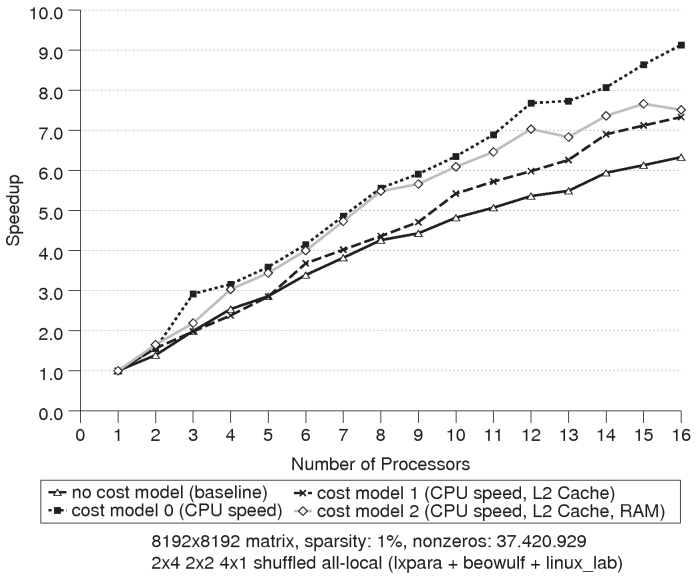


Figure 3: Speedups for the most heterogeneous configuration using *different cost models*

This strategy is tuned to minimise communication cost. However, if we are more concerned with steady and predictable (speedup curve is more smooth) but slightly slower increase in performance on the heterogeneous setups, we should use a shuffled setup.

The results on all-local configurations suggest that our cost model, discussed in Section 3, improves performance on heterogeneous setups and does not impede performance on homogeneous setups. However, if a network hierarchy is introduced we need to account for additional communication overhead for remote nodes.

Figure 3 presents a comparison of different cost models, that are all performing better than the baseline case: on 16 processors, the speedup improves from 6.2, without cost model, to 9.1, with cost model CM0, thus nearly achieving the speedup reached on the homogeneous and flat setup, as observed in the Figure 1 above. With the current settings for the scaling factors, the more advanced cost models do not achieve further improvements in performance, and the simple cost model CM0 performs best. This reflects the intuition that PE speed is the primary performance predictor for computationally intensive applications on all-local setups. Additionally, low variation of the cache-to-RAM ratio leads to the low impact of the respective parameters (cf. last column of Table 1 in Section 4.3). The number of parameters makes the more sophisticated models, which can be tuned to increase performance, more flexible and potentially employable for a wider range of applications. Moreover, application-specific and dynamic system parameters should be taken into account, since they are likely to influence the impact of the architectural parameters.

Partly Remote, Heterogeneous and Hierarchical Case: To investigate the *impact of latency* on parallel performance, we use a partly remote setup where two nodes are emulated to have high latency interconnect. Figure 4 illustrates the importance of taking the latency into account. All the cost models that do not use latency as a parameter send work to remote PEs leading to a parallel slowdown, which can be avoided by latency-aware models by simply not using the remote PEs. In ongoing work, we aim to devise a sophisticated cost model that would send off the work only if it would increase the speedup.

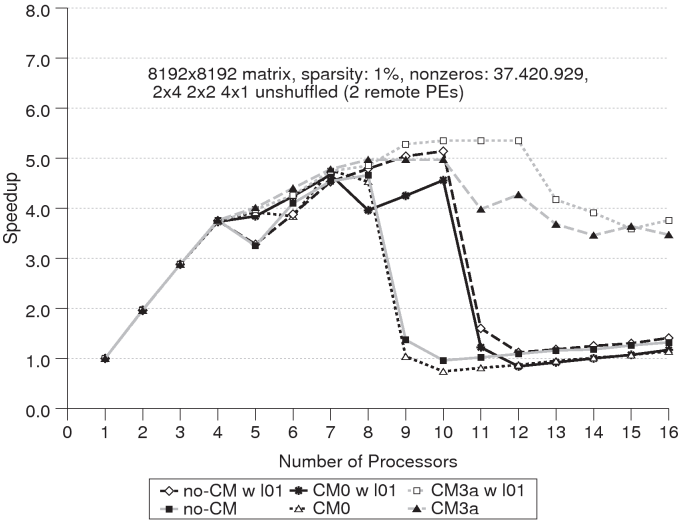


Figure 4: Speedups for the partly remote heterogeneous configuration using *different cost models*

5 Related Work

HeteroMPI [LR06] integrates cost modelling with MPI to exploit heterogeneous clusters more efficiently by automating the optimal selection and placement of a group of processes according to both architecture and application characteristics. It defines an abstraction layer on top of MPI and requires the user to provide a performance model that incorporates the number of PEs, their relative speed and the speed of communication links in terms of latency and bandwidth, as well as the volume of computations, the volume of data to be transferred between each pair of processes in the group, and the order of execution of computations and communications. However, to create such a model the user needs intricate knowledge of the target architecture and of the application. Additionally, familiarity with a special performance modelling language is required, steepening the initial learning curve. Since HeteroMPI employs a partially static model, it may not be suitable to support applications with irregular parallelism. By contrast, we focus on using an architecture-aware but application-agnostic cost model to improve performance portability.

The design of our cost model is based on results in [AMT11], which achieves good speedups using a simple cost model that does not take into account RAM and latency for two applications[4] expressed in terms of skeletons and run on a heterogeneous architecture. While the focus of that work is on hybrid parallel programming models for algorithmic skeletons, we study the behaviour on a standalone parallel application.

6 Conclusion

This paper investigates the feasibility and performance of a programming model that combines high-level cost modelling with low-level, explicit control of coordination using C and MPI to achieve performance portability on modern parallel hardware. More specifically, formal, architecture-aware, static cost models are used to enhance parallel performance on a range of clusters of multi-core machines — an emerging and important class of parallel architectures. We quantify the impact of heterogeneity by comparing the performance of parallel sparse matrix multiplication on a range of clusters of multi-cores: on the most heterogeneous setup with 16 PEs the speedup is merely about half of the speedup on a homogeneous setup. By making decisions on the size and placement of computations using an architecture-aware cost model, the application doesn't have to be changed when moving to a different parallel machine.

Our example program achieves good performance portability even on the most heterogeneous configuration: the speedup on 16 PEs increases from 6.2 (without any cost model) to 9.1 (using a simple, static cost model), thus nearly matching the performance on a homogeneous architecture. Based on one application, these results on a range of configurations with increasing heterogeneity indicate that using a simple cost model parameterised by PE speed, memory size and cache size, can already achieve a high performance portability. Although measured only for up to 16 PEs, the results indicate that the scalability is improved beyond the available number of PEs on the all-local setups. Since there is no need for monitoring or time-consuming calculations, our static cost model has negligible overhead in the still important homogeneous, flat case. Our results also show that using a shuffled placement strategy, which evenly distributes computations across multi-cores rather than fully exploiting individual multi-cores, provides higher predictability whilst moderately impacting performance. Given the trends towards large-scale many-core architectures, this observation is relevant, considering that with increasing numbers of cores per machine, shared-memory access will become a bottleneck, and therefore exploiting all the available cores on one machine may no longer be an optimal strategy. Additionally, our results affirm that latency is critical for load balancing decisions on partly remote setups.

In summary, our findings suggest that architecture-aware cost modelling facilitates the transition to high-level parallel programming improving performance portability by transferring the responsibility for specific policies controlling parallelism from the programmer to an informed cost model. The need of manual code alterations is eliminated as target architecture is exchanged.

[4] computing the Euler Totient sum and the SIFT image processing algorithm

Limitations: The main limitation of our approach to cost modelling is that we do not incorporate any application-specific information or dynamic load information. Although this design decision reduces the overhead associated with the cost model, it limits the flexibility of the system by missing opportunities of dynamic reconfiguration of the parallel computation, which could improve performance in highly dynamic environments. Moreover, we rely on heuristics and emphasise the need of a suitable high-level parallel computational model to allow for more systematic cost modelling.

Future Work: Although we have demonstrated the feasibility of using architecture-aware cost models, more work is needed to generalise the results to other application domains. Including further architectural parameters as well as application specific ones (e.g. cache hit patterns, communication patterns) and dynamic load information in the cost model has the potential for more accurate prediction, which pays off especially in heterogeneous and hierarchical setups. A further interesting direction is the investigation of the use of more sophisticated cost models within a run-time system or as part of algorithmic skeletons in accordance with a semi-implicit approach to parallel programming on the more heterogeneous architectures comprising clusters of GPU-enabled multi-cores.

Acknowledgements

We thank Khari Armih, Artur Andrzejak, and Miroslaw Malek for inspiring discussions and the anonymous reviewers for the helpful comments.

References

[AISS95] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model. In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, July 1995.

[AMT11] Khari Armih, Greg J. Michaelson, and Phil W. Trinder. Cache Size in a Cost Model for Heterogeneous Skeletons. In *Proceedings of the 5th ACM SIGPLAN Workshop on High-Level Parallel Programming and Applications*, September 2011.

[Asa06] Krste Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[BGK03] Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann. Cost Optimality and Predictability of Parallel Programming with Skeletons. *Parallel Processing Letters*, 13(4):575–587, 2003.

[BP06] Jose Luis Bosque and Luis Pastor. A Parallel Computational Model for Heterogeneous Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1390–1400, December 2006.

[CKP+93] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[Dar93] John Darlington et al. Parallel Programming Using Skeleton Functions. In *Proceedings of the 5th International Conference on Parallel Architectures and Languages*, volume 694 of *LNCS*, pages 146–160. Springer, June 1993.

[Fly66] Michael J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Symposium on the Theory of Computing*, pages 114–118, 1978.

[Gor04] Sergei Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, January 2004.

[LMR95] Zhiyong Li, Peter H. Mills, and John H. Reif. Models and Resource Metrics for Parallel and Distributed Computation. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 133–143. IEEE Computer Society Press, 1995.

[LR06] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.

[LTB01] H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4), December 2001. Selected papers from HLPP'01 — International Workshop on High-level Parallel Programming and Applications, Orleans, France, 26-27 March, 2001.

[Qui04] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994.

[ST98] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):124–169, June 1998.

[TCH+11] Phil W. Trinder, Murray I. Cole, Kevin Hammond, Hans-Wolfgang Loidl, and Greg J. Michaelson. Resource Analysis for Parallel and Distributed Coordination. *Concurrency: Practice and Experience*, 2011. Accepted for publication.

[THLJ98] Phil W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Func. Prog.*, 8(1):23–60, January 1998.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of ACM*, 33(8):103–111, August 1990.

[Val11] Leslie G. Valiant. A Bridging Model for Multi-Core Computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011.

Exploring Software Variance with Hypermodelling
An exemplary approach

Tim Frey, Veit Köppen

Institut für Technische und Betriebliche Informationssysteme
Otto-von-Guericke University
Magdeburg, Germany
tim.frey@tim-frey.com; veit.koeppen@iti.cs.uni-magdeburg.de

Abstract: Framework manufacturers face the challenge to determine which parts of frameworks are used and varied. Application developers want to know on which framework elements their application is depending. Currently, programs need to be parsed to extract information about framework usage what consumes time and effort and makes information mining inflexible. Hypermodelling utilizes Data Warehouse technologies for source code investigations to overcome the current limitations. In this paper, we demonstrate that Hypermodelling is suitable to explore software variance. We present reports based on real application data of one project example to reveal multiple facts about the software variance. We show visualizations at different granularity levels. This supports our theory that Hypermodelling can be used to explore software variance in an easy way.

1 Introduction

Source code reuse is one of the key concepts in modern application development. Programmers develop applications by using already encoded functionalities of frameworks. Commonly, object-oriented inheritance of framework classes or interfaces is used to employ predefined functionalities [Sc97]. Without loss of generality, we restrict ourselves to Java as programming language in our examples. The types that get inherited are in Java called superclasses and superinterfaces. In the following, we refer to both as supertypes. Developers vary such supertypes by adding or altering functionalities in subclasses, what we call children or inheritors [BMM10]. Hence, all distinct inheritors, i.e. extenders or implementers, create a huge amount of diversification of the original superclasses of the framework.

Framework manufacturers and application developers face the challenge to understand how, where, and which framework elements are used. For instance, there is the desire to know which methods are commonly implemented and overridden in subclasses. The implemented methods point out a variation of the originally offered methods by the frameworks supertype. Developers want to point out how different frameworks are used within an application. The diversity of inherited classes and implemented methods gives developers details on the variation of the frameworks used within the application. The variation is important to draw conclusions on dependencies within a framework.

However, researchers put a lot of effort on source code mining [BMM10, GHH09, ZZ11]. Source code mining is about extracting facts of source code and associated artifacts to analyze them via machine learning algorithms [BMM10]. A main issue in mining is that there is no standardized infrastructure to extract facts. Thus, before mining starts, facts need to be specified and extracted. This consumes a lot of time and effort. In order to advance source code analysis, we develop the Hypermodelling approach, see for instance [Fr11, FKS11, Fr12]. Hypermodelling utilizes Data Warehouse (DW) technologies to enable and infrastructure an analysis of source code with Online Analytical Processing (OLAP) queries [HKP11]. This enables fast and flexible investigations of source code through queries without having a complex fact-extract scenario.

In our former work on Hypermodelling we visualized rudimentary queries on annotations and inheritance [FKS11]. In this paper, we show that Hypermodelling can be used to explore diverse facts on variances of frameworks. We depict reports and visualizations that reveal facts on software variances at different granularity levels. Hence, our contribution is to demonstrate that Hypermodelling can be used to do investigations on software variance. Our reports give first clues what types of queries can be composed with Hypermodelling. This supports our hypothesis that DW mechanisms can be beneficial for software investigations.

First, we give some background information about Hypermodelling and our used exemplarily used project. In the following, we present and discuss various reports. Then, we describe related approaches and point out their similarity to the usage of our Hypermodelling approach. Lastly, we draw conclusions and give an outlook to future work.

2 Background

In this section, we recap the most important facts of Hypermodelling. Then, we describe details about the project that served as base to generate reports with our Hypermodelling approach.

2.1 Hypermodelling

Hypermodelling is developed on the idea to program analysis and DW. A more detailed description is available in our former work [Fr11, FKS11, Fr12].[1] DW systems are an integrative component in business computing [In05]. They are used to assemble data of different sources together. The integrated data are arranged into multi-dimensional data structures, i.e. so called cubes, which serve as base for queries [HKP11]. The queries allow aggregating different relations and hierarchies that occur in the data. For instance, the revenues for a specific salesman in a specific area can be computed for a given time period. Thereby, this query aggregates the region, the salesman and the time in relation

[1]Http://hypermodelling.com

to revenue indicator. Likewise, hierarchies can be abstracted. For instance, the region of the salesman can be split into continents, countries and counties and the aggregates are associated with the distinct revenues for those. Likewise, this can be done for other hierarchies, like customer group, year, or department. Generally, the idea is that different aggregations enable detailed investigations.

With Hypermodelling we introduce the idea that programming elements, like annotations or classes, are similar to the data that is used within a DW. For instance, classes are defined within a package hierarchy. Annotations are associated with classes and their members. They are also defined in their own package. This is like the association of a salesman to a region, time period, and revenues. The hierarchies in code are similar to hierarchies of region or time. All together, we load source code into a DW and realize the associations of classes, their inheritance, packages, and annotations as a DW cube. This enables us to built queries on top of this cube and it creates the possibility to compute different aggregations for code. For this paper, we combined the cubes that were presented in our former work [FKS11] and built queries on it. The results of queries on this cube are the reports that we present in this paper.

2.2 The Alfresco Project

In order to have valid and real world application data, we downloaded a head revision of the Alfresco project[2] from the corresponding version control system. Alfresco is a popular open source content management system, written in Java. Then, we loaded two subprojects (core and repository) of the whole Alfresco project into our DW. Thereby, we also created types, coming out of other projects or the Java standard that were referenced by classes of the Alfresco project. The following measures were computed by queries against the processed cubes that were generated out of the loaded data. The total amount of types in the Alfresco package is 3,651. These 3,651 types consist out of 2,866 classes, 658 interfaces, 124 enumerations, and 3 annotations. The total amount of members is 42,823. These members are divided into three different annotation parameters, 13,694 fields and 29,126 methods. It is interesting that the interfaces declared in the Alfresco project define 4,612 members from which 3,770 are methods and 842 are fields. This means a huge amount of constants is defined by interfaces. Classes define 25,279 methods and 12,852 fields. Enumerations define 77 methods and zero fields. The three Annotations defined in the project define three parameters altogether. In the following, we use this project to exemplarily present reports with the Hypermodelling approach. Note, other projects could be used similarly, e.g., [FKS11].

3 Hypermodelling Inheriting Variance

In the following, we present different reports based on the loaded project. First, we provide an overview on project inheritance statistics. Then, we discuss how and which packages get varied. Afterwards, we drill down to method and type level.

[2] http://www.alfresco.com

3.1 Inheritance Overview

We present the amount of total inherited types and distinct inherited types in Table 1. Types that are inherited are divided into superclasses and superinterfaces. The total inherited types are the amount of children classes. The distinct types depict how many different types serve as supertypes. The supertype and superinterface row indicates the kinds of used supertypes. It can easily been seen that more classes are extended (2,798) than interfaces are implemented (1,435). We compare the distinct types and see that the situation is different for those. The diversity of classes is less as that for interfaces, but they get more often extended than interfaces get implemented.

Table 1: Which kind of type is used as a supertype

	Total inherited types	Distinct inherited types	Implementation ratio (Total/Distinct)
Superclasses	2,798	377	7.42
Superinterfaces	1,435	625	2.3
Total	4,233	1,002	4.22

We use the indicator implementation ratio. With it, we enable a more concrete comparison of distinct supertypes to the amount of subclassed types: The indicator enables to compare the amount of distinct types with the total inherited types in one number. Since classes or interfaces that implement or extend other types add commonly new functionality to the existing one, they vary the original types. Therefore, we see figure of implementation ratio as one indicator to measure the variance of supertypes.

The figure of the implementation ratio is useful to depict a standard variation. A high or low value is a first indicator how intense supertypes are varied. When the figure is at a high value, we can conclude that the defined standard of the supertype gets aligned a lot. This means that the same types are often implemented or extended. Every time a type is used it gets adapted to the specific application needs. This way, developers have a starting point for further investigations to determine the types, responsible for high variance. This is especially useful, when varied supertypes are updated. If supertypes with a high ratio get updated, many children will be dependent on them. Thus, it is recommendable to investigate how the children adjust the supertype to keep its future version compatible to the inheritors. Furthermore, framework manufacturers can use the ratio indicator to determine which types are mostly adapted. With that information they can investigate how most developers varied the types. It is possible to depict if there is a common use or functionality in the extending types. If so, this functionality can be encapsulated into a new version of the supertype.

Additionally, we provide another perspective in Table 2. There, we show what type of children inherits a supertype. As before, the figure of the implementation ratio is presented. It is important to note that this table does not distinguish between type of the

supertype, i.e., if the supertype is a class or an interface.[3] In total, most of the implementing types are classes, followed by interfaces, and enumerations. Likewise, this is the same for the distinct inherited types.

Table 2: Which kind of types are the children types

	Total inherited	Distinct inherited	Implementation ratio (Total/Distinct)
Implementing classes	3934	967	4,07
Implementing interfaces	171	75	2,28
Implementing enumerations	128	4	32
Total	4233	1002	4,22

The implementation ratio is very interesting in this table. We assume indicators are led by classes, but we were surprised by enumerations. We can see that 128 enumerations exist in the project and extend only four different supertypes. Since the number of implementing interfaces is in total not much higher (171), we do a drill down (refinement in the hierarchy level). There, we see that 104 children are directly derived from the *Enum* class. 18 from the *EnumLabel* enum, two from the *Comparable* interface and two from *Serializable* interface. Enumerations that have no supertype are derived, by the Java language specification, from the "pure" enum supertype. Therefore, the enumerations show a higher diversity than classes or interfaces. However, we learn from the abstract views of Table 1 and Table 2 that the implementation ratio of classes exceeds interfaces. This means: Classes seem to have a higher variance then interfaces.

3.2 Inheritance and Packages

We present a more detailed view of the variance of inherited types at the package level in this section. First, we describe the source data that we used to generate various diagrams. Secondly, we present an analysis which package is used most in total numbers. Afterwards, the same analysis is done for distinct numbers. Finally, we show an overview that depicts packages with a high implementation ratio.

3.2.1 Source Data

Table 3 presents an excerpt of the source data that we use to create our reports. The table shows a query for packages and inherited type indicators. The package hierarchy is expanded to a level where a developer can derive the original or meaning of the package. In total, we split the report into 38 packages that contained the 1,002 distinct supertypes.

