GMD Report 90

GMD –
Forschungszentrum
Informationstechnik
GmbH

François Bry, Ulrich Geske,
Dietmar Seipel (Hrsg.)

14. Workshop Logische Programmierung

Würzburg, 26. - 28. Januar 2000

Januar 2000

GMD – Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

Herausgeber/Editors:
Prof. Dr. François Bry
LMU München

Prof. Dr. Ulrich Geske
GMD – Forschungszentrum Informationstechnik GmbH, Berlin

Prof. Dr. Dietmar Seipel
Universität Würzburg

Die Deutsche Bibliothek - CIP-Einheitsaufnahme
Workshop Logische Programmierung <14, 2000, Würzburg>:
14. Workshop Logische Programmierung : Würzburg, 26.- 28. Januar 2000
/ GMD – Forschungszentrum Informationstechnik GmbH.
François Bry ... (Hrsg.). [Veranst.: Gesellschaft für Logisches
Programmieren (GLP), Untergruppe der Association for Logic Programming
(ALP) und die Fachausschüsse 1.1. und 1.2.
der Gesellschaft für Informatik (GI)] . - Sankt Augustin :
GMD – Forschungszentrum Informationstechnik, 2000
(GMD Report ; 90)
ISBN 3-88457-971-1

ISSN 1435-2702

Abstract

Logic Programming is an advanced paradigm for declarative specification of problems. In the logic programming language Prolog, the basic calculus is a subclass of predicate logics. Extended concepts integrate functions or constraints. This programming method has received more and more attraction in the last years, especially in the domains of data bases, in processing of natural language, and for modeling and processing of combinatorial problems. In this manner, logic programming is an active area, which is a sound basis for integration of modern aspects in software engineering, dynamic programming and communication.

Keywords:
Logic Programming, Constraints, Program Analysis, Artificial Intelligence

Kurzfassung

Logische Programmierung ist eine besonders weitgehende Art, Probleme deklarativ zu spezifizieren. In der Form von Prolog geschieht das durch den Einsatz eines Fragments der Logik. Weitergehende Konzepte integrieren in dieses ursprünglich rein relationale Konzept auch Funktionen und Constraints. Dieses Paradigma gewinnt eine zunehmende Attraktivität, u.a. in den Bereichen Datenbanken und Verarbeitung von natürlicher Sprache und bei der Modellierung und Bearbeitung komplexer kombinatorischer Probleme. Die Logikprogrammierung ist somit ein aktives Gebiet geblieben, das sich zunehmend auch den schwierigen Thematiken der Integration in die übrige Softwarelandschaft, der Behandlung von Dynamik und des Umgangs mit Kommunikation stellt.

Schlagwörter:
Logische Programmierung, Constraints, Progammanalyse, Künstliche Intelligenz

Content

Preface 7

Invited Talks

Slim Abdennadher, Thom Frühwirth (LMU München)
Definition von angepaßten Constraint-Solvern mit Constraint-Handling-Rules 15

David Pearce
Equilibrium Logic: An Extension of Answer Set Programming for Nonmonotonic Reasoning 17

Harold Boley
Beziehungen zwischen Logikprogrammierung und XML 19

Frank Puppe
Übersicht über heuristische, fallbasierte und modellbasierte Problemlösungsmethoden in der Diagnostik 35

Constraint Logic Programming I 43

Ulrich John (GMD-FIRST, Berlin)
Rekonfiguration komplexer industrieller Produkte mittels constraint-logischer Programmierung 43

Hans Schlenker (GMD-FIRST, Berlin)
Reduce-To-The-Max: ein schneller Algorithmus für Multi-Ressourcen-Probleme 55

Markus Hannebauer, Ines Münch (TU Berlin und Humboldt-Univ. Berlin)
Transforming object-oriented domain models into declarative CLP expressions 65

Hans-Joachim Goltz (GMD-FIRST, Berlin)
Über Methoden des constrainbasierten Lösens von Problemen der Stundenplanung 77

Georg Ringwelski, Armin Wolf, Ulrich Geske (GMD-FIRST, Berlin)
Implementierung von built-in Constraints für endliche Wertebereiche in Minerva 89

Annalisa Di Deo, Dmitri Boulanger (GMD-FIRST, Berlin)
Using Objects to Build Constraint Databases 101

Extensions of Logic Programming 115

Clemens Beckstein, Manfred Rahneberg (Univ. Jena)
On the Complexity of Generalized Horn Clause Intuionistic Logic 115

Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer,
and Axel Polleres (TU Wien)
Using the dlv System for Planning and Diagnostic Reasoning 125

Functional Logic Programming 135

Frank Steiner, Michael Hanus (RWTH Aachen)
Type-based Nondeterminism Checking in Functional Logic Programs 135

Michael Hanus (RWTH Aachen)
TkCurry: A Declarative Approach to GUI Programming 149

Logic and AI 161

Steffen Hölldobler (TU Dresden)
Challenge problems for the integration of logic and connectionist systems 161

Pierre Bonzon (Univ. Lausanne)
Deliberate Agent Reconcile Reactive and Goal-Directed Agents 171

Program Analysis and Debugging 183

Marija Kulas (Fern-Univ. Hagen)
Debugging Prolog Using Annotations 183

Gabriella Kokai (Univ. Erlangen)
New Missing Solution Method for Trace Trees 199

System Demonstrations 211

Norbert E. Fuchs (Univ. Zürich)
Attempto Controlled English 211

Michael Hanus (RWTH Aachen)
PACS: The Portland Aachen Curry System 219

Uwe Egly, Thomas Eiter, Hans Tompits, Stefan Woltran (TU Wien, Austria)
Implementing Default Reasoning Using Quantified Boolean Formulae 223

Hans-Joachim Goltz (GMD-FIRST
Constraintbasierte interaktive und automatische Stundenplanung 227

Slim Abdennadher, Matthias Saft und Sebastian Will (LMU München)
Constraintbasierte Raumplanung für Universitäten 229

Gabriella Kokai (Univ. Erlangen)
CAPP: Statische Codeanalyse von PROLOG Programmen 231

Stefan Kral, Fred Mesnard, Ulrich Neumerkel (TU Wien, Austria)
Slicing zur Fehlersuche in Logikprogrammen 241

Preface

This volume contains the papers from the presentations of the *14th Workshop on Logic Programming – WLP*, which was held at the *Julius–Maximilians–Universität Würzburg* on January 26–28, 2000. The WLP–series of workshops on logic programming is organized by the *Gesellschaft für Logische Programmierung e.V. (GLP)* – the German speaking subsection of the international *Association for Logic Programming (ALP)* – in conjunction with the Sections 1.1 and 1.2 of the *Gesellschaft für Informatik (GI)*. Previous Workshops on Logic Programming took place in Berlin (1988–1990), Gosen (1991), Darmstadt (1992), Hagen (1993), Zürich (1994), Wien (1995, 1998), and München (1997).

Sixteen papers and seven proposals for system demonstrations had been submitted for the 14th WLP. Each submission was refereed by at least two reviewers from the program committee or by colleagues from outside the program committee. The fourteen accepted papers and the seven papers about system demonstrations are reprinted in this proceedings, as well as papers (abstracts) for four invited talks. The topics of the presentations can be grouped into three maior areas:

1. *constraint logic programming (CLP)*,
2. *extensions of logic programming*
   (including functional logic programming and non–monotonic reasoning), and
3. *program analysis and debugging*.

There were nine presentations in the field of constraint logic programming including an invited talk by Slim Abdennadher and Thom Frühwirth. CLP–technology has received great interest by researchers, and it has been very successful in practical, industrial applications. Seven presentations were dealing with extensions of logic programming including an invited talk on some aspects of non–monotonic reasoning by David Pearce. The topic of program analysis and debugging for logic programming was covered by four presentations. Finally, there was an invited talk on different types of *problem solving methods for diagnosis* by Frank Puppe, an invited talk on the *relationship between logic programming and Web–technology, namely XML,* by Harold Boley, and one presentation on natural language processing.

We would like to thank all the authors who have submitted papers, and all members of the program committe and external referees for their contributions to the success of the workshop. Finally, we gladly acknowledge the support received by GMD (German National Research Center for Information Technology) for printing the proceedings.

December 1999

François Bry,
Ulrich Geske,
Dietmar Seipel

Schedule of Presentations

Mittwoch, 26. 1. 2000

13:15–14:15 Invited Talk

Slim Abdennadher, Thom Frühwirth (LMU München)
Definition von angepaßten Constraint-Solvern mit Constraint-Handling-Rules 15

14:30–16:00 Session 1: Constraint Logic Programming I

Ulrich John (GMD-FIRST, Berlin)
Rekonfiguration komplexer industrieller Produkte mittels constraint-logischer Programmierung 43

Hans Schlenker (GMD-FIRST, Berlin)
Reduce-To-The-Max: ein schneller Algorithmus für Multi-Ressourcen-Probleme 55

Markus Hannebauer, Ines Münch (TU Berlin und Humboldt-Univ. Berlin)
Transforming object-oriented domain models into declarative CLP expressions 65

16:30–17:30 Session 2: Extensions of Logic Programming

Clemens Beckstein, Manfred Rahneberg (Univ. Jena)
On the Complexity of Generalized Horn Clause Intuionistic Logic 115

Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres (TU Wien)
Using the dlv System for Planning and Diagnostic Reasoning 125

17:30–18:00 System Demonstrations I

Norbert E. Fuchs (Univ. Zürich)
Attempto Controlled English 211

18:30–19.30 GLP-Meeting

Donnerstag, 27. 1. 2000

9.00–10.00 Invited Talk

David Pearce
Equilibrium Logic: An Extension of Answer Set Programming for Nonmonotonic Reasoning 17

10:30–12:00 Session 3: Constraint Logic Programming II

Hans-Joachim Goltz (GMD-FIRST, Berlin)
Über Methoden des constrainbasierten Lösens von Problemen der Stundenplanung 77

Georg Ringwelski, Armin Wolf, Ulrich Geske (GMD-FIRST, Berlin)
Implementierung von built-in Constraints für endliche Wertebereiche in Minerva 89

Annalisa Di Deo, Dmitri Boulanger (GMD-FIRST, Berlin)
Using Objects to Build Constraint Databases 101

14.00–15.00 Invited Talk

Harold Boley
Beziehungen zwischen Logikprogrammierung und XML 21

15.00–16:00 Session 4: Functional Logic Programming 19

Frank Steiner, Michael Hanus (RWTH Aachen)
Type-based Nondeterminism Checking in Functional Logic Programs 135

Michael Hanus (RWTH Aachen)
TkCurry: A Declarative Approach to GUI Programming 149

16:30–17.30 System Demonstrations II

Michael Hanus (RWTH Aachen)
PACS: The Portland Aachen Curry System 219

Uwe Egly, Thomas Eiter, Hans Tompits, Stefan Woltran (TU Wien, Austria)
Implementing Default Reasoning Using Quantified Boolean Formulae 233

17:30–18.30 System Demonstrations III

Hans-Joachim Goltz (GMD-FIRST
Constraintbasierte interaktive und automatische Stundenplanung 227

Slim Abdennadher, Matthias Saft und Sebastian Will (LMU München)
Constraintbasierte Raumplanung für Universitäten 229

Freitag, der 28. 01. 2000

9.00–10.00 Invited Talk

Frank Puppe
Übersicht über heuristische, fallbasierte und modellbasierte Problemlösungsmethoden in der Diagnostik 35

10:30–11.30 Session 5: Logic and AI

Steffen Hölldobler (TU Dresden)
Challenge problems for the integration of logic and connectionist systems 161
Pierre Bonzon (Univ. Lausanne)
Deliberate Agent Reconcile Reactive and Goal-Directed Agents 171

11:30–12.30 Session 6: Program Analysis and Debugging

Marija Kulas (Fern-Univ. Hagen)
Debugging Prolog Using Annotations 183
Gabriella Kokai (Univ. Erlangen)
New Missing Solution Method for Trace Trees 199

14:00–15:30 System Demonstrations IV

Gabriella Kokai (Univ. Erlangen)
CAPP: Statische Codeanalyse von PROLOG Programmen 231
Stefan Kral, Fred Mesnard, Ulrich Neumerkel (TU Wien, Austria)
Slicing zur Fehlersuche in Logikprogrammen 241

Organizers

Dietmar Seipel (Univ. Würzburg)
Ulrich Geske (GMD Berlin)
François Bry (LMU München)

Program Committee

Slim Abdennadher (LMU München)
Christoph Beierle (Fern-Univ. Hagen)
Alexander Bockmayr (Univ. Henri Poincare, Nancy, France)
Dimitri Boulanger (GMD Berlin)
Stefan Brass (Univ. Hannover)
François Bry (LMU München)
Jürgen Dix (Univ. Koblenz)
Thomas Eiter (TU Wien)
Uwe Egly (TU Wien)
Thomas Eiter (TU Wien)
Burkhard Freitag (Univ. Passau)
Thom Frühwirth (LMU München)
Norbert Fuchs (Univ. Zürich)
Ulrich Furbach (Univ. Koblenz)
Ulrich Geske (GMD Berlin)
Wolfgang Goerigk (CAU Kiel)
Michael Hanus (RWTH Aachen)
Steffen Hölldobler (Univ. Dresden)
Ulrich Neumerkel (TU Wien)
Dietmar Seipel (Univ. Würzburg)
Michael Thielscher (Univ. Dresden)
Christoph Weidenbach (MPI Saarbrücken)
Armin Wolf (GMD Berlin)

Definition von angepaßten Constraint-Solvern mit Constraint-Handling-Rules

Slim Abdennadher, Thom Frühwirth
Institut für Informatik, Ludwig-Maximilians-Universität
Oettingenstraße 67
D-80538 München
{abdennad,fruehwir}@informatik.uni-muenchen.de

Equilibrium Logic: An Extension of Answer Set Programming for Nonmonotonic Reasoning

David Pearce
DFKI
Stuhlsatzenhausweg 3
D-66123 Saarbrücken
pearce@dfki.de

Abstract

Although the semantics of answer sets for extended logic programs is now ten years old, there has recently been a revival of interest in answer set programming. This is in part due to the recognition that several well–known types of combinatorial problems have elegant solutions in terms of answer sets. In addition, efficient systems have now become available, notably the dlv system of disjunctive datalog which implements answer sets, making problem–solving and programming in this paradigm now feasible.
In this talk we give an introduction to a formalism called equilibrium logic. This can be viewed as a general purpose approach to nonmonotonic reasoning based on a notion of negation–by–default. It can also be viewed as an extension of answer set programming to arbitrary ground theories in a predicate language with two types of negation. It may therefore be of interest for those wishing to extend answer set programming beyond disjunctive datalog. We give two simple characterisations of equilibrium logic, one semantic, based on minimal models, the other syntactic, based on theory completions. We also describe some generalisations of the logic and we discuss some of its properties and technical applications. We conclude with some remarks on computational complexity and implementation issues.

Beziehungen zwischen Logikprogrammierung und XML*

Harold Boley
Deutsches **F**orschungszentrum für **K**ünstliche **I**ntelligenz GmbH
Email: boley@dfki.de

Zusammenfassung

Wechselseitige Zusammenhänge zwischen Logikprogrammierung und XML werden untersucht. XML-Dokumente werden als linearisierte Ableitungsbäume eingeführt und auf Prolog-Strukturen abgebildet. Umgekehrt führt eine Darstellung von Herbrand-Termen und Horn-Klauseln in XML zu einem puren XML-basierten Prolog (XmlLog). Die dazu benutzten XML-Elemente werden um Verwendungen von XML-Attributen wie id/idref für erweiterte Logiken ergänzt. XMLs Dokument-Typ-Definitionen werden anhand einer ELEMENT-Deklaration für XmlLog und einer ATTLIST-Deklaration für id/idref eingeführt. Schließlich werden Anfragen in Sprachen wie XQL funktional-logisch behandelt und Inferenzen auf der Basis von XmlLog vorgestellt.

1 Einführung

Die Einfachheit des Web-basierten Austauschs von Daten kommt nicht-formalen, semiformalen und formalen Dokumenten zugute. Für formale Spezifikationen und Programme erlaubt das Web eine verteilte Entwicklung, Nutzung und Wartung. Die Logikprogrammierung (LP) hat das Potential, hierfür als einheitliche Sprache zu dienen. Allerdings hat das World Wide Web Consortium (W3C) [http://www.w3.org/] inzwischen HTML - für nicht- und semiformale Dokumente - zur Extensible Markup Language (XML) [Har99] - für semiformale und formale Dokumente - weiterentwickelt.

Somit ergibt sich die Problemstellung zu den Beziehungen zwischen XML und LP. Hat die Logikprogrammierung trotz oder vielleicht gerade wegen XML die Chance, zu einer "Web-Technologie" für formale Dokumente zu werden? Könnte die HTML-artige Syntax von XML dafür durch eine Prolog-artige Syntax ersetzt werden oder in einer solchen editiert bzw. - über ein standardisiertes Stylesheet - präsentiert werden? Ist die SLD-Resolution ein geeigneter Ausgangspunkt für Interpretierer-Semantiken von XML-Anfragesprachen wie XQL [http://www.w3.org/TandS/QL/QL98/pp/xql.html] oder sollte unmittelbar eine LP-orientierte, inferenzielle Anfragesprache in Form eines XML-basierten Prologs entwickelt werden? Solche Fragen werden im Folgenden behandelt, wobei mögliche Wechselwirkungen zwischen XML und LP in beiden Richtungen angesprochen werden.

*Für die Einladung zu diesem Artikel möchte ich mich bei Ulrich Geske und dem Programmkomitee des 14. Workshop Logische Programmierung bedanken. Für die Durchsicht einer ersten Version geht mein Dank an Andreas Klüter und Michael Sintek.

Der bereits abzusehende Erfolg von XML als ein 'universelles' Austauschformat für die kommerzielle Praxis (inkl. E-Commerce) kann auch als Erfolg der deklarativen Repräsentationstechnik gesehen werden, wie sie in etwas anderer Form von der Logikprogrammierung vorgeschlagen wurde. Gemeinsamkeiten und Unterschiede dieser deklarativen Paradigmen werden später ausgeführt. Hier seien zwei Erfolgsfaktoren von XML erwähnt: (1) Eine ausreichende Kompatibilität von XML zum bereits weitverbreiteten HTML, z.B. über XHTML. (2) Die XML-Standardisierung durch das W3C, somit der Einbau von XML-Werkzeugen in gängige Browser wie Internet Explorer und Netscape/Mozilla, und damit das Wachstum des Anteils der XML-Dokumente im Web. Der ISO-Standard von Prolog [DEDC96] kann zwar wegen der fehlenden Web/HTML-Orientierung kaum dieselbe Breitenwirkung erzielen, hat aber eine stärker fokussierte Semantik. Insgesamt wäre also eine Gegenüberstellung der beiden Standards interessant, die detaillierter ist als der Anfang, der im Folgenden gemacht wird. Dadurch wäre der Weg für (partielle) Prolog-XML-Übersetzer, -Schnittstellen etc. in beiden Richtungen geebnet, womit sich (wesentliche) Vorteile beider Paradigmen kombinieren ließen.

Jenseits aller Anfangseuphorie bietet XML neue allgemeine Möglichkeiten, von denen auch die Logikprogrammierung profitieren kann: (1) Definition von selbstbeschreibenden Daten in einem weltweit normierten, nichtproprietären Format. (2) Strukturierter Daten- und Wissensaustausch in Unternehmen verschiedenster Branchen. (3) Integration von Informationen aus unterschiedlichen Quellen zu einheitlichen Dokumenten. Weiterhin ist XML die geeignetste Sprache für logische Wissensbasen im Web, wie bereits in [Bol97] angesprochen. Zusätzliche spezielle LP-Verwendungen von XML wären etwa der Austausch von Wissensbasen zwischen unterschiedlichen Logiksprachen sowie zwischen LP-Wissensbasen und Datenbanken, Anwendungssytemen etc. Selbst die Transformation bzw. Compilation von Logikquellprogrammen könnte auf der Basis von XML-Markup und -Annotationen geschehen. Diese und ähnliche Möglichkeiten werden im Folgenden ausgeführt.

Der vorliegende Text verwendet konkrete Gegenüberstellungen von LP- und XML-Beispielen, um (syntaktische) Unterschiede sowie (semantische) Gemeinsamkeiten herauszuarbeiten. Er wendet sich primär an einen Leserkreis der Grundzüge von XML über LP - und deduktive bzw. relationale Datenbanken - verstehen will; aber auch umgekehrt erscheinen Aspekte der LP über XML in neuem Licht. Die Abschnitte nach diesem (1.) sind wie folgt aufgebaut: 2. Zunächst führen wir 'elementares' XML ein und zeigen eine Element-Darstellung mittels Prolog-Strukturen. 3. Danach interessieren wir uns für die umgekehrte Richtung, wie Herbrand-Terme durch XML-Elemente darstellbar sind. 4. Darauf aufbauend geht es dann um eine XML-Darstellung von Horn-Klauseln (pures XmlLog). 5. Anschließend werden XML-Attribute wie `id/idref` für erweiterte Logiken verwendet. 6. Daraufhin werden XMLs Dokument-Typ-Definitionen für Logiksprachen wie XmlLog genutzt. 7. Sodann werden XML-Anfragesprachen wie XQL und inferenzielle Logikprogramme einander gegenübergestellt. 8. Zum Schluß diskutieren wir einige ähnliche Ansätze und geben einen Ausblick.

2 XML-Elemente als 'Ground'-Strukturen

XML-Dokumente bestehen aus (geschachtelten) Elementen. Jedes Element ist eine Folge der Art <*tag*> . . . </*tag*>, d.h. ein durch *tag*-'gefärbte' Start- und Endeklammern umschlossener *Inhalt* ". . .", der aus Text bzw. wiederum aus Elementen bestehen darf. Somit können ein 'Start-Tag' und ein 'End-Tag' zum Gesamt-Markup eines textlichen Inhalts verwendet

werden, der beliebig viel weiteres wohlgeklammertes Detail-Markup enthalten darf. Mit der Zunahme von XML-Markup in einem Dokument erhöht sich seine 'Formalheit' i.S. des am Anfang von Abschnitt 1 erwähnten Spektrums.

Betrachten wir zunächst ein **nichtformales** Minimal-Markup mit einem einzigen 'Tag'-Paar `<sentence> . . . </sentence>`:

```
<sentence> Onoffbook sold 12417 copies of XML4You online </sentence>
```

Durch grobes Zerlegen erhält man ein **semiformales** `<triple>`-Element, dessen `<predicate>`- und `<object>`-Unterelemente noch unalysierten Text enthalten:

```
<triple>
  <subject> Onoffbook </subject>
  <predicate> sold online </predicate>
  <object> 12417 copies of XML4You </object>
</triple>
```

Schließlich ergibt eine vollständige kontextfreie Zerlegung ein (syntaktisch) **formales** Dokument mit folgender `<s>`-verwurzelten Elementschachtelung:

```
<s>
  <np>
    <noun> Onoffbook </noun>
  </np>
  <vp>
    <vgroup> <verb> sold </verb> <adverb> online </adverb> </vgroup>
    <np>
      <ngroup>
        <card> 12417 </card>
        <noun> copies </noun>
      </ngroup>
      <pp>
        <prep> of </prep>
        <np>
          <noun> XML4You </noun>
        </np>
      </pp>
    </np>
  </vp>
</s>
```

Ein Dokument wird also durch XML in der Art eines Zerlegungs- oder Ableitungsbaums strukturiert, wobei jedes Element `<`*tag*`> . . . </`*tag*`>` ein *tag*-markierter Knoten (ein Nonterminal) ist, der geordnete gerichtete Kanten zu den in ". . ." vorkommenden direkten Unter-Elementen hat (hier durch Einrücken dargestellt). Die in ". . ." vorkommenden Texte sind die Blätter dieses Ableitungsbaums. Im Normalfall enthält jedes XML-Element entweder nur Unterelemente (entsprechend Nonterminalen) oder nur Text (entsprechend einem 'Token' von Terminalen).

Offensichtlich kann ein solches XML-Dokument - etwa in Prolog - als (variablenfreie, d.h.) 'Ground'-Struktur *tag*`( ., ., . )` aufgefasst werden, sofern sich die Texte an den Blättern

als Individuenkonstanten darstellen lassen. Dabei wird *tag* zum Konstruktor und der Inhalt ". . ." zu entsprechend transformierten Argumenten "., ., .". Ohne auf die genauen "Character Escape"-Konventionen von XML und Prolog einzugehen, stellen wir XML-Texte als Prolog-Strings dar; nach vollständiger Analyse verwenden wir Prolog-Individuenkonstanten (mit Kleinbuchstaben).

Die drei Stufen des Zerlegungsbeispiels führen somit in Prolog zu folgenden 'Ground'-Strukturen:

```
sentence("Onoffbook sold 12417 copies of XML4You online")

triple(
  subject("Onoffbook"),
  predicate("sold online"),
  object("12417 copies of XML4You")
      )

s(
  np(noun(onoffbook)),
  vp(
    vgroup(verb(sold),adverb(online)),
    np(ngroup(card(12417),noun(copies)),
       pp(prep(of),np(noun(xml4you))))
    )
 )
```

Unsere linguistische Analyse kann nun von dieser allgemeinen Syntax zu einer speziellen Semantik weitergehen, z.B. für den Diskursbereich "E-Commerce". Analytiker spezifischer Diskursbereiche springen sogar oft - ohne den bisher behandelten natürlichsprachlichen (XML-)Teil - direkt zu solchen Semantiken.

Damit erhalten wir in XML ein (semantisch) formales Dokument mit vier `<sales>`-Unterelementen, die den Domänen einer relationalen `<sales>`-Datenbanktabelle entsprechen:

```
<sales>
  <channel> online </channel>
  <company> Onoffbook </company>
  <item> XML4You </item>
  <quantity> 12417 <quantity>
</sales>
```

Dieses führt dann in Prolog zu folgender 'Ground'-Struktur:

```
sales(
  channel(online),
  company(onoffbook),
  item(xml4you),
  quantity(12417)
     )
```

3 Herbrand-Terme als XML-Elemente

Für die Darstellung von Herbrand-Termen (Individuenkonstanten, logischen Variablen und Strukturen) in XML verwenden wir im Wesentlichen `<ind>`-, `<var>`- und `<struc>`-Elemente.

Wie alle Elemente muss eine *Individuenkonstante* in XML explizit durch ein Markup als solche gekennzeichnet werden, hier durch ein `<ind>`-*Element* der Form `<ind>` ... `</ind>`. So wird etwa die Prolog-Individuenkonstante `channel-tunnel` für den Kanaltunnel zwischen Großbritannien und Frankreich zu dem XML-Element `<ind> channel-tunnel </ind>`.

Statt einem Individuensymbol kann in Prolog auch eine *('Ground'-)Struktur* wie `undersea-connection(britain,france)` zur Denotation eines Individuums verwendet werden. Diese kann man auch als 'Record' mit Komponenten für die verbundenen Länder betrachten. Eine Struktur wird in XML zu einem `<struc>`-*Element* mit einem eingebetteten Element für den Konstruktor, gefolgt von Elementen für seine Argumentterme, hier zwei Individuenkonstanten:

```
<struc>
  <constructor> undersea-connection </constructor>
  <ind> britain </ind>
  <ind> france </ind>
</struc>
```

Da das XML-Markup die Syntax der urspünglichen Prolog-Struktur explizit macht, kann daraus sowohl diese wiedergewonnen als auch - in der Art von Abschnitt 2 - eine Prolog-Struktur konstruiert werden, welche diese Syntax wie folgt beschreibt:

```
struc(
  constructor(undersea-connection),
  ind(britain),
  ind(france)
     )
```

Ist bereits die urspüngliche Struktur geschachtelt wie beispielsweise

```
service-tunnel(
  undersea-connection(
    britain,
    surrounded-country(belgium,luxembourg,germany,switzerland,italy,spain)))
```

führt dies zu eingebetteten `<struc>`-Elementen wie in

```
<struc>
  <constructor> service-tunnel </constructor>
  <struc>
    <constructor> undersea-connection </constructor>
    <ind> britain </ind>
    <struc>
      <constructor> surrounded-country </constructor>
      <ind> belgium </ind>
      <ind> luxembourg </ind>
      <ind> germany </ind>
```

```
      <ind> switzerland </ind>
      <ind> italy </ind>
      <ind> spain </ind>
    </struc>
  </struc>
</struc>
```

Durch die 'Start-Tag'/'End-Tag'-Klammerung werden XML-Elemente immer explizit mit ihrem Typ gekennzeichnet. So sieht man etwa in Prolog den Symbolen `channel-tunnel` und `service-tunnel` nicht unmittelbar an, dass das erste eine Individuenkonstante und das zweite ein Konstruktor ist. In XML werden solche Unterscheidungen wie in streng typisierten LP-Sprachen explizit gemacht - allerdings an jeder Auftrittsstelle eines Symbols. Obiges Beispiel gibt einen Eindruck vom Selbstbeschreibungsvorteil - aber auch vom 'Platzbedarf' - dieses großzügigen Umgangs mit "syntaktischem Zucker". Statt jede schließende XML-Klammer redundant als volles 'End-Tag' zu schreiben, könnte zwar - in Annäherung an Prolog-Strukturen - wie in XML-QL [http://www.w3.org/TR/NOTE-xml-ql/] die 'neutrale' schließende Klammer `</>` verwendet werden, dies ist aber nicht in XML 1.0 standardisiert.