We show, that the supertypes used within the project are from the Alfresco project itself. Nearly, half of the total used types is from org.alfresco. Likewise, over ¾ of the distinct types are out of the project itself. Therefore, we compute the implementation ratio without the project itself: It increases over 11.5.

Remarkably is also the ratio for the java.lang package. We perform a drill down into the

[3]We leaved this out to avoid a complex anything with anything table.

used types and discover that 1,195 classes inherit the object class. Likewise, there are enumerations extending the "main" *Enum* and other classes that are extending language functionalities. We conclude that also the java.lang package should be excluded. When it is excluded the average implementation ratio drops down to 4.68. These operations are comparable to drill down, roll-up, and drill-across in the DW domain. This illustrates the application of the Hypermodelling approach. Note that all indicators are directly computed within the cube and only "navigation" is required.

Table 3: Exemplary source data of inheritance at the package level

Package hierarchy	Total inherited types	Distinct inherited types	Implementation ratio (Total/Distinct)
org.alfresco	1,972	808	2.441
java.lang	1,404	11	127.64
org.springframework	214	40	5.35
junit.framework	184	2	92.0
org.antlr	87	6	14.5
Total	4,233	1,002	4.22
Without org.alfresco	2,261	194	11:65
Without org.alfresco and java.lang	857	183	4.68

3.2.2 Dependency on Total Packages

In Figure 1, we visualize dependencies of the Alfresco project on super types. As described before, the java.lang package and the Alfresco package are dominant, therefore we kept them aside. The three emphasized slices, java.io, java.util, and javax.jcr visualize dependencies on java.lang itself. We can also identify a heavy type usage of the spring framework. Furthermore, we can conclude that the index code base contains unit tests, since nearly a quarter is depending on the junit framework.

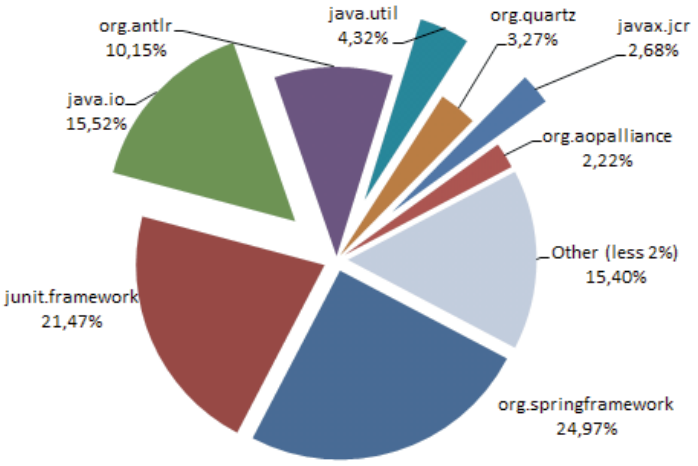


Figure 1: Dependency on total used types

3.2.3 Dependency on Distinct Packages

Figure 2 visualizes the distinct used types within the project. The two main used packages are the Spring framework and the javax.jcr package. The jcr package is widely used, because the analyzed application is a content management system.

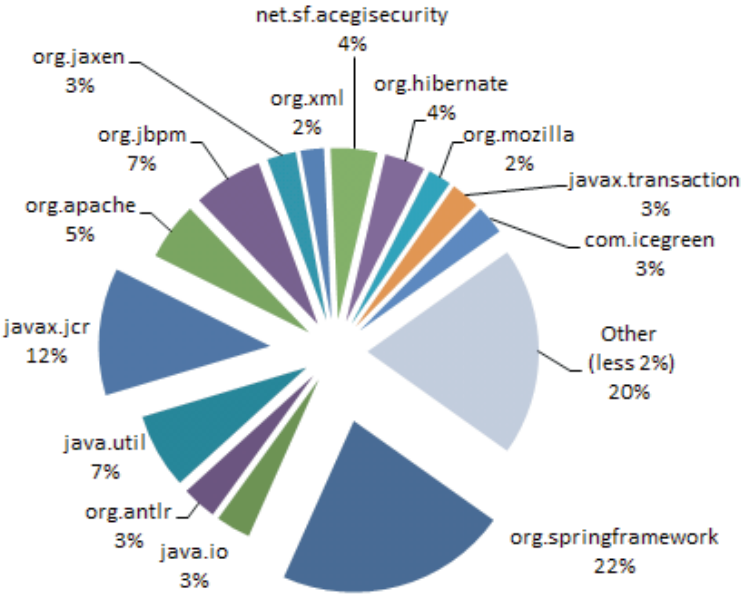


Figure 2: Dependency on distinct types

Jcr contains the java content repository types. The Spring framework is used because it is a web and Java enterprise application. All together it is interesting that the chart of distinct types differs from Figure 1. For instance, the Junit framework that is extensively used does not occur, because not many different types of it seem to be used.

3.2.4 Package Variance

In order to visualize variances of the packages we need a different kind of graphic that does not limit the visualization of one variance on the costs of another. Therefore, we use a bar chart. Figure 3 shows a bar chart of the implementation ratio of the different packages in the project. We only show three packages with the ratio of one as proxy out of space issues. Furthermore, we excluded the java.lang package and the junit.framework package out of their high indicator value. Such outliers would crush the diagram view and differences would hard to be recognized. Lastly, we show the average implementation ratio line with value 4.68 (mean).

Figure 3 enables to depict the packages containing types that have a high variance in their implementation. Therefore, we can see that java.io has a high variation. Likewise, not much distinct types are used often of the aopalliance package. Probably, these are the aspect enhancements. Antlr provides a grammar parser. The high ratio indicates that

various grammars are parsed. Additionally, quartz offers functionality to built timers into the application, what indicates that the Alfresco application uses timers.

Lastly, it is interesting that ibatis, a database table-class mapper, and the Spring framework are used at an average ratio. We interpret this out of the circumstance that spring and ibatis are large frameworks. Thus, the total amount in the package of spring and ibatis is much larger what leads to the different ratio.[4]

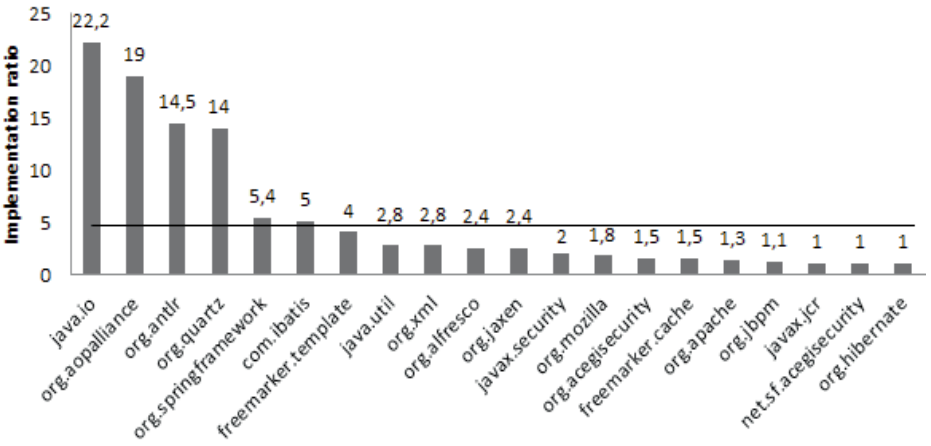


Figure 3: Implementation ratio for packages

However, like all other figures, the data of Figure 3 has been retrieved with a simple multi-dimensional expression (MDX-Queries).[5] The query language is declarative and especially created to query multi-dimensional data easily. In order to give an impression about the used query language, we present exemplarily the query that served as foundation to generate Figure 3:

```
with member [Implementation Ratio] as
 [Measures].[Type Inheritance Count] /
 (count( ([Parent - Type].[Id].children, [Measures].[Type Inheritance Count]) ,
                                                    EXCLUDEEMPTY)

SELECT {[Implementation Ratio]} ON COLUMNS ,
{[Parent - Package hierarchy].[Parent].[All].children} ON ROWS
FROM [Inheritance-Annotations-Cube]
WHERE ([Package Hierarchy].[Parent].&['org.alfresco'])
```

Listing 1: The query of Figure 3

[4]At this point it would be necessary to compare the children packages of the frameworks with the packages of quartz and so on. We do not show them, to stick to the main focus to demonstrate exemplary applications of our approach.

[5]For further information about the MDX language, see the language reference: http://msdn.microsoft.com/en-us/library/ms145595.aspx and the XMLA standard: http://xmlforanalysis.com

In short, the query in Listing 1 computes dynamically the indicator *[Implementation Ratio]*. The quotient is computed by the aggregation of the *[Measures].[Type Inheritance Count])* as dividend and the distinct count of the amount of *[Measures].[Type Inheritance Count])* measures as divisor. The *SELECT* applies the computation to different packages of the corresponding cube (*FROM [Inheritance-Annotations-Cube]*). Lastly, all other packages, except the *org.alfresco* ones, are excluded as inheritors from the computation (*WHERE ([Package Hierarchy].[Parent].&['org.alfresco'])*).

3.3 Method Variance Drilldown

With the information that is uncovered in the previous section we know that a huge amount of types of the Spring framework is used within the application. Furthermore, a few types out of Junit are intensively used. We present the top used types of the two packages in Table 4. Three columns are already known and show the origin package of supertypes, supertype name, and total inheriting types. We introduce a new indicator of distinct method names in inheriting types to credit our drill down. This measure counts the distinct method names occurring in types. This way, we see how many types subclass a supertype and how much a supertype is varied in total.[6]

Table 4: Excerpt of the top used types of the spring and junit package

Supertype package	Supertypename	Distinct method names occurring in inheriters	Total Inheriting
org.springframework	ApplicationContextAware	580	26
	InitializingBean	578	56
	AbstractLifecycleBean	351	42
	ApplicationListener	175	7
	… in total 40 types	… in total 2,374	… in total 214
junit.framework	TestCase	1326	170
	TestSuite	2	14
	Total 2 types	Total ,1328	Total: 184

3.3.1 Method Variance

In order to explore the concrete usage of the InitializingBean and TestCase in detail, we generate a report for supertypes and method names of their children. We present an excerpt of the report, sorted after the most implemented method name, in Table 5. The column distinct inherited types shows the amount of children types that use the same method name. For instance 55 of 56 children of the InitializingBean use the name afterPropertiesSet for a method.

[6]Clearly, this enables to create an abstract indicator similar to the implementation ratio at the type level. We do not show it to focus on further possibilities to stick new ways of exploration.

We also see that a huge amount of 578 method names for the InitializingBean and 1,326 method names for the TestCase turn up in their subclasses. Currently, we can depict common method names from the table. However, we cannot derive if the intention of a developer is actually to override a method of a supertype. Therefore, we show in the following section how we enhance this approach with source code metadata to get common methods for improved reporting.

Table 5: Excerpt of the most used method names in supertypes children

Supertype package	Supertype	Methodname	Distinct inherited types (i.e.: extenders sharing the method name)
org. springframework	InitializingBean	afterPropertiesSet	55
		setAuthenticationService	10
		setNodeService	9
		setBeanName	7
		destroy	6
		… in total 578 distinct member names	Total distinct types: 56
junit.framework	TestCase	setUp	139
		tearDown	73
		testSetUp	35
		create	6
		… in total 1326 distinct member names	Total distinct types: 184

3.3.2 Slice by @Override

Before, we address the problem of the immense amount of different method names within the children of a class, we propose to solve this problem by the usage of metadata annotations that are defined within the Java programming language[7]. Java's annotations allow developers to enrich source code with various structural and logical information. Specifically, the @Override annotation out of the Java standard is of interest in our current case. The Java standard describes the Annotation usage as follows:

@Override
"Indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message."[8]

This means that the override annotation can be used, but there is no compulsion to do so. Thus, @Overwride marked methods override a method for sure, but overriding can also take place without this annotation. In fact, modern development environments, like Eclipse[9] remind programmers through messages and markers to use the annotation.

[7] http://jcp.org/ja/jsr/detail?id=175
[8] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Override.html
[9] http://eclipse.org

Hence, we assume that most overriding methods are marked with the annotation in modern programs. We query how often the annotation is occurring on methods of the Alfresco project. 2,067 methods are annotated with this annotation. In total 3,139 annotations are occurring at methods of the analyzed project. Therefore, we assume that we can use annotation to specialize our former query.

3.3.3 The Report Filtered by @Override

We present the modified query result for the @Override marked methods of children of TestCase and InitializingBean in Table 6. In contrast to our expectation, the most overridden methods of InitializingBean are not marked @Override. In fact, the annotation is used scarcely, like the distinct inherited type numbers indicate. Nevertheless, we get the methods that are intended to be overridden by developers. The most interesting fact about the distribution of the @Override annotation is that 21 distinct types seem to use the override annotation at different methods. Only at four methods the override annotation is occurring at the same method name. In contrast to the *InitializingBean* children, project developers seem to have been more sincere about annotating test cases with the annotation. 113 types out of 184 in total got at least one annotation.

Lastly, the result gets even more interesting, when regarding at the Javadoc of the two parent classes. *InitializingBean* defines only one method: afterpropertiesSet[10]. TestCase[11] defines the two @Override marked methods. Hence, InitializingBean extenders override methods out of the supertypes *supertype*.

Table 6: Excerpt of the most used method names with @Override

Supertype package	**Supertype**	**Methodname**	**Distinct inherited types (In this case: extenders sharing the method name**
org. springframework	InitializingBean	equals	2
		Implementation AllowsGuestLogin	2
		toString	2
		transformInternal	2
		afterPropertiesSet	1
		… in total 21 distinct method	Total distinct types that got an @Override Annotation
junit.framework	TestCase	setUp	95
		tearDown	53
		Total: 2	Total distinct types that got an @Override Annotation: 113

3.3.4 An Additional Type-Method Report Filtered by @Override

Surprised, by the previous results of overridden methods, we depict further inherited types that have a huge amount of overridden methods. Our choice for the concrete types is made out of their name, indicating what they are used for. Additionally, we investigate

[10] http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/beans/factory/InitializingBean.html
[11] http://www.junit.org/apidocs/junit/framework/TestCase.html

the methods that are defined by the supertype themselves and compared those method names with the ones defined by the inheritors. In the following, we present different types shown in Table 7.

Serializable is a well known Java flag interface. However, the Serializable interface does not define any methods. It is interesting that inheritors seem to be attracted to override toString, equals, and hashCode that are originally coming out of the object class. Thus, if we ask an Alfresco programmer which methods serializable class needs, he will probably answer: "toString, equals, and hashCode".

The *TransactionListenerAdapter* offers five methods whereby three get overridden by subclasses. All together, it seems if the "after the transaction methods" are more common to be adjusted by the application logic. Methods of the *Object* class get overridden what brings up the idea that those should maybe excluded in a query if the focus is purely set to be on other types.

The *EntityLookupCallbackDAOAdapter* is highly customized within the application. This is quite logical, since data access objects (DAO) are used to persist different classes of the domain model of an application. All together is seems that the delete methods seem to work more generically than the retrial or update methods since they do not get adapted this much.

Table 7: Common inherited types and @Override marked methods in children

Supertype	Methodname	Overriding Types	Exists in Supertype?	Additional methods in supertype
Serializable	toString	42	no	
	equals	33	no	
	hashCode	32	no	
	getInviteeEmail	2	no	
	getInviteeFirstName	2	no	
	getInviteeLastName	2	no	
	clone	2	no	
	getInvitationType	2	no	
	initialiseHandler	1	no	
Transaction Listener Adapter	afterCommit	21	yes	flush
	afterRollback	11	yes	
	beforeCommit	6	yes	beforeCompletion
	equals	4	no	
	hashCode	3	no	
	toString	1	no	
Entity Lookup Callback DAOAdaptor	updateValue	5	yes	deleteByValue
	findByValue	5	yes	
	getValueKey	5	yes	
	deleteByKey	2	yes	

3.3.4 Method Variance Roll Up Enriched with @Override

Overall, we learn that override annotations are used within the project. Even if they are not applied consequently the amount of 2,067 annotated methods is immense. Hence, we present a roll up to all types that contain overridden methods.

Report Source Data

Table 8 shows an excerpt of the report that we generate through a roll up from the Spring and Junit package. The distinct inheriting types specify the amount of types that extend or implement the supertype. The overridden methods specify how many methods of the children carry the @Override annotation. The total methods amount is sliced by the method names (distinct method names). The C and I indicate if the supertype is a class or a method. Lastly, we introduce the overridden method implementation ratio. This ratio indicates the average amount of overridden methods per type. A higher ratio means that more methods in total of the type get overridden and the variation of the children is probably higher. Such, framework manufacturers can determine the types that get varied the most by developers.

Table 8: Excerpt of the most used method names with @Override

Class/ Interface	Type name	Distinct inheriting types	Total overridden methods	Distinct method names	Implementation ratio (Total methods/ Distinct types)
C	Object	184	353	47	1,918
C	TestCase	113	150	2	1,327
I	Serializable	42	88	9	2,095
C	Transaction ListenerAdapter	26	46	6	1,769

Relations with Figures

We present how different figures of the report relate to each other in Figure 4. We eliminate outliers from the source report data to have a clear distribution of the points of the data. We exclude the *Object* class that serves as parent class, if no other is given and *TestCase* out of its extreme usage. Additionally, we exclude types having less children then three to avoid an intense cluster. All together, 63 adequate values left as base for visualization. Founded on those values, we present the charts to have a first look for trends that maybe excel out of the diverse variance figures.

Figure 4a shows the relation of the total amount of overridden methods to inheriting types. We see that the amount of overridden methods grows with the types. We added a linear trend based on these values. The big picture is that a linear growth seems to exist. We conclude that it seems when types inherit a supertype they also override methods.

Figure 4b shows the relation of inheriting types to distinct overridden methods within the children types. We assume that the circled space marks a cluster. This would be logical, since we guess that the amount of methods of a type is in most cases within a certain range. However, there are outliers in the X and Y direction. The Y direction seems to be the case for a tiny amount of inheritors, what means: A large amount of methods of supertypes is overridden, but not many other types inherit from the same supertype. In

contrast to the Y direction, the X direction shows many inheritors that map to a small amount of distinct overridden method names.

We present the implementation ratio distribution for various types in Figure 4c. The inheritance ratio values are sequenced following the amount of the inheritors of a type. For us it seems that the ratio values are mostly arranged within a certain range and a few outliers exist. All the more, the median of all values is arranged at 3.33, what is on top of the guessed range.

Out of the guessed range of the ratio before, we reorder the ratio values in Figure 4d. The ratio expresses the average overridden methods for a type. Since distinct method names are not expressed in this relation, we order the ratio values following the amount of types in Figure 4d. The outliers seem to grow in value in relation to methods. In the graphic we show a linear trend for this. However, the trend is not supported by many values to make any predictions. There exists a cluster that we emphasize through a circle. This cluster seems to map the prior mentioned range in Figure 4c to one spot. Thus, we assume all our values seem to have a relation and be dependent on each other. Though, more detailed investigations are required to verify our assumptions we depict from the diagrams.

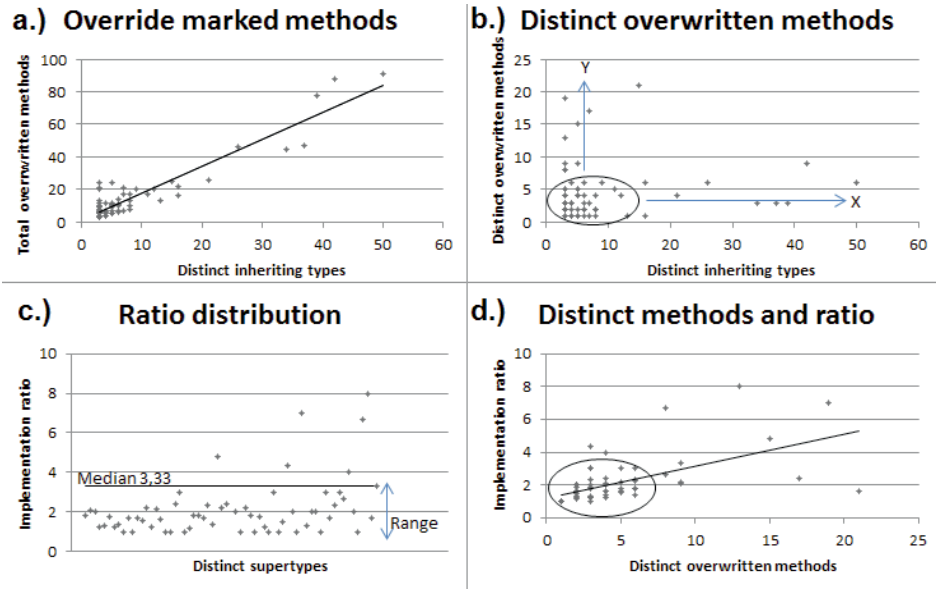


Figure 4: Relations of different inheritance indicators

4 Related Work

Related to the idea of Hypermodelling is the area of source code mining [GHH09]. Our approach to study software variance based on subclassing is especially near the idea to mine subclassing directives from existing code [BMM10]. Mining subclassing directives

is about to uncover which methods are most used by extenders of classes to create a recommender system for developers to assist them in the integrated development environment. In order to generate these directives, the code is parsed and facts get extracted into a binary matrix by static analysis. Then, an algorithm is applied to generate directives. Our Hypermodelling approach overcomes the limitation to do a static analysis to uncover facts. Facts can be revealed via a query from the DW. Furthermore, we show that Hypermodelling is not limited to one fixed fact set: Plenty of facts can be extracted and visualized. Therefore, we already show that there exist more complex subclassing directives. For instance, methods that extend the *Serializable* interface often override three different methods of the object class. The method presented in the mining subclasses paper [BMM10] would not uncover this information. Moreover, our reports showed that different kinds of slices can be built to investigate source code and to visualize them to gain more knowledge. However, we do not see Hypermodelling in competition to such traditional mining approaches. Rather, Hypermodelling as integrative component to access the different facts about source code.