Listen - wie sie z.B. in Prolog verwendet werden - können auch in XML auf geschachtelte `cons`-Strukturen reduziert oder direkt als n-Tupel dargestellt werden [Bol99].

Eine *logische Variable* wird in XML durch ein `<var>`-*Element* der Form `<var>` ... `</var>` markiert. So wird etwa die Prolog-Variable `Xyz` zu dem XML-Element `<var> xyz </var>`, wobei sich durch das `</var>`-Markup Konventionen wie die Großschreibung von Anfangsbuchstaben erübrigen.

Damit wird eine (nicht variablenfreie, d.h.) 'Non-ground'-Struktur wie Prologs `undersea-connection(britain,Xyz)` zu XMLs 'Non-ground'-Element

```
<struc>
  <constructor> undersea-connection </constructor>
  <ind> britain </ind>
  <var> xyz </var>
</struc>
```

4 Horn-Klauseln als XML-Elemente

Für die Darstellung von Horn-Klauseln (Fakten und Regeln) führen wir in XML weitere Elemente ein.

Ein Prädikaten- oder *Relationssymbol* wird in XML zu einem `<relator>`-*Element*. Als Beispiel betrachten wir das Relationssymbol `travel`, welches zu `<relator> travel </relator>` wird.

Die *Anwendung eines Relationssymbols auf Terme* markieren wir in XML mit einem `<relationship>`-*Element*. Eine `travel`-Anwendung `travel(john,channel-tunnel)` auf zwei Individuenkonstanten wird beispielsweise zu

```
<relationship>
  <relator> travel </relator>
  <ind> john </ind>
  <ind> channel-tunnel </ind>
</relationship>
```

Darüber lässt sich ein Horn-Fakt in XML als ein `<hn>`-*Element* assertieren, welches genau ein Unterelement - das `<relationship>`-Element - besitzt. Im Beispiel wird der Prolog-Fakt

```
travel(john,channel-tunnel).
```

zu

```
<hn>
  <relationship>
    <relator> travel </relator>
    <ind> john </ind>
    <ind> channel-tunnel </ind>
  </relationship>
</hn>
```

Zur Darstellung von Regeln werden nun keine weiteren XML-Elementarten benötigt.

Einen *Relationsaufruf* schreiben wir nämlich wiederum als `<relationship>`-*Element*. Der `carry`-Aufruf `carry(eurostar,Someone)` mit einer Individuenkonstanten und einer logischen Variablen wird etwa zu

```
<relationship>
  <relator> carry </relator>
  <ind> eurostar </ind>
  <var> someone </var>
</relationship>
```

Damit lässt sich eine Horn-Regel in XML als ein `<hn>`-*Element* assertieren, welches zwei oder mehr Unterelemente - das Kopf-`<relationship>`-Element gefolgt von mindestens einem Rumpf-`<relationship>`-Element - aufweist. Das obige Beispiel ist somit zu einer Prolog-Regel

```
travel(Someone,channel-tunnel) :- carry(eurostar,Someone).
```

verallgemeinerbar und wird in XML umgeschreiben als

```
<hn>
  <relationship>
    <relator> travel </relator>
    <var> someone </var>
    <ind> channel-tunnel </ind>
  </relationship>
  <relationship>
    <relator> carry </relator>
    <ind> eurostar </ind>
    <var> someone </var>
  </relationship>
</hn>
```

Pures XmlLog - das bisher anhand von Beispielen eingeführte pure Prolog in XML-Syntax - wird in Abschnitt 6 präzise definiert.

5 Attribute für erweiterte Logiken

Obwohl geschachtelte Elemente - als Bäume - die Darstellung beliebiger Informationen erlauben, führen sie in manchen Situationen zu unnötig tief bzw. breit verschachtelten Repräsentationen. Oft lässt sich dann in XML durch *Attribute* Abhilfe schaffen: Ein 'Start-Tag' kann um n Attribute-Wert-Paare der Form a_i=v_i angereichert werden; i.A. ist somit jedes Element eine Folge der Art <*tag* a_1=v_1...a_n=v_n> . . . </*tag*>.

So kann z.B. das `<sales>`-Element am Ende von Abschnitt 2 durch die Umwandlung des `<channel>`-Unterelements in ein `channel`-Attribut mit dem "String"-Wert `"online"` etwas kompakter und intuitiver notiert werden:

```
<sales channel="online">
          <company> Onoffbook </company>
          <item> XML4You </item>
          <quantity> 12417 <quantity>
</sales>
```

Während es dazu in Prolog kein direktes Gegenstück gibt, ermöglichen andere Logikprogrammiersprachen eine entsprechende Umwandlung der ursprünglichen Prolog-Struktur von Abschnitt 2. Im funktional-logischen Relfun [http://www.relfun.org/] bekäme man etwa - abgesehen von der Verwendung eckiger Klammern - die Struktur

```
sales(channel(online))(
          company(onoffbook),
          item(xml4you),
          quantity(12417)
                    )
```

mit einem durch das Attribut parametrisierten Konstruktor `sales(channel(online))`.

Zusätzliche Informationen lassen sich ebenfalls oft am besten unmittelbar als Attribute statt als Unterelemente einführen. Das in Abschnitt 2 verwendete `<noun> copies </noun>` kann beispielsweise durch Attribute für Genus und Numerus zu `<noun genus="neutrum" numerus="plural"> copies </noun>` bzw. `noun(genus(neutrum),numerus(plural))(copies)` erweitert werden. I.A. können damit Feature-Grammatiken in XML und LP abgebildet werden.

Auch unabhängig von Grammatiken lassen sich XML-Attribute in der Logikprogrammierung zur Darstellung von Feature-/Psi-Termen benutzen. Da die Werte in a_i=v_i in XML 1.0 atomar sind, sind auf direkte Weise allerdings nur Feature-/Psi-Terme mit entsprechend atomaren Werten repräsentierbar. Dies ist aber z.B. für Beschreibungslogiken [CDL99] kein Problem.

Eine nützliche Verwendung von XML-Attributen für Prolog sind speziell die Aritätskennzeichnungen bei Relationssymbolen. So könnte man in Abschnitt 4 ein als binär gekennzeichnetes Relationssymbol `travel/2` in XML mit einem `arity`-Attribut als `<relator arity="2"> travel </relator>` schreiben. Gleiches gilt für Aritätskennzeichnungen von Konstruktoren und - in funktional-logischen Sprachen - von (definierten) Funktionssymbolen.

Analog sind über Attribute Annotationen auf beliebigen Elementebenen möglich - u.a. 'Mode'-Deklarationen für logische Variablen, Determinismus-Spezifikationen für Klauseln oder Prozeduren und Kontext-Bedingungen für komplette Wissensbasen.

Schließlich können die Annotationen, welche zur Compilation von Logikquellprogrammen benötigt werden, selbst in einer deklarativen Zwischensprache dargestellt werden, z.B. in Form von "klassifizierten Klauseln" [Kra90]; diese sind nun als attributierte XML-Elemente darstellbar.

Außer den bisher verwendeten Attributen mit Werten vom allgemeinen Typ `CDATA` ("Character Data") kennt XML auch Attribute von spezifischerem Typ. So gibt es insbesondere die Typen `ID` und `IDREF` zur Benennung und Referenzierung von Elementen eines gegebenen XML-Dokuments. Ein `ID`-typisierter Wert muss eindeutig ein Element identifizieren und ein `IDREF`-typisierter Wert muss auch als `ID`-Wert eines Elements vorkommen.

Mit diesen XML-Mitteln lassen sich viele meta- und modallogische Sachverhalte ausdrücken, da z.B. Klauseln benannt und ihre Namen für die Bildung weiterer Klauseln benutzt werden können. Wir nehmen in diesem Abschnitt zunächst an, dass - wie in vielen XML-Werkzeugen - die Attribute `id` bzw. `idref` vom Typ `ID` bzw. `IDREF` sind. In einer solchen Erweiterung unseres puren XmlLogs können wir z.B. den Prolog-Fakt von Abschnitt 4 wie folgt in einer Teilwissensbasis benennen:

```
<hn id="john-channel">
  <relationship>
    <relator> travel </relator>
    <ind> john </ind>
    <ind> channel-tunnel </ind>
  </relationship>
</hn>
```

Von einer anderen Teilwissensbasis aus können wir dann zur Darstellung des "modalen Prolog-Fakts"

```
belief(mary,travel(john,channel-tunnel)).
```

wie folgt auf die Aussage `"john-channel"` zugreifen:

```
<hn>
  <relationship>
    <relator> belief </relator>
    <ind> mary </ind>
    <prop idref="john-channel"/>
  </relationship>
</hn>
```

Hier wird die XML-Abkürzung von leeren Elementen <*tag* ...> </*tag*> zu <*tag* .../> benutzt, um das propositionale Argument des `belief`-Operators als `<prop idref="john-channel"/>` recht kompakt zu notieren. Man beachte, dass obige Prolog-artige Notation für Propositionen als Argumente - ohne Deklaration von Modaloperatoren wie `belief` bzw. Trennung von Relationssymbolen und Konstruktoren - nicht von der für Strukturen als Argumente unterscheidbar ist. Dagegen sind in der selbstbeschreibenden XML-Notation die 'Tags' für `prop` und `struc` klar unterschieden - ganz abgesehen davon, dass wir Propositionen in XML nur durch ihre Namen repräsentiert haben.

Über `ID`/`IDREF` lässt sich die Baumstruktur von XML zu azyklischen gerichteten Graphen (DAGs) verallgemeinern. So kann etwa ein `disbelief`-Fakt für `fred` mit `idref` auf dieselbe

"john-channel"-Aussage zugreifen, d.h. ID/IDREF ermöglichen "Sharing". Darüber hinaus gelangt man so auch zu allgemeinen gerichteten Graphen, mit denen sich z.B. reguläre Bäume in Prolog II+ [Col84] darstellen lassen.

6 Dokument-Typ-Definitionen für Logiksprachen

In den Abschnitten 3 und 4 haben wir eine Hornlogik-Sprache als pures XmlLog anhand von Beispielen eingeführt. XML erlaubt aber auch ihre allgemeine Definition.

Um Sprachen syntaktisch zu definieren, verwendet XML *Dokument-Typ-Definitionen (DTDs)*, die wir zunächst nur als ELEMENT-*Deklarationen* für **nicht-attributierte Elemente** betrachten. Auf der Ebene der Nonterminale entspricht eine DTD einer gewöhnlichen kontextfreien Grammatik in einer abgewandelten (EBNF-)Notation. Auf der Ebene der Terminale werden aber meist beliebige Permutationen des zugrunde gelegten Alphabets (sog. "Parsed Character Data", kurz "PCDATA") verwendet, statt feste Terminal-Folgen zu spezifizieren. Die in DTDs benutzte Abkürzung #PCDATA kann auch als Nonterminal betrachtet werden, das (als Kleenescher Abschluss) zu allen 'Tokens' expandiert, welche mit dem Alphabet gebildet werden können.

Mit einer DTD-Grammatik kann man also kontextfreie Wortmuster mit beliebigen 'Tokens' an den Blättern ableiten. Dies ist aber nur sinnvoll, wenn man dabei die Ableitungsbäume selbst - in einer durch Klammern linearisierten Form - als das Generierungsergebnis auffasst. In XML geschieht dies durch 'Stehenlassen' der bekannten Klammern <*tag*> . . . </*tag*> für jedes Nonterminal *tag*.

Ein gegebenes XML-Element wird als *valid* bzgl. einer DTD bezeichnet, wenn es von der DTD auf diese Weise als linearisierter Ableitungsbaum generiert werden kann.

Als Beispiel einer DTD betrachten wir die syntaktische ELEMENT-Deklaration unseres puren XmlLogs als einer Wissensbasis (kb) von null oder mehr Horn-Klauseln (hn*):

```
<!ELEMENT kb            (hn*) >
<!ELEMENT hn            (relationship, relationship*) >
<!ELEMENT relationship  (relator, (ind | var | struc)*) >
<!ELEMENT struc         (constructor, (ind | var | struc)*) >
<!ELEMENT relator       (#PCDATA) >
<!ELEMENT constructor   (#PCDATA) >
<!ELEMENT ind           (#PCDATA) >
<!ELEMENT var           (#PCDATA) >
```

Mit dieser DTD zeigen wir eine Generierung des XML-linearisierten Ableitungsbaums für die kb-eingebettete Beispielregel in Abschnitt 4, die man auch - wie ein XML-Validierer - von unten nach oben als Zerlegung lesen kann. Man beachte, wie z.B. das Startsymbol kb die von ihm abgeleitete(n) Klausel(n) im Weiteren als Teil der linearisierten 'Start-Tag'/'End-Tag'-Baumrepräsentation <kb> . . . </kb> umklammert. Die ausgelassenen mittleren Schritte sind leicht ergänzbar:

```
kb

<kb> hn* </kb>

<kb> hn </kb>
```

```
<kb> <hn> relationship relationship* </hn> </kb>

<kb> <hn> relationship relationship </hn> </kb>

.  .  .

<kb>
  <hn>
    <relationship>
      <relator> #PCDATA </relator>
      <var> #PCDATA </var>
      <ind> #PCDATA </ind>
    </relationship>
    <relationship>
     <relator> #PCDATA </relator>
     <ind> #PCDATA </ind>
     <var> #PCDATA </var>
    </relationship>
  </hn>
</kb>

<kb>
  <hn>
    <relationship>
      <relator> travel </relator>
      <var> someone </var>
      <ind> channel-tunnel </ind>
    </relationship>
    <relationship>
      <relator> carry </relator>
      <ind> eurostar </ind>
      <var> someone </var>
    </relationship>
  </hn>
</kb>
```

Wir betrachten nun DTDs für **attributierte Elemente**. Dafür verwendet man `ATTLIST`-*Deklarationen*, welche einem Elementnamen einen Attributnamen mit seinem Attributtyp und eventueller Auftretenskennzeichnung zuordnen.

In einem ersten Beispiel deklarieren wir das in Abschnitt 5 verwendete `relator`-Attribut `arity` als `CDATA`-typisiert (analog `#PCDATA`) und optional auftretend (`#IMPLIED`):

```
<!ATTLIST relator       arity CDATA #IMPLIED >
```

Als zweites Beispiel definieren wir das in Abschnitt 5 eingeführte erweiterte XmlLog mit benannten `hn`-Klauseln und eingebetteten Propositionen. Dazu ergänzen wir zunächst obige `ELEMENT`-Deklaration für `relationship` um `prop` und deklarieren letzteres als leeres Element:

```
<!ELEMENT relationship  (relator, (ind | var | struc | prop)*) >
<!ELEMENT prop          EMPTY >
```

Dann fügen wir eine `ATTLIST`-Deklaration an, welche für die `hn`- bzw. `prop`-Elemente die Attribute `id` - als optionalen `ID`-Typ - bzw. `idref` - als mandatorischen `IDREF`-Typ - spezifiziert:

```
<!ATTLIST hn            id    ID    #IMPLIED >
<!ATTLIST prop          idref IDREF #REQUIRED >
```

Mit der gesamten DTD lassen sich nun z.B. der `"john-channel"`-benannte Fakt von Abschnitt 5 und die darauf zugreifenden Fakten generieren/zerlegen.

7 XML-Anfragesprachen und Logikprogrammierung

Bei den XML-Anfragesprachen ist die W3C-Standardisierung noch nicht so weit fortgeschritten wie bei XML selbst. Trotzdem ist bereits ein vorläufiger Vergleich mit der Logikprogrammierung möglich, der hier auf der Basis von XQL [http://www.w3.org/TandS/QL/QL98/pp/xql.html] und XmlLog exemplarisch begonnen wird. Wir betrachten die Beziehungen wiederum in beiden Richtungen.

Für eine LP-Behandlung von XQL-Anfragen legen wir als Beispiel ein Dokument mit der `<s>`-verwurzelten Elementschachtelung von Abschnitt 2 zugrunde, das um Attribute für Genus und Numerus entsprechend Abschnitt 5 erweitert ist. Das Dokument sei über eine Variable `Root` zugreifbar.

Durch die XQL-Anfrage `//`*tag* werden alle Elemente der Form `<`*tag ...*`>` . . . `</`*tag*`>` des Dokuments zurückgegeben. Im Beispiel liefert etwa `//np` folgende drei Elemente, wobei das dritte im zweiten enthalten ist:

```
<np>
  <noun genus="neutrum" numerus="singular"> Onoffbook </noun>
</np>

<np>
  <ngroup>
    <card> 12417 </card>
    <noun genus="neutrum" numerus="plural"> copies </noun>
  </ngroup>
  <pp>
    <prep> of </prep>
    <np>
      <noun genus="neutrum" numerus="singular"> XML4You </noun>
    </np>
  </pp>
</np>

<np>
  <noun genus="neutrum" numerus="singular"> XML4You </noun>
</np>
```

Dieses Verhalten ließe sich in reiner Logikprogrammierung durch einen Relationsaufruf `descendant(Root,np,Res)` realisieren, welcher eine Resultatvariable nichtdeterministisch bindet. Näher an der wertorientierten Denkweise von XQL kann - etwa in einer funktional-logischen Sprache - ein Funktionsaufruf `descendant(Root,np)` verwendet werden, der folgende Prolog-Strukturen als nichtdeterministisch aufgezählte Werte zurückgibt:

```
np(noun(genus(neutrum),numerus(singular))(onoffbook))

np(ngroup(card(12417),noun(genus(neutrum),numerus(plural))(copies)),
   pp(prep(of),np(noun(genus(neutrum),numerus(singular))(xml4you))))

np(noun(genus(neutrum),numerus(singular))(xml4you))
```

Durch die XQL-Anfrage `//`*tag*`[@`*a*`=`*v*`]` werden alle Elemente der Form `<`*tag* *...a=v...*`>` . . . `</`*tag*`>` des Dokuments zurückgegeben, welche das Attribute-Wert-Paar *a*=*v* enthalten. Im Beispiel liefert etwa `//noun[@numerus="plural"]` nur folgendes Element:

```
<noun genus="neutrum" numerus="plural"> copies </noun>
```

Der entsprechende, durch einen Filter parametrisierte Funktionsaufruf `descendant-filter(numerus(plural))(Root,noun)` gibt die korrespondierende Prolog-Struktur zurück:

```
noun(genus(neutrum),numerus(plural))(copies)
```

Die Definition von Anfragefunktionen wie `descendant` und `descendant-filter( )` kann natürlich nicht nur aus (deterministischen!) Prolog-Unifikationen bestehen: Alle Prolog-Unterstrukturen mit den angegebenen Konstuktoren müssen durch rekursives Traversieren gefunden und Filter auf die Konstruktorparameter angewandt werden.

Umgekehrt erlaubt die Unifikation logische Variablen aber nicht nur in der Anfrage, sondern auch im Dokument, wie etwa in dem Element für die 'Non-ground'-Struktur `undersea-connection(britain,Xyz)` von Abschnitt 3. Dadurch wären auch auf Dokumenten echt unifizierende Anfragen entsprechend zu `undersea-connection(Same,Same)` möglich, die `Same` an `britain` bindet. Das abgespeicherte, universell zu lesende 'Non-ground'-Element dokumentiert nämlich u.a. die Behauptung, dass eine britische Insel mit einer anderen unterseeisch verbunden ist. Inwieweit 'Non-ground'-Dokumente - etwa als Formulare mit z.T. identisch auszufüllenden Feldern - in der Praxis Verwendung finden, muss die Erfahrung zeigen.

Wenden wir uns jetzt Anfragen und Inferenzen auf der Basis des in den Abschnitten 4 und 6 definierten puren XmlLogs zu.

Betrachten wir den Fakt `carry(eurostar,fred)`. Er kann in seiner XmlLog-Version

```
<hn>
  <relationship>
    <relator> carry </relator>
    <ind> eurostar </ind>
    <ind> fred </ind>
  </relationship>
</hn>
```

benutzt werden, um die in Abschnitt 4 gezeigte XmlLog-Version der Anfrage `carry(eurostar,Someone)` zu beantworten, welche die XML-Logikvariable `<var> someone </var>` an `<ind> fred </ind>` bindet.

Mit solchen `carry`-Basisfakten ist nun eine Regel verwendbar, um `travel`-Aussagen bedarfsweise dynamisch herzuleiten, statt sie pauschal statisch abspeichern zu müssen: `travel(Someone,channel-tunnel) :- carry(eurostar,Someone).` Ihre XML-Version von Abschnitt 4 ist nämlich für inferenzielle Anfragen wie `travel(fred,Where)` in XML-Versionen wie

```
<relationship>
  <relator> travel </relator>
  <ind> fred </ind>
  <var> where </var>
</relationship>
```

verwendbar. Diese Anfrage unifiziert über `<var> someone </var>` = `<ind> fred </ind>` und `<var> where </var>` = `<ind> channel-tunnel </ind>` mit dem Kopf der Regel. Die mit der ersten Bindung aus dem Rumpf der Regel resultierende interne XML-Anfrage entsprechend `carry(eurostar,fred)` ist mit obigem XML-Fakt erfolgreich. Damit wird die zweite Bindung, d.h. `Where` = `channel-tunnel` als Ergebnis der externen XML-Anfrage geliefert. Insgesamt wird die Inferenz also als ein SLD-Resolutionsschritt [Llo87] durchgeführt.

Allgemein lässt sich die prozedurale Semantik der SLD-Resolution auf XmlLog übertragen. Wird hierfür eine Verteilung der Klauseln über verschiedene Dokumente im Web angenommen, befindet man sich in der in [Bol97] beschriebenen Situation: In dieser "offenen Welt" kann insbesondere kaum noch logische Vollständigkeit gefordert werden. Die (Java-)Realisierung der Semantik in Form eines XmlLog-Interpretierers könnte auf der Client-Seite als Standarderweiterung - oder "Plugin" - eines Web-Browsers geschehen.

8 Verwandte Arbeiten und Schlussfolgerungen

Das hier vorgestellte pure XmlLog ist eine Teilsprache der Relational-Functional Markup Language (RFML) [Bol99], die u.a. auch benutzerdefinierte Funktionen und Operatoren höherer Ordnung zulässt [http://www.relfun.org/rfml.ps].

SHOE (Simple HTML Ontology Extensions) [HHL99] ist die erste bekannt gewordene Sprache, die Horn-Klauseln im Web bereitstellt. Sie wurde - vor der Definition von XML - zunächst in HTML spezifiziert, inzwischen aber auch in XML [http://www.cs.umd.edu/projects/plus/SHOE/]. Ein interessanter Aspekt von SHOE ist, dass Individuenkonstanten durch den (offiziellen) URL/URI für das Individuum repräsentiert werden. In unserem Horn-Regel-Beispiel in Abschnitt 4 treten zwei Individuen auf, die man auf diese Weise als `eurostar` = `http://www.eurostar.com/` und `channel-tunnel` = `http://www.eurotunnel.com/` repräsentieren kann. Damit wird diese Regel in SHOE zu

```
<DEF-INFERENCE DESCRIPTION="travel(?someone,http://www.eurotunnel.com/) if
                            carry(http://www.eurostar.com/,?someone)">
        <INF-IF>
                <RELATION NAME="carry">
                        <ARG POS=1 VALUE="http://www.eurostar.com/">
                        <ARG POS=2 VALUE="someone" USAGE=VAR>
                </RELATION>
        </INF-IF>
```

```
        <INF-THEN>
                <RELATION NAME="travel">
                        <ARG POS=1 VALUE="someone" USAGE=VAR>
                        <ARG POS=2 VALUE="http://www.eurotunnel.com/">
                </RELATION>
        </INF-THEN>
</DEF-INFERENCE>
```

"Courteous Logic Programs" [GLC99] wurden für die priorisierte Konfliktbehandlung bei der Agenten-Kommunikation im E-Commerce (z.B. bei Lieferverträgen) untersucht. Zum Wissensaustauch wird dazu eine XML-basierte Business Rules Markup Language (BRML) entwickelt [http://www.oasis-open.org/cover/brml.html], welche auch die Klauselteilmenge des Knowledge Interchange Format (KIF) [http://logic.stanford.edu/kif/dpans.html] in XML codiert.

Von den vielen vorgeschlagenen XML-Anfragesprachen [http://www.w3.org/TandS/QL/QL98/] haben wir bisher nur XQL [http://www.w3.org/TandS/QL/QL98/pp/xql.html] und XML-QL [http://www.w3.org/TR/NOTE-xml-ql/] erwähnt. Ein alternativer, benutzerorientierter Vorschlag ist EquiX, bei dem aus Anfragen über eine formularbasierte Schnittstelle automatisch Resultatsdokumente mit ihren DTDs konstruiert werden [CYK+99]. Die im funktionalen Haskell implementierte Algebra für XML-Anfragesprachen [http://www.cs.bell-labs.com/who/wadler/papers/xquery-algebra/xquery-algebra.html] ließe sich wohl mit dem Haskell-verallgemeinernden funktional-logischen Curry [Han97] in die Logikprogrammierung übertragen.

Aus Platzgründen können die XML-Namespaces hier nicht behandelt werden, obwohl interessante Zusammenhänge zu den Modulsystemen von LP-Sprachen bestehen. Letztere waren bekanntlich der schwierigere Teil bei der ISO-Standardisierung von Prolog [http://www.logic-programming.org/prolog_std.html]; andererseits sind sie nicht völlig spezifisch für die Logikprogrammierung.

Auch auf die darauf aufbauenden XML-Erweiterungen RDF - zur Metadaten-Beschreibung - und RDF Schema - zur entsprechenden Schema/Ontologie-Spezifikation - kann hier nicht näher eingegangen werden. Für RDF-Anfragen und -Inferenzen kommen aber ebenfalls Techniken aus dem Bereich der Logikprogrammierung in Betracht, etwa eine Verwendung der F-Logic [DBSA97]. Als RDF-Editor wäre eine Variante von Protégé-2000 geeignet [http://smi-web.stanford.edu/projects/protege/protege-rdf/protege-rdf.html].

In diesem Zusammenhang sei bemerkt, dass XML 1.0 nicht die gewohnten eingebauten Typen wie `integer`, `float` etc. anbietet. Daher mussten wir z.B. in Abschnitt 6 das Attribut `arity` mit unspezifischen `CDATA`-"Strings" typisieren, obwohl dafür nur die natürlichen Zahlen in Frage kommen. Hier schafft XML Schema mit dem differenzierten Typsystem in Teil 2 Abhilfe [http://www.w3.org/TR/xmlschema-2/]. Weitere Untersuchungen könnten sich also den Beziehungen von Typ-/Sorten- bzw. Signatursystemen von (Feature-/Psi-Term-)LP-Sprachen zu denen von XML Schema und RDF Schema widmen.

Insgesamt eröffnet XML - in Weiterführung von Internet-Workshops auf LP-Konferenzen seit 1996 - ein neues Untersuchungs- und Anwendungsgebiet für die Logikprogrammierung. Beide Gebiete können von den angesprochenen Wechselwirkungen profitieren. Da manche der erwähnten XML-Erweiterungen als Vorentwürfe noch nicht (W3C-)standardisierungsreif sind, bestünde sogar z.T. noch die Möglichkeit, Einfluss auf ihre endgültige Form zu nehmen.

Literatur

[Bol97] Harold Boley. Wissensbasen im World Wide Web: Eine Herausforderung für die Logische Programmierung. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *Twelfth Workshop on Logic Programming (WLP'97)*. Institut für Informatik, LMU München, September 1997.

[Bol99] Harold Boley. The Relational-Functional Markup Language RFML. Technical report, Stanford Medical Informatics, December 1999.

[CDL99] D. Calvanese, G. De Giacomo, and M. Lenzerini. Representing and Reasoning on XML Documents: A Description Logic Approach. *JLC: Journal of Logic and Computation*, 9, 1999.

[Col84] A. Colmerauer. Equations and Inequations on Finite and Infinite Trees. In *Proc. of the International Conference on Fifth Generation Computer Systems (FGCS-84)*, pages 85–99, Tokyo, Japan, November 1984. ICOT.

[CYK+99] S. Cohen, Y.Kanza, Y. Kogan, W. Nutt, A. Serebrenik, and Y. Sagiv. EquiX – Easy Querying in XML Databases. In *Proc. of the ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Penn., June 1999.

[DBSA97] Stefan Decker, Dan Brickley, Janne Saarela, and Jürgen Angele. A Query and Inference Service for RDF. In *QL'98 - The Query Languages Workshop, http://www.w3.org/TandS/QL/QL98/*. World Wide Web Consortium, 1997.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer Verlag, 1996.

[GLC99] Benjamin N. Grosof, Yannis Labrou, and Hoi Y. Chan. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In *Proc. 1st ACM Conference on Electronic Commerce (EC-99), Denver, Colorado*, 1999.

[Han97] Michael Hanus. A Unified Computation Model for Functional and Logic Programming. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 80–93, Paris, France, January 1997.