The usage of Business Intelligence is already mentioned in the literature. Since Business Intelligence is often used as a term for all associated technologies, like OLAP or DW this can be seen related. Software Intelligence (SI) [HX10] describes this idea on an abstract level. In spite of the relation to SI, Hypermodelling is still unique. SI neither proposes multi-dimensional models nor takes the fact into account how DW technology can be used. Lastly, no reports about code are shown and mentioned.

In [Pa03] a method to control a software development project with metrics is described. There, a Metrics Warehouse is mentioned. Since the measures computed in the paper represent somehow metrics of source code this seems first to be related. However, the metrics described are economic project figures, not representing code metrics. Even though, we think the idea to steer a project based on reports similar to the ones presented in this paper can be an interesting trail for the future.

5 Conclusions and Future Work

We presented different reports based on the Alfresco application. We explored different types of parent – children relations of software variance. We showed a difference in the variance of classes and interfaces at the exemplarily project level. Through drill downs we depicted variance differences of frameworks. Furthermore, we depicted two different figures to express the dependency on a framework. Investigations about the method variance of subclasses illuminated the fact: The methods of subclasses exceed probably a pure relation to supertypes methods. Commonly, there exists an undercurrent that methods of a supertypes subclasses share method names of another supertype. Thereby, we also demonstrated that Hypermodelling reports can be discriminated with dynamic factors, like metadata annotations, to enhance results. We showed diagrams that supported the following: The indicators, used within the paper to draw conclusions about inheritance variance seem to be linked. Lastly, we presented related work that described research that has potentially synergy effects with Hypermodelling.

However, the main intention of this paper is to present how the Hypermodelling approach can be applied to investigate software variance. We described at various points that we adjusted the queries. Thus, the key figure is that all statistics shown in this paper can be redone with queries. Those can easily be adjusted to the specific need. The diversity of the reports showed that we were capable to use the Hypermodelling approach to reveal facts about software variance that would be complex and expensive to be uncovered otherwise. The diversity of the reports was only possible because Hypermodelling is designed to uncover such facts with queries. All together, we see the flexibility of Hypermodelling as support that Hypermodelling can advance source code mining.

We see future work divided into two different trails. First, we think it is necessary to advance the Hypermodelling approach itself. That means that more facts and different cubes need to be built to enable other investigation of source code. Currently, we are planning extensions like caller-callee relationships. Secondly, we see the need to load a larger code base and different projects into the same warehouse to intensify and evaluate our investigations. Thereby, we see the necessity to derive standard reports for software variance. In order to create those, we see the presented reports as origin for further developments.

Nonetheless, Hypermodelling is mainly about the infrastructure to explore source code with queries. Therefore, we feel the emerging need to work together with source code mining experts and widen the possibilities of source code mining.

References

[Sc97] H.-A. Schmid. Systematic Framework Design by Generalization. Communications of the ACM, Vol.40/No.10, Oct. 97, pp. 48-51

[BMM10] M. Bruch, M. Mezini, and M. Monperrus, “Mining subclassing directives to improve framework reuse,” in Proc. Working Conference on Mining Software Repositories. IEEE Computer Society, 2010, pp. 141–150.

[GHH09] M. W. Godfrey, A. E. Hassan, J. D. Herbsleb et al.. Future of mining software archives: A roundtable. IEEE. 2009

[ZZ11] J. Sliwerski, T. Zimmermann, A. Zeller. Don't Program on Fridays! How to Locate Fix-Inducing Changes. In Proceedings of the 7th Workshop Software Reengineering, 2005

[Fr11] T. Frey. Vorschlag Hypermodelling: Data Warehousing für Quelltext. 23rd GI Workshop on Foundations of Databases. CEUR-WS, pages. 2011. pp.55-60.

[FKS11] T. Frey, V. Köppen, G. Saake. Hypermodelling - Introducing Multi-dimensional Concern Reverse Engineering. In 2nd International ACM/GI Workshop on Digital Engineering (IWDE), Magdeburg, Germany, 2011. http://hypermodelling.com

[Fr12] T. Frey, Hypermodelling for Drag and Drop Concern Queries. Proceedings of Software Engineering 2010 (SE2012). Gesellschaft für Informatik (GI), Berlin, Germany. 2012

[In05] W. H. Inmon. Building the Data Warehouse. 4th ed., J.Wiley & Sons, New York. USA. 2005.

[HX10] A. E. Hassan, T. Xie. Software Intelligence: Future of Mining Software Engineering Data. In Proceedings of FSE/SDP Workshop on the Future of Software Engineering Research. Santa Fe. 2010

[Pa03] C. R. Pandian. Software metrics: a guide to planning, analysis and application. Auerbach Publications. 2003

[HKP11] J. Han, M. Kamber, J. Pei. Data Mining: Concepts and Techniques. Morgan Kaufmann. 3rd edition. 2011.

Zertifizierung und modellgetriebene Entwicklung sicherer Software (ZeMoSS)

ZeMoSS-Workshop: Zertifizierung und modellgetriebene Entwicklung sicherer Software

Michaela Huhn[1] Stefan Gerken[2] Carsten Rudolph[3]
[1] Institut für Informatik, Technische Universität Clausthal
[2] IC MOL RA R&D, Siemens AG, Braunschweig
[3] Fraunhofer Institut für Sichere Informationstechnologie (SIT), Darmstadt

Mit dem vielfältigen Einsatz softwaregesteuerter Produkte und Infrastrukturen in unserem Alltag wachsen die Software-Qualitätsanforderungen, sowohl bezüglich der funktionalen Sicherheit als auch bezüglich der Informationssicherheit. In der Luft- und Raumfahrt, der Energieerzeugung und im Schienenverkehr, aber auch in der Medizintechnik, der Automobiltechnik und bei mobilen Systemen sind Zertifizierung und der Nachweis der Sicherheit kritischer Systeme und softwarespezifische Sicherheitsnormen international etabliert und bindend. Zwei aktuelle, domänenübergreifende Herausforderungen bei der Entwicklung sicherer Software werden im Workshop adressiert:

- Modellgetriebene Entwicklung eingebetteter Software wird in der Industrie immer wichtiger und in ihren Grundlagen für höhere Sicherheitsanforderungsstufen seit langem in Sicherheitsnormen als dringend empfohlen klassifiziert. Da Normen aber immer nur die etablierten Regeln der Technik darstellen, entsteht für den Hersteller mit jedem Schritt hin zum erweiterten Einsatz modellgetriebener Methoden und Werkzeuge die Herausforderung, dass diese im Zertifizierungsprozess neu akzeptiert werden müssen, selbst wenn noch keine normativen Aussagen zu ihnen vorliegen.

- Durch die zunehmende Vernetzung kritischer Infrastrukturen und die Anbindung mobiler Endgeräte entstehen neue Risiken aus der wechselseitigen Abhängigkeit von Informationssicherheit und funktionaler Sicherheit. Hier sind eine Integration von Safety- und Security-Prozess und neue Methoden gefragt, die eine verbindende Behandlung von funktionaler Sicherheit und Informationssicherheit in der Risikoanalyse, der Entwicklung und beim Sicherheitsnachweis unterstützen.

Der ZeMoSS-Workshop soll den Austausch über offene Fragen und Lösungsansätze zu diesen Herausforderungen domänenübergreifend zwischen Teilnehmern aus Industrie und Forschung fördern.

Die Einladung zur Einreichung von Beiträgen führte zu sieben Einreichungen. Jede Einreichung wurde von mindestens drei Programmkomiteemitgliedern, assistiert durch externe Gutachter, begutachtet. Das Programmkomitee nahm fünf Beiträge zur Veröffentlichung im Tagungsband an, ein weiterer, nachträglich eingereichter Beitrag konnte für einen Vortrag zugelassen werden. Zusätzlich konnten wir mit Prof. Dr. Jens Braband, einen ausgewiesenen Experten für Sicherheit in der Bahntechnik, eine Key Note gewinnen:

Jens Braband: Security und Safety in der Eisenbahnsignaltechnik
Am Beispiel der Eisenbahnsignaltechnik wird die Wechselwirkung der Produkteigenschaften Safety und Security motiviert und erläutert. Dabei wird sowohl die Normenlandschaft beleuchtet als auch Beispiele für Anwendungen gezeigt. Es werden Erfahrungen bei der Erstellung eines Schutzprofils nach Common Criteria diskutiert.

Die folgenden Beiträge wurden angenommen:

- Kristian Beckers und Stephan Faßbender: Supporting the Context Establishment according to ISO 27005 using Patterns
- Marcus Mews und Steffen Helke: Towards Static Modular Software Verification
- Frank Ortmeier, Simon Struck und Michael Lipaczewski: Using model-based analysis in certification of critical software-intensive systems
- Patrick Werner, Stefan Gerken und Michaela Huhn: GSN$_M$-Edit: Ein modellgetriebener Editor für modulare GSN-Argumentationen
- Christian Wessel, Thorsten Humberg, Sven Wenzel und Jan Jürjens: Frühzeitige modellbasierte Risikoanalyse für mobile, verteilte Anwendungen

Wir möchten allen Beteiligten für ihren Beitrag zum Gelingen des ZeMoSS-Workshops danken: Unser Dank gilt den Autoren und dem eingeladenen Vortragenden für ihre Beiträge zum Programm. Wir danken den Programmkomiteemitgliedern und externen Gutachtern. Sie haben die Einreichungen in kurzer Zeit begutachtet und den Autoren nützliches Feedback gegeben. Das Programmkomitee, geleitet durch Michaela Huhn, Stefan Gerken und Carsten Rudolph, bestand aus:

Jan Jürjens	Technische Universität Dortmund
Volkmar Lotz	SAP AG
Marco Hauri	Ascom Systec AG
Hardi Hungar	Offis, Oldenburg
Stephan Katzenbeißer	Technische Universität Darmstadt
Lothar Pfeifer	Esterel Technologies
Heiko Saalbach	Movares Deutschland GmbH
Bernhard Schätz	fortiss, München

Schließlich möchten wir den lokalen Organisatoren der SE'12 für ihre Unterstützung im gesamten Prozess danken.

Wir hoffen, dass der ZeMoSS-Workshop für alle Teilnehmenden ein interessantes und stimulierendes Ereignis wird, aus dem sich neue Perspektiven für das Gebiet ergeben.

Februar 2012

Michaela Huhn (TU Clausthal)
Stefen Gerken (Siemens AG)
Carsten Rudolph (Fraunhofer SIT)
PC Chairs ZeMoSS'12

Supporting the Context Establishment according to ISO 27005 using Patterns

Kristian Beckers, Stephan Faßbender
paluno - The Ruhr Institute for Software Technology -
University of Duisburg-Essen, Germany

Kristian.Beckers,Stephan.Faßbender@paluno.uni-due.de

Abstract: The documentation of an information and communication system according to the requirements of the ISO 27005 standard is difficult, because the standard only provides sparse descriptions.

We propose the use of specific patterns for the ISO 27005 standard, which can be instantiated for any given information and communication system. Each of our pattern will cover a section of the standard. In this paper we present one pattern for Section 7 of the standard, the context establishment. This is one of the initial steps of the standards and it is the input for following steps, e.g., the asset identification.

1 Introduction and Background

Establishing trust of customers is essential for a company. The security level of a company is a decisive factor in establishing this trust. The ISO 27000 series of security standards supports to achieve this goal.

In this paper we provide patterns for the context establishment of the security information risk management process according to the ISO 27005 [ISO08] standard. The importance of this step is obvious, because later steps depend upon it. Beckers et al. [BKFS11] proposed a common pattern for the cloud computing domain to support context establishment and asset identification of the ISO 27000 series. We built upon this work an present a more general approach that is not limited to cloud computing systems, but support any kind of ICT system. Moreover, the work in [BKFS11] only extends to parts of the context establishment, the patterns presented in this work cover the entire context establishment.

The ISO 27001 defines the requirements for for establishing and maintaining an ISMS [ISO05]. In particular the standard describes the process of creating a model of the entire business risks of a given organization and specific requirements for the implementation of security controls. The resulting ISMS provides a customized security level for an organization.

The ISO 27001 standard is structured according to the "Plan-Do-Check-Act" (PDCA) model, the so-called *ISO 27001 process* [ISO05]. In the *Plan* phase an ISMS is established, in the *Do* phase the ISMS is implemented and operated, in the *Check* phase the ISMS is

monitored and reviewed, and in the *Act* phase the ISMS is maintained and improved. In the *Plan* phase, the *scope and boundaries* of the ISMS, its *interested parties*, *environment*, *assets*, and all the *technology* involved are defined. In this phase also the ISMS *policies*, *risk assessments*, *evaluations*, and *controls* are defined. Controls in the ISO 27001 are measures to *modify risk*. The ISO 27005 [ISO08] refines this process for risk management and extends it with a pre-phase for information gathering.

The process starts with a gathering of information about the organization. The next activity is the *context establishment*. This initial step is important for all following steps and is responsible, whether the risk management can be implemented in a sufficient extent and on a sufficient level of detail. The next step is the *risk identification*, that determine potential loss. Then *risk estimation* tries to rate the consequences of loss on a qualitative or quantitative scale as well as the likelihood of occurrence. The *risk evaluation* step compares the level of risks against the risk acceptance criteria, defined during the context establishment. Then, the *risk treatment* step sets up controls. In the *risk acceptance* step, residual risks have to be accepted by managers of the organisation.

2 Supporting Context Establishment using Patterns

In our previous work [BKFS11] we presented a cloud system analysis pattern that provides a conceptual view on cloud computing. The pattern supports to systematically instantiate stakeholders, cloud technology, and relations between these. The instantiated pattern allows a structured analysis of cloud stakeholders and the cloud system. The documentation can be used for parts of the early phases of the ISO 27005 standard. We propose in this work a generic version of the cloud system analysis pattern, which can be instantiated for any given ICT system, depict in Fig. 1.

An ICT system is a software system that processes data for stakeholders. The software system consists of resources that have a location and the system consists of hardware and software. The system is embedded into a direct system environment, which contains stakeholders that have a direct relation to the system. The direct system environment is further embedded into the indirect system environment, which contains stakeholders that have no direct relation to the system.

The *System Owner* owns the ICT system, the *Administrator* maintains the resources of the software system. The *Developers* develops software for the ICT system. The system has an *Internal Users* that works for the System Owner and exchanges data with the ICT system. In addition, *External Users* work for a *Customer* of the System Owner.

The stakeholder of the indirect environment are the legislator, a set of all relevant laws from a specific country. The template can be instantiated with multiple legislator of all the countries relevant for the ICT system. The domain represents a set of all relevant regulations for, e.g., the finance domain.

We accompany the system analysis pattern by templates to systematically gather domain knowledge about the direct and indirect system environments based upon the stakeholders' relations to the system and other stakeholders. This is also similar to the approach in

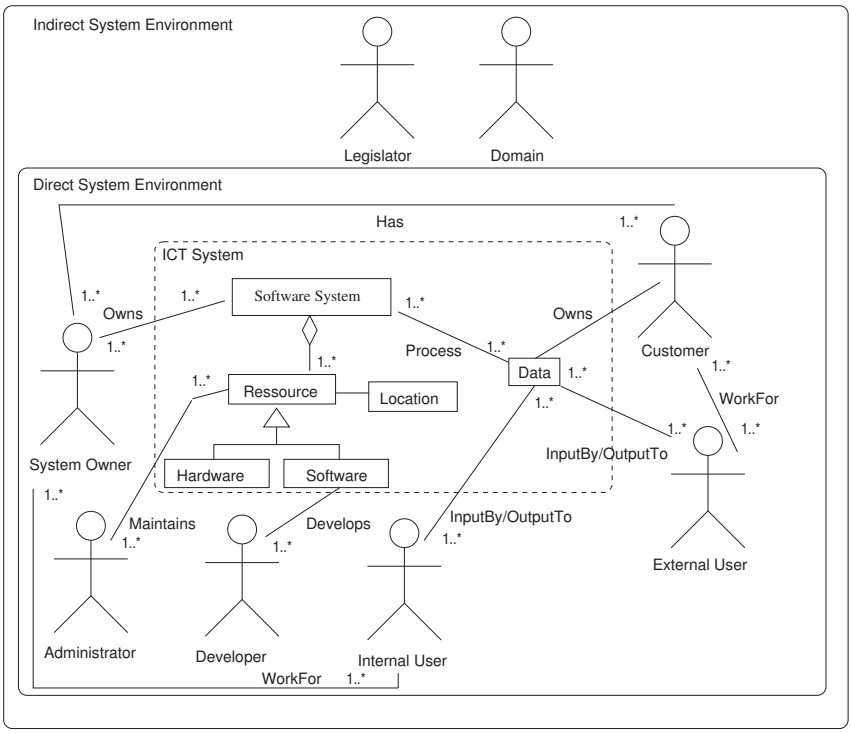


Figure 1: System Analysis Pattern

[BKFS11].

We present a template for the administrator as an example for the templates for the indirect stakeholders:

Name Administrator

Description The administrator repairs the system and he/she executes maintenance efforts.

Relations to the ICT system The administrator has access to the entire data in the system.

Motivation The administrator earns money by maintaining the resources of the ICT system.

Relations to other direct stakeholders The administrator has a contract with the System Owner to maintain the system. There might also be further contracts, e.g., preventing the administrator from accessing the Customers data.

Assets The administrator has no direct assets in the ICT system. However, if the ICT system does not exist, he/she would not earn money. Hence, the system itself could be considered the asset of the administrator.

When the gathering of information about the organization is done, the ISO 27005 demands a context establishment as a next step. We present an *ISO 27005 Context Establishment Pattern*, depict in Fig. 2. This contains several vital parts of the ISO 27005 context establishment and considers the basic process of this part of the standard. This pattern can be

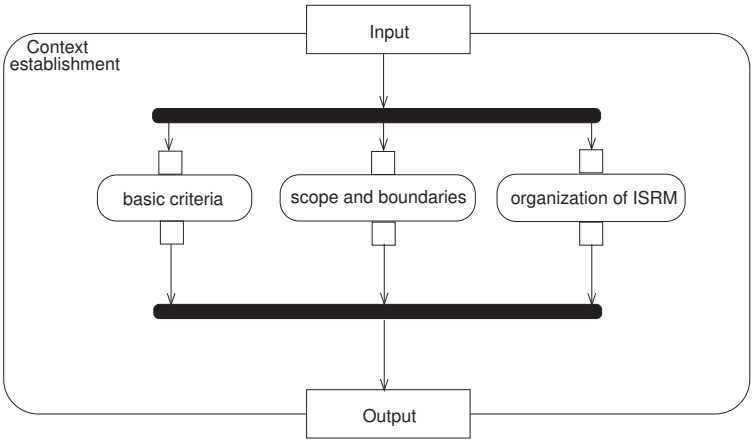


Figure 2: ISO 27005 Context Establishment Pattern

instantiated systematically for a given organization. The resulting instance of the pattern can be used as documentation for the standard implementation. The pattern contains three elements of the standard, which have to be instantiated. These parts are the *basic criteria*, the *scope and boundaries* and the *organization of information security risk management (ISRM)*.