[Har99] Elliotte Rusty Harold. *XML Bible*. IDG Books Worldwide Inc., San Mateo, CA, USA, 1999.

[HHL99] Jeff Heflin, James Hendler, and Sean Luke. SHOE: A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078, University of Maryland, College Park, October 1999.

[Kra90] Thomas Krause. Klassifizierte relational/funktionale Klauseln: Eine deklarative Zwischensprache zur Generierung von Register-optimierten WAM-Instruktionen. Technical Report SWP–90–04, University of Kaiserslautern, Department of Computer Science, May 1990.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Heidelberg, New York, 1987.

Diagnostische Problemlösungsmethoden

Frank Puppe
Universität Würzburg, Lehrstuhl für Künstliche Intelligenz und Angewandte Informatik
http://ki.informatik.uni-wuerzburg.de/

1 *Einleitung*

Problemlösungsmethoden (problem solving methods) bestimmen, wie Wissen zur Problemlösung verwendet wird. Während das Interesse in den siebziger Jahre ausschlielich allgemeinen, universell anwendbaren Problemlösungsmethoden wie Mittel-Zweck-Analyse im "General Problem Solver" oder Forwärts– bzw. Rückwärtsverkettung von Regeln galt, und anschlieend das andere Extrem dominierte, die Betonung von speziellem Domänenwissen unter Vernachlässigung der Verarbeitungsmethode ("in the knowledge lies the power"), wird inzwischen das Zusammenspiel von Wissen und Methoden als entscheidend angesehen. Dabei stehen nicht mehr universelle Problemlösungsmethoden im Vordergrund, sondern problemspezifische Methoden, die nicht nur dazu dienen, bereits erworbenes Wissen zu interpretieren, sondern auch bei der Akquisition und Strukturierung des Wissens helfen. Insbesondere Problemlösungsmethoden, die mit den mentalen Modellen von Fachexperten übereinstimmen, vereinfachen den Wissenserwerbsproze erheblich. Daher kommt der Auswahl bzw. Konstruktion einer angemessenen Problemlösungsmethode für einen Anwendungsbereich eine entscheidende Rolle bei einer ökonomischen und erfolgreichen Wissensakquisition zu. Aufgrund der umfangreichen Forschung insbesondere in den neunziger Jahren gibt es heute umfangreiche Sammlungen von Problemlösungsmethoden, die für bestimmte Anwendungsbereiche geeignet sind.

Deren wichtigste Unterscheidung ist die in analytische oder synthetische Anwendungsbereiche. Bei den analytischen wird eine Problemlösung ausgewählt ("Klassifikation, Diagnostik"), bei den synthetischen wird sie aus primitiven Bausteinen konstruiert ("Konstruktion"). Während die Problemklasse Klassifikation ein relativ gut verstandener Bereich ist, ist die Problemklasse Konstruktion insgesamt wesentlich heterogener mit eigenständigen Teilbereichen wie Konfiguration, Planung, Scheduling usw. Innerhalb dieser Hauptklassen unterscheiden sich Problemlösungsmethoden vor allem in ihren Annahmen über den Anwendungsbereich sowie darin, welche Art von Wissen sie benutzen, z.B. Fall–orientiertes Wissen, Erfahrungswissen oder modellbasiertes Wissen. Anwendungsbereiche, die mit derselben Problemlösungsmethoden lösbar sind, können zu Problemlösungstypen zusammengefat werden. Eine andere ebenfalls gebräuchliche Unterteilung orientiert sich weniger an den Methoden, sondern an Aufgabentypen oder Problemtypen (vgl. [Puppe 90, Fensel 2000]). Dazu gehören z.B. im Bereich der Klassifikation Fehlersuche (Diagnostik im engeren Sinne), Produktauswahlberatung, Objektidentifikation, Bewertung usw.

Problemlösungsmethoden sind doppelt parametrisierte Algorithmen, die neben der normalen Parametrisierung durch Daten auch entsprechend dem Domänenwissen parametrisiert sind. So ist es für die Diagnosemethode grundsätzlich unerheblich, ob Fehler in Autos oder Krankheiten beim Menschen diagnostiziert werden. Jedoch gibt es in beiden Bereichen Problemmerkmale wie "Auto springt nicht an" oder "Fieber" und Problemlösungen wie "Zündkerzen verbraucht" oder "Hepatitis". Die Abbildung von konkreten Domänenbegriffen auf abstrakte Metaklassen (Konzepte) ist daher zentral zur Beschreibung einer Problemlösungsmethode. Da ähnliche Problemlösungsmethoden sich in vielen Inferenzschritten gleichen, ist es ökonomisch, diese ebenfalls zu standardisieren und bei der Beschreibung der Methoden wiederzuverwenden. Die Inferenzschritte haben als Ein- und Ausgabe jeweils Metaklassen. Neben der Aufzählung ihrer Inferenzschritte ist schlielich zur Beschreibung einer Problemlösungsmethode noch die Kontrollstruktur notwendig, die mit den üblichen Konstrukten wie Komposition, Verzweigung, Schleife usw. beschrieben wird. Diese drei Beschreibungsebenen – Domänen–, Inferenz– , und Aufgaben– oder Kontrollebene – sowie eine grafische Notation dafür wurden schon in den ersten KADS–Beschreibungen [Wielinga & Breuker 86] eingeführt und sind heute weit verbreitet (vgl. Breuker 94). Im folgenden beschreiben wir Problemlösungsmethoden für die Problemklasse Klassifikation. Für die Problemklasse Konstruktion wird auf [Biundo et al. 1995, Günter & Kühn 99] verwiesen.

2 *Inferenzschritte für diagnostische Problemlösungsmethoden*

Klassifikationsprobleme bestehen aus zwei endlichen, disjunkten Mengen von Problemmerkmalen (Merkmale, Symptome) und Problemlösungen (Lösungen, Diagnosen) und aus Wissen über die Beziehungen zwischen Merkmalen und Lösungen. Ein Problem ist durch eine eventuell unvollständig gegebene Teilmenge von Merkmalswerten charakterisiert und das Ergebnis ist die Auswahl einer oder mehrerer der Lösungen, wobei ggf. zusätzlichen Merkmale zur Verbesserung der Qualität der Problemlösung angefordert werden müssen.

Das erste Inferenzdiagramm zur Klassifikation geht auf [Clancey 85] zurück und enthält drei Inferenzschritte: die *Datenabstraktion* von beobachteten Merkmalen zu diagnostisch höherwertige Merkmalsabstraktionen, die (heuristische) *Abbildung* der Merkmalsabstraktionen in Lösungsklassen und die *Verfeinerung* der Lösungsklassen zu Lösungen (Abb. 1). Die Datenabstraktion ermöglicht die Integration scheinbar gegensätzlicher Anforderungen an Klassifikationssysteme: einerseits sollen die Merkmale leicht erfat werden, d.h. die Systemfragen sollen ohne besonderes Vorwissen des Benutzers oder der Megeräte beantwortet werden können, andererseits sollen sie möglichst aussagekräftig im Hinblick auf die Bewertung der Lösungen sein. Während ersteres umgangssprachliche Begriffe oder Zahlen (bei Mewerten) erfordert, wird letzteres durch fachsprachliche Begriffe erreicht, z.B. die Bewertung von Mewerten (wie die Herzfrequenz) in Kategorien wie normal, zu hoch (Tachykardie), zu niedrig (Bradykardie) usw., in die auch Kontextwissen eingeht (z.B. ob der Patient ein Sportler ist). Die Art des Match der Merkmalsabstraktionen auf Lösungsklassen hängt von der Art des benutzten Wissens ab (s. Kap. 3). Die Verfeinerung der Lösungsklasse zu einer Lösung erfordert meistens weitere Merkmale oder Merkmalsabstraktionen, was das Hufeisendiagramm nicht anzeigt.

Daher kann man diesen Inferenzschritt auch als einen Speziallfall des Match betrachten, die als Eingabe dann zusätzlich Lösungsklassen enthält.

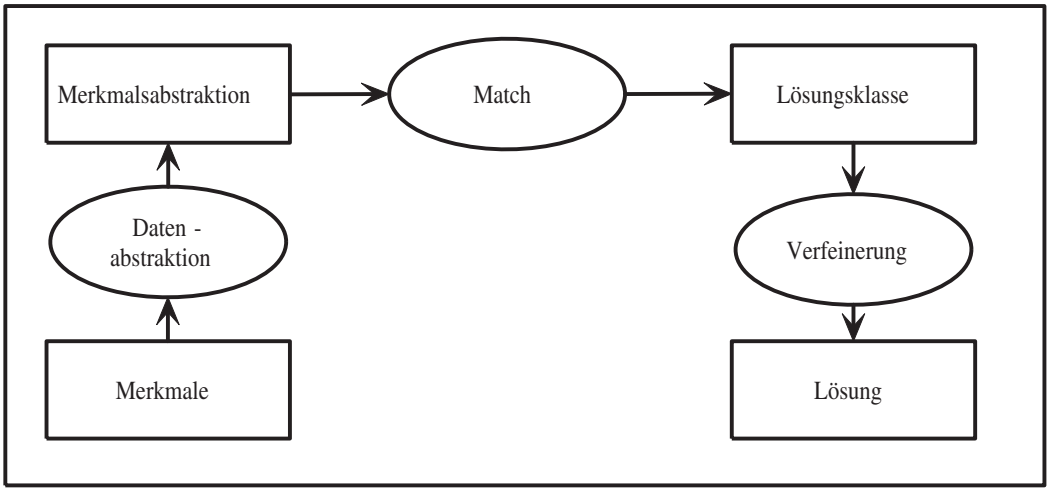


Abb. 1: Hufeisenförmiges Inferenzdiagramm zur Klassifikation nach [Clancey 85]

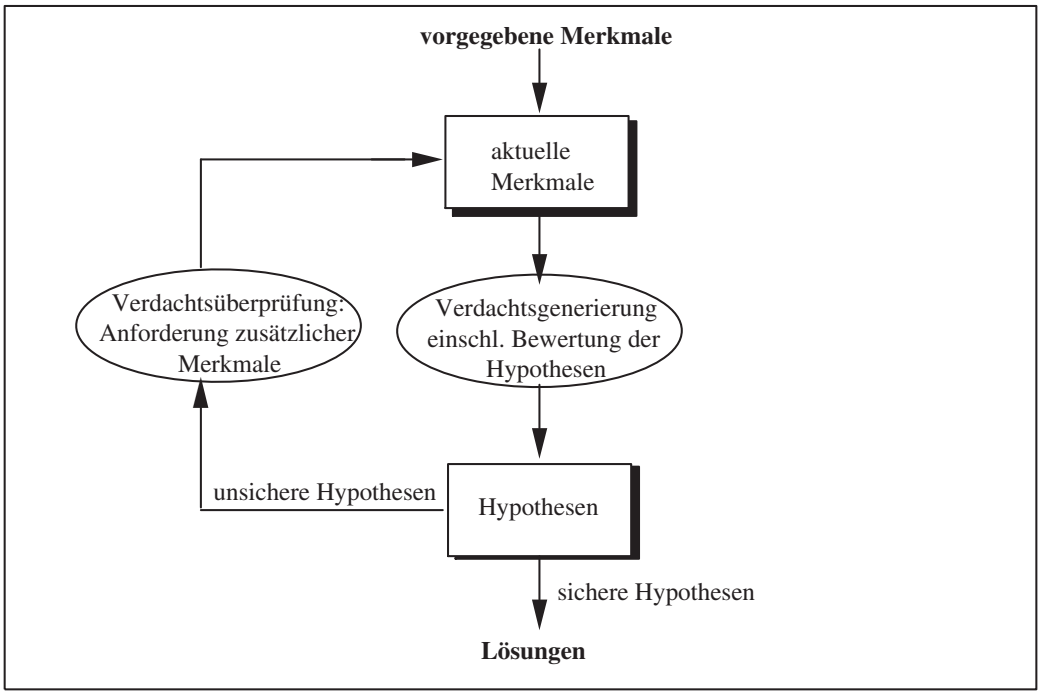


Abb. 2: Inferenzdiagramm der Hypothesize–and–Test–Strategie

Das Hufeisendiagramm aus Abb. 1 berücksichtigt nicht die Anforderung zusätzlicher Merkmale. Die wichtigste Methode dazu ist die hypothetisch–deduktive Vorgehensweise, bei der

aufgrund der Eingangsdaten Verdachtsdiagnosen generiert werden, zu deren Bestätigung oder Ausschlu gezielt weitere Merkmale angefordert werden. Abb. 2 zeigt das entsprechende Inferenzdiagramm mit zwei Inferenzschritten, der Verdachtsgenerierung von Hypothesen aufgrund der aktuellen Merkmale und der Verdachtsüberprüfung, die aufgrund der Hypothesen neue Merkmale anfordert.

Ein weiterer wichtiger Inferenzschritt für die Klassifikation ist die Datenerfassung selbst, die auer einer Benutzungsschnittstelle auch eine Plausibilitätskontrolle der Eingabedaten umfassen sollte (z.B. ist eine Herzfrequenz von 500 beim Menschen im allgemeinen unmöglich). Schlielich lohnt sich häufig eine Unterscheidung zwischen sicherer (kategorischer) und unsicherer (probabilistischer) Herleitung von Lösungen, wobei ersteres zwar untypisch für die meisten Anwendungsbereiche ist, aber immer dem letzeren vorzuziehen ist, auch wenn sie nur Teillösungen liefert. Zusammen ergeben sich daraus folgende fünf Basis–Inferenzschritte für die Klassifikation, die durch eine einfache Schleife zu einer Problemlösungsmethode erweitert ist:

Solange keine Lösung mit ausreichender Sicherheit hergeleitet wurde, tue:

1. **Auswahl anzufordernder Daten**
2. **Datenerfassung (Benutzungsschnittstelle, Plausibilitätskontrolle)**
3. **Datenabstraktion**
4. **Sichere Lösungsbewertung**
5. **Unsichere Lösungsbewertung**

Für den ersten Schritt, der Auswahl anzuforderner Daten gibt es auer der hypothetisch–deduktiven Vorgehensweise auf der Basis der Verdachtsdiagnosen noch weitere Alternativen, z.B. durch ein standardisiertes Vorgehen, das bei Herleiten bestimmter Zustände immer bestimmte Merkmale erfragt, durch Benutzersteuerung oder durch Mischformen.

3 *Wissensarten für diagnostische Problemlösungsmethoden*

Am gröten ist jedoch die Variationsbreite für den letzten Schritt, die unsichere Lösungsbewertung. Im folgenden geben wir eine bersicht über mögliche Alternativen (vgl. Abb. 3), die auf unterschiedlichen Wissensarten basieren (ausführliche Darstellung in [Puppe et al. 96, Puppe 98, Dressler & Puppe 99]]). Dabei ist zu beachten, da man auch ganz ohne unsichere Lösungsbewertung auskommen kann, z.B. bei Entscheidungstabellen und Entscheidungsbäumen.

- Heuristische Klassifikation mit Wissen der Art "Wenn ⟨Merkmalskonstellation⟩ dann ⟨Lösung⟩ mit ⟨Evidenzwert⟩", der von Experten geschätzt wird. Lösungen werden

entsprechend ihrer akkumulierten Evidenz bewertet.

Kausale Klassifikation mit verschiedenen Arten von Modellen:

- berdeckende Klassifikation [Reggia et al. 83] mit Wissen der Art "Wenn ⟨Lösung⟩ dann ⟨Merkmal⟩ mit ⟨Typikalität⟩", die von Experten geschätzt wird. Lösungen werden danach bewertet, wie gut sie die beobachteten Merkmale überdecken (erklären), wobei letztere nach ihrer Bedeutung gewichtet sein können.
- Funktionale Klassifikation [Abu-Hanna et al. 91] auf der Basis eines Modells der normalen Systemfunktion, das typischerweise aus aktiven Komponenten und passiven Materialien aufgebaut ist. Wenn eine Inkonsistenz zwischen den Beobachtungen und dem Systemmodell festgestellt wird, wird eine minimale Menge von Modelländerungen gesucht (z.B. indem die Verhaltensbeschreibung einer Komponenten so geändert wird, da sie defektes Verhalten modelliert), die die Inkonsistenzen aufhebt. Die Modelländerungen entsprechen den Lösungen.
- Verhaltensbasierte Klassifikation, die ähnlich zu funktionaler Klassifikation ist, aber mit einem Systemmodell auf der Basis von physikalischer Prinzipien beruht, wobei die intendierte Funktion nur eines von vielen Verhaltensweisen der Struktur ist.

- Fälle–orientiertes Schlieen, bei dem Fälle die primäre Wissensquelle darstellen. Eine systematische Evaluation von ca. 20 verschiedenen Algorithmen aus den im folgenden genannten Bereichen findet sich z.B. in [Michie et al. 94]):
  - Statistische Klassifikation mit Wissen über die Apriori Wahrscheinlichkeit von Lösungen P(L) und bedingten Wahrscheinlichkeiten P(M/L) von Merkmalen unter der Annahme von Lösungen. Die Wahrscheinlichkeiten werden aus einer repräsentativen Fallsammlung berechnet. Auf deren Basis werden die Lösungswahrscheinlichkeiten P(L/M1 ... Mn) nach dem Theorem von Bayes oder nach der Theorie Bayes'scher Netze [Heckerman 91] berechnet.
  - Neuronale Netze, die eine Fallsammlung zur Adaption ihrer Gewichte verwenden. Dabei werden verschiedene Lernregeln und Netztopologien eingesetzt.
  - Fallbasiertes Schlieen, bei dem eine Fallsammlung direkt mit Wissen über ein hnlichkeitsma verwendet wird. Zu einem neuen Fall werden die ähnlichsten Fälle aus der Fallsammlung gesucht und ihre Lösungen für den neuen Fall übernommen oder adaptiert.
  - Data–Mining und Lernverfahren, mit denen aus groen Fallsammlungen interessante Zusammenhänge gesucht werden, die dann mit einem der anderen Verfahren wie Entscheidungsbäume oder heuristische Klassifikation abgearbeitet werden; z.B. mit induktiven Lernalgorithmen wie ID3 usw.

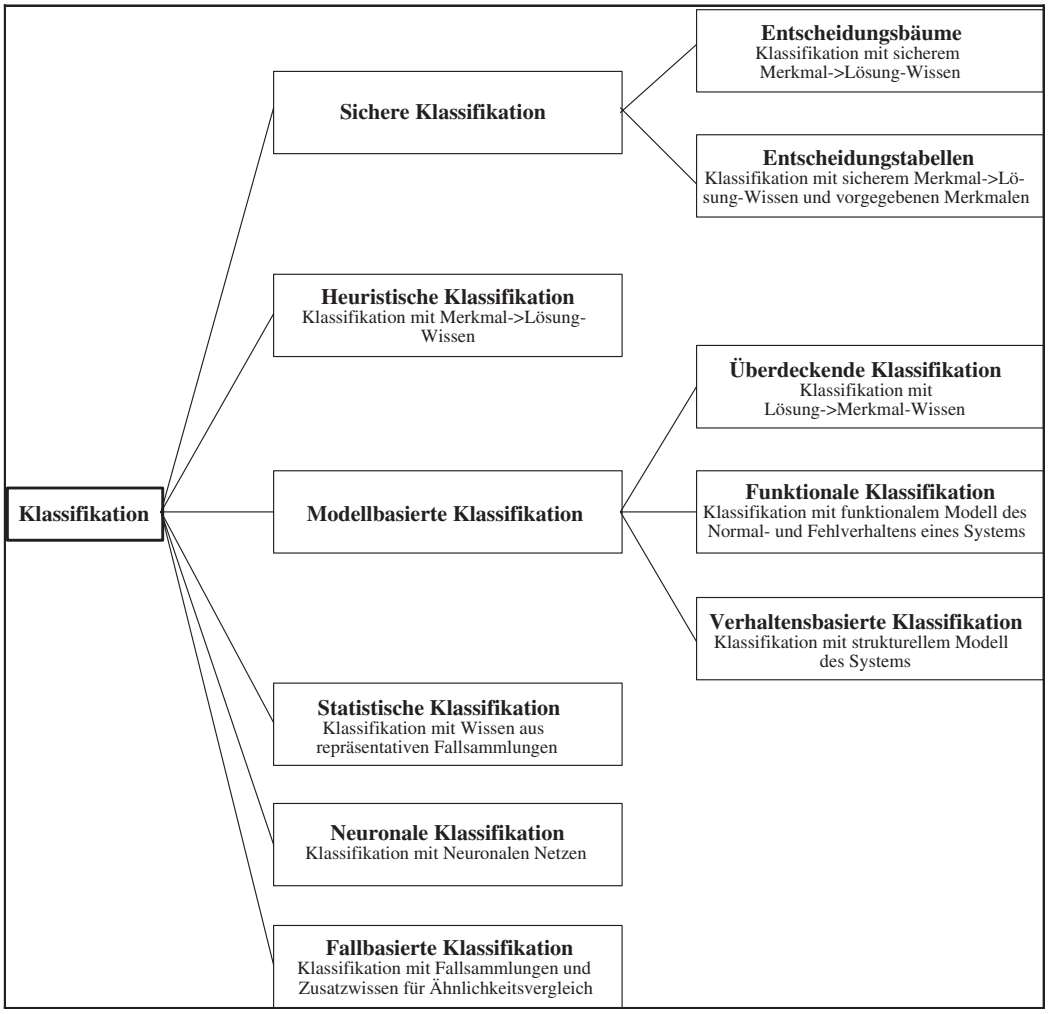


Abb. 3: bersicht über Wissensarten für die Problemklasse Klassifikation (Diagnostik).

Die Alternativen sind nach den drei wichtigsten Wissenstypen, Fälle, Erfahrungswissen und Modelle, gegliedert. *Fälle* verbinden Beobachtungen vergangener Probleme mit deren Lösungen und können auf verschiedene Art zur Lösung neuer Probleme verallgemeinert werden. Bei der Kodierung von *Erfahrungswissen* orientiert man sich an den mentalen Modellen von Fachexperten. Bei viele technischen Anwendungen verfügt man über gute *Modelle* des zu diagnostizierenden Systems, die eine systematische Fehlersuche ermöglichen. Meist sind nicht alle Wissensarten verfügbar, z.B. fehlen Fälle und Erfahrungswissen bei der Diagnose neuer technischer Systeme, während bei komplexen biologischen Systemen das kausale Verständnis begrenzt ist. Wenn Fallsammlungen existieren, hängt deren Nützlichkeit von ihrer Quantität, Qualität und dem Grad der Formalisierung ab. Erfahrungswissen kann unzugänglich sein, wenn Fachexperten nicht kooperieren wollen oder aus Zeitmangel nicht können. Wenn in einer Anwendung mehrere Wissenstypen parallel vorliegen, dann ist es oft vorteilhaft, sie zu kombinieren, z.B. indem man für verschiedene Teilbereiche verschiedene Wissenstypen benutzt, oder indem man das Wissen redundant kodiert und die mit unterschiedlicher Problemlösungsmethoden hergeleiteten Lösungen vergleicht (vgl. [Puppe 98].

Da der Aufwand für die Entwicklung wissensbasierter Systeme entsprechend ihrer Komplexität meist beachtlich ist, versucht man diesen durch Wiederverwendung zu verringern. Während die Wiederverwendung von Problemlösungsmethoden insbesondere durch problemspezifische Werkzeuge (Shells) weit fortgeschritten ist (z.B. für die Klassifikation Shells wie D3 (vgl. [Puppe et al. 96]) oder für die Konfiguration KONWERK (vgl. [Guenter & Kühn 99]), ist die Wiederverwendung von Wissen schwieriger, da die gewählte Wissensrepräsentation mit fremden Werkzeugen meist inkompatibel ist. Daher ist es auch interessant, die Problemlösungsmethoden flexibler zu gestalten, um Brücken zwischen verschiedenen Terminologien zu ermöglichen. Dazu werden in Fensel [2000] sogenannte Adapter vorgeschlagen, die eventuell vorhandenes Wissen, da aus Sicht der Aufgaben beschrieben wurde (Task–Ontologien) in die Repräsentation der Problemlösungsmethoden (Methoden–Ontologien) zu überführen.

4 *Literatur*

Abu-Hanna, A., Benjamins, R., Jansweijer, W.: Device Understanding and Modeling for Diagnosis, IEEE–Expert 6, 33–40 (1991)

Biundo, S., Günter, A., Hertzberg, J., Schneeberger, J. und Tank, W.: Planen und Konfigurieren, in Görz , G. (Hrsg.): Einführung in die Künstliche Intelligenz, Addison-Wesley, 2. Auflage, Kap. 7.2, 1995.

Breuker, J. and van de Velde, W. (Hrsg.): CommonKADS Library for Expertise Modelling: Reusable Problem Solving Components, IOS Press, 1994.

Clancey, W.: Heuristic Classification. Artificial Intelligence 20, 215–251, 1985.

Dressler, O., Puppe, F.: Knowledge–Based Diagnosis – Survey and Future Prospects, in Proc. XPS-99, 5th Biannual Conference on Knowledge–Based Systems: Survey and Future Directions, Springer, LNAI 1570, 24–46, 1999.

Fensel, D.: Understanding, Developing and Reusing Problem–Solving Methods, to appear as Lecture Notes of Artificial Intelligence (LNAI), Springer, 2000.

Heckermann, D.: Probabilistic Similarity Networks, Cambridge: MIT Press 1991.

Günter, A. und Kühn, C.: Knowledge–Based Configuration – Survey and Future Directions, in: Proc. XPS-99, 5th Biannual Conference on Knowledge–Based Systems: Survey and Future Directions, Springer, LNAI 1570, 47–66, 1999.

Michie, D. et al. (eds.): Machine Learning, Neural and Statistical Classification. Ellis Horwood 1994.

Puppe, F., Gappa, U., Poeck, K., Bamberger, S.: Wissensbasierte Diagnose– und Informationssysteme. Berlin: Springer 1996.

Puppe, F.: Problemlösungsmethoden in Expertensystemen, Springer, 1990.

Puppe, F.: Knowledge Reuse among Diagnostic Problem Solving Methods in the Shell–Kit D3. Intl. Journal of Human–Computer–Studies 49, 627–649, 1998.

Reggia, J., Nau, D., Wang, P.: Diagnostic Expert Systems Based on a Set Covering Model, Intl. Journal of Man–Machine–Studies 19, 437–460, 1983.

Wielinga, B. and Breuker, J.: Models of Expertise, European Conference on Artificial Intelligence (ECAI-86), 306–318, 1986.

Rekonfiguration komplexer industrieller Produkte mittels constraint-logischer Programmierung

U. John
GMD — Forschungszentrum Informationstechnik GmbH
GMD FIRST, Kekuléstr. 7, D-12489 Berlin

e-mail: john@first.gmd.de

Zusammenfassung

Für die Rekonfiguration von industriellen komplexen Produkten gibt es zwei Hauptmotive. Einerseits bedingt der Ausfall von Produktkomponenten, die auf dem Markt nicht mehr verfügbar sind, nichttriviale Rekonfigurationen. Andererseits ist es oft wünschenswert, die Funktionalität von Produkten zu ändern, was ebenfalls einen "Umbau" des jeweiligen Produktes erfordert. Naheliegende Zielstellung für die Rekonfigurationsprozesse ist es, die erforderlichen Änderungen mit minimalen Kosten durchzuführen.
Aufbauend auf unserem constraint-basierten Konfigurationsmodell ConBaCon für industrielle Produkte wird in diesem Paper ein praktikabler Rekonfigurationsansatz vorgestellt. Dieser basiert auf einem Constraint-Hierarchie-Transformator und entsprechenden Wartungsstrategien für Produktdatenmodelle.

1 Einführung

Konfigurationssoftware für Produkte ist nach wie vor ein Gegenstand intensiver Forschungs- und Entwicklungstätigkeiten. Gründe hierfür sind Defizite der bestehenden Konfigurationsprozesse (vgl. z.B. [19, 1, 25]) und die sich anbietende Verlagerung bzw. weitgehende Automatisierung von Marketing und Verkaufsaktivitäten, die allein in US-Unternehmen durchschnittlich 40% der Unternehmensbudgets beanspruchen ([18]). Für den Konfigurationssoftware-Markt wird z.Zt. ein jährliches Wachstum von ca. 35% prognostiziert, wobei für das Jahr 2000 ein Umsatz von 2,5 Mrd. US$ erwartet wird ([18]).

Im Laufe der letzten zwei Jahrzehnte wurden, beginnend mit der Entwicklung des bekannten regel-basierten Konfigurationssystems XCON für die Konfiguration von DEC-Computern im Jahre 1981, verschiedene Ansätze für die wissensbasierte Konfiguration von industriellen Produkten entwickelt und untersucht. Diese umfassen zahlreiche regel-basierte, fall-basierte und gegenwärtig mehr und mehr constraint-basierte Ansätze. Überblicke über verschiedene Ansätze und Systeme bieten zum Beispiel [25, 24, 22].

Betrachtet man kommerziell verfügbare Systeme (CeBIT'98) und publizierte Forschungsansätze, so kann man folgende Defizite analysieren: Die Problemspezifikation ist nichtdeklarativ und meist in Form eines Entscheidungsbaumes gegeben, was eine schwere Wartbarkeit bedingt. Oft ist die Reihenfolge von Experteninteraktionen während des Konfigurationsprozesses fest vorgegeben. Dadurch bedingt, ist ein flexibler Konfigurationsprozeß, wie

er vom System *ConBaCon* unterstützt wird, nicht durchführbar. Die Simulation verschiedener Auswirkungen von alternativen Interaktionsentscheidungen ist selten möglich und die *Unterstützung von guten Rekonfigurationen*, wie sie durch die Industrie benötigt werden, ist *ungenügend bzw. nicht existent*. Präferenzen und weiche Constraints können nicht verarbeitet werden. Das Finden optimaler bzw. optimumnaher Konfigurationen ist unmöglich, etc.

Betrachtet man laufende Entwicklungen in Industrie und Forschung, so scheint sich die Erkenntnis zu formieren, daß sich zur Entwicklung hoch-qualitativer Konfigurationssysteme besonders die Constraint-Programmierung eignet (z.B. [10, 2, 4, 5, 21, 16, 3, 27, 20, 18]). Dies erkennt man auch an der Tatsache, daß die leistungsstärkeren kommerziellen Systeme sich als "constraint-basiert" vermarkten, wobei diese Titulierung in den meisten Fällen irreführend ist, da sie keinen Constraint-Löser verwenden (es findet keine suchraumeinschränkende Propagation statt), sondern eine constraint-basierte Problemspezifikation durch reine Constraintprüfung verarbeiten (vgl. auch [1]). Publikationen bezüglich der "echt" constraint-basierten Konfigurationssysteme und Forschungsprototypen besitzen oft sehr allgemeinen Charakter oder dokumentieren Einschränkungen hinsichtlich der Qualität der Suchraumeinschränkung oder der behandelbaren Problemklasse. Insbesondere parametrierte Komponenten, wie sie zum Beispiel für industrielle Kontrollsysteme benötigt werden, werden selten von konfigurationsrelevanten Publikationen (vgl. auch [25]) behandelt.

Das System ConBaCon, das auf der constraint-logischen Programmiersprache CHIP basiert, überwindet o.g. Defizite zu großen Teilen (vgl. auch [12, 13]).

Der nächste Abschnitt gibt einen kurzen Überblick über die Spezifikationssprache ConBaConL zur Spezifikation technischer Konfigurationsprobleme, wobei insbesondere die Erweiterungen zur Dokumentation von Produkttaxonomie-Modifikationen für die Thematik des Papers relevant sind.

Abschnitt 3 beschreibt unser constraint-basiertes Konfigurationsmodell sowie dessen Realisierung ConBaCon, das die effiziente Konfiguration von industriellen Produkten ermöglicht.

Das auf dem präsentierten Problemlösungsmodell basierende Rekonfigurationsmodell wird in Abschnitt 4 behandelt. Die Arbeit endet mit einem Ausblick auf zukünftige Erweiterungsmöglichkeiten.

2 Problemspezifikation mit ConBaConL

Ausgehend von praktischen Konfigurationsproblemen aus dem Gebiet der Prozeßleittechnik wurde nach Entwicklung eines formalen Konfigurationsmodells die weitgehend deklarative Spezifikationssprache ConBaConL entwickelt, mit der technische Konfigurationsprobleme spezifizierbar sind. ConBaConL-Spezifikationen bestehen neben dem Bezeichner des zu konfigurierenden Produktes aus drei Blöcken: der Objekthierarchie, den kontextunabhängigen Constraints sowie den kontextabhängigen Constraints. Jedes technische Objekt, was hinsichtlich der Konfiguration relevant ist, wird bzgl. seiner Struktur spezifiziert. Ein Objekt kann entweder aus verschiedenen Komponenten im Sinne der *consist_of*-Beziehung bestehen (*Aggregat*), wobei einzelne Komponenten optional sein können, oder aber verschiedene Spezialisierungen im Sinne der *is_a*-Relation besitzen (*Objektklasse*). Darüber hinaus werden die Objektattribute spezifiziert, wobei eine Aufzählung von möglichen Attributwerten erfolgen kann, sofern diese explizit bekannt sind.

Eine korrekte kontextunabhängige Repräsentation des Konfigurationsproblems ergibt sich aus dem vorliegenden semantischen Netz durch Spezifikation der Beziehungen (*Constraints*) zwischen verschiedenen Attributwertemengen, sowie der Abhängigkeiten bzgl. der Existenz bzw. Nichtexistenz von technischen Modulen (Objekten) in der Problemlösung. Gibt es bestimmte Randbedingungen aus dem Kontext (etwa kundenspezifische Wünsche), so werden diese als problemspezifische Constraints spezifiziert. Die Unterscheidung von problemspezifischen und problem- bzw. kontextunabhängigen Constraints ist sinnvoll, da für die technische Korrektheit des zu konfigurierenden Objektes lediglich die kontextunabhängigen Constraints zu erfüllen sind.

Die in ConBaConL spezifizierbaren Constraints lassen sich in *Simple Constraints*, *Compositionale Constraints* und in *Conditionale Constraints* unterscheiden, die meisten werden im folgenden informal eingeführt.

2.1 Simple Constraints

Attributwerte- und Existenzconstraints
[o, Attr, VS] (*not([o, Attr, VS])*)
≡ das Attribut *Attr* des Objektes *o* muß (darf nicht) einen Wert aus VS besitzen,
exist(Objectlist) (*noexist(Objectlist)*)
≡ alle Objekte aus *Objectlist* müssen (dürfen nicht) in der Lösung enthalten sein.
Relationale Constraints zwischen Attributwertemengen und Tabellen-Constraints
eq(T1,T2), neq(T1,T2), lt(T1,T2), let(T1,T2), gt(T1,T2), get(T1,T2).
Darüber hinaus ist es möglich, Gleichungen über Attributen zu spezifizieren.

In gebräuchlichen industriellen Richtlinien zur Konfiguration technischer Systeme werden Zusammenhänge oft in tabellarischer Form dargestellt. Um manuelle (fehleranfällige) Transformationen der Tabellen in andere Arten von ConBaConL-Constraints zu vermeiden, wurde ein Tabellen-Constraint (vgl. [13]) eingeführt.

2.2 Compositionale Constraints

Compositionale Constraints sind neben den Simplen Constraints Kompositionen von compositionalen Constraints: *and([*$Cons_1, \ldots, Cons_n$*])*, *or([*$Cons_1, \ldots, Cons_n$*])*, *xor([*$Cons_1, \ldots,$ $Cons_n$*])*,
at_least([$Cons_1, \ldots, Cons_n$*],N)/ at_most([*$Cons_1, \ldots, Cons_n$*],N)/ exact([*$Cons_1, \ldots, Cons_n$*],N)*
≡ mindestens/höchstens/genau N der aufgeführten Constraints müssen bzw. dürfen erfüllt sein[1].

2.3 Conditionale Constraints

[if($Comp\_Cons_1$*), then(*$Comp\_Cons_2$*)]* (*[iff(*$Comp\_Cons_1$*), then(*$Comp\_Cons_2$*)]*)
(Genau dann) Wenn das compositionale Constraint $Comp_Cons_1$erfüllt ist, muß das Compositionale Constraint $Comp\_Cons_2$ erfüllt sein.

[1] Bis jetzt wurde die Verarbeitung von or-, xor-, at_least-, at_most-, exact-Constraints über Existenz- bzw. Nichtexistenz-Constraints von Objekten realisiert.

2.4 Modifikationen

In der industriellen Praxis unterliegen die Spezifikationen von Produkttaxonomien bzw. Komponentenkataloge einer hohen Modifikationsrate. Z.B. mußten 40% der 30000 in R1/XCON verwendeten Komponententypen von DEC-Computern jährlich aktualisiert werden (cf. [7]). Es gibt verschiedene Gründe für die Notwendigkeit, Produkttaxonomien und die damit verbundenen Problemspezifikationen zu ändern. Hauptgrund ist die Tatsache, daß neue technische Module verfügbar werden, wohingegen andere vom Markt verschwinden. Ein spezieller Fall davon ist das sogenannte Versioning. Ein weiterer Grund ist das Ändern von kontextunabhängigen Constraints, etwa auf Grund von Gesetzesänderungen. Will man spätere Rekonfigurationen ermöglichen, so dürfen veraltete Informationen in der Spezifikation nicht einfach gelöscht werden. Statt dessen sollten veraltete Module und Constraints in der Spezifikation bzw. Produkttaxonomie mit dem Schlüsselwort *obsolete* markiert werden.

Neue Module sowie neue Constraints können problemlos zur Spezifikation hinzugefügt werden (Abbildung 1). Zwei Fälle lassen sich hinsichtlich der Integration eines neuen Modules *new_o* als Alternative zu einem bereits existierenden spezifizierten Modul o_x unterscheiden. Im ersten Fall (I.) ist o_x eine Spezialisierung des Objektes o. Im zweiten Fall (II.) ist o_x eine Komponente von o. Ein Hilfsobjekt h muß an die Position von o_x eingefügt werden und erhält die Spezialisierungen *new_o* und o_x.

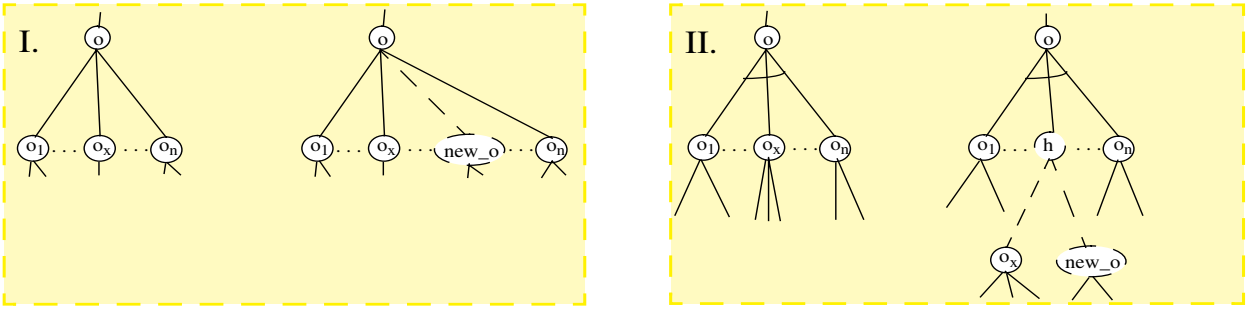


Abbildung 1: *Integration von neuen Modulen*

Andere wichtige Elemente von ConBaConL sind Präferenzen und Optimierungsziele, auf die in dieser Arbeit nicht genauer eingegangen wird.

Eine beispielhafte ConBaConL-Spezifikation (Grundstromrichterkonfiguration für große elektrische Antriebe), die anhand von industriellen Daten erstellt wurde, findet man, gemeinsam mit der Problemlösung durch ConBaCon, in [13].

3 Problemlösungsmodell

Ziel ist es, ausgehend von einer Problemspezifikation ein Modell bzw. eine Modellinstanz zu erhalten, das eine effiziente Problemlösung ermöglicht und bei Bedarf einen hohen Grad von Interaktionen mit dem Experten unterstützt. Als Grundlage für das vorgestellte Lösungsmodell dient ein CLP-Modell über endlichen Domänen. Somit kann das Modell auch als *globales Constraint für strukturelle Konfigurationsprobleme* gesehen werden. Jedes spezifizier-

te Objekt[2] wird in ein "Modul-Objekt" des Problemlösungsmodells transformiert. Zusätzlich wird jedes Attribut eines Objektes zu einem "Attributobjekt" des Problemlösungsmodells transformiert. Das heißt, ein mit n Attributen spezifiziertes Objekt wird zunächst durch $n+1$ Objekte im Problemlösungsmodell repräsentiert (Abbildung 2). Hierin verbirgt sich ein wesentlicher Unterschied zu publizierten Repräsentationen. Parametrierte Module werden in konfigurationsrelevanter Literatur selten berücksichtigt (vgl. [25]). Zum Beispiel wird in [5] unkommentiert davon ausgegangen, daß es in technischen Domänen nur parameterlose Module gibt. Das hieße für Problembereiche, in denen Objektparameter ausschlaggebend sind, etwa in der Prozeßkontrollindustrie, daß für jede Parameterkombination eines technischen Objektes ein gesondertes Objekt bzw. eine gesonderte Constraint-Variable im Problemlösungsmodell existieren muß, was eine kombinatorische Explosion zur Folge hätte. Die Objekte

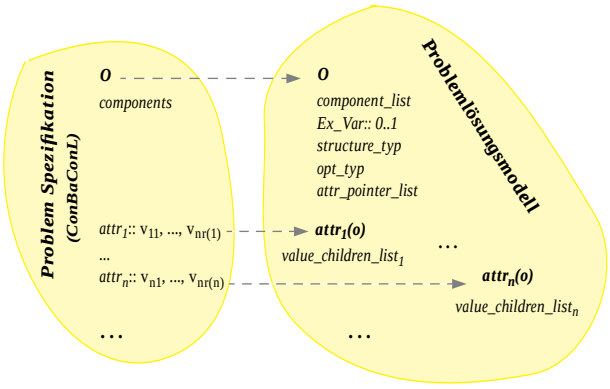


Abbildung 2: *Objekttransformation*

des Problemlösungsraumes werden mit entsprechenden Attributen versehen. Das Attribut *komponentenliste* enthält die Bezeichner der Komponenten bzw. der Spezialisierungen des Objektes o. Der Strukturtyp (Aggregat oder Objektklasse) wird im Attribut *struktur_typ* verzeichnet. Ex_Var gibt Auskunft über die Existenz des Objektes in der Problemlösung. Ist Ex_Var null, so existiert o nicht in der Lösung, hat Ex_Var hingegen den Wert eins, so ist o Bestandteil der Lösung[3] . Das Attribut *opt_typ* gibt an, ob o ein optionales Element der Lösung ist, oder nicht. Die Bezüge zu den zugehörigen Attributobjekten werden durch *attr_pointerliste* hergestellt. Jedes Attributobjekt speichert die möglichen Werte für das Attribut in einer *wert_kind_liste* sowie in der Domäne einer zugehörigen Constraintvariable. Darüber hinaus sind in *wert_kind_liste* die Bezeichner der mit den einzelnen Werten verbundenen "Kinderknoten" enthalten, sofern der jeweilige Knoten o verschiedene Spezialisierungen besitzt, wobei sich die Attributwerte dann aus der Vereinigung der Attributwerte seiner Spezialisierungen ergeben.

Neben den Objekten des Problemlösungsmodells sind modellinhärente Constraints notwendig, die die intuitiv verständlichen Zusammenhänge zwischen den Objekten des Problemlösungsmodells durchsetzen und damit die Korrektheit von Lösungen bzgl. der Problemspezifikation sowie die Vollständigkeit des Lösungsraumes garantieren. Diese Constraints bezeichnen wir als *Konsistenzsichernde Constraints* (CE-Constraints).

[2] Dieses beschreibt im ingenieur-technischen Sinne einen technischen Modul.

[3] Die Beschränkung auf die Werte null und eins ist eine Vereinfachung des realisierten Modells. In diesem gibt es mehr Werte. Dies erlaubt die Modellierung verschiedener technischer Bezeichner für ein technisches Objekt, die zum Beispiel abhängig von den konkreten Parameterwerten gelten.

3.1 Konsistenzsichernde Constraints

CE-Constraints werden als logische Zusammenhänge zwischen den Werten von Ex_Var-Attributen sowie zwischen Attributwertemengen verschiedener Attributobjekte realisiert. Die wesentlichsten CE-Constraints sind in Abbildung 3 schematisiert. Kann ein Objekt nicht Be-

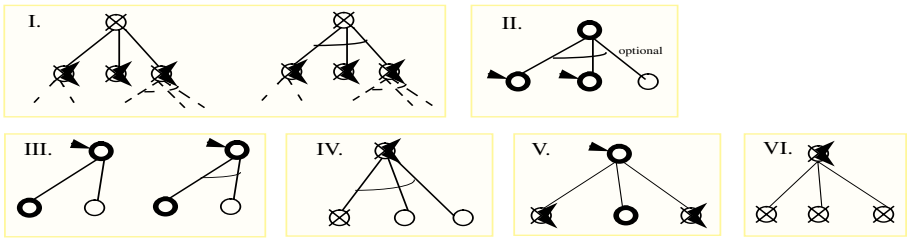


Abbildung 3: *Konsistenzsichernde Constraints*

standteil der Problemlösung sein, so können auch die Komponenten bzw. die Spezialisierungen des Objektes nicht in einer Lösung enthalten sein (I). Wenn klar wird, daß ein Objekt in der Lösung enthalten ist ($Ex\_Var = 1$), so muß durch CE-Constraints gesichert werden, daß alle nichtoptionalen Komponenten des Objektes in der Lösung ebenfalls vorkommen (II). Die Existenz eines Objektes zieht in jedem Falle zwingend die Existenz des "Vaterobjektes" nach sich (III). Kommt eine nichtoptionale Komponente eines Objektes o in keiner Lösung vor, so kommt auch o in keiner Lösung vor (IV). Ist eine Spezialisierung eines Objektes Bestandteil der Lösung, so kommt keine andere Spezialisierung des Objektes in der Lösung vor (V). Steht für alle Spezialisierungen eines Objektes fest, daß sie nicht Bestandteil der Lösung sind, so kann auch das Objekt selbst nicht Bestandteil der Lösung sein (VI).

Die Attributwertemengen werden durch eine gesonderte Klasse von CE-Constraints konsistent gehalten. Wird aus der Attributwertemenge eines Objektes o, welches in der Problemspezifikation Spezialisierung eines "Vaterobjektes" v ist, ein Wert gelöscht, so wird in der entsprechenden Attributwertemenge von v dieser Wert ebenfalls gelöscht, sofern v keine andere "noch gültige" Spezialisierung besitzt, die den gelöschten Wert in der entsprechenden Attributwertemenge verzeichnet[4] . Ist o hingegen selbst "Vaterobjekt" mit verschiedenen Spezialisierungen, so wird der gelöschte Wert in der entsprechenden Attributwertemenge aller Spezialisierungen von o gelöscht. Wird eine Attributwertemenge eines Objektes leer, so wird über ein spezielles CE-Constraint das Folgern der Nichtexistenz des Objektes gesichert.

Die strukturbegründeten Zusammenhänge zwischen spezifizierten Objekten bzgl. der Existenz bzw. Nichtexistenz und bzgl. der Attributwertemengen werden durch die vorgestellten CE-Constraints gesichert. Darüber hinaus müssen die in der Problemspezifikation formulierten Constraints in Constraints des Problemlösungsmodells überführt werden.

3.2 Spezifizierte Constraints

Attributwerte- und Existenzconstraints (siehe 2.1) laufen auf ein Löschen von Attributwerten im Problemlösungsmodell bzw. auf das Setzen der Modul-Attribute Ex_Var hinaus. *Relationale Constraints* zwischen Attributwertemengen werden im Problemlösungsmodell durch

[4] Um ein kostenintensives Abtesten zu verhindern, wird nach jedem Löschen eines Attributwertes einer Spezialisierung, das *wert_kind_liste*-Attribut des entsprechenden, dem "Vaterobjekt" zugeordneten, Attributobjektes aktualisiert.

Löschen bezüglich der Relation ungültiger Attributwerte umgesetzt. Gibt es im Anschluß daran Wertetupel, die die Relation nicht erfüllen , so werden entsprechende Dämonen generiert, die die Einhaltung der Relationalen Constraints bei Änderungen der betroffenen Attributwertemengen überwachen. Transformierte *Tabellen-Constraints* stellen im Problemlösungsmodell eine Verbindung zwischen den beteiligten Attributwertemengen und den Existenzinformationen (Ex_Var) genannter Objekte dar. Änderungen beteiligter Attributwertemengen bzw. Änderungen des Existenzstatus beteiligter Objekte führen zur Ungültigkeitsmarkierung entsprechender Tabellenzeilen. Sind alle Tabellenzeilen ungültig, so gilt das Constraint als nicht erfüllt. Umgekehrt wird gesichert, daß die beteiligten Attributwertemengen nur Werte enthalten, die in gültigen Tabellenzeilen verzeichnet sind. Die *Compositionalen Constraints* (vgl. 2.2) werden im wesentlichen durch Gleichungen bzw. Ungleichungen über den entsprechenden Objektattributen Ex_Var realisiert. Für jede Nicht-Existenzaussage wird statt Ex_Var der Term "1 - Ex_Var" bezüglich des entsprechenden Objektes in der Gleichung bzw. Ungleichung verwendet. *Conditionale Constraints* (vgl. 2.3) werden zu bedingten Transitionen des Problemlösungsmodells transformiert, die den spezifizierten logischen Zusammenhang innerhalb des Problemlösungsmodells absichern. Um eine möglichst große Problemraumeinschränkung innerhalb des Modells zu erreichen, werden die Kontrapositionen der spezifizierten Conditionalen Constraints ebenfalls zu Bestandteilen des Problemlösungsmodells transformiert.

3.3 Problemlösung

Basierend auf dem skizzierten Problemlösungsmodell kann nun ein flexibler, effizienter Problemlösungsprozeß (Abbildung 4) durchgeführt werden. Nach Generierung der Objekte des

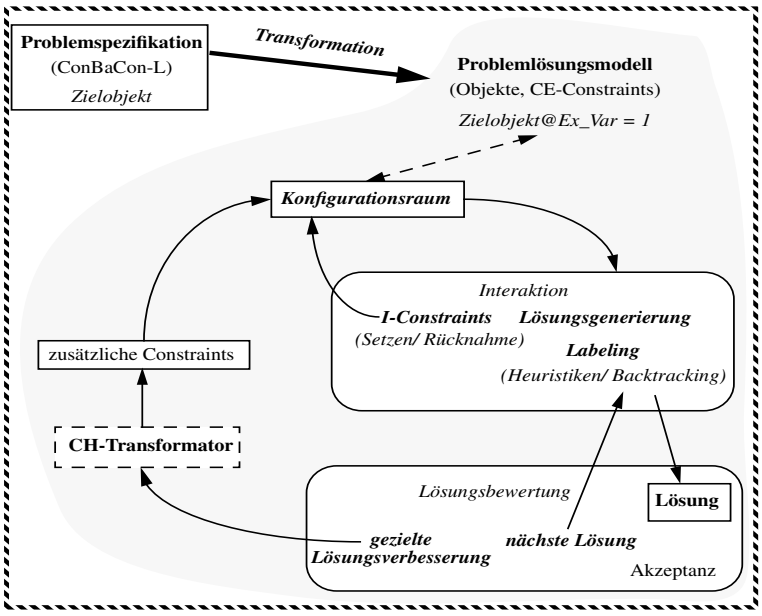


Abbildung 4: *Problemlösungsprozeß*

Problemlösungsmodells und Absetzen der abgeleiteten CE-Constraints sowie der Transformate der spezifizierten Constraints, wird dem Ex_Var-Attribut des Zielobjektes der Wert eins zugewiesen. Auf Grundlage des generierten Modells, insbesondere durch die CE-Constraints,

wird eine weitgehende Suchraumeinschränkung herbeigeführt. Die Menge der jeweils aktiven Modul-Objekte des Problemlösungsmodells[5] bezeichnen wir als *Konfigurationsraum*. Ausgehend vom Konfigurationsraum ist es möglich interaktiv weitere Constraints bezüglich der Existenz bzw. Nichtexistenz von *beliebigen* Objekten des Konfigurationsraumes bzw. bezüglich der Ausprägung derer Attributwertemengen zu fordern. Die durch die modellinhärenten Constraints realisierte Inferenz führt dann zu einem "neuen Konfigurationsraum". Die Möglichkeit, frühere interaktive Constraints wieder zu löschen, verleiht dem Problemlösungsmodell die Eigenschaft der Simulationsunterstützung (vgl. [11]), was Grundlage für einen hochflexiblen Konfigurationsprozeß ist. Werden keine weiteren interaktiven Constraints gefordert, so kann eine Lösung generiert werden. Dabei wird ein Labeling über den Ex_Var-Attributen der (noch) aktiven Objekte des Problemlösungsmodelles angestoßen. Das Labeling kann durch Heuristiken gesteuert werden. So ist es z.B. möglich, bestimmte Präferenzen in Form von Präferenzregeln für den Labelingprozeß zu berücksichtigen[6]. Findet die Lösung keinen Gefallen bzw. besteht sie nicht die Lösungsgüteprüfung, so können via Backtracking weitere Lösungen erzeugt werden. Ist es ausreichend, die Lösung partiell zu verbessern, so kann eine gezielte Lösungsverbesserung mittels der Spezifikation und Verarbeitung einer Constraint-Hierarchie vorgenommen werden[7], d.h. die unbedingt zu erfüllenden Constraints werden als *harte* Constraints spezifiziert, die möglichst beizubehaltenden Lösungsbestandteile als *weiche* Constraints, wobei unterschiedliche Wichtungen spezifiziert werden können. Die so spezifizierte Constraint-Hierarchie wird einem "Fehlerminimierungsprozeß" unterworfen, woraus die Generierung äquivalenter harter Constraints resultiert. Detailliertere Ausführungen zur Realisierung und Anwendung von Constraint-Hierarchien im auf dem Problemlösungsmodell basierenden Prototypen findet man in [23]. Eine exemplarische Konfiguration aus dem Gebiet der Prozeßleittechnik mit ConBaCon ist in [12] dargestellt.

4 Rekonfiguration

Es gibt zwei plausible Gründe für die Rekonfiguration von existierenden Systemen resp. Produkten. Einerseits fallen, über einen längeren Zeitraum betrachtet, Teile bzw. Komponenten aus und müssen ersetzt werden, wobei des öfteren eine Reparatur ohne Beeinflussung von anderen Modulen des Systems nicht möglich ist. Andererseits besteht oft der Wunsch, ein existierendes System mit einer neuen/ modifizierten Funktionalität zu versehen, was Erweiterungen und Ersetzungen notwendig machen kann. Verschiedene zusätzliche Ziele für den Rekonfigurationsprozeß sind denkbar. So könnte es sinnvoll sein, die Anzahl der notwendigen Ersetzungen/ Änderungen minimal zu halten. Diese Zielstellung kann als ein Ersatzziel für das Ziel Kostenminimalität angesehen werden. Ein anderes Ziel könnte sein, so wenig wie möglich Änderungen von existierenden Parameterwerten vorzunehmen. Die Änderung von vielen Parameterwerten eines rekonfigurierten sicherheitsrelevanten Kontrollsystems, z.B., würde teure Feldsimulationen erfordern.

[5] Das sind die Objekte, deren Ex_Var-Attribut (noch) nicht den Wert null besitzt.

[6] Bisher wurde in ConBaCon nur die Verarbeitung von Präferenzen in Form von Constraint-Hierarchien realisiert.

[7] Analog zur gezielten Lösungsverbesserung können Konfigurationen für kostenarme Umbauten bzw. Erweiterungen industrieller Systeme unterstützt werden. Dies wird in [25] als offenes Problem beschrieben.

4.1 Minimale Ersetzung

Wir wollen uns hier zunächst auf das Ziel "minimale Ersetzungen/ Änderungen" beschränken. Eine Erweiterung dessen, die Behandlung ressourcen-zentrierter Module, wird im nächsten Abschnitt präsentiert. Abbildung 5 schematisiert den Rekunfigurationsprozeß für ein System s. Die exakte Spezifikation des existierenden Systems s muß mit dem derzeitigen Produkt-

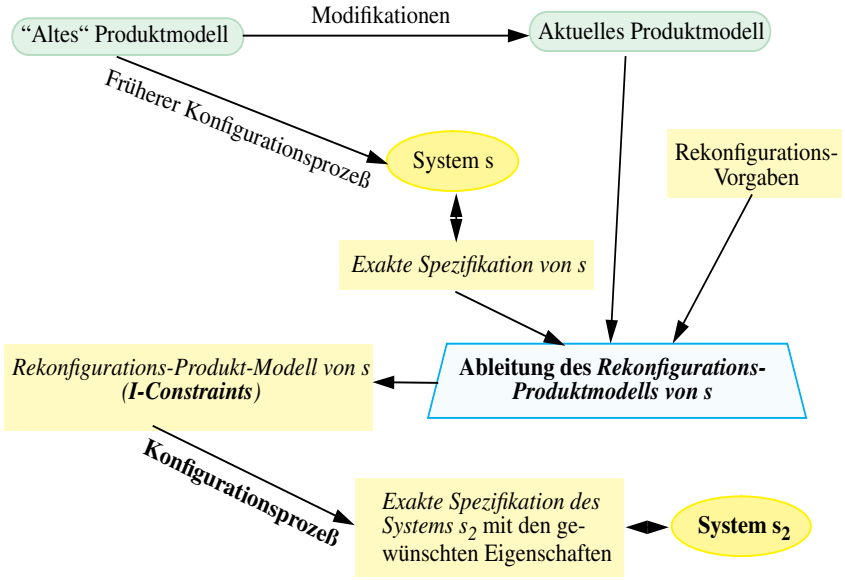


Abbildung 5: *Rekonfiguration eines technischen Systems*

modell von s folgenderweise gemischt werden. Ausgehend von dem derzeitigen Produktmodell müssen alle technischen Module/ Objekte des Systems s, *die nicht kaputt sind* und *deren Austausch in Rekonfigurationsforderungen nicht explizit gefordert ist* als *gültig* markiert werden, falls sie derzeit als *obsolete* (vgl. 2.4) markiert sind. Sie sind verfügbar, da sie bereits in s integriert sind. Spezifizierte kontextunabhängige Constraints werden in ähnlicher Weise behandelt. Für jedes technische Modul von s, das nicht kaputt ist und dessen Austausch bzw. Entfernen nicht explizit in den Rekonfigurations-Vorgaben gefordert wird, wird ein *weiches Existenz-Constraint* generiert. Für jedes optionale Objekt des resultierenden Produktmodells, das nicht Teil von s ist und dessen Existenz nicht explizit in den Rekonfigurations-Vorgaben verlangt wird, wird ein *weiches Nicht-Existenzconstraint* abgeleitet. Das resultierende *Rekonfigurations-Produkt-Modell*, das ebenfalls die Rekonfigurations-Vorgaben (ConBaConL-spezifizierte Constraints) und die generierten weichen Constraints enthält, ist die neue Problemspezifikation. Nach der Transformation der Spezifikation in ein adäquates Problemlösungsmodell (siehe 3) werden die weichen Nicht-Existenzconstraints und die weichen Existenzconstraints zur Summe über die Existenzvariablen Ex_Var der benannten Objekte bzw. die Terme "$1 - Ex\_Var$" für die weichen Existenzconstraints transformiert. Die resultierende Summe wird durch *restricted forward labeling* der involvierten Existenzvariablen minimiert. Nach durchzuführendem Backjumping wird der Wert des (nicht gelabelten) Summentermes mit dem ermittelten Minimum gleich gesetzt. Dadurch wird ein Konfigurationsraum erzeugt, der durch interaktive Vorgaben weiter reduziert werden kann (vgl. 3.3). Für jede Lösung ist die Minimalität der Ersetzungen/ Änderungen bzgl. des Systems s, des gegewärtigen Produktmodelles und der Rekonfigurations-Vorgaben gesichert.