The basic criteria is instantiated with a set of resources, which have to be elicited before the instantiation. Resources according to the ISO 27005 standards are all relevant elements of an organization to the ISRM. We propose to use our system analysis pattern as a source for these resources. The basic critera requires further that information assets have to be derived from the set of resources. Afterwards the resulting set of information assets has to be refined to classes of similar information assets. For each of these classes an assesment of security breaches has to be conducted. This shall be done for at least the following kinds of security issues: CIA breaches, financial and business loss, compliance breaches. Afterward criteria have to developed for each class of information assets based on the assessed breaches and losses. Those criteria should include risk evaluation criteria, impact criteria and risk acceptance criteria.

The scope and boundaries has to be instantiated with collected information artifacts about organization. These artifacts have to be collected from: business objectives, strategies, information security policies, business processes, organizational functions and structure, legal and contractual requirements, stakeholders and their expectations, socio-cultural environment, and interfaces between environment and organization. For each collected information artifact, it has to be decided whether it shall be included in the scope and boundaries (or not).

An information security risk management process has to be defined and documented, as part of the description of the organization of the ISRM. Such a process should contain the stakeholders of the system, the roles they enact, the activities they execute, the relations

between stakeholders, roles and the organization, and the records to be kept. Our system analysis pattern already defines most of the needed information in a coarse grained way.

We accompany the ISO 27005 Context Establishment Pattern also with templates to systematically gather the requested knowledge for the basic criteria, scope and boundaries, and organization of ISRM. We provide the template for the organization of the ISRM process as an example:

Stakeholder As defined in the system analysis pattern

Roles

Rolename A short name of the role

Description A summarizing description of the role

Stakeholder A list of stakeholders which can enact the role

Precondition Conditions to be fulfilled to enact this role.

Responsibilities A description for which assets and resource the role is responsible

Rights A description of rights the role has to have to fulfill the responsibilities.

Excluded roles A list of role,s which are not allowed to be bound to a stakeholder, who enacts the role at hand, at the same time.

Resources And Assets

Name Identifier for the resource or asset

Description A summarizing description of the resource or asset

Stakeholder A list of stakeholders, which have a relation to the resource or assets. The relation has also to be described

Records

Identifier Identifier for the record

Description A summarizing description of record

Template A template to generate a record

Instantiation Description A description to instantiate te template

Activities

Activity A short name of the activity

Description A summarizing description of the activity

Precondition Activities and constrains to be fulfilled before this activity can be executed.

Affected Stakeholders A list of stakeholders which are affected by the activity

Roles The roles which are allowed to execute this activity

Resources and Assets A list of resources and assets which are necessary for the activity

Records List of records to be generated when this activity is executed

Postcondition Activities and constrains to be fulfilled after this activity has been executed

3 Related Work

Cheremushkin et al. [LCAS11] present a UML-based meta-model for several terms of the ISO 27000, e.g., assets. These meta-models can be instantiated and, thus, support the refinement process. However, the authors do not present a holistic approach to information security. The work mostly constructs models around specific terms in isolation. Mondetino et al. investigate possible automation of controls that are listed in the ISO 27001 and ISO 27002 [MF11]. Their work can complement our own.

4 Conclusions and Future Work

We presented a pattern for the context establishment section for the ISO 27005 standard. Our work shows that the ISO 27005 can benefit from a system of patterns that can be instantiated for any given ICT system. Our approach comprises the following main benefits: A generic context establishment based on patternsa and systematic pattern-based documentation of ICT systems. This benefits ease the burden of establishing the context for a ISO 27005 certification. In the future we will extend our work for the entire ISO 27005 standard. In addition, we intend to develop a general method to support the establishment of security standards, and apply it for instance as validation to the BSI Standard 100-2.

References

[BKFS11] Kristian Beckers, Jan-Christoph Küster, Stephan Faßbender, and Holger Schmidt. Pattern-Based Support for Context Establishment and Asset Identification of the ISO 27000 in the Field of Cloud Computing. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, pages 327–333. IEEE Computer Society, 2011.

[ISO05] ISO/IEC. Information technology - Security techniques - Information security management systems - Requirements. ISO/IEC 27001, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2005.

[ISO08] ISO/IEC. Information technology - Security techniques - Information security risk management. ISO/IEC 27005, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2008.

[LCAS11] Alexander Lyubimov, Dmitry Cheremushkin, Natalia Andreeva, and Sergey Shustikov. Information security integral engineering technique and its application in ISMS design. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, pages 585–590. IEEE Computer Society, 2011.

[MF11] Raydel Montesino and Stefan Fenz. Information security automation: how far can we go? In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, pages 280–285. IEEE Computer Society, 2011.

Towards Static Modular Software Verification

Marcus Mews, Steffen Helke

Department of Software Engineering
Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin
{mews, helke}@cs.tu-berlin.de

Abstract: The paper presents our first work in progress results of an approach to verify the correct use of software libraries in target projects. Therefor the project's source code is analyzed and checked against the library's behavior specification, called interface grammar. This grammar is formalized using annotated state diagrams, and the verification analysis is based on static control flow, data flow and alias analyses. The paper illustrates the presented approach using a small-sized Java library example. In the end, we give a brief outlook to necessary enhancements.

1 Introduction

When developing software, in many cases software engineers include and reuse software libraries. But reusing third party's libraries necessitates a thorough understanding of the software library. Without proper care, misused libraries can lead to errors and exceptions at runtime, and can thus endanger the safety of the developed software. Hence the question arises, whether the included software libraries are utilized correctly and how to get prove.

In our context, utilizing a software library means nothing else but calling a library's interface methods. Usually, most software libraries provide a documentation including e. g. its methods, which are intended to be called in a specific order. This grammar is part of the interface specification and its violation can cause the library and/or its caller to fail.

We address the issue of wrong calling orders of library methods and present a static source code analysis for modular software verification. Inputs to this analysis are the interface grammar and the complete source code which utilizes the library. In this paper, we use state machines to specify the interface grammar. As a result of the analysis, two succeeding library calls are detected which may lead to a violation of the library's specification at runtime. Since the analysis relies on naturally imprecise control flow, data flow and alias analyses, its results can contain false positives. Nevertheless, the presented analysis can give sound evidence that a library is utilized correctly, if no errors are detected.

2 Static Software Utilization Verification

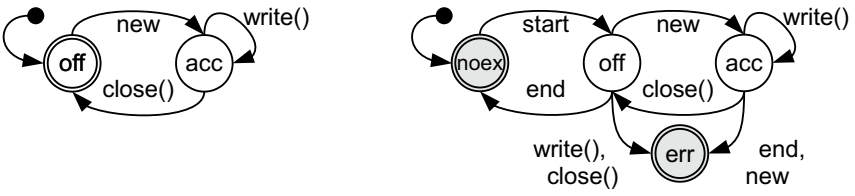
Our static modular software verification is presented in two steps: First, we show how to derive possible misuses from the interface grammar. Then, we explain how we verify whether the software source code contains any of these misuses. But before, have a look at Java Listing 1: Our approach finds the two FileOutputStream misuses: accessing the same file twice at the same time and omitting to close the second file stream.

```
1 public class FileOutputStream_Error {
2     public static void main(String[] args) throws IOException {
3         File file = new File("c:/line.txt");
4         FileOutputStream fos1 = new FileOutputStream(file);
5         FileOutputStream fos2 = new FileOutputStream(file);
6         fos2.write("Hallo Welt".getBytes());
7         fos1.close();
8 }   }
```

Listing 1: This compiling code contains two library misuses (one throwing a runtime exception)

2.1 Step 1: Find Error Paths

Misusing a software library means that the library's interface methods are called in a wrong order, or the library is not shut down appropriately before the program terminates. We call a sequence of succeeding interface events (method calls or program start/termination) leading to an error state an error path. In this section, we outline how to derive an error path from the interface grammar.



(a) Original interface specification L_{FOS}

(b) Extended specification L_{FOS_X}

Figure 1: Reduced interface grammar of the Java FileOutputStream library

We use state machines to specify the interface grammar: $L = (Q_L, \Sigma_L, \Delta_L, q_{0_L}, F_L)$ (Fig. 1). Q_L contains all states, Σ_L contains all input symbols and $\Delta_L \subseteq Q_L \times \Sigma_L \times Q_L$ contains all transitions. Δ_L maps from a start state and an input symbol ($dom(\Delta_L) \subseteq Q_L \times \Sigma_L$) to a target state in Q_L. Σ_L is the set of qualified interface constructor and method names (abbreviated in Fig. 1). q_{0_L} is the initial state and F_L contains all finite states. We simplify the task and derive error paths with length of two, only. Therefor, we restrict L so that all transitions with the same input symbols lead to the same state:

$$\frac{(q_1, \sigma_1, q'_1), (q_2, \sigma_2, q'_2) \in \Delta_L}{(\sigma_1 = \sigma_2) \Rightarrow (q'_1 = q'_2)}$$

Next, we derive a second state machine $L_X = (Q_X, \Sigma_X, \Delta_X, q_{0_X}, F_X)$ based on L (Fig. 1(b)). The purpose of L_X is to enrich L with information about the program start and terminate events, and an error state and its attached transitions. Therefor, in exchange for start and finite state markings we add a no execution state q_{noex}, the input symbols σ_{start}, σ_{end}, and transitions Δ_{noex} to and from q_{noex}. We also add an error state q_{err} and transitions Δ_{err} from every state to q_{err}: If a state lacks an outgoing transition that fires on an input symbol σ_i, a new transition to q_{err} is added. The state machine L_X remains deterministic and Δ_X still has only one target state for every tuple in its domain. The initial state now is $q_{0_X} = q_{noex}$, and the and the finite states are $F_X = \{q_{noex}, q_{err}\}$.

$Q_X \triangleq Q_L \cup \{q_{err}, q_{noex}\}$, $\Sigma_X \triangleq \Sigma_L \cup \{\sigma_{start}, \sigma_{end}\}$, $\Delta_X \triangleq \Delta_L \cup \Delta_{noex} \cup \Delta_{err}$

$\Delta_{noex} \triangleq \{(q_{noex}, \sigma_{start}, q_{0_L})\} \cup \{(q_f, \sigma_{end}, q_{noex}) \mid q_f \in F_L\}$

$\Delta_{err} \triangleq \{(q_i, \sigma_i, q_{err}) \mid q_i \in Q_L \wedge \sigma_i \in (\Sigma_L \cup \{\sigma_{end}\}) \wedge (q_i, \sigma_i) \notin dom(\Delta_L \cup \Delta_{noex})\}$

At last, we calculate error paths using L_X. As a benefit of the state machine restriction mentioned above, we can reduce complexity and length of the error paths. An error path $p \in P$ is a list of succeeding interface events, and in our case defined as $P \subseteq \Sigma_X \times \Sigma_X$, containing only two events in a row. P_{FOS} shows all error paths of the Java File Stream library of L_{FOS_X}, and $P_{Listing}$ shows the two error paths that can be found in Listing 1.

$P \triangleq \{(\sigma_i, \sigma_j) \mid \delta_m, \delta_n \in \Delta_X \wedge \delta_m = (q_i, \sigma_j, q_{err}) \wedge \delta_n = (q_k, \sigma_i, q_i)\}$

$P_{FOS} \triangleq \{(\sigma_{start}, \sigma_{write()}), (\sigma_{start}, \sigma_{close()}), (\sigma_{close()}, \sigma_{write()}), (\sigma_{close()}, \sigma_{close()})\}$
$\cup\{(\sigma_{new()}, \sigma_{new()}), (\sigma_{new()}, \sigma_{end}), (\sigma_{write()}, \sigma_{new()}), (\sigma_{write()}, \sigma_{end})\}$

$P_{Listing} \triangleq \{(\sigma_{new()}, \sigma_{new()}), (\sigma_{write()}, \sigma_{end})\}$

2.2 Step 2: Check Project

With the error paths at hand, we analyse the program and detect possible library misuses. The library interface methods can be either static or bound to receiver objects. Since we support multiple library instances, library misuses have to be checked for every library instance and its aliases. Thus, aliasing and control flow problems are tackled now.

2.2.1 Alias Analysis

The flow insensitive may alias analysis respects the following assignments: ordinary variable assignments, parameter assignments of method calls, assignments from return statements to method declarations, and from method declaration to all possibly bound method calls. The analysis uses symbols $s \in S$ for variables and methods calls/declarations. We refer to every kind of assignment from symbol s_1 to s_2 with the fact notation $assigned_d(s_1, s_2)$. We then specify transitive assignments with $assigned(s_1, s_2)$, and define that two symbols s_x and s_y do alias when they both have an assignment symbol o in common.

$assigned(s_1, s_2) \triangleq \exists s_i \in S \mid assigned_d(s_1, s_2) \vee (assigned_d(s_1, s_i) \wedge assigned(s_i, s_2))$

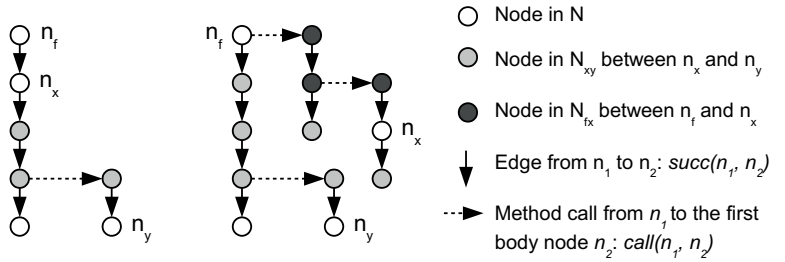
$alias(s_x, s_y) \triangleq \exists o \in S \mid assigned(s_x, o) \wedge assigned(s_y, o)$

2.2.2 Control Flow Analyses

The goal of the control flow analysis is to find two directly succeeding interface events n_x and n_y in the source code. This means that other library events n_B may not be fired in between those two events. More precisely: There exists at least one control flow path from n_x to n_y so that no other n_B is in between. In this subsection, we first describe how we abstract from the source code, and then give a specification of our control flow analysis.

We transform the source code to a data structure $G = (M, B, N, E, C, n_0, F_M, F_P)$ with M as methods, N as nodes, $n_0 \in N$ as the program start node, and $F_P \subset N$ as the program terminal nodes. $E \subseteq N \times N$ is a relation that represents edges from one node to other nodes, and $C \subseteq N \times M$ is a relation that maps method calls from nodes to methods and respects polymorphism by mapping each node to all possible called methods. $B \subseteq M \times N$ is a function that maps every method to its first node, and $F_M \subseteq M \times N$ is a relation that maps every method to all its exit nodes. Additionally, $M_{LE} \subset M$ references all methods that invoke library events like methods of the analyzed library or methods that exit the program. In other words, G contains ordinary control flow graphs for every method of the program, and all Java statements/expressions are abstracted to nodes. Further, the following rules apply: (1) We begin at the first node of every method; (2) every node points to its predecessor(s) (except the last node in a method); (3) every method call node relates additionally to all possibly bound methods (C); (4) every `switch` condition statement node points to all of its conditional bodies and the next mandatory node if no default body was declared; and (5) every `if` condition statement points either to its two conditional bodies, or to its single conditional body and to the next mandatory node. To free G from loops, (6) there are no edges that point to previous nodes. Further, (7) the bodies of loop statements are copied once so that the loop statement node points to both, the original loop body b and a copy bb which is a concatenated version of two times b. Unrolling loop bodies to bb suffices since the error paths only have a length of two. Additionally, (8) conditional loops point to the next mandatory node, since they are not necessarily executed.

Additionally, the methods $start$ and end (representing the symbols σ_{start} and σ_{end} of L) are added to M. As a predecessor we insert a new first node that calls the method $start \in M$. And complementary, we add after every node that can be the last node of a regular program execution, a new succeeding last node that calls the method $end \in M$.



(a) direct (b) indirect (c) Legend

Figure 2: There are two error path types from node n_x to n_y

Fig. 2 depicts two control flow structures that could be specified in G. In the following, we call nodes connected by edges of G *succeeding nodes*. If a node calls a method, we say that the control flow path *descends*. Note that the control flow path between n_x and n_y in Fig. 2(a) is directly constructable by succeeding or descending to the next nodes. In contrast, the control flow path in Fig. 2(b) first needs to return to the previous call site n_f from where it can reach n_y (and even n_x) directly. Two nodes (n_x, n_y) can be connected by arbitrary control flow paths. Each control flow path has a set of nodes N_{xy} that contains all nodes in between.

With error tuples like $(\sigma_{write()}, \sigma_{end}) \in P$ from Sec. 2.1 we call the code analysis method $Path_{Lib}(\sigma_{write()}, \sigma_{end})$. To ensure that there are no library events between n_x and n_y, we detect on one path all nodes N_{xy} in between (using $Path(n_x, N_{xy}, n_y)$) and demand that they do not invoke library events (using $noLib(N_{xy})$). In the case $Path_{Lib}(\sigma_a, \sigma_b) \wedge (\sigma_a, \sigma_b) \in P$ is true, we successfully detected a possible error path in the source code.

$$Path_{Lib}(n_x, n_y) \triangleq \exists N_{xy} \subseteq N \mid Path(n_x, N_{xy}, n_y) \wedge noLib(N_{xy})$$

$$Path(n_x, N_{xy}, n_y) \triangleq \exists n_f \in N, \exists N_{fx}, N_{fy} \subseteq N \mid$$
$$AllPaths_{Desc}(n_f, N_{fx}, n_x) \wedge Path_{Desc}(n_f, N_{fy}, n_y) \wedge N_{xy} = N_{fy} \setminus (N_{fx} \cup \{n_x\})$$

$$Path_{Desc}(n_x, N_B, n_y) \triangleq Path_{Succ}(n_x, N_B, n_y) \dot{\vee} Path_{Call}(n_x, N_B, n_y)$$

$$Path_{Succ}(n_x, N_B, n_y) \triangleq n_x \notin dom(C) \wedge ((succ(n_x, n_y) \wedge N_B = \emptyset)$$
$$\vee (\exists n_i \in N, \exists N_{B*} \subseteq N \mid succ(n_x, n_i) \wedge Path_{Desc}(n_i, N_{B*}, n_y) \wedge N_B = \{n_i\} \cup N_{B*}))$$

$$Path_{Call}(n_x, N_B, n_y) \triangleq (call(n_x, n_y) \wedge N_B = \emptyset) \vee ($$
$$\exists n_t, n_i, n_j \in N, \exists m \in M, \exists (n_x, m) \in C, \exists (m, n_t) \in F_M, \exists N_{B*}, N_{B**} \subseteq N \mid$$
$$(call(n_x, n_i) \wedge Path_{Desc}(n_i, N_{B*}, n_y) \wedge N_B = \{n_i\} \cup N_{B*}) \vee$$
$$(call(n_x, n_i) \wedge Path_{Desc}(n_i, N_{B*}, n_t) \wedge ((succ(n_x, n_y) \wedge N_B = \{n_i, n_t\} \cup N_{B*})$$
$$\vee (succ(n_x, n_j) \wedge Path_{Desc}(n_j, N_{B**}, n_y) \wedge N_B = \{n_i, n_j, n_t\} \cup N_{B*} \cup N_{B**}))))$$

$$noLib(N_B) \triangleq \forall n_i \in N_B \mid (n_i, m) \in C \wedge m \in (M \setminus M_{LE})$$

$$call(n_1, n_2) \triangleq \exists m \in M \mid (n_1, m) \in C \wedge (m, n_2) \in B$$

$$succ(n_1, n_2) \triangleq (n_1, n_2) \in E$$

Descending the control flow path is easy using G, but ascending again is only possible if one keeps track with the call sites: Only if the call sites in a generic path are known, the next node after a return node can be determined. To keep track with call sites, the analysis specifies a generic path from n_x to n_y based on two descending paths. Both of them start at the same node n_f that precedes n_x and n_y, and that is located at a higher level in the call graph hierarchy. Since the control flow graphs *may* be forked at a node n_f (as shown in Fig. 2(b)), we call n_f fork node. The nodes $n \in N_{xy}$ can then be specified using the difference of two descending path node sets: The minuend is the set of nodes N_{fy} between the n_f and n_y (Fig. 2: grey/dark nodes and n_x); and the subtrahend is the set of nodes N_{fx} between n_f and n_x, including n_x (Fig. 2: dark nodes and n_x). But N_{fx} and N_{fy} are of different kind: While both of them contain nodes between n_f and n_x or n_y, respectively, N_{fx} contains the nodes of all paths between n_f and n_x (specified in $AllPaths_{Desc}$). In

contrast, N_{fy} only contains the nodes of one single path between n_f and n_y (specified in $Path_{Desc}$). In the formalization above, $Path_{Desc}$ is stated in detail, and $AllPaths_{Desc}$ is omitted, but can be specified analogously.

The specification $Path_{Desc}$ always respects methods calls when determining next nodes. If a node n_x does not call a method, then $Path_{Desc}$ is based on $Path_{Succ}$. Otherwise – if n_x calls a method – $Path_{Desc}$ is based on $Path_{Call}$. With regard to all possible locations of n_x and n_y in a descending control flow path, $Path_{Succ}$ and $Path_{Call}$ are defined. $Path_{Succ}$ first considers the case that n_x and n_y follow each other directly and hence have no nodes in between. The second case is that n_y follows n_x at some point later in the control flow graph, and a recursive definition is used. Hence, the nodes in between are the union of the directly succeeding node n_i and all the following nodes in N_{B*}. In style of $Path_{Succ}$, $Path_{Call}$ is specified similarly.

The remainder of the specification above states a succession and a call relation. $succ(n_1, n_2)$ is true when the node n_2 succeeds n_1. $call(n_1, n_2)$ is true when n_1 calls a method and n_2 is the first node of this method's body.

2.3 Evaluation

For evaluation we implemented our approach using JTransformer [KHR07] as a meta programming and analysis tool for Java. To verify the implementation we used a test suite that tests every possible correct and incorrect library use of our example, and Java language features like program calls, conditional blocks and loops. To evaluate performance and scalability[1], we extended the Soot framework's analysis source code [VRHS+99] that has a big connected call graph, using polymorphy etc. Table 1 indicates that the performance does not depend on the code size but on the call graph size due to its depth and numerous calls to the same methods.

Project	Lines of Code	Performance (sec.)
Single test case	22	0.004
Test suite (22 test cases)	959	0.312
Soot	12515	4874

Table 1: Scaling performance of the analysis

3 Conclusion

Like Ball et al. [BR02] and others before, we use an API grammar to specify correct behavior. Our work also is related to the work of Hughes et al. [HB07], Tkachuk et al. [TD03], and Jin [Jin07], but for verification we use static code analyses instead of model checking or formal methods.

[1] Tested on an Intel i5 Processor, 4GB RAM; JTransformer's fact building time not included.

Our implementation currently supports libraries that use static and instance methods, and parameters. In addition, language features like polymorphy, condition and loop statements are respected. On the downside, the implementation ignores threads and exception handling, permits recursion and poorly scales to large programs. Nevertheless, our approach as presented here is capable of analysing simple but essential libraries like file stream or socket libraries based on static analyses, and identifies their misuses.

In the future, we will work on supporting error paths of length greater than two and extend the interface grammar to provide additional features to express method parameter constraints or even dependencies of multiple library instances. Regarding the implementation, we concentrate on switching to Soot as an analysis tool, and use collapsed call graphs and more precise static code analyses that take object or control flow context information into account [Mil05].

Acknowledgements

This work is carried out as part of the VirtuOS project. The VirtuOS project is financed by TSB Technologiestiftung Berlin – Zukunftsfonds Berlin Co-financed by the European Union – European fund for regional development.

References

[BR02] Thomas Ball and S K Rajamani. SLIC: A Specification Language for Interface Checking (of C). *Techn Report MSRTR2001*, 21(MSR-TR-2001-21), 2002.

[HB07] Graham Hughes and Tevfik Bultan. Interface grammars for modular software model checking. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 39–49, New York, NY, USA, 2007. ACM.

[Jin07] Ying Jin. Formal Verification of Protocol Properties of Sequential Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 475 –482, july 2007.

[KHR07] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*, LATE '07, New York, USA, 2007. ACM.

[Mil05] Ana Milanova. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol*, 14:2005, 2005.

[TD03] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 188–197, New York, NY, USA, 2003. ACM.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

Using model-based analysis in certification of critical software-intensive systems

Frank Ortmeier, Simon Struck and Michael Lipaczewski
Computer Systems in Engineering
Otto-von-Guericke University of Magdeburg

Abstract: Software is taking over more and more functionality in most technical systems, which leads to the term software-intensive or cyber-physical systems. Although this offers many exciting new opportunities, it also makes precise analysis of safety and reliability goals much more complicated. Well-known traditional techniques often reach their limits. Model-based approaches on the other hand can be useful for solving some of these problems.

However, in industrial practice answering the question alone is often not sufficient. It is also necessary to explain how answers were found. In this paper, we will show some of the capabilities of modern model-based analysis methods and highlight how they possibly could be used in safety engineering resp. what obstacles need to be avoided.

1 Introduction

Safety critical applications need certification prior to system launch. Based on a safety analysis the certification ensures that the system will not cause any harm to human beings. Considering modern software-intensive systems the safety analysis is a challenging task. Research developed new analysis methods to cope with the increasing complexity. However these methods and techniques face some difficulties in the traditional certification process. In this paper we point out the current state, advantages and difficulties of the application of modern model-based safety-analysis techniques in the certification process of software-intensive safety-critical systems. Section 2 introduces software intensive systems. Section 3 gives an overview of the certification process and Section 3.1 points out how the safety analysis is used during certification. After that Section 3.2 introduces chances and challenges of modern model-based safety analysis techniques. Finally Section 4 summarizes the content presented in the paper.

2 Software-intensive systems

A *software-intensive system* is a technical system in which important parts are realized in terms of software and thus the software interacts with other systems, the environment and human beings [WH06]. This especially implies embedded computer applications. Such systems often consist of sensors, a programmable processing unit and actors. Besides the

basic physical model of the sensors and actors the behavior of the system is heavily dependent on the programming of the control unit.

Along with decreasing costs and increasing capabilities of semiconductor technology, software-intensive systems become more and more important in modern engineering. An important aspect of many software-intensive systems is their nature as reactive systems. Based on sensor values the system controls its actors so that it properly reacts on the input values. If the system also covers an internal state its output not only depends on the current input, but also on a history of input values. Reactive systems often cannot be analyzed in isolation, because their output typically influences their inputs via the systems environment. Therefore it is mandatory to analyze the system together with its surrounding components. The environment may cover other system components as well as the surrounding nature.

3 Certification process

In general, the goal of certification is to ensure the correctness and safety of a system. For successful certification a comprehensive a-priori statement about the system safety is necessary. This means the safety of the system is demonstrated and documented in an objective way prior to the system launch. Depending on the stakeholder there are different reasons why (safety critical) systems require certification:

- The producer of a system (i.e. the company that develops and builds the system) needs the certification as permission to sell/install/use the resulting product. Of course a company is also interested in the safety of their products (as this may have impact on the reputation), but basically they get paid for the product and not its safety. So the main goal of producers is to minimize the risk of failing to successfully certify their system.
- The certification body (usually in terms of a national agency) has to ensure the safety of the society and especially all human beings. The main interest of the certification body is in certifying *only* solidly safe systems. The secondary (but very important) interest is to make an *objective* decision. In particular, a-posteriori (after an accident has happened) certification bodies need to show how and why they approved the system. So their main goal is to make legally solid and objective arguments as claims for certification.

These are somehow contradictory interests from which different requirements can be derived. The company has to demonstrate the safety of their products and wants to do this by minimal costs. On the other hand there is the certification body which wants to see a comprehensive and objective demonstration about the safety of the product in question.

Mathematically spoken, verification of software can be used to ensure the functionality of a software system with respect to a specification (for example Hoare logic). This proves the correctness of the system in a mathematical way and is thus absolutely correct. However,

deducing a mathematical proof is often very complicated and time consuming. In the case of complex systems it even might not always be feasible. In any case, it is a very expensive task and only done for small but highly critical system components.

In addition, mathematical verification of a system is often hard to understand. The lack of traceability as well as high complexity of the (mathematical) reasoning makes understanding of this type of safety arguments hard. This is especially a problem when the safety case needs to be accepted by the certification body. Imagine that an accident with the system happened and an a-posteriori analysis showed an error in the mathematical analysis. Who takes responsibility then? The analyst (for making the error), the safety manager (for not finding the error) or the certification body (for allowing hard to understand/evaluate arguments). Things become even worse if arguments have been calculated (semi-) automatically, so that the designer/producer of the tool can be held responsible (for distributing erroneous software).

3.1 Safety analysis and certification

Safety analysis is the process of a-priori measuring/determining the safety of a system. Certification bodies often rely on standardized tools and analysis techniques. Different standards, with the focus on specific needs form different domains, exist. The IEC 61508 [Lad08] as a generic norm, DO178B/C [RTC92] for avionics and the upcoming ISO26262 [Int09] for automotive. However, in practice safety analysis is not only done by certification bodies nor only during certification.

Figure 1 depicts a certification process from the railroad domain. The system is developed by the system engineers (not depicted in the figure) led by a technical project leader. The quality manager observes the overall design and implementation process and organizes the (software) tests. The validator evaluates the results of the (software) tests and compares the results with hardware/system in the loop tests. More important for the context of this paper is the safety manager. He coordinates the safety related processes within the project. The technical project leader exchanges design documents and implementation plans with the safety manager and in return gets early feedback on the system's safety aspects. On the other hand, the safety manager combines the quality management report and the design related documentation into a safety case that is passed to the safety assessor. The safety assessor evaluates the overall systems safety by examining the validation plan and validation report from the validator and the safety case from the safety manager. As a result the safety assessor creates a safety assessment that is passed to the certification body. If the safety assessment is accepted, the system gets certified.

The interaction between the technical project leader, the safety manager and the safety assessor relates to the two different interests for certification mentioned in the last section. The safety manager gives feedback to the system engineers and thereby helps to pass the safety certification. On the other side the safety manager provides important input for the safety assessment and thus fulfills the requirements from the certification body.

For the safety assessment to be accepted it must provide comprehensive information about

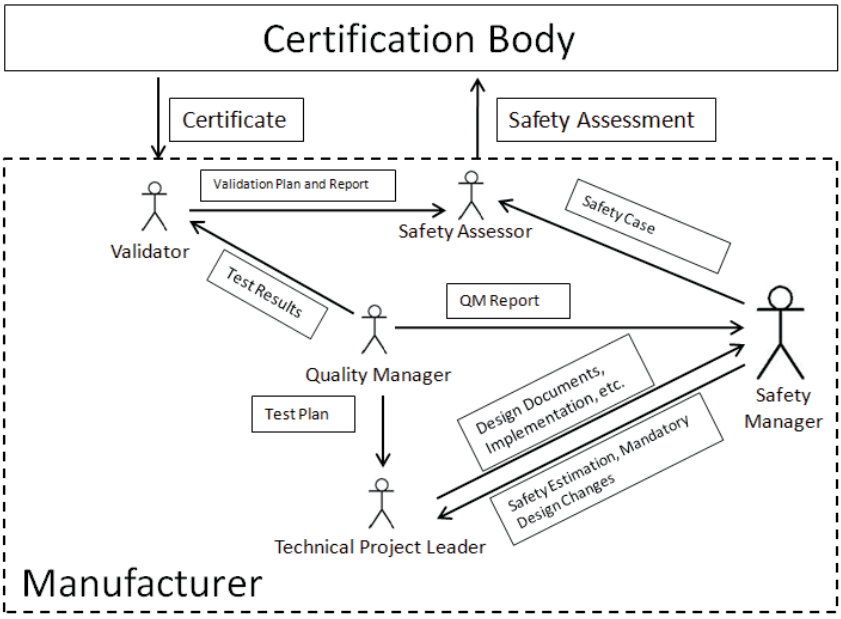


Abbildung 1: Interaction between all participants in the certification process

the systems safety. This implies that the used analysis techniques are comprehensible, even to people who are not experts in safety analysis domain. Therefore safety engineers often rely on common safety analysis techniques like fault tree analysis (FTA) [VDF+02] and failure modes and effects analysis (FMEA) [MMB96]. Besides being standardized, these analysis techniques have proved their use in numerous of applications and are easy to understand.

3.2 Chances and challenges for model-based methods

However, traditional safety analysis methods have some disadvantages. They were mostly developed decades ago. Thus they were designed for mechanical systems with only few or no software and might no longer fit the needs of modern software-intensive systems.

Model-based safety analysis methods where developed to cope with software-intensive systems. There are several modeling languages and safety analysis tools available, that were already successfully applied during system development: COMPASS/SLIM[BCK+09] and SCADE/Lustre[1] and some others.

A clear advantage of the model-based analysis methods is that they are highly automated. Thus they are performed very fast, and cost efficient. This is a great advantage in modern

[1]Feb. 2012: `http://www.esterel-technologies.com/products/scade-suite/`

software development processes, where the analysis/evaluation is not only applied once at the end of the development. As a highly automated process, model-based analysis is also less error prone and also leads to more accurate results. Finally the close relation between the model and the design/implementation helps to improve the maintainability of the model.

If model-based analysis methods are used during the system design time, they can give very early and very precise feedback about the system safety and thus assist the design process. Imagine a design related safety issue. If the issue is only found during the certification (where safety analysis must be done), it is often very expensive to correct a failure. However if model-based safety analysis points out that there is an issue, this can be considered in the design process and thus will not lead to complications in the certification process. This fits with the interests of the project/company which wants to pass the certification on the first approach.

The main challenge for the application of model-based analysis techniques is the demonstration of its cost-benefit ratio. Due to the deterministic and highly automated process more accurate results are possible. These can even be gained at lower costs than with traditional analysis methods. In contrast to the certification body the overall costs are one of the main issues for the company developing the system.

Objectively measuring the safety of a system, also meets the demands of the certification body. The main challenge for the application of model-based analysis during the certification process is that they are hardly accepted by the certification body. On a long term the model-based analysis techniques should be qualified as tool for safety certification. This, however, is a difficult and long process. On shorter term, the model-based analysis can be used in combination with traditional analysis methods. Imagine a FTA, where the safety engineer has to decide how detailed the analysis is performed. This can be a serious source of failure in the analysis. If the results of the FTA are evaluated with the results from a model-based analysis, this could point out where the safety engineer abstracted to far from the system. The FTA can then be corrected, and thus becomes more reliable. In this scenario the certification body still gets a traditional analysis as safety case.

4 Conclusion

Safety analysis is used during the design time and for the certification of safety critical systems. Traditional analysis techniques do not cope with the complexity of modern software-intensive systems. Thereto model-based analysis techniques where created. However, these methods are barely used in industrial practice.

Model-based analysis techniques can assist during the design of a system. Possible design failures can be detected and corrected early in the development process. Thus the system can pass certification without expensive corrections (in a late development phase). This decreases production costs, and thus suits the economic interests of a company that is producing safety-critical systems.

After the development of a safety critical system, certification is required as permission to

sell/launch the system. The certification process uses safety analysis to measure/determine the system safety. Despite the advantages of model-based analysis techniques, they are even less used in the certification process than during the system design. Qualifying model based analysis is a long and difficult process. An intermediate approach is to combine model-based and traditional analysis methods. Thus the accuracy of the model-based analysis can be expressed in terms of traditional analysis methods, which are accepted by the certification body.

Acknowledgments

Michael Lipaczewski is sponsored by the Deutschen Ministerium fr Bildung und Forschung in the ViERforES project (BMBF, project-Nr.: 01IM08003C).

Simon Struck is sponsored by the German Research Foundation (DFG) within the ProMoSA project.

Literatur

[BCK+09] M. Bozzano, A. Cimatti, J.P. Katoen, V. Nguyen, T. Noll und M. Roveri. The COMPASS approach: Correctness, modelling and performability of aerospace systems. *Computer Safety, Reliability, and Security*, Seiten 173–186, 2009.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani und A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV 2002)*, Jgg. 2404 of *LNCS*. Springer, 2002.

[Est11] Esterel Technologies, http://www.esterel-technologies.com/?id=13281, accessed Feb. 28. 2011. SCADE Suite, 2011.

[GO10] Matthias Güdemann und Frank Ortmeier. A Framework for Qualitative and Quantitative Model-Based Safety Analysis. In *Proceedings of the* 12^{th} *High Assurance System Engineering Symposium (HASE 2010)*, Seiten 132–141, 2010.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond und D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Int09] International Organisation for Standardization. ISO 26262: Road Vehicles-Functional Safety, Draft International Standard (DIS), 2009.

[KNP02] Marta Kwiatkowska, Gethin Norman und David Parker. PRISM: Probabilistic Symbolic Model Checker. In T. Field, P. Harrison, J. Bradley und U. Harder, Hrsg., *Proceedings of the* 12^{th} *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2002)*, Jgg. 2324 of *LNCS*, Seiten 200–204. Springer, 2002.

[Lad08] Peter B. Ladkin. An Overview of IEC 61508 on E/E/PE Functional Safety, 2008.

[MMB96] Robin E. McDermott, Raymond J. Mikulak und Michael R. Beauregard. *The Basics of FMEA*. Quality Resources, 1996.

[ORS06] F. Ortmeier, W. Reif und G. Schellhorn. Deductive Cause-Consequence Analysis (DCCA). In *Proceedings of the* 16^{th} *IFAC World Congress*. Elsevier, 2006.

[RTC92] RTCA. DO-178B: Software Considerations in Airborne Systems and Equipment Certification, December, 1st 1992.

[VDF+02] Dr. W. Vesley, Dr. Joanne Dugan, J. Fragole, J. Minarik II und J. Railsback. *Fault Tree Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance, August 2002.

[WH06] M. Wirsing und M. Holzl. Software intensive systems. *Draft Report on ERCIM âBeyond the Horizon Thematic Group*, 6, 2006.

GSN_M-Edit: Ein modellgetriebener Editor für modulare GSN-Argumentationen

Patrick Werner[1] Stefan Gerken[2] Michaela Huhn[3]
[1] Institut für Theoretische Informatik, Technische Universität Braunschweig
[2] IC MOL RA R&D, Siemens AG
[3] Institut für Informatik, Technische Universität Clausthal

Abstract: In diesem Paper wird ein Konzept und eine prototypische Implementierung von GSN_M-Edit vorgestellt, einem Editor, der eine Modularisierung von GSN-Strukturen unterstützt. Modularisierung von Argumentationstrukturen in Sicherheitsnachweisen erleichtern die Wiederverwendbarkeit von Modulen in verschiedenen *Goal Structures*. Das vorgestellte Konzept wird anhand einer konkreten Implementierung - sowie anschließender Anwendung auf eine Argumentation evaluiert.

1 Einleitung

In sicherheitskritischen Systemen kommt dem Nachweis der funktionalen Sicherheit eine große Bedeutung zu. Ein wesentliches Kriterium für eine erfolgreiche Sicherheitsnachweisführung ist eine nachvollziehbare, überzeugende und stringente Argumentation, die belegt, dass die Sicherheitsziele von dem technischen System erreicht werden. Hierzu bietet die *Goal Structuring Notation* (GSN) [Kel98] eine anerkannte, graphische Strukturierungsmöglichkeit, die eine Argumentationskette übersichtlich visualisiert.

Größere technische Systeme und Anlagen setzen sich in der Regel aus vielen bereits erprobten Komponenten zusammen. Der Nachweis der Sicherheitsziele nimmt Bezug auf die Systemarchitektur. Daher bietet es sich an, auch die Argumentation modular aufzubauen, bereits exisitente Nachweise für Komponenten einzubeziehen und den Nachweis an der Systemstruktur des technischen Systems auszurichten.

Dies gilt umso mehr, wenn man sich vergegenwärtigt, dass etwa Ein-/Ausgabemodule technischer (Sub-)Systeme in der Regel für unterschiedliche Funktionen eingesetzt werden, und dabei auch in unterschiedlichen sicherheitstechnischen Funktionen mitwirken und deren individuelle Sicherheitsziele erfüllen müssen. Die Sicherheitsargumente für derartige Module werden daher in einer Sicherheitsargumentation an einer Vielzahl von Stellen referenziert.

Um Sicherheitsargumentationen zu modularisieren, werden Werkzeuge zum Editieren benötigt, die die Modularisierung von Zielen und das Einfügen von Referenzen an verschiedenen Stellen einer Argumentationsstruktur unterstützen. Ein solches Werkzeug ist die Grundlage für die effiziente Erstellung von Sicherheitsargumentationen, da bei einer architekturorientierten Strukturierung das verteilte Erstellen und die Wiederverwendung von

Nachweisen für Teilsysteme erheblich vereinfacht wird. Allerdings steigt mit einem modularen Ansatz auch die Notwendigkeit, Änderungen in ihren Auswirkungen nachverfolgen zu können.

Diese Überlegungen führten zu GSN$_M$-Edit, einem Editorprototypen für GSN-Strukturen, der unter Verwendung eines verbreiteten und offenen modellbasierten grafischen Frameworks, genau diese Art der konsistenten Modularisierung über Referenzen auf Teilargumente unterstützt.