4.2 Ressourcen-zentrierte Rekonfiguration

Im Fall, daß für ein existierendes System eine neue oder erweiterte Funktionalität gefordert ist, ist es nicht immer sinnvoll, Systemelemente zu ersetzen. Statt dessen kann es billiger sein, lediglich einige Elemente zum bestehenden System hinzuzufügen. Wenn zum Beispiel die Kapazität einer Spannungsversorgung erhöht werden soll, könnte es sinnvoller sein, eine zusätzliche Spannungsversorgung parallel zu der existierenden hinzuzufügen, als die existierende durch eine leistungsstärkere zu ersetzen.

Will man solche ressourcen-zentrierten Rekonfigurationen unterstützen, so muß ConBaConL und das Problemlösungsmodell um sogenannte *generative Objekte* (siehe auch [6], [9]) erweitert werden.

Wir definieren generative Objekte als komplexe Objekte, die aus gleichartigen Komponenten bestehen, wobei diese in spezifizierter Weise miteinander verbunden sind. Wenn die Komponentenanzahl nicht durch ein spezielles ConBaConL-Constraint begrenzt ist, bleibt diese zunächst unbekannt. Die Eigeschaftswerte generativer Objekte werden von den Eigenschaftswerten ihrer Komponenten durch spezifizierte Formeln berechnet.

Für die geeignete Behandlung generativer Objekte im initialen Konfigurationsprozeß muß das Konfigurationsmodell teilweise reorganisiert werden. Zum Beginn des Konfigurationsprozesses hat jedes generative Objekt eine Komponente. Falls keine Problemlösung gefunden werden kann, wird ein neuer Konfigurationszyklus gestartet, wobei jedes generative Objekt zwei gleichartige Komponenten besitzt, wobei jeweils eine Komponente optional ist und darüber hinaus mit einem weichen Nicht-Existenzconstraint versehen ist. Wenn mit der neuen Modellinstanz auch keine Lösung gefunden werden kann, so wird für jedes generative Objekt eine neue optionale Komponente mit zugehörigem weichen Nicht-Existenzconstraint generiert, und so weiter. Der Zyklus wird solange wiederholt, bis eine Lösung gefunden wird, oder bis die Anzahl der Komponenten der generativen Objekte unrealistisch groß wird. Da in den meisten praktischen Konfigurationsproblemen die maximale Anzahl von gleichartigen Komponenten bekannt ist und somit durch eine feste Anzahl von optionalen Komponenten in der initialen Problemspezifikation spezifiziert werden kann, ist die Verwendung generativer Objekte lediglich für Rekonfigurationsaufgaben notwendig.

Die ressourcen-zentrierte Rekonfiguration beginnt mit der Generierung eines Rekonfigurationsmodells, wie im vorangegangenen Abschnitt beschrieben. Zusätzlich wird jede Komponente eines generativen Objektes ergänzt durch eine gleichartige optionale Komponente sowie ein dazugehöriges weiches Nicht-Existenzconstraint. Die Eigeschaftsformeln zwischen jedem generativen Objekt und seinen Komponenten müssen entsprechend adaptiert werden. Das resultierende Modell ist die Problemspezifikation für die ressourcen-zentrierte Rekonfiguration des existierenden Systems und wird wie oben beschrieben verarbeitet. Wenn jemand das Hinzufügen von mehr als einer gleichartigen Komponente pro generativem Objekt zulassen will, muß er weitere analoge Erweiterungen der generierten Rekonfigurations-Problem-Spezifikation durchführen.

5 Ausblick

Es wurde ein Rekonfigurationsansatz vorgestellt, der auf unserem constraint-logik-basierten Problemlösungsmodell für die Konfiguration von technischen Systemen basiert. Die Beschrei-

bung der Hauptelemente der zum Lösungsmodell korrespondierenden Spezifikationssprache ConBaConL sollte einen Eindruck über die beherrschbare Klasse von Konfigurationsproblemen vermitteln. Der auf dem vorgestellten Problemlösungsmodell basierende Prototyp ConBaCon wurde auf dem Gebiet industrieller Kontrollsysteme erfolgreich für die Konfiguration von Stromversorgungssystemen für große elektrische Antriebe getestet. Durch signifikante Reduzierung des Suchraumes, sichert das Problemlösungsmodell, zusammen mit dem darunterliegenden CLP-System, einen effizienten Konfigurationsprozeß, der durch Nutzerinteraktionen flexibel gesteuert werden kann. Es ist gesichert, daß jede gefundene Lösung korrekt bzgl. der Problemspezifikation und bzgl. des verwendeten Constraint-Lösers ist. Darüber hinaus ist die theoretische Vollständigkeit des Lösungsprozesses garantiert. Die Integration eines Constraint-Hierarchie-Transformators gestattet die Auswertung von interaktiv gegebenen "Verbesserungsweisungen" sowie von spezifizierten Präferenzen und stellt zugleich eine solide Basis für den vorgestellten Rekonfigurationsansatz dar.

Durch die Integration eines grafischen Problemeditors in das System ConBaCon, ist das System in der Lage, innovative Konfigurationsprozesse zu unterstützen, was für viele praktische Design-Probleme relevant ist (siehe [12]).

Ein Ansatz, das vorgestellte Lösungsmodell auch für große Konfigurationsprobleme nutzen zu können, ist die Modellclusterung, die in [15] beschrieben wird. Für die optimumnahe Lösung sehr großer komplexer Probleme ist eine zusätzliche Verteilung des existierenden Problemlösungsmodells unumgänglich. Dazu müssen geeignete Problemzerlegungsmethoden und Modelle korrespondierender Agentensysteme entwickelt werden. Ein weiterer Grund für die perspektivische Verteilung des Modells ist der Wunsch, existierende Teamstrukturen in konfigurationsrelevanten Unternehmen zu unterstützen. Ansätze hinsichtlich der Verteilung findet man in [8, 17, 14].

Literatur

[1] T. Axling: Collaborative Interactive Design in Virtual Environments. www.sics.se/∼axling/3dobelics.html.

[2] T. Axling, S. Haridi: A Tool for Developing Interactive Configuration Applications. J. of Logic Programming, 1996.

[3] T. Darr, M. Klein, D.L. McGuinness: Configuration Design. Journal on AI for Engineering Design, Analysis and Manufacturing, September 1998.

[4] E. Gelle, R. Weigel: Interactive Configuration using Constraint Satisfaction Techniques. Proc. of PACT'96.

[5] Boi Faltings, Rainer Weigel: Constraint-Based Knowledge Representation for Configuration Systems. Technical Report TR-94/59 des EPFL, 1994.

[6] G. Fleischanderl, G. Friedrich, A. Haselböck, M. Stumptner: Configuring Large Systems Using Generative Constraint Satisfaction. IEEE- Int. Systems 4/ 1998.

[7] Eugene C. Freuder: The Role of Configuration Knowledge in Business Process. IEEE-Int. Systems 4/ 1998.

[8] L. Gupta, J.F. Chionglo, Mark S. Fox: A Constraint Based Model of Coordination in Concurrent Design Projects. www.ie.utoronto.ca/EIL/DITL/WET-ICE96/ProjectCoordination/. 1996.

[9] A. Haselböck, M. Stumptner: A Constraint-Based Architecture for Assembling Large-Scale Technical Systems. Proc. of Int. Conf. on Expert Systems Applications/ AI in Engineering, Edinburgh, June 1993.
[10] P. van Hentenryck, V. Saraswat: Constraint Programming: Strategic Directions. J. of Constraints, 2/ 1997.
[11] U. John: Constraint-Based Simulation of Configuration Processes. Proc. of IMACS'97, August 1997.
[12] U. John: Constraint-Based Design of Reliable Industrial Control Systems. In "Advances in Systems, Signals, Control and Computers (V. Bajic, Ed.)". IAAMSAD. Durban, South Africa. September 1998.
[13] U. John: Model and Implementation for Constraint-Based Configuration. Proc. of IN-AP'98.
[14] U. John, U. Jähnichen: Agenten-orientierte Konfiguration Industrieller Produkte. Proc. des Vorbereitungs-WS zum DFG-Schwerpunktprogramm "Intelligente Softwareagenten und betriebswirtschaftliche Anwendugsszenarien." Ilmenau, Juli 1999.
[15] U. John: Solving Large Configuration Problems Efficiently by Clustering the ConBaCon-Model. Submitted to IEA/AIE-2000.
[16] D. Mailharro: A Classification and Constraint-Based Framework for Configuration. Journal on AI for Engineering Design, Analysis and Manufacturing, September 1998.
[17] Van Parunak et al: Distributed Component-Centered Design as Agent-Based Distributed Constraint Optimization. Proc. of WS Constraints and Agents on AAAI'97.
[18] A. J. Pasik: The Configuration Invasion. Report, Lazard Frères & Co. LLC, www.selectica.com/html/articles/Lazard1.html. New York, September 1998.
[19] D. Sabin: www.cs.unh.edu/ccc/config. 1996.
[20] M. Rafea: Towards A Configuration Specification Language Based On Constraint Logic Programming. Proc. of WS on Configuration at AAAI'99, Orlando, 1999.
[21] D. Sabin, E.C. Freuder: Configuration as Composite Constraint Satisfaction. Proc. of AAAI'96.
[22] D. Sabin, R. Weigel: Product Configuration Frameworks - A Survey. IEEE Int. Systems 4/ 1998.
[23] A. Schiemann, U. John, U. Geske, D. Boulanger: Realization and Application of Constraint Hierarchies for Configuration of Technical Systems with ConBaCon (In German). Proc. of 12th Workshop Logic Programming (WLP'97). Munich 1997.
[24] M. Stumptner: An Overview of Knowledge-Based Conf.. AI Communications. Vol. 10 No. 2, 1997.
[25] J. Tiihonen, T. Soininen, T. Männistö, R. Sulonen: State-of-the-Practice in Product Configuration - A Survey of 10 Cases in the Finnish Industry. In [26], 1996.
[26] T. Tomiyama, M. Mäntylä, S. Finger (Eds.): Knowledge Intensive CAD. Capman & Hall, 1996.
[27] M. Veron, H. Fargier, M. Aldanondo: From CSP to configuration problems. Proc. of WS on Configuration at AAAI'99, Orlando, 1999.

Reduce-To-The-Max: ein schneller Algorithmus für Multi-Ressourcen-Probleme

Hans Schlenker
GMD FIRST
Kekuléstraße 7
12489 Berlin
hs@first.gmd.de

Zusammenfassung

Mit Reduce-To-The-Max stellen wir einen Algorithmus vor, der für spezielle Probleme der diskreten kombinatorischen Optimierung den zwar endlichen aber in der Eingabegröße exponentiellen Suchraum derart einschränkt, dass (fast) nur noch gute Lösungen enumeriert werden. Das gilt für Multi-Ressourcen-Probleme, bei denen sowohl Tasks als auch Ressourcen geteilt werden können, und Tasks oder Ressourcen in möglichst großen Einheiten zugeordnet werden sollen. Wir entwickeln hier den Algorithmus, beleuchten empirisch seine Qualität, theoretisch seine Korrektheit und zeigen mögliche Anwendungen auf.

1 Einleitung

Die meisten Zuordnungs-Probleme (siehe z.B. [2, 17, 8, 9]) sind NP-hart. Für solche existieren vor allem klassische Lösungsverfahren aus dem *Operations Research* (*OR*, [13]) und eben auch — seit einigen Jahren — der elegante Constraint-Satisfaction-Ansatz, der zusammen mit der Logikprogrammierung seine volle Durschlagskraft entfaltet und deshalb meist in diesem Zusammenhang realisiert wird: *Constraint-Logik-Programmierung* (*CLP*, [16]).

Zuordnungsprobleme sind gegeben durch eine Menge von Aufgaben (*Tasks*), eine Menge von Ressourcen, eine Menge von Bedingungen und eine Menge von Optimierungsfunktionen, die die entsprechenden Kriterien formalisieren. Das Problem ist dann, die Aufgaben so auf die Ressourcen zu verteilen, dass alle Bedingungen erfüllt sind und die Optimierungsfunktionen optimale Werte liefern.

Viele aktuelle Lösungsalgorithmen benutzen zwei Grundbedingungen: sie betrachten Aufgaben als unteilbar und (fast immer:) sie betrachten Ressourcen als unteilbar (z.B. [1]). Für einige Probleme sind das auch sinnvolle Voraussetzungen: so können etwa Maschinen in der Regel nur genau ein Werkstück zu einer Zeit bearbeiten.

Das Weglassen dieser Bedingungen eröffnet entscheidende Möglichkeiten: die Algorithmen werden allgemeiner, die Klasse der damit lösbaren Probleme wird größer. Probleme, die diese Bedingungen nicht haben, sind u.a. Verschnittprobleme, Lagerhaltung, Transport- und Flussprobleme, Investitionsprobleme ([13]) und die Belegung von Gewächshäusern als Teilproblem der Gärtnereiplanung.

Für diese Problemklasse haben wir einen effizienten Labeling-Algorithmus entwickelt, der eine Optimierungsheuristik verwendet und damit den Suchraum erheblich einschränkt. Er

eignet sich vor allem für Probleme, bei denen die gegebenen Aufgaben in möglichst großen Einheiten auf die Ressourcen verteilt werden sollen.

Unser Papier ist wie folgt gegliedert: Wir geben zunächst eine sehr kurze Einführung in Constraint-Logik-Programmierung und Zuordnungsprobleme, anschliessend beschreiben wir den Algorithmus Reduce-To-The-Max und diskutieren ihn. Schliesslich fassen wir zusammen und blicken etwas in die Zukunft.

2 Constraint-Logik-Programmierung und Zuordnungsprobleme

2.1 CSP und CLP

Ein *Constraint Satisfaction Problem* (CSP) ([16]) besteht aus einer Menge von Variablen und einer Menge von Relationen zwischen diesen Variablen, den *Constraints*. Eine *Belegung* der Variablen weist jeder Variablen einen Wert aus ihrem Wertebereich zu. Eine *gültige Belegung* berücksichtigt dabei alle diesbezüglichen Constraints. Eine *Lösung* eines CSP ist eine gültige Belegung aller Variablen des CSP. Eine *optimale Lösung* des CSP ist eine Lösung, bei der die Optimierungsfunktionen optimale Werte ergeben.

Constraint Logic Programming (CLP) ([16]) ist eine logikbasierte Umsetzung des Constraint-Paradigmas. Dabei wird ein zentrales Konzept der Logikprogrammierung, die Unifikation, durch das allgemeinere des *Constraint Solving* ersetzt. *Constraint Solver* sorgen für die Einhaltung der Bedingungen und für optimale Verbreitung (*Propagation*) der impliziten Effekte, i.e. Bedingungen. Technische Realisierungen solcher CLP-Systeme bedienen sich der Sprache *Prolog* ([7]) und sind v.a.: CHIP ([4]), ECLIPSE ([10]), IF/Prolog ([11]), SICSTUS ([15]) und GNU-Prolog ([5]).

2.2 Modellierung von Zuordnungsproblemen

Zuordnungsprobleme können eingermassen direkt als CSP formuliert werden. Generell gibt es dazu zwei Ansätze:

- Tasks Ressourcen zuordnen:
  Aufgaben (die in diesem Fall auch *Jobs* genannt werden) in (eine fest vorgegebene Anzahl von) Teilaufgaben (hier: Tasks) zerlegen und für jede Teilaufgabe und jede Ressource eine Variable definieren. Der Wert der Variablen beschreibt dann den Anteil, den die Teilaufgabe an der Ressource hat.

- umgekehrt Ressourcen Tasks zuordnen:
  Ressourcen in (eine fest vorgegebene Anzahl von) Teil-Ressourcen zerlegen und für jede Teil-Ressourcen und jeden Task eine Variable definieren. Der Wert der Variablen beschreibt dann den Anteil der Aufgabe, der von der Teil-Ressource erbracht wird.

Uns interessieren jetzt vor allem Probleme, bei denen Ressourcen in möglichst großen Einheiten vergeben bzw. Aufgaben in möglichst großen Einheiten erledigt werden sollen. Durch obige Modellierung als CSP werden maximale Einheiten durch Belegung der entsprechenden Variablen mit maximalen Werten erreicht.

Bei der Lagerhaltung bespielsweise sollen möglichst große Losgrößen erzeugt werden ([13]), bei der Belegung von Gewächshäusern sollen zusammengehörende Pflanzungen in möglichst

wenigen Gewächshäusern untergebracht werden, jede Teilpflanzung also möglichst groß ausfallen. In der Regel haben gerade bei diesen Problemen die Variablen des CSP besonders große Wertebereiche, wodurch die Suche nach einer möglichen und guten Lösungen die Benutzung trivialer Verfahren verhindert.

Ein Belegungs-Verfahren, das von vornherein dieses Optimierungskriterium berücksichtigt, eine geeignete Heuristik also, würde automatisch zuerst oder sogar ausschliesslich die besten Werte für die entsprechenden Otimierungsfunktionen liefern. In letzterem Fall würde sogar der gesamte Suchraum erheblich eingeschränkt. Unser Verfahren Reduce-To-The-Max realisiert diesen Fall.

3 Der Reduce-To-The-Max-Algorithmus

3.1 Grundidee

Die Grundidee von Reduce-To-The-Max (RTTM) ist, die Variablen des Teilproblems des CSP, für das wir maximale Belegung erreichen wollen, *ausschliesslich* mit ihren jeweils *maximal möglichen* Werten zu belegen, bei Backtracking also nicht über die Werte der Domänen der Variablen zu iterieren, sondern nur über die Reihenfolge der Behandlung der Variablen. Diese maximalen Werte bedeuten für das konkrete Problem maximale Einheiten. Der Constraint-Solver sorgt dafür, dass jeweils der maximal *mögliche* Wert genommen wird[1].

Abbildung 1 beschreibt die Belegungsstrategie von RTTM.

- $V :=$ alle Variablen des Teilproblems
- selektiere aus V die Variable v mit (zuerst) der größten Domäne und (dann) dem größten Domänen-Maximum
- belege diese Variable mit ihrem Domänen-Maximum
- (gehe in die Tiefe[2]) belege $V' := V \setminus \{v\}$ nach der gleichen Strategie
- bei Backtracking (in die Breite[2] des Suchbaums): selektiere aus V' nach obiger Strategie die beste Variable w und gehe mit $V'' := V' \cup \{v\}$ in die Tiefe.

Abbildung 1: Belegungsstrategie

[1] Natürlich findet die Auswahl der *möglichen* Werte in den Grenzen des aktuell gegebenen Constraint-Solving, also der konkreten Propagation, statt (siehe auch Abschnitt 5).

[2] *Tiefe* und *Breite* des Suchraums beziehen sich hier auf die übliche Darstellung des Suchbaums, nach der die Suche von oben nach unten die Variablen belegt, so dass jedes Blatt eine Lösung representiert, und in der Breite des Baums (von links nach rechts) die Alternativen besucht werden.

3.2 RTTM$_0$

Der grundlegende RTTM-Algorithmus sieht also wie folgt aus: Abbildung 2 beschreibt ihn als Prolog-Prädikat. `select_var(Vars,V,Max,Rest)` selektiert aus der Liste `Vars` die Variable `V` mit ihrem Domänen-Maximum `Max` entsprechend obiger Selektionsregel. `Rest` ist `Vars` ohne `V`. Die Liste `Pres` ist die Liste der Variablen, die schon als die besten selektiert und untersucht wurden.

```
rttm0(Vars) :- walk(Vars,[]).

walk(Vars,Pres) :-
    select_var(Vars,V,Max,Rest),
    ( % depth
        V = Max,
        append(Pres,Rest,All),
        walk(All,[])
    ; % breadth
        walk(Rest,[V|Pres])
    ).
walk([],[]).
```

Abbildung 2: rttm0

3.3 RTTM$_1$: 1. Optimierung

Das Problem von RTTM$_0$ ist, dass es tatsächlich alle möglichen Permutationen der Variablen durchprobiert. Das muss aber nicht sein.

Wir beobachten, dass bei der Belegung der Variablen irgendwann ein Punkt erreicht ist, bei dem alle „noch zu belegenden“ Variablen (die aus V' bzw. `[V|Pres]`) schon belegt sind (durch Propagation). Der Algorithmus permutiert dann über die Reihenfolge der restlichen Variablen, ohne dass dadurch eine neue Lösung generiert wird.

Wir schneiden die Suche und Permutation hier entsprechend ab. Das ist sehr einfach, da wir ja immer die Variable mit der größten Domäne selektieren, und wenn deren Domäne einelementig ist, ist sie und alle restlichen Variablen instantiiert.

3.4 RTTM$_2$: 2. Optimierung

Eine weitere Optimierung, die zweite, ergibt sich aus einer Verallgemeinerung dieser Idee.

Wir benutzen im folgenden einige Abkürzungen: $\langle\theta\rangle$ bezeichne den Zustand $\theta \in \Theta = \{x, y, z, \theta'\}$ des Constraint-Netzwerks, $\phi := \psi$ die Belegung der Variablen ϕ mit dem Wert ψ, $\langle\alpha\rangle\ \phi := \psi\ \langle\beta\rangle$ den Übergang des Constraint-Netzes vom Zustand α in den Zustand β durch die Variablenbelegung (und natürlich Propagation). Für einen Zustand $\theta \in \Theta$ bezeichne $\theta(\tau)$ den Wert des Terms τ im Zustand θ. Außerdem ist $\mathrm{dom}(\phi)$ die Menge der Werte, mit denen ϕ noch belegt werden kann. Das ist eine Funktion (je)des Constraint-Solvers.

Betrachten wir folgende Situation: gegeben zwei Variablen a und b und folgende Belegung:

$$\langle x \rangle \;\; a := \max(\mathrm{dom}(a)) \;\; \langle y \rangle \;\; b := \max(\mathrm{dom}(b)) \;\; \langle z \rangle$$

Außerdem gelte

$$x(\max(\mathrm{dom}(b))) = y(\max(\mathrm{dom}(b)))$$

Wir nennen a und b in diesem Fall *unabhängig*. Man beachte hier, dass zwischen $a := \max(\mathrm{dom}(a))$ und $b := \max(\mathrm{dom}(b))$ nichts passiert! Konkret ist das bei unserem Algorithmus die Situation dass `walk` „in die Tiefe geht".

Unsere Annahme ist nun, dass wenn a und b unabhängig sind, in der umgekehrten Situation (der Algorithmus ist durch Backtracking in den Zustand x zurückgekommen)

$$\langle x \rangle \;\; b := \max(\mathrm{dom}(b)) \;\; \langle y' \rangle \;\; a := \max(\mathrm{dom}(a)) \;\; \langle z' \rangle$$

gilt:

$$y'(\max(\mathrm{dom}(a))) = x(\max(\mathrm{dom}(a)))$$

Diese Annahme wollen wir *Unabhängigkeitsannahme* nennen und darauf gleich nochmal zurückkommen.

Wenn diese Annahme gilt, dann gilt aber auch[3]

$$z = z'$$

und damit insbesondere, dass alle weiteren Belegungen, die nach z durchgeführt werden, genauso (nachdem sie ja immer nach derselben deterministischen Strategie erzeugt werden) nach z' durchgeführt werden und damit hier derselbe Such-Teilbaum nochmal erzeugt wird.

Und deshalb können wir hier wiederum abschneiden: wir merken uns nach jeder Variablenbelegung die davon unabhängigen Variablen und verhindern bei der umgekehrten Belegung auf derselben Ebene des Suchbaums die nochmalige Berücksichtigung der entsprechenden Variablen.

4 Diskussion

4.1 Unabhängigkeitsannahme und Propagation

Die Unabhängigkeitsannahme ist plausibel. Beweisidee:

Gegeben wie oben unabhängige Variablen a und b: $\langle x \rangle \;\; a := \max(\mathrm{dom}(a)) \;\; \langle y \rangle \;\; b := \max(\mathrm{dom}(b)) \;\; \langle z \rangle \;\; \wedge\, x(\max(\mathrm{dom}(b))) = y(\max(\mathrm{dom}(b)))$. Angenommen nun, die Unabhängigkeitsannahme gelte *nicht*, in

$$\langle x \rangle \;\; b := \max(\mathrm{dom}(b)) \;\; \langle y' \rangle \;\; a := \max(\mathrm{dom}(a)) \;\; \langle z' \rangle$$

gelte also:

$$y'(\max(\mathrm{dom}(a))) \neq x(\max(\mathrm{dom}(a)))$$

Dann gibt es zwei Möglichkeiten:

[3] Obwohl das nicht unmittelbar einleuchtend ist, ist es doch nicht schwierig: anhand z.B. des CLP-Kalküls aus [6], sieht man gleich, dass die entsprechenden Formeln logisch äquivalent sind.

1. $y'(\max(\mathrm{dom}(a))) < x(\max(\mathrm{dom}(a)))$
   Dann aber hätte $b := \max(\mathrm{dom}(b))$ eine Propagation verursacht, die mit $a := x(\max(\mathrm{dom}(a)))$ inkonsistent wäre und dann wäre auch die erste Belegung gescheitert und damit a und b ohnehin nicht unabhängig (auf dieser Ebene, da es diese Ebene ja dann nicht mehr gäbe).

2. $y'(\max(\mathrm{dom}(a))) > x(\max(\mathrm{dom}(a)))$
   Das würde aber bedeuten, dass durch Instantiierung einer Variable die Domäne einer anderen des Constraint-Netzes vergrößert wird, und das gibt es nicht.

Zwei Variablen können in obigem Sinne nur dann unabhängig sein, wenn sie unmittelbar hintereinander belegt werden. Das bringt ein Problem mit sich: es wird damit schwierig, RTTM mit anderen Verfahren zu kombinieren, die andere Variablen-Selektionsstrategien verwenden. Hier müssen weitere Untersuchungen ansetzen.

Generell stützt sich unser Verfahren stark auf gute Propagation. Wir müssen noch theoretisch untersuchen, welche Voraussetzungen (welche Art von Konsistenzen, etc.) unser Algorithmus benötigt, um den Suchraum optimal einschränken zu können.

4.2 Ein Beispiel: Gärtnereiplanung

Für uns ist vor allem das Problem der Gärtnerei-Planung interessant, die vorliegende Arbeit ist ja aus der Beschäftigung damit hervorgegangen. Neben anderen ist die Belegung von Gewächshäusern ein zentrales Teilproblem der Gärtnerei-Planung. Sie sieht grob wie folgt aus.

Gegeben sei eine Menge $P = \{1, ..., n\}$ von Pflanzungen. Jede Pflanzung $p \in P$ ist definiert durch einen Startzeitpunkt $s_p \in S \subset \mathbb{N}$, eine Dauer $d_p \in D \subset \mathbb{N}$ und Platzbedarf $r_p \in R \subset \mathbb{N}$. Außerdem sind gegeben eine Menge $H = \{1, ..., m\}$ von Gewächshäusern und jedes Gewächshaus $h \in H$ ist definiert durch eine Fläche $l_h \in L \subset \mathbb{N}$. Nachdem Pflanzungen teilbar sind (eine Pflanzung kann auf mehrere Gewächshäuser verteilt werden), müssen wir auch noch festlegen, wieviel Fläche jedes Gewächshauses auf welche Pflanzung entfällt: für jede Pflanzung $p \in P$ und jedes Gewächshaus $h \in H$ einen Wert $t_{p,h} \in T \subset \mathbb{N}$.