Der Beitrag gliedert sich wie folgt: In Kapitel 2 werden die Konzepte der GSN eingeführt. Kapitel 3 betrachtet existente GSN-Editoren bezüglich der Modularisierung. Kapitel 4 stellt die Anforderungen und das Konzept für einen GSN-Editor vor, der Modularisierungsfunktionen unterstützt. In Kapitel 5 wird die Implementierung als Eclipse-Plugin beschrieben. Eine erste Evaluierung, an einem publizierten Sicherheitsnachweis für einen Herzschrittmacher, wird in Kapitel 6 skizziert. Kapitel 7 fasst den Beitrag zusammen und benennt die nächsten Schritte.

2 Die Goal Structuring Notation

Die Goal Structuring Notation (GSN) [Kel98, KW04, GSN11] unterstützt die graphische Darstellung von Argumentationsstrukturen durch die Darstellung der einzelnen Elemente eines Arguments und ihre Beziehungen untereinander. Die GSN wird insbesondere in Europa von einer größer werdenden Anzahl von Firmen verwendet, die sicherheitskritische Systeme herstellen oder betreiben, etwa in den Domänen Luftfahrt, Schienenverkehr, Verteidigung und Medizintechnik. Einige veröffentlichte Ergebnisse über industrielle Erfahrungen mit der Verwendung der GSN in Sicherheitsnachweisen finden sich in [CPK04]. Abbildung 1 zeigt anhand von Beispielinstanzen die Grundelemente der GSN.

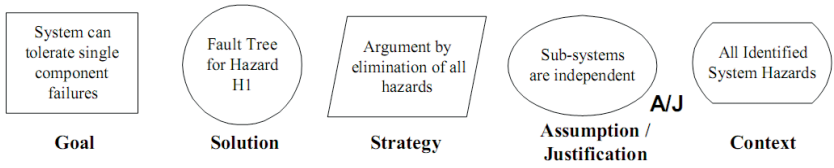


Abbildung 1: Grundsätzliche Elemente der GSN

In der GSN wird eine Argumentationsstruktur als ein gerichteter azyklischer Graph, *directed acyclic graph* (DAG), dargestellt, der auch *'Goal Structure'* genannt wird. Das vorrangige Ziel einer goal structure ist zu zeigen, wie *Goals* (Aussagen zur Systemsicherheit) angemessen und überzeugend argumentativ nachgewiesen werden können, indem sie auf eine Menge von evidenten Belegen (*Solution*) zurückgeführt werden. Dies funktioniert im allgemeinen über eine mehrstufige Argumentation mit weiteren *Sub-goals*. Dabei ist es möglich, die Argumentationsstrategie (*Strategy*) herauszustellen, die bei der Zerlegung in weitere Unterziele der Argumentation verwendet wurde. Die grundlegende GSN Notati-

on sieht darüber hinaus vor, Ziele in einem bestimmten Kontext (*Context*) zu definieren, und bisher nicht explorierte, also noch zu entwickelnde Aussagen zu kennzeichnen. GSN erlaubt grundsätzlich zwei Arten von Relationen zwischen Elementen: Die *SupportedBy* Beziehung (ausgefüllter Pfeil) beschreibt eine *wird-gestützt-von* Beziehung zwischen Strategien und Goals, sowie Solutions und Strategien bzw. Goals. Die *InContextOf* Beziehung (unausgefüllter Pfeil) beschreibt eine kontextuelle, einschränkende Beziehung, bspw. zwischen einem Goal und einem Context. Ein Beispiel für eine vollständige *goal structure* findet sich in [KW04].

Die graphische Darstellung der Argumentationsstruktur mit der GSN hat sich bei der Erstellung von Sicherheitsnachweisen als vorteilhaft erwiesen. Sie erleichtert allen an einem Projekt beteiligten Stakeholdern, die Sicherheitsargumente zu kommunizieren und zu verstehen. Eine Schwäche der GSN in ihrer Grundform ist die fehlende Modularisierung. Alle bisher vorgestellten Elemente zielen darauf ab, Sicherheitsnachweise mit Hilfe von GSN in einer monolithischen *Goal Structure* zu entwickeln.

Bei großen und komplexen Systemen können die Sicherheitsargumentationen aber stark anwachsen, was die Verständlichkeit deutlich mindert. Daher sind einige Erweiterungen der GSN vorgeschlagen worden, um Sicherheitsnachweise durch Modularisierung weiter zu strukturieren und die Wiederverwendbarkeit zu erhöhen [Kel01, GSN11]. Abbildung 2 zeigt die Elemente dieser Erweiterung.

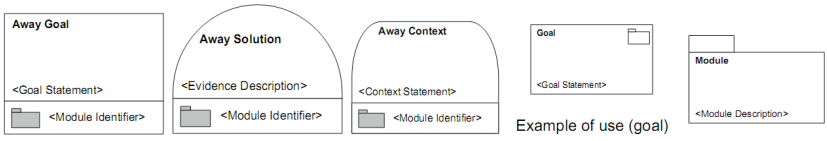


Abbildung 2: Erweiterte Elemente der GSN für Modularisierung

Module werden als zentrales neues Konzept eingeführt. Zusätzlich werden Elemente eingeführt, um Elemente aus (externen) Modulen zu referenzieren: Ein *Away-Goal* stellt dabei ein in einem anderen Modul fortgeführtes Goal dar. Analog werden *Away-Context* und *Away-Solution* Elemente eingeführt, die ebenfalls ihre jeweiligen Grundelemente in anderen Modulen referenzieren. Welche Elemente referenziert werden können, wird explizit durch einen *PublicIndicator* festgelegt, der in der oberen rechten Ecke jedes Goals, Context oder Solution der Basisnotation definiert werden kann. Abbildung 3 zeigt ein Beispiel einer solchen erweiterten *Goal Structure*. Zusätzlich beschreibt der GSN Standard [GSN11] generierte, nicht editierbare Overviewfunktionen. Abbildung 4 zeigt ein Beispiel eines solchen Overviews über alle in einer Goal Structure referenzierten Module.

3 Verwandte Ansätze

Im Zuge der stärkeren Verbreitung der GSN Notation und dem industriellen Einsatz sind mehrere Werkzeuge entstanden, um die Entwicklung von Argumentationsstrukturen in der

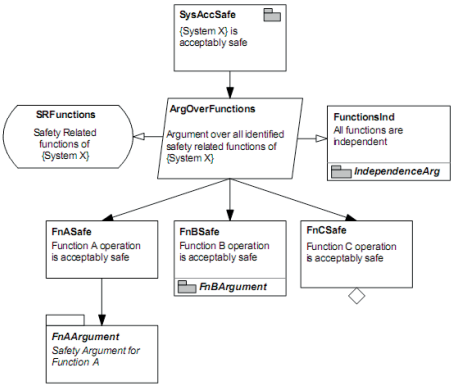


Abbildung 3: Inhalt eines Moduls

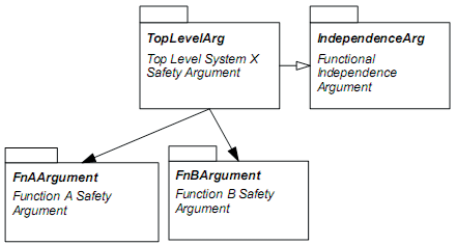


Abbildung 4: Externe Ansicht der Module und Beziehungen

GSN zu unterstützen: Der D-Case Editor [MT11], [D-C] und der ASCE Editor [ASC] stellen Funktionen bereit, um *Goal Structures* zu modellieren und auf GSN konforme Strukturen zu überprüfen. Wie bereits erwähnt, können vor allem größere Sicherheitsnachweise in diesen Tools unübersichtlich werden, da die Konzepte der Modularisierung wie sie in [GSN11] beschrieben sind, nicht umgesetzt sind.

Es ist zwar möglich, Strukturen in sogenannten Templates vorzudefinieren und anschließend in eigene Modelle einzufügen, die Anwendungen von konkreten Referenzierungen ist aber in beiden Implementierungen nicht möglich. Das führt dazu, dass vorhandene Sicherheitsnachweise, etwa von Komponenten, in komplexen Systemen nicht wiederverwendet werden können. Zwar ist es möglich, externe Goal Structures per 'copy-paste' einzufügen, dies führt aber schnell zu unübersichtlichen, nicht navigierbaren Strukturen. Darüber hinaus wird für jede Verwendung eine eigene Kopie benötigt, was erfahrungsgemäß nach wenigen Arbeitsschritten zu Abweichungen und Inkonsistenzen führt.

Der Atego GSN Modeler [Ate] liegt momentan als frühe Beta-Testversion vor. Er erlaubt die Modellierung von modularen *Goal Structures* und stellt darüber hinaus wiederverwendbare Argumentationspattern bereit, ermöglicht aber die Referenzierung von Elementen nur innerhalb eines Projektes. Dies macht eine konsistente, projektübergreifende Datenhaltung - insbesondere bei Mehrfachreferenzierungen - schwierig. Durch die gewählte Darstellungsform werden alle Module eines Projektes gemeinsam angezeigt, was ebenfalls zu unübersichtlichen Strukturen führt.

Unsere eigenen, bisherigen Arbeiten [HZ09, ZH09] zu einem GSN-Profil und Werkzeugunterstützung als Plug-In von Papyrus UML [Pap] zielen auf die enge Integration zwischen Sicherheitsnachweisführung und einem Architekturmodell in UML. Die Modularisierung von GSN-Strukturen wird nicht addressiert.

4 Unterstützung für GSN-Module

In diesem Abschnitt wird ein Konzept für einen GSN-Editor beschrieben, der die Modularisierung von Goal Structures unterstützt. Dafür wird die grundlegende Struktur von Argumentationen auf Projekte und Module erweitert. Ein Projekt stellt die Argumente eines Sicherheitsnachweises dar und besteht aus mehreren Modulen. Um die Modellierung mehrmoduliger Sicherheitsnachweise in GSN auch praktisch zu unterstützen, wird ein erweiterbarer Editor auf der Grundlage modellgetriebener Verfahren entwickelt [SV05].

4.1 Anforderungen

Ausgehend auf den bisher in anderen Werkzeugen nicht umgesetzten Funktionalitäten, dem GSN Standard [GSN11] und der Anwendung in einem praktischen Umfeld ergeben sich die folgenden Anforderungen an die Implementierung des GSN_M-Edit:

- Unterstützung der Erstellung, Bearbeitung und projektübergreifenden Wiederverwendung von GSN-Modulen, inklusive Überblicksfunktionen über die verwendeten Module (Overviews)
- Leichte Erweiterbarkeit des Editors, z.B. um domänen- oder projektspezifische Pattern oder Exportfunktionen
- Automatische Strukturanalyse für GSN-Strukturen mit Fehlerlokalisierung

4.2 Konzepte für die Modularisierung

Grundlage für die modellgetriebene Entwicklung eines GSN-Editors ist ein Metamodell, das die Elemente der Notation und ihre Beziehungen definiert. Zusätzlich zu den Grundelementen der GSN können auch Elemente verwendet werden, die in der modularen Erweiterung des GSN Standards [GSN11] definiert sind. Module dienen der Zusammenfassung kohärenter Teilargumentationen. Wir setzen voraus, dass ein Modul eindeutig einem Projekt zugeordnet wird. Die anderen Elemente für die Modulerweiterung erlauben, Elemente in anderen Modulen zu referenzieren. Dabei kann auch in andere Projekte referenziert werden. Auf diese Weise lassen sich Referenzen zwischen Modulen und auch zwischen Sicherheitsnachweisen erstellen. Diese Referenzen ermöglichen es nicht nur, Sicherheitsnachweise weiter zu strukturieren, sondern auch vorhandene Argumentationsstrukturen wiederzuverwenden. Abbildung 5 zeigt den ersten Teil des Metamodells, das die grundlegendenen Elemente der GSN Notation innerhalb eines Moduls beschreibt.

Alle Elemente innerhalb des Ausschnitts des Datenmodells in Abbildung 5 gehören zu einem Modul. Ein Nachweis besteht aus einer Menge von Modulen, die sich gegenseitig referenzieren können. Innerhalb des Datenmodells finden sich sowohl die grundsätzlichen Elemente der GSN, als auch die Elemente der modularen Erweiterung. Um es dem Be-

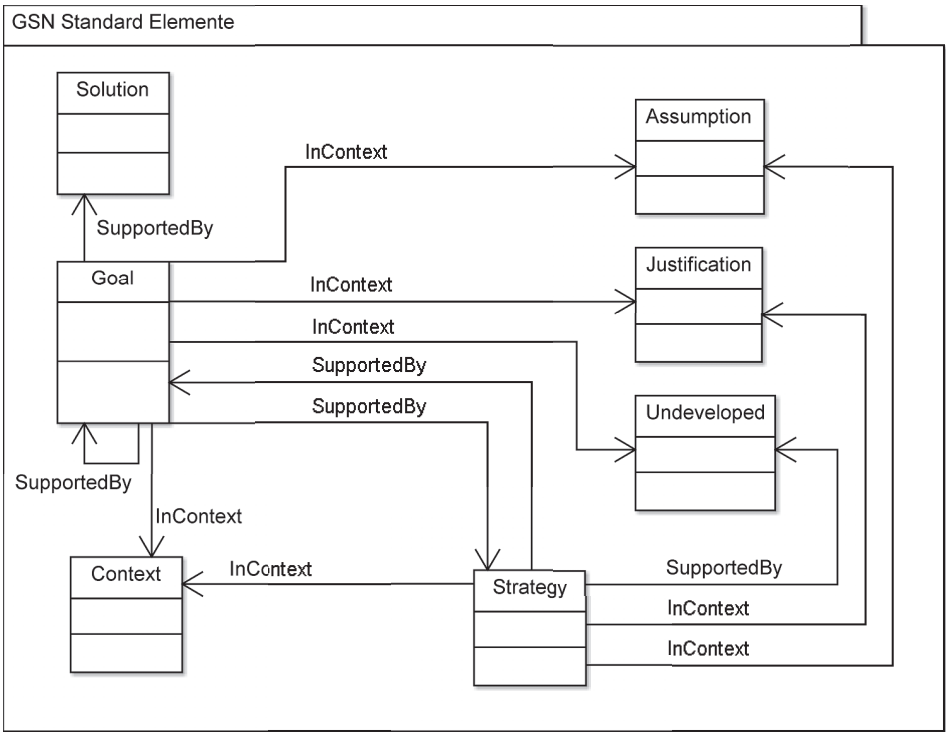


Abbildung 5: Datenmodell GSN - Standardelemente

nutzer einfacher zu machen, diese Elemente zu benennen und mit externen Quellen zu versehen, erben alle Datenelemente von einem Standardelement, mit Ausnahme des *Undeveloped* Elements, das lediglich eine noch nicht beendete Argumentationskette markiert. Dieses Standardelement ordnet jedem Knoten eine Identifikation, einen Inhalt und eine externe Pfadangabe innerhalb des aktuellen Projektes zu. In diesem Pfad kann jeder Knoten eine externe Quelle angeben, auf die er sich bezieht. Um die Übersichtlichkeit zu gewährleisten, kann maximal eine externe Quelle angegeben werden. Mit Hilfe der grundlegenden GSN-Elemente ist es möglich, Goal Structures eines Moduls zu beschreiben: So kann sich ein Goal in mehrere neue Goals aufteilen, bzw. über eine bis mehrere Strategien weiter dekomponiert werden oder durch Solutions belegt werden. Jede Strategie und jedes Goal kann von Assumptions oder Justifications eingeschränkt oder erläutert werden, bzw. in einem bestimmten Context stehen.

Die erweiterten Elemente *AwayGoal*, *AwayContext AwaySolution* und *ModuleReference* stellen in ihrem zugehörigen Modul eigenständige Elemente dar, referenzieren aber genau ein Element aus einem anderen Modul oder, im Fall einer *ModuleReference*, ein komplettes Modul. Dadurch wird eine Referenzierung auf verschiedenen Granularitätsebenen unterstützt. Wie bei den grundlegenden Elementen der GSN werden für referenzierte Elemente SupportedBy- und InContext-Beziehungen angeboten.

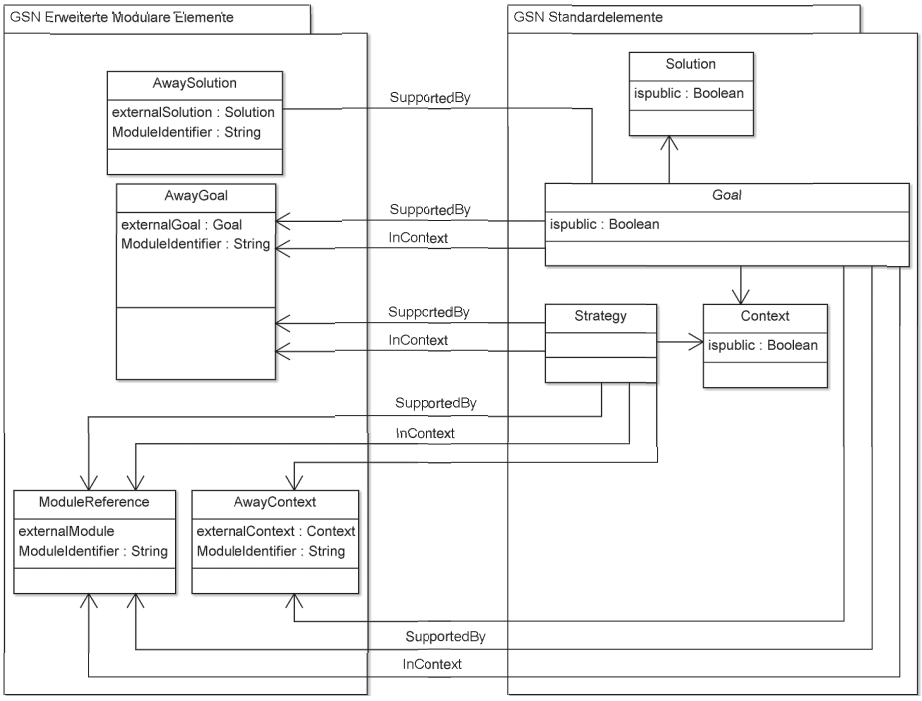


Abbildung 6: Datenmodell GSN - Erweiterung für Modularisierung

Alle Elemente, die in dieser Erweiterung referenziert werden können (Goal, Module, Context, Solution), erhalten in dem erweiterten Datenmodell (siehe Abbildung 6) ein boolesches Attribut, das von Anwender gesetzt werden kann. Dieses *ispublic* Attribut beschreibt, ob das jeweilige Element referenzierbar ist oder nicht.

Die erweiterten Elemente besitzen keine unabhängigen Attribute. Ihre Attribute sind von den von ihnen referenzierten Elementen abhängig. Zusätzlich enthalten sie ein weiteres Attribut. Dieses wird in Abhängigkeit der externen Referenz gesetzt und enthält den Namen des Moduls, in dem sich das referenzierte Objekt befindet. *AwayGoals* und *ModuleReferences* erlauben das Unterteilen des großen Modells in mehrere kleine Module, indem sie die Schnittstelle zwischen den Modulen darstellen. Die Elemente *AwayContext* und *AwaySolution* werden dann verwendet, wenn ein Solution- oder Context-Element in verschiedenen Modulen mehrfach verwendet wird. Diese Struktur hilft mehrfach vorkommende Elemente über Modulgrenzen hinaus einheitlich zu verwalten und Inkonsistenzen zu vermeiden. Die Verbindungen zwischen den Objekten innerhalb eines Moduls werden der Einfachheit halber als unidirektionale Verbindungen umgesetzt. Verbindungen über Modulgrenzen hinaus werden bidirektional modelliert, um die Beziehung in beiden beteiligten Modulen zu speichern. Über diese bidirektionale Modellierung werden im Falle einer Editierung des referenzierten Objektes alle referenzierenden Objekte angepasst.

4.3 Überprüfung von GSN-Strukturen

Erfahrungsgemäß findet die Erstellung eines komplexen Sicherheitsnachweises über mehrere Iterationen statt. Daher können Goal Structures während dieses Entstehungsprozesses von dem GSN Metamodell abweichen. Um dem Benutzer bei der Modellierung von Nachweisen möglichst große Freiheiten zu lassen, überprüft das hier vorgestellte Metamodell nur die Art der Referenzen, bzw. ob nötige Referenzen vorhanden sind oder nicht. Weitere Einschränkungen, die sich nicht kanonisch im Metamodell umsetzen lassen, werden in einer Menge von Regeln formuliert. Diese Regeln werden im Rahmen einer Strukturanalyse überprüft:

- Jedes Modul besitzt ein Top-Goal.
- Jedes Element mit Ausnahme von *solution*-Elementen besitzt *innerhalb seines Moduls* ein eindeutiges Vaterelement.
- Jedes Element ist mit dem aktuellen TopGoal - über eine Kette von Relations - verbunden.
- Jedes Element der modularen Erweiterung referenziert genau ein zugehöriges Grundelement in einem anderen Modul.
- Jedes Goal und jede Strategy sind entweder über SupportedBy-Beziehungen weiter mit Belegen versehen oder als *Undeveloped* markiert.
- Eine Goal Structure ist zyklenfrei bzgl. der SupportedBy- und InContext-Relationen.

Wird eine dieser Regeln innerhalb eines Moduls verletzt, wird eine entsprechende Fehlermeldung ausgegeben und somit Hinweise für nötige Korrekturen gegeben. Zusätzlich zu den hier definierten Regeln überprüft der Syntaxcheck aus dem Metamodell stammende Einschränkungen. Jedes Element der modularen Erweiterung muss genau eine Referenz auf ein zugehöriges Standardelement enthalten. Bei Anwendung der Strukturanalyse wird überprüft, ob diese Referenzen existieren. Alle hier definierten Regeln arbeiten jeweils auf einem bestehenden Modul nicht auf Beziehungen zwischen Modulen. Daher unterliegt es dem Anwender sicherzustellen, dass keine Zyklen über Modulgrenzen hinaus existieren.

4.4 Konzepte für die Wiederverwendbarkeit

Unser Ansatz ermöglicht die Wiederverwendbarkeit von Argumentationsstrukturen. Durch die Verwendung von Mehrfachreferenzierungen können vorhandene *Goal Structures* in neuen Sicherheitsnachweisen verwendet werden, ohne Redundanz zu erzeugen, aber auch referenzierte Module editiert werden, ohne Inkonsistenzen zu verursachen. Abbildung 7 zeigt einen Datenbestand aus zwei Systemen A und B, einem Subsystem C und zwei Komponenten D und E. Der Sicherheitsnachweis des Subsystems C wird hier in zwei Systemen A und B referenziert.

Um den Überblick über alle verwendeten Module zu ermöglichen, wird ein Overview verwendet wie er in Abbildung 4 skizziert ist. Um die Verständlichkeit von bestehenden Sicherheitsnachweisen zu erleichtern, werden zwei verschiedene Overviews angeboten: (1) Die Projektansicht beschreibt den Zusammenhang von Modulen innerhalb eines Projektes, d.h. alle Module innerhalb des Systems A aus Abbildung 7. Diese Ansicht ist hilfreich, um den internen Zusammenhang der Module innerhalb eines Projektes darzustellen. (2) Die Komplettansicht zeigt alle Module, die in der aktuellen Argumentationsstruktur über die Projektgrenzen referenziert werden, und hilft die Referenzen zwischen Projekten zu visualisieren. Die Komplettansicht von System A aus Abbildung 7 zeigt alle Module der Projekte von System A, Subsystem C, den Komponenten D und E und deren Zusammenhang an.

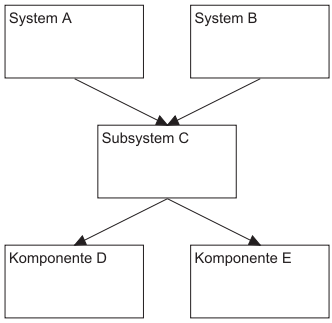


Abbildung 7: Beispiel - Wiederverwendbarkeit

5 Implementierung

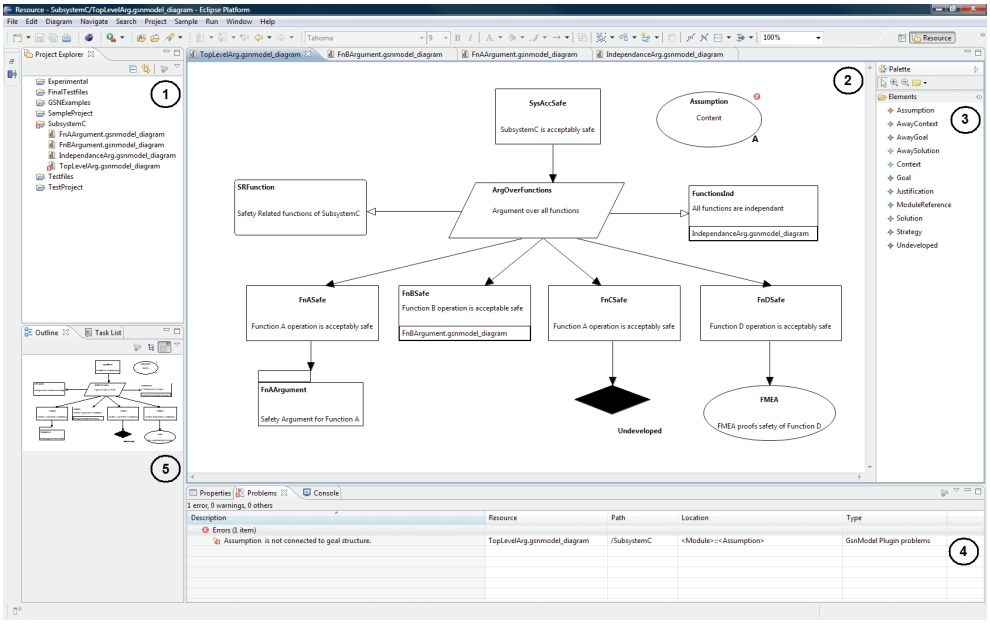


Abbildung 8: Screenshot der Implementierung - Subsystem C - Modul TopLevelArgument

Bei der Implementierung des vorgestellten Konzeptes wird ein modellgetriebener Ansatz

verwendet. Der Editor wird als Eclipse-Plugin implementiert. Dabei liegt die Priorität darauf, möglichst viele Funktionen unter Verwendung generativer Werkzeuge umzusetzen. Um effizient einen lauffähigen Prototypen zu generieren, verwenden wir EuGENia aus der Werzeugfamilie von Epsilon [KRP11]. EuGENia ist ein Front-End für das Graphical Modeling Framework (GMF) [Gro09] und zielt darauf ab, den Entwicklungsprozess von GMF basierten Editoren zu beschleunigen. Dabei wird ein lauffähiger Editor auf Basis eines vorhandenen Eclipse Modeling Framework (EMF) Metamodells erstellt [Gro09]. EuGENia erlaubt einige Annotationen von EMF Modellen, die den weiteren Generationsvorgang beeinflussen und damit weitere manuelle Anpassungen auf Generator- oder Codeebene minimieren. Das innerhalb des Konzepts vorgestellte Metamodell wird als grundlegendes EMF Modell verwendet. EuGENia generiert daraus zuerst die benötigen Dateien, die wiederum benötigt werden, um einen lauffähigen GMF Editor zu generieren. Diese im Zwischenschritt erstellten Dateien werden über Skripte angepasst, um den späteren Anpassungsaufwand auf Codeebene gering zu halten. Die Verwendung von EMF basierten Tools sorgt für Erweiterbarkeit, da alle mit Hilfe des Editors erstellten Instanzen im verbreiteten XML Metadata Interchange (XMI) Format [GDB02] gespeichert werden. Die Overviews werden - aufbauend auf diesen erstellten XMI-Dateien - als separates Eclipse-Plugin implementiert. Zusätzlich zur Codegenerierung bietet Epsilon weitere Funktionen, darunter die Epsilon Validation Language (EVL), eine modellbasierte Validierungssprache zur Konsistenzüberprüfung von Modellen mit Hilfe von definierten Regeln. EVL wird zur Implementierung der automatischen Strukturanalyse verwendet. Abbildung 8 zeigt einen Screenshot des Prototyps. Auf der linken Seite der Abbildung (1) sieht man den Projektexplorer, der die erstellten GSN Projekte im aktuellen Workspace zeigt. Im Zentrum (2) steht das Editorfenster zur Modellierung von *Goal Structures*. Dabei können alle hier beschriebenen GSN-Elemente aus der Palette (3) verwendet werden. Im unteren Teil sieht man zwei weitere Fenster, eines davon zeigt einen Überblick über das aktuelle Modul (5), das andere zeigt alle bei einer Strukturanalyse gefundenen Fehler an (4).

6 Evaluation

Bei der Evaluation wird untersucht, ob sich die GSN Notation [GSN11] im erstellten Editor abbilden und anwenden lässt. Dafür verwenden wir einen komplexeren bereits existierenden Sicherheitsnachweis eines Herzschrittmachers [JLS10]. Anders als in bisher bestehenden Werkzeugen, kann dieser Sicherheitsnachweis in unserer Implementierung entsprechend der in [JLS10] vorgeschlagenen Struktur modularisiert werden. Die Anwendung verlief erfolgreich und erlaubt die Modellierung einer übersichtlichen strukturierten Abbildung dieses Nachweises in dem erstellten Editor. Zur Evaluation wurden einige strukturelle Fehler in die Argumentation eingefügt, die von der anschließenden Strukturanalyse korrekt erkannt wurden. Nach einer Korrektur der gemeldeten Fehler verlief eine erneute automatische Analyse erfolgreich. Auch die Wiederverwendbarkeit wurde adressiert, indem ein Teil des Beispiels in weiteren Projekten referenziert wurde. Dabei zeigte sich der Vorteil der unterschiedlichen Overviews bezüglich des Verständnisses und Überblicks über die erstellen Projekte, aber auch der Nachteil fehlender Versionierungs-

funktionen. Dies ist insbesondere relevant, wenn Argumentationselemente in verschiedenen Versionen eines Sicherheitsnachweises benötigt werden. Daraus resultiert die Anforderung einer anwendungsbezogenen Versionsverwaltung im nächsten Schritt. Im Review mit Domänenexperten erwiesen sich die Modularisierungsmöglichkeiten und die Wiederverwendbarkeitsfunktionen als vielversprechend. Insbesondere die Lesbarkeit und Navigierbarkeit größerer Strukturen, unterstützt durch die Verwendung der Elemente der modularen Erweiterung, unterlag positiven Rückmeldungen. Eine umfassende Evaluation anhand eines großen, praktisch relevanten Projektes ist geplant.

7 Zusammenfassung und Ausblick

Das Eclipse Modeling Framework mit seinen Ergänzungen GMF und Epsilon bietet eine gute Basis, um die GSN modellbasiert in einem grafischen Editor umzusetzen und damit Argumentationen zu modularisieren und zu strukturieren. Das realisierte Plugin GSN$_M$-Edit unterstützt die Wiederverwendung von Argumenten an verschiedenen Stellen eines Sicherheitsnachweises und die zentrale Pflege eines Arguments. Aufgrund der offenen Architektur bietet Eclipse einfache Schnittstellen zu Versionsverwaltungssystemen und anderen Werkzeugen zur Softwareentwicklung und Verifikation. Daher ist die Erweiterung von GSN$_M$-Editors mit weiteren Ansätzen zur Modularisierung nach gegenwärtiger Einschätzung vielversprechend.

Als nächste Schritte sind geplant:

(1) Versionierung von Argumentstrukturen mit einer Änderungsauswirkungsanalyse, mit der sich die von Änderungen betroffenen Argumentfolgen identifizieren lassen. Ggf. soll es möglich sein, verschiedene Versionen eines Arguments gleichzeitig zu verwenden, falls in einem technischen System unterschiedliche Versionen einer Komponente eingesetzt werden.

(2) Verknüpfung der Argumentstruktur mit der Systemarchitektur und Realisierung von Plausibilitätsüberprüfung.

(3) Unterstützung von Argumentationsmustern für dedizierte Eigenschaften, um eine Best-Practice zu bewahren und Konsistenz und Effizienz der Nachweiserstellung zu verbessern.

Literatur

[ASC] ASCE Safety Case Tool. `http://www.adelard.com/web/hnav/ASCE/`.

[Ate] Atego GSN Modeler. `http://http://www.atego.com/download-center/product/atego-gsn-modeler/`.

[CPK04] Paul Chinneck, David Pumfrey und Tim Kelly. Turning Up The HEAT On Safety Case Construction. In *In Practical Elements of Safety: Proceedings of the Twelfth Safety-critical Systems Symposium*, Seiten 223–240. Springer, 2004.

[D-C] D-Case Editor A Typed Assurance Case Editor. http://www.il.is.s.u-tokyo.ac.jp/deos/dcase/.

[GDB02] Timothy J. Grose, Gary C. Doney und Stephen A. Brodsky. *Mastering XMI: Java Programming with XMI, XML and UML.* Wiley, 1. Auflage, April 2002.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project*. Addison-Wesley Longman, Amsterdam, 2009.

[GSN11] GSN Community Standard Version. http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf, November 2011.

[HZ09] Michaela Huhn und Axel Zechner. Analysing Dependability Case Arguments Using Quality Models. In *28th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, Jgg. 5775 of *LNCS*, Seiten 118–131, 2009.

[JLS10] Eunkyoung Jee, Insup Lee und Oleg Sokolsky. Assurance Cases in Model-Driven Development of the Pacemaker Software. In *Proc. of the 4th Intern. Conf. on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA, Vol. II)*, Seiten 343–356, 2010.

[Kel98] Tim Kelly. *Arguing Safety – A Systemic Approach to Managing Safety Cases*. Dissertation, University of York, September 1998.

[Kel01] Tim Kelly. Concepts and Principles of Compositional Safety Cases. Bericht COMSA/2001/1/1, May 2001.

[KRP11] Dimitrios Kolovos, Louis Rose und Richard Paige. The Epsilon Book. http://www.eclipse.org/gmt/epsilon/doc/book/, 2011.

[KW04] Tim Kelly und Rob Weaver. The Goal Structuring Notation - A Safety Argument Notation. In *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[MT11] Yutaka Matsuno und Kenji Taguchi. Parameterised Argument Structure for GSN Patterns. In *11th Intern. Conf. on Quality Software (QSIC)*, Seiten 96–101, 2011.

[Pap] Papyrus for UML. http://www.eclipse.org/modeling/mdt/papyrus/.

[SV05] Thomas Stahl und Markus Völter. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, 1. Auflage, 2005.

[ZH09] Axel Zechner und Michaela Huhn. Structural Analysis of Safety Case Arguments in a Model-based Development Environment. In *Dagstuhl-Workshops Modellbasierte Entwicklung eingebetteter Systeme (MBEES V)*, 2009.

Frühzeitige modellbasierte Risikoanalyse für mobile, verteilte Anwendungen

Christian Wessel[1] Thorsten Humberg[2] Sven Wenzel[1]
Jan Jürjens[1]

[1]Technische Universität Dortmund, 44227 Dortmund
[2]Fraunhofer-Institut für Software- und Systemtechnik (ISST), 44227 Dortmund

Abstract: Anwendungen, die mobile Komponenten beinhalten, sind besonderen Risiken ausgesetzt. Dabei stellt beispielsweise die Kommunikation bei mobilen, verteilten Anwendungen eine Herausforderung an die Daten- und Abhörsicherheit dar. Im Kontext dieser Anwendungen ist daher eine gesonderte Schutzbedarfsanalyse erforderlich. In diesem Papier wird am Beispiel eines Informationssystems mit mobil angebundenen Clients gezeigt, wie eine durchgängige Sicherheits- und Risikoanalyse in den modellbasierten Entwicklungsprozess eines solchen Systems integriert werden kann. Mögliche Schwachstellen des Systems können so bereits in frühen Entwicklungsphasen erkannt und beseitigt werden.

1 Motivation

War bis vor einigen Jahren noch undenkbar, dass beispielsweise Mobiltelefone zu mehr zu gebrauchen sind als nur für Telefonate, so werden Smartphones, Tablet-PCs und andere mobile Endgeräte heute zunehmend in komplexe IT-Systeme und Geschäftsprozesse integriert. Einhergehend mit diesen neuen Möglichkeiten kommen natürlich auch neue Risiken ins Spiel. Während man ein lokal begrenztes System ausreichend und zuverlässig gegen Angriffe und Ausfälle schützen kann, steht man bei mobil verteilten Systemen vor neuen Herausforderungen, wenn z.B. sensible Daten die sicheren Firmennetze verlassen [Eck06]. Die Übertragung der Daten von/zu einem Außendienstmitarbeiter kann z.B. durch geeignete Verschlüsselung geschützt werden. Dass sie mit dem Verlust/Diebstahl des Geräts aber doch in falsche Hände gelangen können oder das gestohlene Gerät an sich ein Angriffspunkt für die Firmen-IT darstellt, bleibt oft unberücksichtigt. Den o.g. Risiken und Sicherheitslücken kann bereits in der Planungsphase eines IT-Systems entgegen gewirkt werden. Ein Risiko im Sinne dieser Arbeit ist eine sicherheitsrelevante Schwachstelle eines Computersystems oder die ungenügend gesicherte Kommunikation zwischen diesen. Die Sicherheit von Softwaresystemen sollte bereits in der Planungsphase berücksichtigt werden, da das Durchsetzen von Sicherheitsrichtlinien im Nachhinein nur sehr schwer möglich, bzw. unmöglich ist. Insbesondere bei modellgetriebenen Entwicklungsmethoden bieten die existierenden Softwaremodelle eine gute Grundlage für erste fundierte Risiko- und Sicherheitsanalysen. Ausgehend von den bestehenden Arbeiten zur modellbasierten Sicherheits- und Risikoanalyse (Abschnitt 2) schlagen wir im Folgenden eine neue Me-

thodik vor, mit der entsprechende Analysen bereits in besonders frühen Planungsphasen, also noch weit vor dem detaillierten Softwareentwurf, möglich werden, wobei auch Complianceanforderungen berücksichtigt werden. Kern unserer Methodik (Abschnitt 3) ist die Integration von Geschäftsprozessen, die z.B. als BPMN-Modelle [FR10] vorliegen, und der geplanten Verteilung auf verschiedene Systemkomponenten, z.B. mit UML Deployments [OMG05]. Auf Basis dieser einfachen Modelle können bereits grundlegende Risiken lokalisiert werden, wie wir am Beispiel eines IT-Systems für den Direktvertrieb zeigen werden. Abschließend diskutieren wir die vorgeschlagene Methodik und erörtern offene Forschungsfragen (Abschnitt 4).

2 Bestehende Ansätze

Es existieren bereits verschiedene Ansätze zur modellbasierten Sicherheitsanalyse. Zwei bekannte Beispiele, um Sicherheitsanforderungen in Softwaremodellen zu spezifizieren, stellen UMLsec [Jür04] und SecureUML [LBD02] dar. Während SecureUML eine Erweiterung der UML zur Zugriffsverwaltung und -kontrolle (RBAC) darstellt, ermöglicht UMLsec auch die Spezifikation zahlreicher weiterer Sicherheitseigenschaften in UML-Modellen. Existierende Werkzeuge[1] erlauben eine automatisierte Prüfung dieser Eigenschaften, z.B. durch Konsistenzprüfungen. Eine Anwendung von UMLsec zur Analyse mobiler Endgeräte wurde bereits in [Bar06] gezeigt.

Auch zur allgemeinen IT-Risikoanalyse existieren erste Werkzeuge. Ein Beispiel ist der *RiskFinder*, der UML-Modelle auf sicherheitsrelevantes Vokabular hin untersucht und mögliche Gefahrenquellen bzw. Risiken hervorhebt [PHJB11]. Schneider et. al. schlagen in [SKH+11] eine heuristische Suche vor, welche Sicherheitsanalysen auf Grundlage von Bayes-Filtern durchführt. *HeRA* stellt einen Feedback-basierten Ansatz zur Sicherheitsprüfung während der Anforderungsanalyse zur Verfügung [KLM09]. Der Ansatz stellt zwar mächtige Regeln bereit, die auch auf dem verwendeten Vokabular arbeiten, jedoch beziehen sich diese stets auf einzelne Wörter und beziehen keine Textdatenbanken ein.

Es existiert ebenfalls ein Ansatz in [Wol08] um Sicherheitsanforderungen in BPMN-Modellen darzustellen. Diese beziehen sich jedoch auf die Darstellung von Sicherheitsmaßnahmen in einem geschlossenen System. Der in dieser Arbeit vorgestellte Ansatz berücksichtigt ebenfalls die spätere Verteilung der Softwarekomponenten.

3 Frühzeitige modellbasierte Sicherheits- und Risikoanalyse

Prozessmodellierungssprachen, wie z.B. BPMN, dienen der Visualisierung von Geschäftsprozessen oder Arbeitsabläufen. Es können z.B. Dokumente oder Informationen modelliert werden, die während eines Prozesses ausgetauscht werden. Verschiedene Akteure, die an einem Prozess beteiligt sind, können durch sogenannte *Swimlanes* dargestellt werden.

[1] z.B. `http://www.umlsec.de/`

Geschäftsprozessmodelle sind i.d.R. schon vor Erstellung eines IT-Systems vorhanden, insbesondere wenn das System zur Unterstützung dieser Prozesse realisiert werden soll.