Hier muss für jedes Gewächshaus die Bedingung gelten, dass die Belegung mit (Teilen von) Pflanzungen zu keiner Zeit die verfügbare Fläche überschreiten darf. Wir benutzen das beliebte Constraint `cumulative`[4]:

$$\forall h \in H : \texttt{cumulative}((s_p)_{p\in P}, (d_p)_{p\in P}, (t_{p,h})_{p\in P}, l_h)$$

Außerdem muss der Platz der Teilpflanzungen einer Pflanzung p genau den Platzbedarf dieser Pflanzung ergeben:

$$\forall p \in P : \sum_{h\in H} t_{p,h} = r_p$$

[4]Formal: Gegeben ein Tupel $(s_1, \ldots, s_n)$ von Startzeitpunkten $s_i : S \subset \mathbb{N}$, ein Tupel $(d_1, \ldots, d_n)$ von Laufzeiten $d_i : D \subset \mathbb{N}$, ein Tupel $(r_1, \ldots, r_n)$ von Verbrauchswerten $r_i : R \subset \mathbb{N}$ und $t : T \subset \mathbb{N}$, die Verfügbarkeit der gemeinsamen Ressource, dann bezeichnet $\texttt{cumulative}((s_1, \ldots, s_n), (d_1, \ldots, d_n), (r_1, \ldots, r_n), t)$ die Menge $\left\{((s_1, \ldots, s_n), (d_1, \ldots, d_n), (r_1, \ldots, r_n), t) \in (S^n \times D^n \times R^n \times T) \middle| \forall k \in \left[\min_{i\in[1,n]}\{s_i\}, \max_{i\in[1,n]}\{s_i + d_i\}\right] : \sum_{j\in[1,n]|s_j\le k\wedge k\le s_j+d_j} r_j \le t\right\}$

Die Modellierung als CSP erfolgt direkt: für alle Werte s_p, d_p, r_p, l_h und $t_{p,h}$ eine logische Variable. Obige Bedingungen können direkt als Constraints hingeschrieben werden: wir benutzen `cumulative` und das Gleichheitsconstraint `#=` der endlichen Domänen.

Ein konkretes Problem wird spezifiziert durch Belegung der Variablen für r_p und l_h und durch Belegung, Bereichsbeschränkung oder sonstige Bedingungen (z.B. Verknüpfung von zwei Pflanzungen i und j durch das Constraint s_i `#=` $s_j + d_j$) der Variablen für s_p, d_p und evtl. sogar $t_{p,h}$. Eine konkrete Belegung aller Variablen ist dann eine Lösung des konkreten Problems.

Sehen wir uns noch ein konkretes Beispielproblem an, das wir *Referenzbeispiel* nennen wollen: Abbildung 3. Wir geben das Problem der Übersichtlichkeit halber in unserer Problembeschreibungssprache als Prolog-Fakten an.

```
requests([
    request(rose1,[_,_],100,100,9..10,_),
    request(aster1,[_,_],50,100,22..25,A),
    request(aster2,[_,_],200,A,6..7,_),
    request(kohl1,[_,_],200,108..110,13..15,_),
    request(begonie1,[_,_],30,115,19..20,_)
    ]).
greenhouses([ greenhouse(a,100), greenhouse(b,200) ]).
```

Abbildung 3: Referenzbeispiel

Ein Gewächshaus wird hier spezifiziert durch einen Term `greenhouse`(h, l_h), eine Pflanzung durch einen Term `request`$(p, [t_{p,\mathtt{a}}, t_{p,\mathtt{b}}], r_p, s_p, d_p, e_p)$. e_p bezeichnet den Endzeitpunkt einer Pflanzung p und wird durch das Constraint e_p `#=` $s_p + d_p$ realisiert.

Hier sehen wir auch Beispiele für die oben erwähnten Einschränkungen der Variablen: die Pflanzungen `aster1` und `aster2` sind über die gemeinsame Variable `A` zeitlich verknüpft, $d_{\mathtt{rose1}}$ und andere sind bereichsbeschränkt[5], $s_{\mathtt{rose1}}$ ist instantiiert. Alle $t_{p,h}$ und die meisten e_p sind an keine zusätzlichen Bedingungen gebunden.

4.3 Komplexität und Vollständigkeit

Unser Verfahren kann für die beschriebenen Probleme den Suchraum gegenüber trivialen Verfahren erheblich einschränken. Das gilt u.a. dann, wenn die Anzahl der möglichen Werte der Variablen (das Maximum der Größen der Wertebereiche) sehr viel größer als die Anzahl der Variablen ist. Wie gesagt ist gerade diese Eigenschaft oft bei konkreten Anwendungen gegeben. Im schlechtesten Fall, wenn also die Optimierungen von RTTM_2 nicht zum Tragen kommen, enumeriert Reduce-To-The-Max alle möglichen Permutationen der Variablen (nicht aber ihrer Werte)!

In der Praxis zeitigen aber gerade die Optimierungen enorme Reduktionen des Suchraums. Empirisch ergeben sich für das Referenzbeispiel folgende Ergebnisse[6]:

[5] Ein Term $a..z$ steht in CLP üblicherweise für einen FD-Bereich.

[6] Die Ergebnisse des Referenzbeispiels sollen hier nur einen Eindruck vermitteln, *wie weit* RTTM den Suchraum einschränken kann.

- Es hat potentiell einen Suchraum von ungefähr 10^{12}.
- RTTM_0 generiert hierfür noch $10! \approx 3.6 * 10^6$ Lösungen[7]. Davon sind aber nur 32 unterschiedlich!
- RTTM_1 erzeugt noch 992 Lösungen,
- RTTM_2 gerade noch 68.

Andere empirische Untersuchungen ergaben entsprechende Ergebnisse. Konkret hängt die Komplexität des Algorithmus, nämlich die Zahl der erzeugten Lösungen, von der Problemmodellierung und der Implementierung der verwendeten Constraints ab. Das verbietet zum Teil eine genaue theoretische Analyse.

Offensichtlich ist RTTM unvollständig in dem Sinn, dass nicht alle möglichen Variablenbelegungen enumeriert werden. Das aber ist ja gerade unser Ziel: nämlich nur diejenigen Variablenbelegungen zu erzeugen, die korrekte und gute Lösungen ergeben. Die Lösungen sind wie gesagt genau dann *gut*, wenn es für das Problem gut ist, Zuordnungen in maximal großen Einheiten zu haben.

Leider macht es keinen Sinn, RTTM *perfekt* zu machen, in dem Sinne, dass er für das Referenzbeispiel nur die 32 verschiedenen Lösungen generiert. Dazu müssten nämlich jeweils alle schon belegten Kombinationen gespeichert und entsprechend überprüft werden, was offenbar exponentiellen Aufwand bedeutet. Einfacher ist es auch hier (in der Logikprogramierung), den berühmten Trick der Dynamischen Programmierung: die *Memoisierung* (z.B. [14, 3]) zu verwenden.

4.4 Mögliche weitere Anwendungen

Natürlich ist Gärtnereiplanung nicht die einzige Anwendung, für die Reduce-To-The-Max geeignet ist. Wie gesagt, kommen alle Multi-Ressourcen-Probleme, bei denen sowohl Aufgaben als auch Ressourcen geteilt werden können in Betracht, insbesondere solche, bei denen möglichst große zusammenhängende Einheiten gefunden und zugeordnet werden müssen. Wir haben oben bereits andere mögliche Einsatzgebiete genannt: Verschnittprobleme, Lagerhaltung, Transport- und Flussprobleme und Investitionsprobleme. [1] nennt unter dem Stichwort *Scheduling with Continuous Resources* noch weitere.

Natürlich ist Gärtnereiplanung nicht die einzige Anwendung, für die Reduce-To-The-Max geeignet ist. Wie gesagt, kommen alle Multi-Ressourcen-Probleme, bei denen sowohl Aufgaben als auch Ressourcen geteilt werden können in Betracht. RTTM reduziert dann für genau die Probleme, bei denen möglichst große zusammenhängende Einheiten gefunden und zugeordnet werden müssen, die Suche auf die Möglichkeiten, bei denen jeweils genau die größtmöglichen Einheiten belegt werden. Wir haben oben bereits andere mögliche Einsatzgebiete genannt: Verschnittprobleme, Lagerhaltung, Transport- und Flussprobleme und Investitionsprobleme. [1] nennt unter dem Stichwort *Scheduling with Continuous Resources* noch weitere.

Wir können uns aber auch ein unter unseren Voraussetzungen bisher ungenanntes Anwendungsgebiet vorstellen: Scheduling von Computer-Prozessen. Diese benutzen offenbar ([1]) immer die Grundbedingung der Unteilbarkeit von Prozessoren. Wir denken aber an eine mehrstufige Modellierung: auf einer höheren Ebene betrachten wir einen Prozessor tatsächlich als

[7] Das Teilproblem des CSP, für das wir maximale Belegung erreichen wollen, ist hier gegeben durch die 10 Variablen $t_{p,h}$.

teilbar. Dieser Ansatz entstammt der Beobachtung, dass heutige Workstations virtuell ja auch zu einer Zeit mehr als einen Prozess ausführen können. Damit könnte man auf der höheren Ebene grobe Prozessor-Zuordnungen planen, die Prozessen Anteile an Prozessorzeit zuweist. Auf einer niedrigeren Ebene müsste man dann aus dem groben Plan konkrete Task-Umschaltungen generieren.

5 Zusammenfassung und Ausblick

In diesem Artikel haben wir einen Algorithmus vorgestellt, der für spezielle Probleme der diskreten kombinatorischen Optimierung, Multi-Ressourcen-Probleme, bei denen sowohl Aufgaben als auch Ressourcen geteilt werden können und mindestens eins von beiden in möglichst großen Einheiten zugeordnet werden soll, den zwar endlichen aber in der Eingabegröße exponentiellen Suchraum derart einschränkt, dass nur noch gute Lösungen enumeriert werden. Die beschriebene neuartige Belegungsstrategie, nicht über die Domänen der Variablen zu iterieren, sondern über die Reihenfolge ihrer Belegung, ergibt zusammen mit den entwickelten Reduktionen unnötiger Permutationen einen im empirischen Durchschnittsfall sehr effizienten Algorithmus. Obwohl die Begründung für diese Effizienzsteigerung fast sofort evident ist, soll neben empirischen Studien vor allem der geplante formale Beweis dieses Ergebnis erhärten.

Neben unserem Einsatzgebiet der Gärtnereiplanung, deren schwerwiegendstes Teilproblem der Flächenbelegung wir kurz dargestellt haben, ergeben sich viele weitere interessante Einsatzgebiete für RTTM.

Zu den wichtigen anstehenden Arbeiten zählen wir die genauere theoretische Untersuchung der Eigenschaften und Voraussetzungen unseres Verfahrens, sowie mögliche Verbesserungen und Verallgemeinerungen, potenziell in einem abstrakteren Zusammenhang wie dem in [12] angedachten. Für interessant halten wir auch die oben erwähnte Betrachtung von Problemen des Prozessor-Schedulings unter unseren erweiterten Rahmenbedingungen. Auf diese Fragestellungen werden sich weitere Untersuchungen konzentrieren.

Literatur

[1] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes.* Springer-Verlag, Berlin, Heidelberg, 1996.

[2] P. Brucker. *Scheduling Algorithms.* Springer-Verlag, Berlin, Heidelberg, 1998.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge (Mass.), 1990.

[4] Cosytec, http://www.cosytec.fr. *The CHIP Universe.*

[5] D. Diaz. *GNU Prolog - a Prolog Compiler.* Free Software Foundation, http://www.gnu.org/software/prolog/.

[6] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung.* Springer-Verlag, Berlin, Heidelberg, 1997.

[7] U. Geske. *Prolog.* Akademie Verlag, Berlin, 1993.

[8] H.-J. Goltz. Reducing domains for search in CLP(FD) and its application to job-shop scheduling. In *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming (CP95)*, volume 976 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 1995. Springer-Verlag.

[9] H.-J. Goltz and U. John. Methods for solving practical problems of job-shop scheduling modelled in CLP(FD). In *Proceedings of the 2nd International Conference on the Practical Application of Constraint Technology (PACT'96)*, Blackpool, UK, 1996. The Practical Application Company.

[10] IC-Parc, http://www.icparc.ic.ac.uk/eclipse/. *The ECLiPSe Constraint Logic Programming System.*

[11] IF/Computer, http://www.ifcomputer.com/Products/IFProlog/. *IF/Prolog.*

[12] F. Labourthe and Y. Caseau. Salsa: A language for search algorithms. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP98)*, volume 1520 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 1998. Springer-Verlag.

[13] K. Neumann and M. Morlock. *Operations Research.* Hanser Verlag, Wien, 1993.

[14] P. Rao, I. V. Ramakrishnan, T. Swift, and D. S. Warren. Dynamic argument reduction for in-memory data queries. In U. Geske and D. Seipel, editors, *Proceedings of the Workshop on Deductive Databases and Logic Programming*, volume 231 of *GMD Studien*, Sankt Augustin, 1994. Gesellschaft für Mathematik und Datenverarbeitung.

[15] SICS, http://www.sics.se/sicstus.html. *SICSTUS.*

[16] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge (Mass.), 1989.

[17] M. Wallace. Applying constraints for scheduling. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, volume 131 of *Advanced Science Institutes Series F.* NATO, 1993.

Transforming Object-Oriented Domain Models into Declarative CLP Expressions

Markus Hannebauer
Technical University Berlin
Department of Computer Science
Software Engineering Laboratory
Franklinstr. 28/29, 10587 Berlin, Germany
hannebau@cs.tu-berlin.de

Ines Münch
Humboldt University Berlin
Department of Computer Science
Artificial Intelligence Laboratory
Rudower Chaussee 25, 12489 Berlin, Germany
muench@informatik.hu-berlin.de

Abstract

An increasing number of applications of information technology is characterized by the need for sophisticated methods of knowledge-based problem solving. Hence, though most of the recent software systems are written in traditional programming languages there has been increasing awareness of the demand for more abstract and declarative languages.

In this paper, we will present a multi-stage framework for transforming object-oriented domain models, as they can often be found in information systems, into more declarative constraint logic expressions to solve complex problems. This framework is tailored to the needs of distributed constraint satisfaction among several agents. We will exemplify the usage of our concepts by following a real-world case study in medical appointment scheduling.

1 Introduction

An increasing number of industrial and administrative applications of information technology is characterized by the need for sophisticated methods of problem solving, such as planning or scheduling. Hence, though most of the recent software systems are written in traditional programming languages, like C(++), Java or Visual Basic, there has been increasing awareness of the demand for more abstract and declarative languages, especially in knowledge-based problem solving. *Constraint logic programming* (CLP, [10, 2]) is a very successful representative of these programming paradigms. Nevertheless, practice has proven it to be utopian to expect a general change in commercial software development customs. Therefore, a hybrid approach to these application problems is necessary, marrying conventional programming style and domain representations with more sophisticated knowledge-based paradigms for problem solving.

Since domain knowledge and models are often encoded into (relational) databases and conventional class/object models, we propose an approach to transform such specific domain-dependent models to more generic CLP expressions that can be used for solving complicated problems. Several papers, mostly application-oriented, mention a specific transformation of real-life data to more abstract CLP expressions (rf. e. g. to [3, 7]). In this paper, we do not propose such a specific transformation process. Also, we do not propose a generic automatic transformation process from all kinds of object models to CLP expressions. Rather, we present a generic transformation framework that ensures that domain-dependent transformation procedures are integrated to form a unique process with well-defined interfaces to the client program, e. g. agent. Our transformation framework also differs in its strict design for applicability in collaborative problem solving, i. e. problem solving among autonomous intelligent agents that share the common goal to find consensus.

In section 2 we will briefly introduce our approach to collaborative problem solving including the short description of our main protocol for distributed constraint satisfaction. A major part of this protocol is an efficient procedure for transforming traditionally encoded domain models to exchangeable constraint representations. After a short overview to the transformation framework in section 3, section 4 will present a domain model that can be found very often in information, control and decision support systems, like ERP or workflow systems. The sections 5 and 6 will then detail the main contribution of this paper, introducing *constraint factories* and *constraint objects* as key concepts in our transformation framework. Despite the generality of our concepts, we will use a specific problem in medical appointment scheduling to show the practical usefulness of our proposal. Section 7 will apply the transformation framework to a case study within the research project *ChariTime* that aims at designing, realizing and evaluating a system of autonomous agents that try to negotiate on medical appointments. We will finish our paper with concluding remarks and a short glimpse on future work.

2 Collaborative problem solving

Many problems in managing business processes, like allocating and scheduling tasks and resources, can be adequately modeled as *constraint satisfaction problems* (CSPs, [9, 12]) on finite domains. To put it into a nutshell, CSPs consist of a set of variables, each possessing a certain set of possible values, and a set of restricting constraints. The task is then to find for each variable one value from the variable's set of possible values such that all constraints are satisfied.

All problems modeled this way can be solved by a monolithic CLP solver, like ECL^{i}PS^e [13]. Though this possibility exists in principle, it is often not practicable in real settings. Traditional monolithic systems often scale purely in measure of process instantiations and they usually ignore the problem of restricted information distribution. The information on variables and constraints is spatially distributed among several organizational units. In case of a monolithic solver, one would have to collect all the information from its several sources, transfer it to the solver, solve the problem and again distribute the results of optimization among the different users. Hence, even in case of a central optimizer, one has to cope with communication and information consistency problems. There are other, much stronger arguments for distributing the solution procedure of real-life CSPs to several entities, including efficiency, privacy issues and responsiveness. More on this can be found in [5].

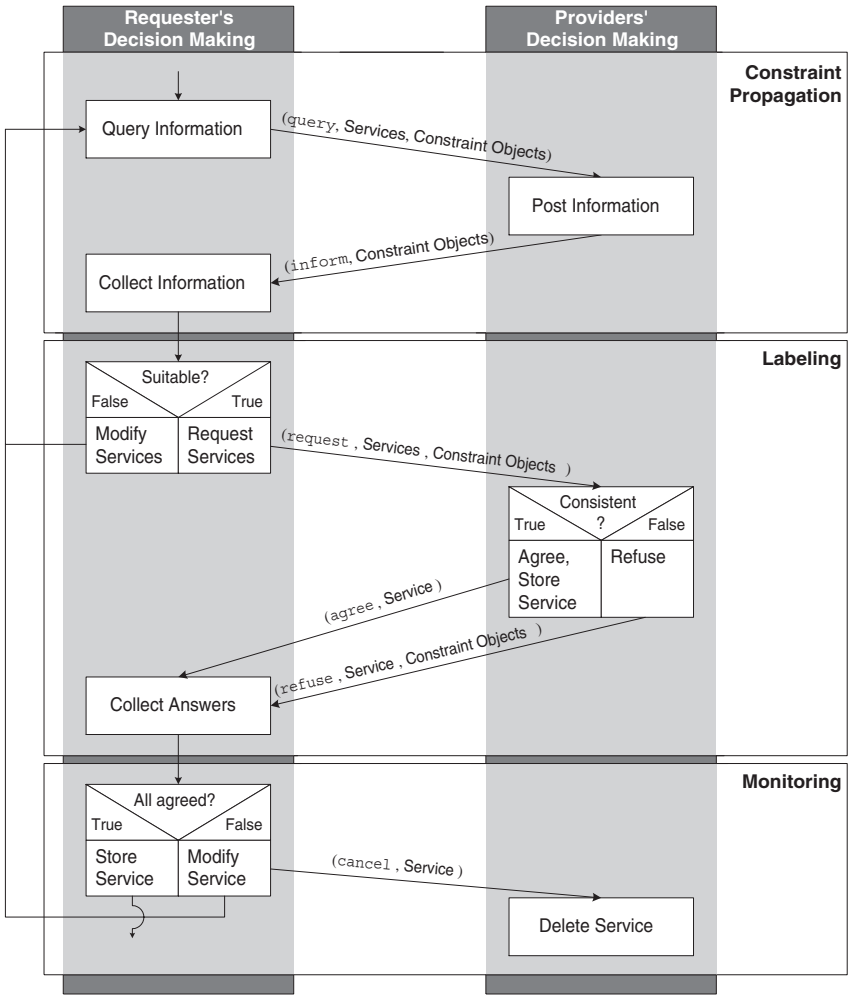


Figure 1: Coarse Overview to the Consensus Finding Protocol

Despite the advantages of distribution, such systems also have major disadvantages. The complexity that has been saved within the several solvers is transferred to the collaboration process. Due to this, investigations on todays distributed solver systems often report poor optimization results or vast communication overhead. Concepts for architectures of agents and multi agent systems that support dynamic reconfiguration of the system structure to cope with this overhead can be found in [6]. In [4] we propose a powerful consensus finding protocol for collaborative problem solving that is related to research in distributed constraint satisfaction [14, 8]. This protocol is transparent to the current agent system configuration and nevertheless efficient. Transparency means that in the case of collaborative problem solving it should be the same whether the constraints between subproblems are internal to a unique agent or externally distributed among several agents. The consensus finding protocol should allow for both. For efficiency, the protocol shall reduce communication effort by preventing exhaustive asynchronous search for solutions and shall improve results by making decisions more well-informed.

Figure 1 provides a coarse overview to this protocol. It shall not be discussed in detail, but one can easily observe that it uses three main phases of negotiation. The first two of

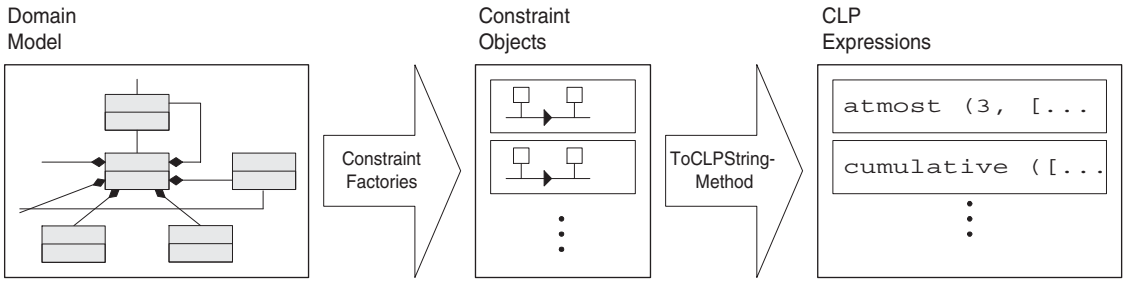


Figure 2: Transformation Framework

these phases strongly rely on the possibility to exchange variable and constraint information efficiently between agents. Both on the requester side and on the provider side agents use the exchanged variable and constraint information to compute locally good answers for the next negotiation round. In the next sections we will discuss how an agent can transform its domain knowledge to constraint objects that can easily be exchanged between neighbors and also easily transformed to CLP expressions for using them in local constraint solvers.

3 Overview to the framework

The task of an agent involved in a consensus finding protocol is to make rational decisions on how to answer queries, proposals, requests, agreements, refusals and other messages. In our setting, these decisions are made with the help of constraint logic programming. To use CLP, we have to assume the existence of a problem model appropriate for solving the problem by CLP. Usually, pure declarative CLP problem models consist of a set of variable declarations together with a set of constraints both denoted by a specialized form of an enriched first order logic calculus. We will not detail here the usual additional demand for an operational specification of a labeling procedure. Rather, we will focus on the problem to derive declarative CLP expressions by transforming the above described domain model appropriately. Figure 2 provides an overview to the whole transformation framework.

The starting point of transformation is the object-oriented domain model. This domain model is investigated by domain-dependent constraint factories that use the knowledge encoded in the domain model to produce several constraint objects. These constraint objects are exchangable and can be used as a basis for the before-mentioned consensus finding protocol. Constraint objects provide a method `ToCLPString` that can be finally used to produce valid CLP expressions. In the following section we will detail this transformation framework further.

4 Domain models

In typical information, control and administration systems the data and domain knowledge is not stored in a format that is directly accessible for AI approaches, e. g. rule-based systems, heuristic search or constraint programming. The formats are usually not rule-based, deductive or case-based in nature, but mostly relational and more seldom, but increasingly, object-oriented. One could try to permanently transform this kind of commercially popular representations to representations that are more easily accessible for AI methods. The

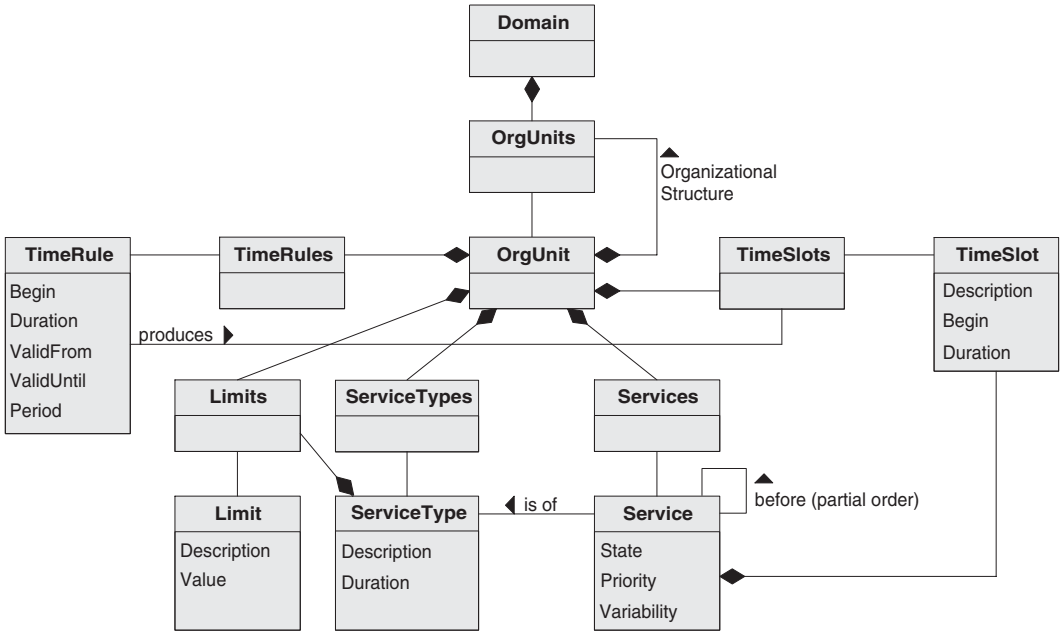


Figure 3: Exemplary Object Model

main problem of this approach lies in maintenance. System administrators are usually not willing to administrate several heterogeneous data collections with different representations. Another problem is interoperability with legacy systems and third-party systems.

Hence, the approach presented in this paper does not propose to permanently transform relational or object-oriented domain models to AI representations, but to do online transformation in case of a current need. In our approach we are using UML notation [11] to analyze and design domain knowledge. Figure 3 shows a simplified example for such an object model. It is intentionally created on top of a very abstract domain ontology that describes relations of different organizational units playing the roles of providers of and requesters for services. The network of providers, requesters and connecting services underlies a variety of constraints that are divided into constraints related to time slots and constraints associated with other resource and capacity limits.

All collections of objects (usually 1-n aggregations) are denoted explicitly in the model because these collections play an important role in our approach to transformation. Collections are denoted by plural wording. The root of the domain object model is an object called Domain. It contains a collection of organizational units each of which can again subsume several organizational units. This relation encodes the hierarchical structure that can be often found in human organizations. The OrgUnit is the central object in our model. In our medical appointment scheduling problem wards, diagnostic units, patients and so on are represented by organizational unit objects. Each organizational unit contains a collection of TimeSlot objects that can have different semantics according to the role of the attached organizational unit. For example for providers these objects represent off-times and other periods of time that influence the availability calendar of the provider but do not need working resources. For requesters, these objects determine desired periods of time for service.

In addition to the concrete periods of time, there are abstract rules for time periods. These are encoded by TimeRule objects. A common example for a time rule is “every Monday from

7 AM to 11 AM is an off-time". If applied to a concrete time horizon this rule will produce a certain collection of concrete TimeSlot objects. Organizational units also possess several restricting values represented by Limit objects. A typical example is a restriction on the number of services that can be provided on each day.

ServiceType objects are used to encode the available services at a providing organizational unit. They are also subject to restrictions encoded in limits and define the functionality of a service provider. They also contain standards for typical values of services like estimated duration, estimated resource usage and so on.

All the named objects represent constraints or organizational relations among constraints. The decision variables of our scheduling problem are encapsulated in Service objects. To use a common metaphor in agent-oriented development, our notion of services denotes contracts in different states, like open, requested, committed and so on. In case of a committed contract, both the requester and the provider(s) keep track of this contract by inserting an appropriate Service object into their Services collection by building a customized copy of the ServiceType object. Hence, services at provider side represent a major part of the calendar, and services at requester side represent some kind of simple workflows. To further the workflow metaphor, services are allowed to be related by a given partial order.