Um zu planen, welche Programmkomponenten auf welche Teile des Systems (insb. Hardware) verteilt werden, bieten sich UML Deployment-Diagramme an. Hier steht jedoch nicht die feingranulare Verteilung einzelner Artefakte im Vordergrund, sondern der grundlegende Aufbau (Grobentwurf) eines Systems soll betrachtet werden. So wird z.B. identifiziert, dass es Tablet-PCs für Außendienstmitarbeiter geben wird. Wir können daher annehmen, dass BPMN-Modelle und UML Deployment-Diagramme bereits in sehr frühen Entwicklungsphasen gegeben sind.

Beispiel. Abbildung 1 zeigt ein Beispiel in Anlehnung an den Bestellprozess der fiktiven Direktvertriebsfirma *Eisfrost*. Eine Bestellung kann dabei entweder sofort ausgeführt werden, falls sich die gewünschte Ware im Fahrzeug befindet, oder für die nächste Tour des Verkaufsfahrers vorgemerkt werden. Das dazugehörige Deployment-Diagramm zeigt die Systemstruktur. Dabei handelt es sich lediglich um einen möglichen Entwurf des Systems und stellt noch nicht die endgültige Architektur dar. Der Verkaufsfahrer benutzt für die Bestellung des Kunden vor Ort einen Tablet-PC, um die Kundendaten auszuwählen und die Bestellung entgegen zu nehmen. Der Tablet-PC kommuniziert per Bluetooth mit der *Central Car Unit (CCU)*, die sich im Fahrzeug befindet. Die CCU übernimmt die Kommunikation mit dem *Enterprise Resource Planning* (ERP)-System in der Firmenzentrale, um die Kundendaten abzurufen und die Bonität des Kunden vor der Ausführung der Bestellung zu prüfen.Das Ergebnis der Prüfung wird dann an den Tablet-PC des Verkaufsfahrers gesendet, der auf Grundlage der Daten entscheidet, den Bestellvorgang durchzuführen oder abzubrechen.

Vorgehensweise. Das oben gezeigte Beispiel beinhaltet einige Risiken und Sicherheitsprobleme, die man mit einer systematischen Analyse aufdecken kann. Die hier vorgeschlagene Vorgehensweise zur Untersuchung der Modelle beinhaltet im Wesentlichen drei Schritte:

1. Untersuchung der (physikalischen) Verteilung
2. Untersuchung der Kommunikation
3. Analyse (funktionaler und nicht-funktionaler) Risiken

Die einzelnen Schritte sollen im Folgenden eingehender betrachtet werden.

Untersuchung der Verteilung. Die Verteilung der beteiligten Komponenten auf physikalische Systeme ist von zentraler Bedeutung für die Sicherheit eines Systems. Insbesondere mobile Komponenten unterliegen Gefährdungen, die auf stationäre Systeme nicht zutreffen. Beispielsweise können mobile Systeme verloren gehen oder gestohlen werden. Auch das Abhören oder Manipulieren der Kommunikation der mobilen Komponente stellt

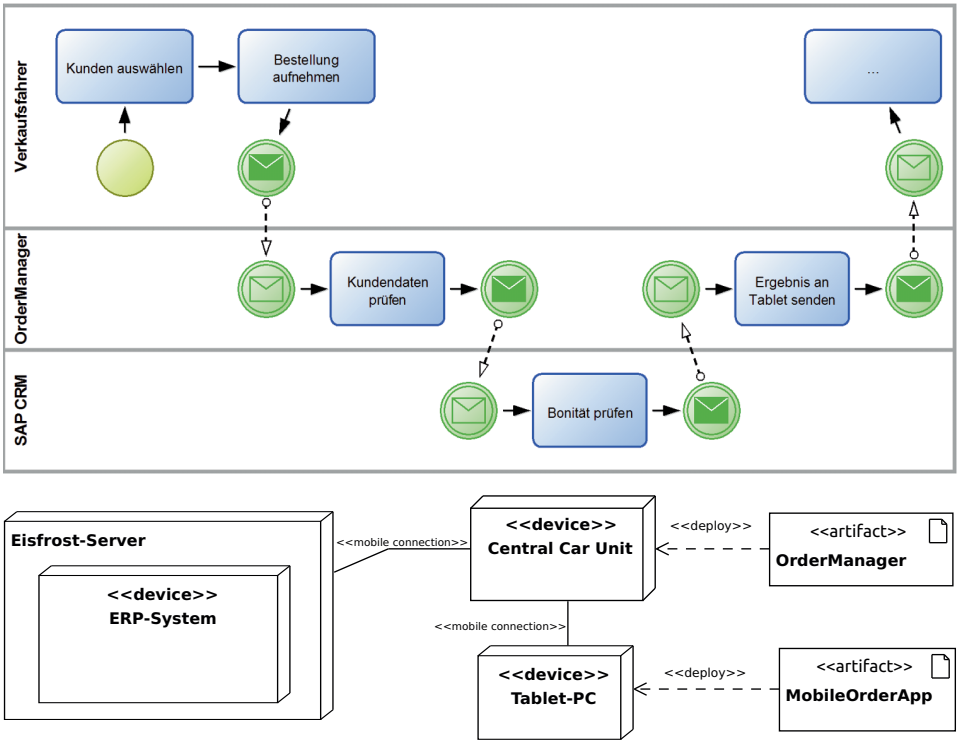


Abbildung 1: Exemplarischer Kundendatenabruf durch ein mobiles System vom zentralen Server

ein erhebliches Risiko dar. Die Verteilung des Systems kann anhand von Deployment-Diagrammen identifiziert werden.

Sobald die Klassifizierung feststeht, kann eine Zuordnung der Komponenten des Deployment-Diagramms auf die Akteure des BPMN-Diagramms vorgenommen werden. Für die Komponenten und Akteure, welche eine erhöhte Aufmerksamkeit hinsichtlich ihrer Gefährdung benötigen, ist es empfehlenswert, diese entsprechend zu visualisieren, um so den Fokus auf diese zu lenken.

Die Zuordnung von Akteuren aus dem BPMN-Modell auf die Komponenten des Deployment-Diagramms kann in Form einer einfachen Tabelle erfolgen. Ein Beispiel für das in Abbildung 1 gezeigte Szenario ist in Tabelle 1 ersichtlich. Mit Hilfe dieser Zuordnung ist es möglich, eine kombinierte Sicht aus Geschäftsprozess und Verteilungsdiagramm zu generieren und die Sicherheitsanalyse darauf auszuführen.

Untersuchung der Kommunikation. Auch die verschiedenen Arten der Kommunikation sind wichtige Merkmale, da diese unterschiedlichen Risiken ausgesetzt sein können. Diese lassen sich ebenfalls aus dem Deployment-Diagramm extrahieren. Ein mögliches Beispiel ist die Unterteilung der Kommunikationsverbindungen in Funk, lokales Netzwerk und Internet. Somit steht für den Nachrichtenaustausch ebenfalls fest, welches Kommu-

Geschäftsprozess	Verteilungsdiagramm
SAP CRM	Eisfrost-Server/ERP-System
OrderManager	OrderManager
Verkaufsfahrer	MobileOrderApp

Tabelle 1: Zuordnung der Akteure des Geschäftsprozesses zu den Komponenten des Verteilungsdiagramms

nikationsmedium dafür benutzt wird. Die besonders schützenswerten Kommunikationskanäle können dann ebenfalls visualisiert werden.

In UMLsec existiert für diesen Zweck den Stereotyp `<<Secure Links>>`. Mit diesem werden Sicherheitsanforderungen für Datenübertragungen gefordert. Jede Verbindung zwischen zwei Knoten kann dann mit dem jeweiligen Leitungstyp, z.B. `<<Internet>>` oder `<<encrypted>>`, annotiert werden. Die übertragenen Daten können z.B. als geheim (`<<secrecy>>`) markiert werden, so dass eine automatische Prüfung erfolgen kann, ob eine Verbindung für entsprechende Daten geeignet ist [Jür04].

Im Beispiel ist ersichtlich, dass eine drahtlose Kommunikation zwischen dem Tablet-PC und der CCU und damit auch mit dem ERP-System stattfindet. Um dieses Schutzbedürfnis im Modell zu manifestieren, schlagen wir eine Erweiterung für BPMN vor, die analog zu UMLsec definiert ist. Dabei geht es in erster Linie aber nicht um die konkrete Kommunikationsart, sondern eher um die Art der Daten, die übertragen werden sollen. Beispielsweise bei personenbezogenen Daten sollte dieses entsprechend im BPMN-Modell annotiert werden, indem der Nachrichtenfluss mit einer besonders gearteten Textannotation versehen werden kann, deren Semantik analog zu einem UMLsec-Stereotyp sein kann. Mit dieser Erweiterung können dann auch Schutzbedarfsanalysen direkt auf dem Geschäftsprozessmodell durchgeführt werden.

Analyse der Risiken. In den vorhergehenden beiden Schritten wurden Gefährdungen betrachtet, die sich aus der Verteilung des Systems und dessen Kommunikation ergeben. Es existieren jedoch weitere Risiken aufgrund der ausgeführten Aktivitäten selbst, deren Identifikation wir wie folgt ermöglichen wollen.

Basierend auf etablierten Standards der IT-Sicherheit, z.B. den Katalogen und Standards des *Bundesamtes für Sicherheit in der Informationstechnik* (BSI)[2] und den ISO-Normen der 27000-Reihe [ISO05], lassen sich sicherheitsrelevante Aktivitäten identifizieren, indem wir das Vokabular der einzelnen Aktivitäten analysieren. Hierzu kann der *RiskFinder* eingesetzt werden [PHJB11]. Wir schlagen jedoch eine Abstraktion von konkreten Notationen vor. So besteht ein zu untersuchender Prozess lediglich aus einer Menge (unabhängiger) *Aktivitäten*, denen jeweils eine Menge von Texten zugeordnet ist. In üblichen Notationen sind dies Titel von Aktivitäten und ggf. ergänzende Kommentare. Dies ist vor allem für die Untersuchung von Geschäftsprozessen geeignet, wo eine Vielzahl formaler und semi-formaler Notationen verwendet werden [Fra11]. Kann einer Aktivität ein Risiko zugeordnet werden, so kann sie entsprechend markiert werden.

[2] `https://www.bsi.bund.de`

Darüber hinaus können auch kritische Strukturen im Prozess aufgedeckt werden, die aus dem Fehlen von Aktivitäten resultieren. Im vorgestellten Beispiel wird bei der Übermittlung der Daten des Kunden keine gesonderte Authentifizierung gefordert. Es muss also davon ausgegangen werden, dass das Login implizit geschieht und die dafür nötigen Zugangsdaten im Gerät gespeichert sind. Bei Verlust des Tablet-PCs oder der CCU, z.B. durch Diebstahl, ist es also möglich, ohne entsprechendes Login auf Firmendaten zuzugreifen.

Weiterhin kann die Risikoanalyse die zuvor betrachtete Untersuchung der Kommunikation unterstützen. So können z.B. Nachrichten, die von Aktivitäten ausgelöst werden, welche schützenswerte Daten verarbeiten, selbst als kritisch gekennzeichnet werden.

4 Diskussion und offene Forschungsfragen

Die Risiko- und Sicherheitsanalyse ist ein unumgänglicher Schritt bei der Entwicklung von Informationssystemen. Insbesondere bei mobilen, verteilten Anwendungen entstehen neue Risiken. In diesem Papier haben wir eine neue Methode zur modellbasierten Analyse solcher Systeme vorgeschlagen, die bereits auf frühphasigen Dokumenten wie Prozessmodellen und groben UML Deployment-Diagrammen angewendet werden kann.

Die Geschäftsprozessmodelle sollten in der Regel schon vorhanden sein und Verteilungsdiagramme lassen sich auch ohne größere Detailkenntnisse entwerfen. Unser Vorschlag ist dabei jedoch nicht auf BPMN-Modelle festgelegt. Für die reine Risikoanalyse sind allein die Bezeichner der Prozessschritte/Aktivitäten ausreichend. Die aktuelle Version des *RiskFinders* wird derzeit dahingehend überarbeitet, dass auch andere Datenquellen möglich sind. Zudem integrieren wir Textdatenbanken zur Auswertung z.B. von synonymen und kookkurenten Begriffen, sodass die Trefferquote zusätzlich präzisiert wird.

Ein Vorteil unseres Ansatzes gegenüber den bestehenden ist dabei, dass die Sicherheitspattern systematisch auf die gefundenen Aktivitäten angewendet werden und damit eine umfassende Sicherheitsanalyse ermöglicht.

Eine weitere nötige Erweiterung des RiskFinders besteht darin, auch das Nichtvorhandensein bestimmter Eigenschaften aufzudecken (vgl. die fehlende Login-Informationen im Beispiel). Dies ist kein triviales Problem, da eine negierte Suche nach Schlüsselwörtern nicht ausreichend ist. Vielmehr muss der Kontext des Nichtvorhandenseins betrachtet werden, um nicht zu viele Falschmeldungen zu generieren.

Die Untersuchung der Kommunikation ist für UML Entwurfsmodelle bereits realisiert worden [Jür04]. Wie oben gezeigt, lässt sich das Konzept problemlos auf Geschäftsprozesse übertragen. Hier ist noch genauer zu evaluieren, wie die Angaben zum Schutzbedarf von Nachrichten sinnvoll in die Modelle integriert werden können. Erweiterungsmechanismen analog zu UML-Stereotypen gibt es in BPMN nicht, die Anmerkungselemente könnten sich evtl. dafür anbieten. Außerdem ist noch eine geeignete Heuristik zu entwerfen, mit der der *RiskFinder* den Schutzbedarf für Datenübertragungen vorschlägt. Insbesondere Transitivitätseigenschaften sind noch zu diskutieren, da nicht immer klar ist, welche Informationen eines Prozessschritts in späteren Schritten weitergenutzt werden. Schlussendlich ist noch eine entsprechende Werkzeugunterstützung für die Konsistenzprüfung zwischen

Schutzbedarf der Nachrichten und verwendetem Kommunikationskanal zu realisieren.

Insgesamt soll die Werkzeugunterstützung dahingehend verbessert werden, dass ein integriertes Werkzeug zur Verfügung steht. Wir denken dabei an eine Integration in das Analysewerkzeug *CARiSMA*[3], um dem Anwender eine ganzheitliche Sicht zu bieten. Hierbei besteht noch Bedarf an einer geeigneten grafischen Benutzerschnittstelle.

Es wäre ebenfalls interessant zu diskutieren, in weit der Ansatz in schon vorhandene Szenarien skaliert, in denen Geschäftsprozessmodelle zur Orchestrierung verwendet werden. Ein verwandter Ansatz zu diesem Thema ist bereits in [Men09] genannt worden.

Literatur

[Bar06] P. Bartmann. *Modellbasierte Sicherheitsanalyse mobiler Endgeräte*. Diplomarbeit, Technische Universität München, 2006.

[Eck06] C. Eckert. *IT-Sicherheit*. Oldenbourg, 2006.

[FR10] J. Freud und B. Rücker. *Praxishandbuch BPMN 2.0*, Jgg. 2. Carl Hanser Verlag, 2010.

[Fra11] Fraunhofer IAO. *Business Process Management Tools 2011*. Fraunhofer-IRB-Verlag, 2011.

[ISO05] ISO. *ISO27001: Information Security Management System (ISMS) standard*, 2005.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer, 1. Auflage, 11 2004.

[KLM09] E. Knauss, D. Lubke und S. Meyer. *Feedback-driven requirements engineering: The Heuristic Requirements Assistant*. ICSE '09, 2009.

[LBD02] T. Lodderstedt, D. A. Basin und J. Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. UML 2002. Springer-Verlag.

[OMG05] Object Management Group. *Unified Modeling Language Specification v2.0*, 2005.

[PHJB11] M. Peschke, M. Hirsch, J. Jürjens und S. Braun. *Werkzeuggestützte Identifikation von IT-Sicherheitsrisiken*. 2011.

[SKH+11] K. Schneider, E. Knauss, S. Houmb, S. Islam und J. Jürjens. *Enhancing security requirements engineering by organizational learning*. *Requirements Engineering*, 2011.

[Men09] Michael Menzel, Ivonne Thomas and Christoph Meinel. *Security Requirements Specification in Service-Oriented Business Process Management*. *ARES*, 2009.

[Wol08] Christian Wolter and Michael Menzel and Christoph Meinel. *Modelling Security Goals in Business Processes*. *Modellierung*, 2008.

[3] `http://carisma.umlsec.de/`

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Lan-guage to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Business 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Er-fahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT:
Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting. CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik –
Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^{2}CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.)
10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on Electronic Voting 2010
co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2010)
Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.)
German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.)
perspeGKtive 2010
Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science – Proceedings of ISSS and BPSC

P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.)
Vom Projekt zum Produkt
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschafts-informatik (WI-MAW), Aachen, 2010

P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM`2010
Second International Workshop on Formal Methods and Agile Methods

P-180 Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, Holger Schwarz (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW)
14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)

P-181 Michael Clasen, Otto Schätzel, Brigitte Theuvsen (Hrsg.)
Qualität und Effizienz durch informationsgestützte Landwirtschaft, Fokus: Moderne Weinwirtschaft

P-182 Ronald Maier (Hrsg.)
6th Conference on Professional Knowledge Management
From Knowledge to Action

P-183 Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.)
Software Engineering 2011
Fachtagung des GI-Fachbereichs Softwaretechnik

P-184 Ralf Reussner, Alexander Pretschner, Stefan Jähnichen (Hrsg.)
Software Engineering 2011
Workshopband
(inkl. Doktorandensymposium)

P-185 Hagen Höpfner, Günther Specht, Thomas Ritz, Christian Bunse (Hrsg.)
MMS 2011: Mobile und ubiquitäre Informationssysteme Proceedings zur 6. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2011)

P-186 Gerald Eichler, Axel Küpper, Volkmar Schau, Hacène Fouchal, Herwig Unger (Eds.)
11th International Conference on Innovative Internet Community Systems (I^2CS)

P-187 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
4. DFN-Forum Kommunikations-technologien, Beiträge der Fachtagung 20. Juni bis 21. Juni 2011 Bonn

P-188 Holger Rohland, Andrea Kienle, Steffen Friedrich (Hrsg.)
DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik
der Gesellschaft für Informatik e.V.
5.–8. September 2011, Dresden

P-189 Thomas, Marco (Hrsg.)
Informatik in Bildung und Beruf
INFOS 2011
14. GI-Fachtagung Informatik und Schule

P-190 Markus Nüttgens, Oliver Thomas, Barbara Weber (Eds.)
Enterprise Modelling and Information Systems Architectures (EMISA 2011)

P-191 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2011
International Conference of the Biometrics Special Interest Group

P-192 Hans-Ulrich Heiß, Peter Pepper, Holger Schlingloff, Jörg Schneider (Hrsg.)
INFORMATIK 2011
Informatik schafft Communities

P-193 Wolfgang Lehner, Gunther Piller (Hrsg.)
IMDM 2011

P-194 M. Clasen, G. Fröhlich, H. Bernhardt, K. Hildebrand, B. Theuvsen (Hrsg.)
Informationstechnologie für eine nachhaltige Landbewirtschaftung
Fokus Forstwirtschaft

P-195 Neeraj Suri, Michael Waidner (Hrsg.)
Sicherheit 2012
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 6. Jahrestagung des Fachbereichs Sicherheit der
Gesellschaft für Informatik e.V. (GI)

P-197 Jörn von Lucke, Christian P. Geiger, Siegfried Kaiser, Erich Schweighofer, Maria A. Wimmer (Hrsg.)
Auf dem Weg zu einer offenen, smarten und vernetzten Verwaltungskultur
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2012

P-198 Stefan Jähnichen, Axel Küpper, Sahin Albayrak (Hrsg.)
Software Engineering 2012
Fachtagung des GI-Fachbereichs Softwaretechnik

P-199 Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.)
Software Engineering 2012
Workshopband

P-200 Gero Mühl, Jan Richling, Andreas Herkersdorf (Hrsg.)
ARCS 2012 Workshops

P-201 Elmar J. Sinz Andy Schürr (Hrsg.)
Modellierung 2012

P-202 Andrea Back, Markus Bick, Martin Breunig, Key Pousttchi, Frédéric Thiesse (Hrsg.)
MMS 2012:Mobile und Ubiquitäre Informationssysteme

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de

Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into
• seminars
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: http://www.gi.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-293-2

This volume contains papers from the Software Engineering 2012 conference held in Berlin from February 27th to March 2nd 2012.The topics covered in the papers range from software requirements, technologies or development strategies to reports that discuss industrial project experience.