5 Constraint factories

An important part of the mentioned transformation is the knowledge where to collect the data from the domain model to form a declarative problem model. This knowledge is of course domain-dependent and hence differs from application to application and has to be encoded in some kind of program. One significant idea of our approach to transformation is not to hardwire this knowledge within the agent, but to encapsulate it into so-called constraint factories. Constraint factories are dynamically loadable components, that fulfill the following simple IDL interface.

```
interface IConstraintFactory
{
  HRESULT Init ([in] IDomain *domain, [in] long orgunitid);
  HRESULT SetServices ([in] ICollection *services);
  HRESULT CreateContext ();

  HRESULT ProduceConstraintObjects ([out, retval] ICollection **objects);
};
```

Each problem solving agent can posses an arbitrary number of different constraint factory instances and types. Using constraint factories is easy for an agent. Prior to first usage, each constraint factory is initialized with interface access to the root node of the agent's domain model by calling `Init`. This initialization links the factory to the proper point in the object model and equips it with domain information that is always up-to-date. In constraint satisfaction one important thing to specify are the constrained variables. In our framework, these decision variables are encoded in Service objects. Hence, the variable specification is done via the method `SetServices`. Finally for initialization, the agent calls `CreateContext` to force the constraint factory to build its production context, for example by exploring the

given domain model and the given variables. The importance of this context will be explained in the next section.

After initialization, the agent can simply call the `ProduceConstraintObjects` method for commanding the factory to produce a proper collection of constraint objects. Every constraint factory knows on its own what parts of the domain model have to be investigated to produce proper constraint objects. Because of the monotony of constraint sequences, the agent can simply traverse its constraint factories without even knowing about their purpose and collect the resulting constraint objects. This is the big advantage compared to hardwired transformation within the agent. Because constraint factories are dynamically loadable components, the set of constraint factories acquainted to an agent can even change dynamically.

6 Constraint objects

In a monolithic, centralized problem solver it would be possible to realize constraint factories that directly produce CLP expressions, but ChariTime is purposed to be a distributed system. To support the consensus finding protocol briefly mentioned in section 2, agents have to exchange knowledge on variables, domains and constraints. CLP expressions, especially from the finite domains constraint system, are not directly suitable, since they usually contain implicit contextual knowledge that cannot be derived from the pure CLP expressions.

Here is a typical example: CLP problem models in scheduling usually assume a certain discrete scheduling horizon, often starting from zero and ending at a however found upper bound. Additionally, they assume a certain, fixed granularity of the scheduling horizon that is determined by the size of the minimum necessary time unit. For example, one day with a scheduling granularity of 15 minutes would imply a scheduling horizon of $[0, \ldots, 96]$. If all variables and constraints were defined using this horizon the situation would be consistent. But variables and constraints are distributed among several agents, and different variables and constraints may imply different scheduling horizons. Hence, the contextual information on absolute time points and absolute granularity underlying the horizon has to be transfered together with the set of constraints such that the agent receiving variable and constraint information can deduce the underlying horizon.

In our framework, so-called constraint objects are responsible for encapsulating this kind of contextual information. They are designed to contain as much information as necessary to make clear the semantics of the transfered variables and constraints to the receiving agent, but not more. The receiving agent shall not have insight on facts in the domain model of the sending agents that have nothing to do with the transfered information. Therefore, constraint objects also have a security and privacy purpose in constraint communication. Constraint objects are as well as constraint factories realized as dynamically loadable components. All constraint objects implement the following IDL interface, which is derived from an interface for streamable objects.

```
interface IConstraintObject : IStreamable
{
  HRESULT GetHorizon ([out] DATE *mindate,
                      [out] DATE *maxdate,
                      [out] long *granularity);
  HRESULT UpdateHorizon ([in, out] DATE *mindate,
                         [in, out] DATE *maxdate,
```

```
                        [in, out] long *granularity);

    HRESULT ToCLPString([out, retval] BSTR *string);
  };
```

This interface ensures that every constraint object implements the ability to determine its minimum absolute scheduling date (including time), its maximum absolute scheduling date and its granularity. The according set of methods is used by a constraint factory to adjust the horizon of the currently produced constraint objects correctly by traversing them. `UpdateHorizon` pushes the minimum date to its minimum over all constraint objects, analogously for the maximum date and the granularity is successively set to the greatest common divisor of all granularities. This information is stored in every constraint object.

Similar to constraint factories, constraint objects need to know which variables they have to restrict. In our framework, all constraint objects are defined on Service objects. This may not always be the case, but it is in our applicational setting. Nevertheless, we handle this as a special case and introduce another inherited interface allowing the definition of a Service collection to be restricted by the constraint object.

```
interface IConstraintServices
  : IConstraintObject
{
  HRESULT AddService ([in] IService *service);
  HRESULT SetServices ([in] ICollection *services);
};
```

With the help of this interface constraint factories can set the definition set of constraint objects. Constraint factories directly hand over their service collections, received in the initialization phase described above, to the constraint objects. Some constraint objects need a collection of fixed arity, but most accept an arbitrary arity.

After adjusting the horizon correctly and indicating the set of affected Service objects, the method `ToCLPString` can be used to produce a string containing a sequence of arbitrary CLP expressions that describe the meaning of the constraint object. This is also shown by figure 2. To make variable names in CLP expressions unique for the whole system, *globally unique identifiers* (GUIDs) are used. The CLP representation of the beginning of a certain time slot is for example given by `T_F927FDA0-A421-11d3-9E6D-00A024AF15AE_begin`. For briefness, we will use shorter IDs in the following examples. The CLP expressions produced by the `ToCLPString`-method of constraint objects can directly be concatenated and used for solving the specified problem with a traditional solver.

7 Case Study

Scheduling patients is a promising application for CLP and is currently used as a case study by several researchers from Humboldt University and Technical University Berlin in cooperation with the cardiological clinic of Charité Berlin, Europe's biggest hospital. The cardiological clinic of Charité consists of five wards with a capacity of altogether over 80 patients, four outpatients' facilities, in which different types of medical consulting are done in parallel, and eight diagnostic units, some of which with several workplaces. The diagnostic units perform

over 100 diagnostic examinations each day. These examinations are requested by the wards, the outpatients' department and other clinics of Charité.

The present problem is the coordination between the requesting and serving units. Spatial and organizational distribution of the named units results in distributed knowledge, distributed control and hence suboptimal patient throughput and resource usage. As many industrial and administrative planning and scheduling problems, this problem can be expressed by a CSP on finite domains. Considering all the arguments for collaborative problem solving given in section 2, we have decided to design, realize and evaluate a truly distributed multi agent system, which is supposed to run on 25 to 30 computers all over the whole cardiological clinic. The system shall be local, flexible and permanently active to allow the dynamic allocation of actors and resources to diagnostic tasks, while coping with failures and emergency cases. We have modeled the given problem as a collaborative problem solving task among autonomous but benevolent agents. In the following, we will exemplify the use of our transformation framework in this case study.

The object-oriented domain model of ChariTime directly corresponds to the one shown by figure 3. There are of course many other entities representing further information on organizational units and services, but the ones important for the pure scheduling problem can be found in the figure. Constraint factories and constraint objects are realized using Microsoft's *distributed component object model* (DCOM, [1]).

Prior to requesting a set of appointments, a requester specifies certain constraints, like the partial order and desires for appointments. The requester uses a constraint factory to produce a set of constraint factories of the following types. For a detailed introduction to the used types of constraints please take a look at [5].

`IConstraintPatientDesires` Ensures a domain for the begin variables of the affected Service objects. This domain directly corresponds to patient desires for appointments.

```
T_3_begin::[2..16,19..33]
```

`IConstraintPatientBefore` Ensures directed order between the two affected Service objects.

```
T_3_begin + T_3_duration #<= T_4_begin
```

`IConstraintPatientCorrectSchedule` Ensures non-overlapping of all affected `Service` objects representing the appointments of a single patient. This constraint object encodes the "atomic character" of the "resource" patient.

```
cumulative ([T_3_begin,T_4_begin], [T_3_duration,T_4_duration], [1,1], 1)
```

We will take a closer look at the reasoning process necessary for a provider (a diagnostic unit) to decide whether and when to schedule a set of requested appointments. The request for scheduling a set of appointments reaches the provider via a set of Service objects each describing the demand for a specific examination together with a set of constraint objects restricting the choice of the provider. The task of the provider agent is then to derive all constraints from its domain knowledge restricting the Service objects and hence to build the solution space for scheduling the appointments. We have designed three types of constraint factories to accomplish this task: a constraint factory defining basic domains and relations among decision variables, a constraint factory for workplaces producing workplace-dependent constraint objects and a resource factory ensuring the proper resource usage in the diagnostic unit.

The basic constraint factory explores the domain knowledge on the workplaces connected to the diagnostic unit. This can be done by traversing the organizational structure on OrgUnit objects in the domain model. The context (scheduling horizon) of this constraint factory is determined by the desires of the patients. After precomputation, the factory produces constraint objects of the following types.

`IConstraintDiagUnitWorkplaces` Restricts the workplace variable of the affected Service objects to be from a certain domain of workplaces.

```
T_3_workplace::[0,1]
```

`IConstraintDiagUnitTransformed` Binds the begin variables of the affected Services objects to transformed begin variables that support the choice among different workplaces in the same diagnostic unit (alternatives).

```
T_3_begintrf #= T_3_begin + T_3_workplace * 96
```

`IConstraintDiagUnitDays` Binds the begin variables of the affected Service objects to according day variables by equation constraints.

```
T_3_rest::[0..23], T_3_begin #= 24 * T_3_day + T_3_rest
```

After the basic specification of variables and their relations, a constraint factory for every workplace is used to produce workplace-dependent constraint objects. The choice of a concrete workplace for a service by labelling the `T_n_workplace` variable entails a set of conditional constraints, including the restriction to certain free time slots for automatic appointments, the service duration, setup times between services of the same type and the calendar consistency. The information for these constraint objects can be found in TimeSlot, Limit and Service collections connected to the OrgUnit object of the workplace. Workplace constraint factories examine this information and build constraint objects of the following types.

`IConstraintWorkplaceTimes` Ensures a workplace-dependent domain for the begin variables of the affected Service objects. This domain directly corresponds to time slots available for automatic appointments.

```
T_3_workplace #= 0 #=> T_3_begintrf::[7..12,18..35]
```

`IConstraintWorkplaceDuration` Defines conditional constraints, that bind duration variables of the affected Service objects to the choice of the workplace variables.

```
T_3_workplace #= 0 #=> T_3_duration #= 2
```

`IConstraintWorkplaceSetup` Ensures a workplace-dependent setup time between all pairs out of the affected Service objects. Holds only in case of two services sharing the same workplace.

```
(T_3_workplace#=0 #/\ T_4_workplace#=0) #=>
(T_3_begin + T_3_duration + 5 #<= T_4_begin #\/
T_4_begin + T_4_duration + 5 #<= T_3_begin)
```

`IConstraintWorkplaceCorrectSchedule` Ensures non-overlapping of all affected `Service` and `TimeSlot` objects based on the transformed variables.

```
cumulative ([T_3_begintrf,15], [T_3_duration,2], [1,1], 1)
```

Finally, a resource constraint factory produces constraint objects that ensure the proper use of resources within the diagnostic unit taking into account already scheduled appointments. The information on this can be found in the Limit collection of the OrgUnit object of the diagnostic unit and in the Service collections of the OrgUnit objects of its workplaces. The resource constraint factory produces constraint objects of the following types.

`IConstraintDiagUnitMaxPerDay` Ensures that at most n day variables of the affected Service objects are labeled with the same given value. Because of the usage of the original day variables, services are affected by this constraint object independently from their assigned workplace.

```
atmost (5, [T_3_day,T_4_day], 2)
```

`IConstraintDiagUnitCorrectSchedule` Ensures that there is not more cumulated resource usage by the affected Service objects than specified. This constraint object considers all workplaces of a single diagnostic unit, since resources are assigned to diagnostic units and not to single workplaces.

```
cumulative ([T_3_begin,T_4_begin], [T_3_duration,T_4_duration], [2,1], 2)
```

After having produced all constraint objects, the provider agent can either use the constraints delivered by the `ToCLPString` method to find a consistent schedule for the requested appointments or can transfer the constraint objects to the requester to allow a further selection on requester side.

8 Conclusion

In this paper, we have presented a multi-stage framework for transforming object-oriented domain models, as they can often be found in information, control and workflow systems, into more declarative CLP expressions to solve complex problems. This framework is tailored to the needs of distributed constraint satisfaction, since it allows for the dynamic extention of the transformation knowledge by encapsulating it into constraint factories. The concept of constraint objects provides a good foundation for constraint communication since it preserves all the information needed to interpret the transmitted constraints, like information on the scheduling horizon, but allows to obey privacy and security matters. The usage of this concept set has been exemplified by applying it to medical appointment scheduling.

Though in our medical appointment scheduling example we often have almost a direct correspondence between constraint objects and global constraints known from sophisticated CLP languages, this is not necessary. Constraint objects may also produce more complicated CLP expressions, for example defining high-level predicates or even whole CLP programs. There is always a trade-off in finding the right balance between using constraint factories, constraint objects and CLP expressions. To reduce complexity in the upper layers of our multi-stage transformation framework one could transfer the complexity to more complex CLP expressions, doing a lot of work for the upper layers. On the other hand, one could only use very primitive CLP expressions and do all the transformation work in the upper layers. This has to assessed depending on the domain. We plan to empirically assess the right balancing for our application soon. The named constraint objects have been fully implemented, the constraint factories are still under development. After having finished this, we can start the overall evaluation of our approach to collaborative problem solving. Forthcoming papers will report on our progress in this area.

References

[1] D. Box. *Essential COM.* Object Technology. Addison Wesley, 1998.

[2] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung.* Springer, 1997.

[3] H.-J. Goltz, G. Küchler, and D. Matzke. Constraint-based timetabling for universities. In *Proceedings of the Eleventh International Conference on Applications of Prolog (INAP-98)*, pages 75–80, 1998.

[4] M. Hannebauer. Multi-phase consensus communication in collaborative problem solving. In *Proceedings of the Third Workshop on Communication-based Systems (CBS-2000, to appear)*, pages 131–146. Kluwer, 2000.

[5] M. Hannebauer and U. Geske. Coordinating distributed CLP-solvers in medical appointment scheduling. In *Proceedings of the Twelfth International Conference on Applications of Prolog (INAP-99)*, pages 117–125, Tokyo, Japan, 1999.

[6] M. Hannebauer and R. Kühnel. Dynamic reconfiguration in collaborative problem solving. In H.-D. Burkhard, L. Czaja, H.-S. Nguyen, and P. Starke, editors, *Proceedings of the Eighth Workshop on Concurrency, Specification and Programming (CS&P-99)*, pages 71–82, Warsaw, Poland, 1999.

[7] U. John. Model and implementation for constraint-based configuration. In *Proceedings of the Eleventh International Conference on Applications of Prolog (INAP-98)*, Tokyo, Japan, 1998.

[8] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. Research Report RR-93-74, Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, UK, 1993.

[9] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wiley-Interscience Publication, second edition, 1992.

[10] K. Marriot and P. J. Stuckey. *Programming with Constraints — An Introduction.* MIT Press, 1998.

[11] B. Oestereich. *Objektorientierte Softwareentwicklung — Analyse und Design mit der Unified Modeling Language (UML).* Oldenbourg, 1998.

[12] E. P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[13] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, UK, 1997.

[14] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and DATA Engineering*, 10(5), 1998.

Über Methoden des constraintbasierten Lösens von Problemen der Stundenplanung

Hans-Joachim Goltz
GMD – Forschungszentrum Informationstechnik GmbH
GMD-FIRST, Kekuléstr. 7, 12489 Berlin
e-mail: goltz@first.gmd.de

1 Einleitung

Ein Stundenplan ist eine Zuordnung von Ereignissen zu Zeitpunkten bzw. Zeitintervallen, so daß die geforderten Bedingungen (Constraints) erfüllt sind. Bei dieser Zuordnung sind problemspezifische Bedingungen (Constraints) zu beachten. Neben den Bedingungen, die immer erfüllt sein müssen und auch harte Constraints genannt werden, existieren auch Bedingungen, die möglichst erfüllt sein sollen und als weiche Constraints bezeichnet werden. Seit langem ist bekannt, daß das Stundenplanungsproblem zu der Klasse der NP-vollständigen Problemen gehört (siehe z.B. [4]). Für die automatische Stundenplanung, die ein aktuelles Forschungsgebiet ist, existieren verschiedene grundlegende Softwaretechniken und Ansätze (siehe z.B. [10]). Die constraintlogische Programmierung (CLP) konnte in vielen Anwendungen zur Lösung komplexer diskreter Probleme sehr erfolgreich eingesetzt werde. Verschiedene CLP-Ansätze werden beispielsweise in [1, 7, 8, 9, 11] diskutiert.
Ein Schwerpunkt unserer Forschungsarbeiten ist die Entwicklung von Methoden, Verfahren und Konzepten für eine interaktive, automatische Stundenplanung, die in Universitäten und Schulen angewendet werden können. Die Systeme der Stundenplanung sollen so flexibel sein, daß Besonderheiten der Anwender berücksichtigt und Constraints leicht geändert werden können, falls keine grundsätzliche konzeptionelle Änderung der Stundenplanung erforderlich ist. Die constraintlogische Programmierung bildet die Grundlage zur Verwirklichung unserer Forschungsziele. Für die Realisierung einer effizienten automatischen Lösungssuche ist neben der Verwendung geeigneter Suchstrategien die Problemmodellierung von entscheidender Bedeutung. Deswegen sind für uns die Entwicklung von Konzepten und Methoden der Modellierung von Problemen der Stundenplanung und Untersuchungen zum Einfluß verschiedener Problemmodellierungen auf die Lösungssuche wichtige Forschungsschwerpunkte. Auch die Entwicklung und der Vergleich verschiedener Lösungsstrategien ist ein Bestandteil unserer Forschungsarbeiten.
Im Rahmen unserer Forschungen entwickelten wir ein System für die interaktive, automatische Stundenplanung an der Medizinischen Fakultät Charité der Humboldt Universität zu Berlin. Als Implementationssprache wurde die constraintlogische Programmiersprache CHIP (Version 5.2) der Firma COSYTEC aus Frankreich gewählt. Auch die grafischen Schnittstellen wurden in CHIP implementiert. Neben den globalen Constraints erwies sich die objektorientierte Komponente von CHIP besonders vorteilhaft für die Implementation.
Die Stundenplanung der Charité wird seit Sommersemester 1998 mit unserem System erzeugt. Seitdem wurde das System kontinuierlich weiterentwickelt. Die Vorteile einer kombinierten interaktiven und automatischen Stundenplanerzeugung konnten eindeutig nachgewiesen werden. Die erzeugten Pläne werden als HTML-Dateien ausgegeben und sind im Internet unter `http://www.first.gmd.de/plan/charite` zu finden.

Das Stundenplanungssystem besteht aus verschiedenen Komponenten. In [6] wird das Gesamtsystem vorgestellt. Auf ein Teilproblem, der Planung von Blockpraktika, wird dort nicht eingegangen. In dieser Arbeit diskutieren wir Methoden zum Lösen dieses Teilproblems. Im nächsten Abschnitt wird eine kurze Problembeschreibung gegeben. Im 3. Abschnitt wird die Modellierung der geforderten Bedingungen beschrieben. Anschließend werden mögliche redundante Constraints angegeben. Lösungsstrategien diskutieren wir im 5. Abschnitt. Für verschiedene Methoden und 6 Beispiele werden dann Laufzeitergebnisse der Lösungssuche dargestellt und ausgewertet.

2 Problembeschreibung

In der medizinischen Ausbildung müssen die Studenten relativ viele Pflichtveranstaltungen absolvieren. Neben den Vorlesungen eines Semesters finden Seminare, Untersuchungskurse und Praktika statt, die in Gruppen von bis zu 20 Studenten durchgeführt werden. Dazu werden die Studenten eines Semesters in Seminargruppen unterteilt. In bestimmten Studienabschnitten müssen die Studenten Praktika verschiedener Fächer absolvieren, die in der Regel jeweils an aufeinanderfolgenden Werktagen stattfinden, wobei die Anzahl der Tage bzw. Wochen vorgegeben ist. Diese Praktika werden als Blockpraktika bezeichnet.
Der gesamte Stundenplan besteht aus verschiedenen Komponenten. Für die Planung kann jede Komponente relativ unabhängig betrachtet werden, wobei aber eine bestimmte Reihenfolge einzuhalten ist. Zuerst sind die Vorlesungen für alle Semester zu planen. Anschließend können dann für jedes Semester die Lehrveranstaltungen geplant werden, die in Seminargruppen durchgeführt werden, wobei die relevanten Vorlesungszeiten als "Sperrzeiten" zu berücksichtigen sind. Die Planung der Blockpraktika kann völlig unabhängig erfolgen, da diese in vorgegebenen Wochen tageweise stattfinden.
Im folgenden betrachten wir nur die Planung von Blockpraktika. Diese Lehrveranstaltungen finden in einem vorgegebenen Zeitraum statt und werden in Gruppen von bis zu 20 Studenten durchgeführt. Die Dauer der Praktika ist jeweils vorgegeben. Bei der Planung von Blockprakika sind nur die Tage zu bestimmen, an denen sie stattfinden sollen. Die zugeordnete Tageszeit ist für die Planung nicht von Bedeutung. Ein gegebenes Praktikum ist von alle Gruppen oder einer angegebenen Menge von Gruppen zu absolvieren. Als *Praktikumseinheit* oder *Einheit eines Praktikums* bezeichnen wir im folgenden das Praktikum, das für eine bestimmte Gruppe stattfindet. Folglich kann jedem Praktikum eine Menge von *Praktikumseinheiten* zugeordnet werden.
Zwei Arten von Blockpraktika können unterschieden werden: *Wochen-Praktika* und *Tage-Praktika*. Den Wochen-Praktika sind eine bestimmte Anzahl von Wochen zuzuordnen. Wenn mehr als eine Woche zuzuordnen ist, müssen diese Wochen aufeinanderfolgen. Diese Praktika beginnen immer am ersten Werktag der Woche. Bei der Planung dieser Praktikumsart werden Feiertage ignoriert. Folglich kann die Anzahl der Tage, an denen diese Praktika tatsächlich stattfinden, unterschiedlich sein. Dagegen ist bei den Tage-Praktika diese Anzahl fest. Den Tage-Praktika ist eine bestimmte Anzahl von aufeinanderfolgenden Werktagen zuzuordnen, wobei freie Tage (Wochenende, Feiertage) nicht mitgezählt werden. Praktika dieser Art können an jedem Werktag beginnen. Es wird vorausgesetzt, daß es in jedem Zeitabschnitt (mit einer Länge, die der Dauer entspricht), in dem ein Wochen-Praktikum durchgeführt werden kann, weniger als fünf Feiertage gibt (d.h., weniger als Praktikumstage einer Woche). Die wichtigsten Bedingungen für Blockpraktika sind außerdem:

1. die Praktikumseinheiten, die einer Gruppe zugeordnet sind, dürfen sich nicht überlappen;
2. die Anzahl der Einheiten eines Praktikums, die zur gleichen Zeit stattfinden dürfen, ist beschränkt; diese Schranke muß nicht für die gesamte Zeit gleich sein;
3. es können Praktika existieren, die erst nach der Absolvierung von anderen Praktika stattfinden können;
4. es können Praktika existieren, bei denen jeweils ein direkt nachfolgendes Praktikum vorgegeben ist;
5. einige Praktika dürfen an bestimmten Tagen nicht beginnen;
6. Für jeweils zwei Einheiten eines Praktikums kann die folgende Bedingung gefordert werden: Entweder sie beginnen am gleichen Tag oder sie dürfen sich nicht überlappen;
7. die Anzahl der Tage, an denen die Praktikumseinheiten eines Praktikums beginnen dürfen, kann als minimal oder maximal vorgegeben sein, wobei auch eine genaue Anzahl dieser Tage vorgegeben werden kann.

3 Modellierung der Bedingungen

Der Constraintlöser über endlichen Domänen von CHIP bildet die Grundlage unserer Problemmodellierung. Die gewählte Modellierung eines Problems der Stundenplanung besitzt einen großen Einfluß auf die Propagationseigenschaften des Constraintlösers und damit auf die Effizienz der Lösungssuche. Die Problemmodellierung ist auch abhängig von den Möglichkeiten und Propagationseigenschaften des gewählten Constraintlösers. Die in CHIP enthaltenen globalen Constraints haben sich für unsere Modellierung als sehr wertvoll erwiesen.
Für jede Gruppe, die ein gegebenes Blockpraktikum absolvieren muß, wird eine Praktikumseinheit definiert. Allen Praktikumseinheiten sind Zeiten so zuzuordnen, daß alle Bedingungen erfüllt sind. Da die Dauer eines Praktikums vorgegeben ist, genügt es, jeweils die *Startzeiten* der Praktikumseinheiten zu bestimmen. Die Startzeit wird als Domänenvariable betrachtet. Die möglichen Ausgangswerte einer Domäne ergeben sich aus der fortlaufenden Numerierierung aller Tage, an denen Praktika stattfinden können. Die Domäne der Startzeitvariable eines Wochen-Praktikas besteht anfangs nur aus den natürlichen Zahlen, die der Menge der ersten Werktage der gegebenen Wochen entsprechen.
Weil die Dauer der Wochen-Praktika in Abhängigkeit von der zugeordneten Startzeit durch mögliche Feiertage variieren kann, wird die Dauer von Praktikumseinheiten solcher Praktika als Domänenvariable betrachtet. Für die Modellierung der Beziehung zwischen Dauer und der Startzeitvariable kann das vordefinierte symbolische Constraint `element/3` verwendet werden. Zum Beispiel gilt `element(N,List,Value)` genau dann, wenn das `N`-te Element der nichtleeren Liste `List` natürlicher Zahlen gleich `Value` ist, wobei `N` und `Value` Domänenvariablen sein können.

Die in der Problembeschreibung genannten Bedingungen können direkt durch vordefinierte Constraints formuliert und vor der Lösungssuche erzeugt werden. Für die Formulierung der Bedingungen 1 und 2 kann das globale Constraint `cumulative` direkt verwendet werden. Dieses Constraint sichert, daß von einer Ressource zu jedem Zeitpunkt nicht mehr als die angegebene Anzahl benötigt wird. Falls diese Schranke für den gesamten Zeitraum nicht

gleichmäßig ist, können "Dummy"-Einheiten definiert werden, die zu den entsprechenden Zeiten Kapazitäten blockieren. Die Bedingungen 3 und 4 können einfach durch Gleichungen und Ungleichungen formuliert werden. Durch Streichen von Elementen aus den Domänen von Startzeitvariablen kann gesichert werden, daß die Bedingung 5 erfüllt ist.

Die Modellierung der Bedingungen 6 und 7, die etwas komplizierter ist, wird im folgenden etwas genauer betrachtet. Sei P ein Praktikum, bei dem die Praktikumseinheiten die Bedingung 6 erfüllen müssen. Jeweils 2 Einheiten dieses Praktikums müssen folglich genau parallel stattfinden oder sie dürfen sich nicht überlappen. $X_1, \ldots, X_n$ seien die Tage, an denen mindestens eine Praktikumseinheit beginnt. Somit existiert für jede Praktikumseinheit genau ein X_i, das mit der Startzeit dieser Einheit übereinstimmt. Ohne Beschränkung der Allgemeinheit kann vorausgesetzt werden, daß $X_i < X_j$ für $i < j$ gilt. Wenn D die Dauer der Praktikumseinheiten ist, gilt die Bedingung 6 genau dann, wenn für jedes $i, j \in \{1, \ldots, n\}$ mit $i < j$ die Ungleichung $X_i + D \leq X_j$ erfüllt ist. Wenn P ein Wochen-Praktikum ist, muß für D der minimalste Wert einer Dauer gewählt werden. Die Ungleichung gilt dann auch für diesen Fall, da eine Beschränkung der maximalen Anzahl von Feiertagen innerhalb eines Duchführungszeitraumes vorausgesetzt wurde. Diese Bedingungen und Zusammenhänge werden nun durch Constraints ausgedrückt.
Sei $Starts$ die möglichen Startzeiten und n die Anzahl der möglichen verschiedenen Startzeiten der zugeordneten Praktikumseinheiten. Wenn $AnzT$ die Gesamtzahl der Tage ist, an denen überhaupt ein Praktikum durchgeführt werden kann, ist die Zahl n durch den Quotienten $AnzT/D$ beschränkt. Die möglichen verschiedenen Startzeiten $X_1, \ldots, X_n$ werden als Domänenvariablen mit der Domäne $Starts$ definiert und für jedes $i, j \in \{1, \ldots, n\}$ mit $j = i + 1$ wird das Constraint $X_i + D \leq X_j$ erzeugt. Für jede Praktikumseinheit P_i des Praktikums P sei $Start_i$ die Startzeitvariable mit der Anfangsdomäne $Starts$. Um die Beziehungen zwischen den Startzeitvariablen und den Variablen $X_1, \ldots, X_n$ ausdrücken zu können, wird für jede Praktikumseinheit eine weitere Domänenvariable definiert, die mit *StartNr* bezeichnet und Startnummervariable genannt wird. Die Anfangsdomäne jeder dieser Variablen wird mit $1 \ldots n$ festgelegt. Für jede Praktikumseinheit P_i wird nun das folgende Constraint[1] formuliert:

$$\texttt{element}(StartNr_i, [X_1, \ldots, X_n], [0, \ldots, 0], Start_i, \ [\texttt{all,all,all,all}])$$

Dieses Constraint sichert, daß die Beziehung $Start_i = X_{StartNr_i}$ gilt. Das 3. Argument ist im allgemeinen eine Liste von natürlichen Zahlen der Länge n, durch die eine konstante Abweichung von dieser Gleichung ausgedrückt wird. In unserem Fall ist diese Abweichung jeweils gleich 0. Für die Kontrolle der Propagation wird das letzte Argument verwendet. Durch die gewählte Angabe, wird das Constraint immer dann aktiviert, wenn sich die Domäne einer der beteiligten Variablen ändert.

Für die Modellierung der 7. Bedingung verwenden wir die eingeführten Startnummervariablen. Somit werden auch die Constraints vorausgesetzt, die die Beziehungen zwischen den Startzeitvariablen und den Startnummervariablen charakterisieren. Falls ein Praktikum die Bedingung 6 nicht erfüllen muß, ist die Anzahl n der möglichen verschiedenen Startzeiten nicht durch den Quotienten $AnzT/D$ beschränkt und die Ungleichung $X_i + D \leq X_j$ ist durch die Ungleichung $X_i < X_j$ zu ersetzen. Wenn in der Bedingung 7 eine maximale Anzahl verschiedener Startzeiten der Einheiten eines Praktikums gefordert ist, so kann dies durch die Wahl der Zahl n und somit durch die Wahl der Domänen für die Startnummervariable

[1] Das symbolischen Constraint `element/5` ist in der Version 5.2 von CHIP erstmalig enthalten.

gesichert werden.
Wenn eine minimale Anzahl Min verschiedener Startzeiten für ein Praktikum P gefordert ist, bedeutet dies, daß die Menge $\{StartNr_1, \ldots, StartNr_m\}$ mindestens Min verschiedene Elemente enthält, wobei m die Anzahl der Praktikumseinheiten von P sei. Zuerst bestimmen wir für jedes $k \in \{1, \ldots, n\}$ die Anzahl N_k, die aussagt, wie oft der Wert k in der Liste $[StartNr_1, \ldots, StartNr_m]$ vorkommt. Sei $MaxPar$ die Anzahl der Praktikumseinheiten von P, die höchstens parallel stattfinden dürfen, und für jedes $k \in \{1, \ldots, n\}$ sei N_k eine Domänenvariable mit der Domäne $0 \ldots MaxPar$. Für jedes $k \in \{1, \ldots, n\}$ wird folgendes `among`-Constraint erzeugt[2]:

$$\texttt{among}(N_k, [StartNr_1, \ldots, StartNr_m], [0, \ldots, 0], [k], \texttt{all})$$

Dieses Constraint gilt, wenn von den Variablen $StartNr_1, \ldots, StartNr_m$ genau N_k-vielen der Wert k zugeordnet wird. Das 3. Argument ist im allgemeinen eine Liste von natürlichen Zahlen der Länge m, durch die eine Abweichung vom Wert k für die entsprechende Variable ausgedrückt werden kann. In unserem Fall wird keine Abweichung erlaubt. Durch die Angabe "`all`" im letzten Argument wird dieses Constraint immer dann aktiviert, wenn sich die Domäne einer der relevanten Variablen ändert. Für die Domänenvariablen $N_1, \ldots, N_n$ muß außerdem die folgende Gleichung gelten: $N_1 + \ldots + N_n = m$. Durch ein weiteres `among`-Constraint kann nun ausgedrückt werden, wieviele der Domänenvariablen $N_1, \ldots, N_n$ höchstens den Wert 0 annehmen dürfen, wobei Y eine Domänenvariable mit der Domäne $0 \ldots (n - Min)$ ist: $\texttt{among}(Y, [N_1, \ldots, N_n], [0, \ldots, 0], [0], \texttt{all})$.

4 Redundante Constraints

Redundante Constraints sind zusätzliche Constraints, die logisch aus den anderen Constraints folgen. Solche Constraints werden hinzugefügt, um die Propagationseigenschaften zu verbessern. Folglich könnte dadurch der Suchraum reduziert werden. Durch Verwendung redundanter Constraints ist im allgemeinen aber eine Erhöhung der Rechenzeit für einen Suchschritt zu erwarten, wobei eine Verringerung des Suchraumes nicht garantiert werden kann. Für ein gegebenes Problem ist somit zu untersuchen, für welche redundante Constraints im Durchschnitt ein besseres Gesamtergebnis zu erwarten ist. Einige Beispiele redundanter Constraints für das gegebene Problem werden im folgenden beschrieben. Anschließend werden einige Testbeispiele mit verschiedenen Varianten des Hinzufügens redundanter Constraints angegeben und die Ergebnisse werden ausgewertet.
Die Anzahl der Einheiten eines Praktikums, die zur gleichen Zeit stattfinden dürfen, ist beschränkt (2. Bedingung). Diese Kapazitätsbeschränkung kann mittels des globalen Constraints `cumulative` auch bezüglich der Variablen $StartNr$ eines Praktikums formuliert werden, falls diese Variablen relevant sind. Dieses Constraint folgt logisch aus den Beziehungen der Variablen $StartNr$ und $Start$ und aus der constraintlogischen Modellierung der 2. Bedingung bezüglich der Variablen $Start$. Im folgenden setzen wir voraus, daß für alle Praktika, für die $StartNr$ eine Domänenvariable ist, dieses redundante Constraint erzeugt wird.
Die Beschränkung der Kapazität von Ressourcen kann auch mittels dem globalen Constraint `diffn` ausgedrückt werden. Durch dieses globale Constraint können abstrakte Bedingungen

[2] In CHIP kann diese Menge von `among`-Constraints zu einem speziellen `among`-Constraint zusammengefaßt werden.

der folgenden Art direkt modelliert werden: *Eine Liste n-dimensionaler Rechtecke ist überlappungsfrei in einem gegebenen n-dimensionalen Rechteck zu plazieren.* Solche Constraints könnten auch die `cumulative`-Constraints ersetzen. In diesem Bericht werden wir nur die Möglichkeit diskutieren, dieses Constraint bei den Ressourceneinschränkungen für die erste und zweite Bedingung zusätzlich in allen Fällen zu verwenden, wobei jeweils 2-dimensionale Rechtecke betrachtet werden. Bei der Modellierung der ersten Bedingung ist die Zeit eine Dimension, wobei die Dauer die Länge des Rechtecks bestimmt. In der anderen Dimension unterscheiden sich die Rechtecke nicht (z.B. Länge gleich 1 und Plazierung bei 0). Für die zweite Bedingung können `diffn`-Constraints für die Variablen *Start* und für die Variablen *StartNr* erzeugt werden. Diese Variablen bestimmen auch die Werte in der einen Dimension. Die Werte der anderen Dimension werden durch die mögliche Anzahl paraller Einheiten bestimmt. Dazu wird jeder Einheit eine Parallelitätsnummer zugeordnet und die Länge des entsprechenden Rechtecks in dieser Dimension beträgt 1.

Durch `among`-Constraints kann indirekt ausgedrückt werden, wieviele verschiedene Werte mindestens eine Liste von Domänenvariablen annehmen muß. Dies ergibt sich aus der Modellierung der 7. Bedingung. Auch wenn für ein Praktikum keine minimale Anzahl verschiedener Startzeiten gefordert wird, können `among`-Constraints so erzeugt werden, wie im 3. Abschnitt beschrieben wurde, wobei auf das `among`-Constraint bezüglich $N_1, \ldots, N_n$ verzichtet wird. Das Constraint $N_1 + \ldots + N_n = m$ wird aber erzeugt. Analog könnten solche Constraints auch bezüglich den Startzeitvariablen erzeugt werden.

Für zwei beliebige Praktikumseinheiten P_i und P_j eines Praktikums können die Beziehungen zwischen den Startzeitvariablen und den Startnummervariablen zusätzlich durch folgende bedingte Constraints ausgedrückt werden:

`if` $Start_i < Start_j$ `then` $StartNr_i < StartNr_j$ `else` $StartNr_i \geq StartNr_j$

`if` $StartNr_i < StartNr_j$ `then` $Start_i < Start_j$ `else` $Start_i \geq Start_j$

Wenn alle Möglichkeiten dieser Art der Beziehungen betrachtet werden sollen, sind im allgemeinen eine relativ große Anzahl solcher bedingter Constraints zu erzeugen.

5 Lösungssuche

Constraintlöser über endlichen Domänen sind nicht vollständig. Für die Erzeugung einer Lösung ist i. allg. Suche erforderlich, die oft durch *"Labelling"* realisiert wird. Dabei werden schrittweise den Constraintvariablen Werte aus ihren Domänen zugeordnet. Als Modifizierung dieser Suchmethode haben wir in [5] vorgeschlagen, daß die Wertzuweisung durch eine Einschränkung der Domäne der ausgewählten Variable ersetzt wird. Diese Methode der backtrackbaren Domänenreduzierung ist eine Verallgemeinerung der Labelling-Methode. Die Domänenreduzierung wird auch bei einigen Strategien, die wir im folgenden betrachten, verwendet. In der Lösungssuche sind zwei Arten von Nichtdeterminismus enthalten: Auswahl einer Domänenvariable und Auswahl eines Wertes aus der relevanten Domäne bzw. Auswahl der Domäneneinschränkung für die ausgewählte Variable. Natürlich ist es i. allg. nicht möglich, alle Auswahlmöglichkeiten der Suche zu probieren. Für die Lösungssuche sind folglich Heuristiken erforderlich, die die jeweilige Auswahl unterstützen. In diesem Beitrag werden wir einige Varianten von unterschiedlichen Suchstrategien diskutieren.

Bei der Planung der Blockpraktika ist eine Menge von Praktikumseinheiten gegeben. Den Domänenvariablen *Start* und *StartNr* jeder Praktikumseinheit müssen Werte aus ihren

Domänen so zugeordnet werden, daß alle Constraints erfüllt sind. In einer Suchstratgie muß auch festgelegt werden, in welcher Reihenfolge diese beiden Arten von Variablen bei der Suche einbezogen werden. Wir betrachten nun die folgenden Suchstrategien, wobei eine sortierte Liste von Praktikumseinheiten als gegeben vorausgesetzt wird:

S 1: für jede Einheit: Wertzuweisung von *Start* und gleich anschließend Wertzuweisung von *StartNr*;

S 2: erst für alle Einheiten Wertzuweisung von *StartNr* und anschließend für alle Einheiten Wertzuweisung von *Start*;

S 3: erst für alle Einheiten Wertzuweisung von *Start* und anschließend für alle Einheiten Wertzuweisung von *StartNr*;

S 4: erst für alle Einheiten Domänenreduzierung von *Start* und anschließend für alle Einheiten Wertzuweisung von *StartNr* und von *Start*;

S 5: für jede Einheit zuerst Domänenreduzierung von *Start* und dann gleich Wertzuweisung von *StartNr*, anschließend für alle Einheiten Wertzuweisung von *Start*;

S 6: für jede Einheit zuerst Wertzuweisung von *StartNr* und dann gleich Domänenreduzierung von *Start*, anschließend für alle Einheiten Wertzuweisung von *Start*

S 7: erst für alle Einheiten Domänenreduzierung von *Start* und dann für alle Einheiten Wertzuweisung von *StartNr*; und anschließend für alle Einheiten Wertzuweisung von *Start*.

Für die Sortierung aller Praktikumseinheiten bieten sich zunächst zwei Grundvarianten an: *Sortierung nach Gruppen* und *Sortierung nach Praktika*. Bei unseren Testversuchen erwies sich die 2. Variante als wesentlich besser. Deswegen werden wir hier folgendes Ordnungsprinzip voraussetzen. In der sortierten Liste der Praktikumseinheiten sind die Einheiten eines Praktikums zusammenhängend angeordnet, wobei sich die Reihenfolge durch die Priorität des Praktikums und durch die Gruppenzuordnung ergibt. Die Priorität der Praktika ist entscheidend für die Reihenfolge der Suche. Diese Reihenfolge besitzt natürlich einen großen Einfluß auf die Lösungssuche und kann auch über Erfolg oder Mißerfolg der Suche entscheiden. In diesem Bericht werden wir dies aber nicht genauer untersuchen. In den Beispielen, die im nächsten Abschnitt betrachtet werden, wird die Reihenfolge der Praktikumseinheiten für jedes Beispiel jeweils als gegeben und fest betrachtet.

6 Vergleich verschiedener Methoden

In diesem Abschnitt werden verschiedene Methoden an einigen Testergebnissen verglichen. Grundlage der Beispiele sind zwei praktische Probleme der Blockpraktikaplanung, die sich bei der Stundenplanung für das Sommersemester 1999 und für das Wintersemester 1999/2000 ergaben. Die erzeugten Pläne dieser Probleme sind auch unter der in der Einleitung gegebenen Internetaddresse zu finden. In den Abbildungen 1 und 2 sind diese Lösungen auch dargestellt. Wegen Feiertagen und Schulferien ist die ungleichmäßige Verteilung gewünscht. Bei beiden Problemen sind die Blockpraktika für das 5. klinische Semester zu planen. Diese Blockpraktika sind in 14 bzw. 16 Wochen zu absolvieren, wobei die gesamte Praktikumszeit die letzten 6 bzw. 8 Wochen des aktuellen Semesters und die ersten 8 Wochen des nächsten Semesters umfaßt. Bei diesen beiden Problemen kann vorausgesetzt werden, daß alle Praktika Wochen-Praktika

sind. Die beiden Probleme unterscheiden sich bezüglich der Anzahl der Gruppen und der Anzahl der erlaubten parallen Einheiten eines Praktikums. Für jedes dieser beiden Probleme betrachten wir im folgenden 3 Beispiele, die jeweils aus dem Problem abgeleitet wurden.

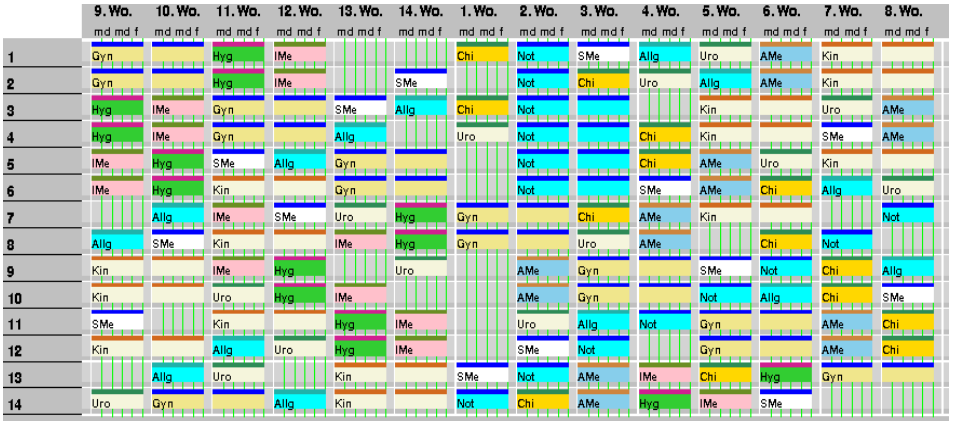


Abbildung 1: Eine Lösung der Beispiele Bsp 11, Bsp 12, Bsp 13

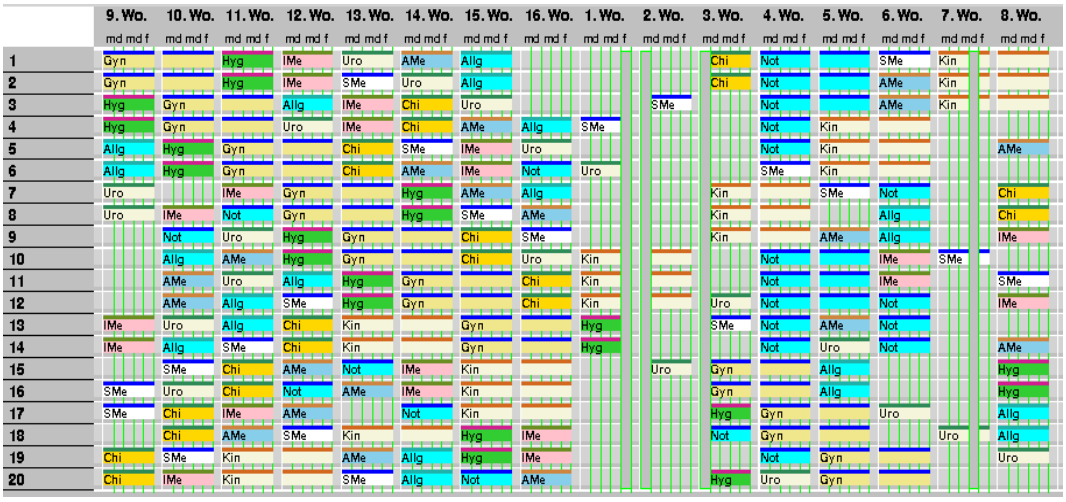


Abbildung 2: Eine Lösung der Beispiele Bsp 21, Bsp 22, Bsp 23

Wenn ein Praktikum für alle Gruppen zu absolvieren ist, könnte bei der Modellierung dieses Praktikum auch in 2 Praktika aufgeteilt und diesen beiden Praktika disjunkte Mengen von Gruppen zugeordnet werden. Die Beispiele Bsp 11 und Bsp 12 unterscheiden sich durch eine solche unterschiedliche Modellierung. Außerdem sind die zugeordneten Prioritäten der Praktika verschieden. Nur eine unterschiedliche Prioriorität eines Praktikums ist die Differenz zwischen den Beispielen Bsp 12 und Bsp 13 . Die anderen 3 Beispiele gehören zum 2. Problem, bei dem 20 Gruppen die Praktika absolvieren müssen. Bei diesem Problem wird

für zwei Praktika gefordert, daß in jeder Woche jeweils wenigstens eine Praktikumseinheit durchgeführt wird. Diese Bedingung kann der 7. Bedingung zugeordnet und entsprechend modelliert werden. Eine andere Möglichkeit ist, ein solches Praktikum in zwei Praktika zu teilen, wobei einem Praktikum 16 Gruppen und dem anderen die restlichen 4 Gruppen zugeordnet werden. Für die beiden neu entstandenen Praktika wird dann jeweils gefordert, daß keine Praktikumseinheiten parallel durchgeführt werden dürfen. Bei Bsp 21 wurden diese beiden Praktika in dieser Weise geteilt und bei Bsp 22 wurde ein Praktikum so geteilt. Diese Art der Teilung wurde bei Bsp 23 nicht verwendet. Sonst unterscheiden sich diese 3 Beispiele nicht, auch die Prioritäten der Praktika sind identisch.

Methode		Bsp 11		Bsp 12		Bsp 13		Bsp 21		Bsp 22		Bsp 23	
		#	sec	#	sec	#	sec	#	sec	#	sec	#	sec
S 1	a	–	14,5	–	15,7	–	23,9	–	11,5	692	14,5	667	12,5
	b	11	1,1	–	30,4	145	3,5	134	3,9	692	14,7	667	12,4
	c	11	1,1	–	30,9	83	3,0	24	2,1	4	1,8	184	5,9
	d	0	0,8	–	37,1	31	1,8	0	1,6	1	1,7	1	1,6
S 2	a	–	28,4	–	12,0	–	10,5	–	11,4	–	10,6	69	2,6
	b	–	28,3	–	14,5	–	12,2	33	2,5	8	1,6	69	2,6
	c	–	29,9	–	15,6	–	13,2	33	2,6	8	1,7	69	2,8
	d	0	0,8	–	34,0	14	0,9	0	1,6	27	2,9	0	1,7
S 3	a	2	0,6	186	3,3	–	27,8	–	11,5	0	1,4	–	26,2
	b	1	0,6	11	0,8	53	1,8	0	1,3	0	1,4	–	26,4
	c	1	0,6	11	0,8	44	2,0	0	1,4	0	1,5	–	54,1
	d	1	0,7	10	0,9	23	1,4	0	1,4	0	1,5	–	52,2
S 4	a	6	0,7	185	3,1	–	22,6	–	12,6	–	29,1	–	30,4
	b	5	0,7	11	0,9	136	3,0	0	1,4	–	29,1	–	30,3
	c	5	0,7	11	0,9	136	3,2	0	1,5	–	32,3	–	45,8
	d	3	0,7	10	1,0	28	1,3	0	1,5	34	2,3	–	45,8
S 5	a	–	15,2	–	15,1	–	20,2	–	12,7	–	16,2	–	13,5
	b	11	1,1	–	23,9	21	1,2	123	4,4	–	16,3	–	13,7
	c	11	1,1	–	24,4	20	1,3	44	1,9	–	30,3	–	46,6
	d	0	0,8	–	37,9	15	1,2	4	1,8	3	1,8	1	1,7
S 6	a	–	14,0	–	12,7	–	20,0	–	12,8	–	11,4	–	12,5
	b	18	1,1	–	16,0	21	1,0	132	4,0	–	11,4	–	12,2
	c	18	1,2	–	16,4	20	1,1	62	1,8	–	23,3	–	26,6
	d	0	0,8	–	35,3	14	1,0	0	1,7	1	1,8	0	1,8
S 7	a	–	24,4	–	13,1	–	11,4	–	11,8	–	12,1	69	2,9
	b	–	25,8	–	16,2	–	13,4	–	9,4	8	1,7	69	2,9
	c	–	27,5	–	17,1	–	14,4	33	2,9	8	1,9	69	3,0
	d	0	0,9	–	38,0	14	1,1	0	1,7	27	3,2	0	1,8

Tabelle 1: Ergebnisse des ersten Vergleiches

Für den ersten Vergleichstest setzen wir voraus, daß als redundante Constraints die `diffn`-Constraints und die bedingten Constraints (wie im 4. Abschnitt beschrieben) hinzugefügt werden. Bezüglich der Hinzunahme von zusätzlichen `among`-Constraints betrachten wir verschiedene Varianten:

a: *keine solchen redundanten Constraints werden erzeugt*;

b: *nur für kritische Praktika* (Praktika, bei denen in fast allen Wochen Einheiten stattfinden müssen);

c: *für alle Praktika bezüglich der Startzeitvariablen*;

d: *für alle Praktika bezüglich der Startnummervariablen.*

Für alle Strategien, die im 5. Abschnitt beschrieben wurden, alle 6 Beispiele und die 4 Varianten der Hinzunahme zusätzlicher `among`-Constraints werden in der Tabelle 1 die Ergebnisse der Lösungssuche dargestellt. Die Suche wurde abgebrochen, wenn eine Lösung nicht innerhalb von 1000 Backtracking-Schritten für Entscheidungen der Lösungssuche gefunden wurde. In den Spalten mit der Bezeichnung # ist die Anzahl der benötigten Backtracking-Schritte angegeben, falls die Suche erfolgreich war. Das Zeichen – wurde eingetragen, wenn die Suche abgebrochen wurde. Die benötigte Zeit für die Lösungssuche (auf einer SUN-*Ultra-Sparc 1*) sind in Sekunden angegeben.

Methode		Bsp 11		Bsp 12		Bsp 13		Bsp 21		Bsp 22		Bsp 23	
		#	sec	#	sec	#	sec	#	sec	#	sec	#	sec
S 1	d	0	0,8	–	37,1	31	1,8	0	1,6	1	1,7	1	1,6
	e	1	0,6	–	28,9	39	1,6	0	1,3	2	1,4	2	1,4
	f	482	9,6	–	23,4	814	10,3	0	0,9	1	1,0	1	1,0
	g	212	2,6	–	13,8	328	2,6	0	0,6	2	0,6	2	0,6
S 2	d	0	0,8	–	34,0	14	0,9	0	1,6	27	2,9	0	1,7
	e	0	0,7	–	28,9	14	0,8	0	1,4	27	2,6	0	1,4
	f	259	4,4	–	9,6	242	3,0	0	0,8	27	1,5	0	0,8
	g	259	3,3	–	8,0	246	2,2	0	0,7	27	1,1	0	0,6
S 3	d	1	0,7	10	0,9	23	1,4	0	1,4	0	1,5	–	52,2
	e	25	0,8	34	1,1	36	1,4	–	20,7	–	17,0	–	20,7
	f	1	0,3	–	13,6	84	1,3	0	0,7	0	0,7	–	25,8
	g	25	0,4	–	7,4	84	0,9	–	9,6	–	8,7	–	10,1
S 4	d	3	0,7	10	1,0	28	1,3	0	1,5	34	2,3	–	45,8
	e	–	10,6	34	1,2	28	1,1	–	12,4	–	14,5	–	27,4
	f	3	0,4	–	6,0	92	1,3	0	0,8	34	1,1	–	21,8
	g	–	7,3	–	4,5	96	0,8	–	8,1	–	8,9	–	11,6
S 5	d	0	0,8	–	37,9	15	1,2	4	1,8	3	1,8	1	1,7
	e	1	0,7	–	29,3	16	1,1	1	1,4	2	1,5	1	1,4
	f	452	9,7	–	24,2	559	8,3	4	1,0	3	1,0	1	1,0
	g	212	2,7	–	14,4	304	2,3	1	0,6	2	0,6	1	0,7
S 6	d	0	0,8	–	35,3	14	1,0	0	1,7	1	1,8	0	1,8
	e	0	0,7	–	28,9	14	0,9	0	1,4	1	1,4	0	1,4
	f	–	11,7	–	15,9	272	3,5	0	0,9	1	1,1	0	1,0
	g	–	7,0	–	9,6	274	1,9	0	0,6	1	0,7	0	0,6
S 7	d	0	0,9	–	38,0	14	1,1	0	1,7	27	3,2	0	1,8
	e	0	0,7	–	29,2	14	0,8	0	1,4	27	2,5	0	1,4
	f	259	5,4	–	12,3	236	3,6	1	1,0	27	1,8	0	0,9
	g	259	3,3	–	8,1	246	2,1	0	0,6	27	1,1	0	0,6

Tabelle 2: Ergebnisse des zweiten Vergleiches

Aus der Tabelle 1 ist sofort ersichtlich, daß die Variante "d " der Hinzunahme zusätzlicher `among`-Constraints die beste ist. Bei dem zweiten Vergleich wird deshalb diese Variante vorausgesetzt und es werden Varianten der Hinzunahme der anderen redundanten Constraints betrachtet, wobei die Variante "d" in beiden Tabellen identisch ist:

d: *mit* diffn-Constraints und *mit* den bedingten Constraints;

e: *mit* diffn-Constraints und *ohne* die bedingten Constraints;

f: *ohne* diffn-Constraints und *mit* den bedingten Constraints;

g: *ohne* diffn-Constraints und *ohne* die bedingten Constraints.

Aus diesen beiden Tabellen können unter anderem die folgenden Schlußfolgerungen gezogen werden:

- Unter den Suchstrategien S 1 – S 7 gibt es keine, welche für alle Beispiele durchgängig die beste wäre.
- Einige Suchstrategien zeigen unter einander ein relativ ähnliches Verhalten. Mit Abstrichen können diese Suchstratgien bezüglich ihrer Ergebnisse in zwei Gruppen eingeteilt werden: { S 1, S 2, S 5, S 6, S 7 } und { S 3, S 4 }.
- Obwohl sich durch Hinzunahme redundanter Constraints die Zeit der Lösungssuche verlängern kann, ist dies im allgemeinen vorteilhaft.
- Zusätzliche `among`-Constraints sollten für alle Praktika bezüglich der Startnummervariablen erzeugt werden.
- Die Hinzunahme redundanter Constraints verringert nicht in jedem Fall den Suchraum.
- Obwohl relativ viele redundante bedingte Constraints zu erzeugen sind (1708 für die ersten 3 Beispiele und 3268 für die anderen Beispiele), ist die für die Lösungssuche zusätzlich benötigte Zeit verhältnismäßig gering.
- Bei Vergleich der Suchergebnisse der Beispiele Bsp 12 und Bsp 13 wird deutlich, welche Auswirkungen eine kleine Änderung der Reihenfolge der Domänenvariablen bewirken kann.

7 Zusammenfassung und Ausblick

Die Ergebnisse der Lösungssuche zeigen, daß unsere grundsätzliche Vorgehensweise bei der Lösungssuche richtig ist (siehe auch [6]): die Anzahl der erlaubten Backtracking-Schritte für Entscheidungen der Lösungssuche wird stark eingeschränkt und es werden verschiedene Heuristiken ausprobiert. Somit wird ein Backtracking über verschiedene Heuristiken durchgeführt. In unserer Standardeinstellung ist die Anzahl der erlaubten Backtracking-Schritten für Entscheidungen der Lösungssuche auf 100 begrenzt. Aus den dargestellten Ergebnissen ergeben sich für die Heuristiken, die ausprobiert werden sollten, folgende Schlußforgerungen:

- Alle redundanten Constraints sollten hinzugefügt werden (`among`-Constraints bzgl. Startnummervariablen); evtl. kann im ersten Versuch auf das Erzeugen der zusätzlichen `diffn`-Constraints und bedingten Constraints verzichtet werden.

- Aus den beiden Gruppen "ähnlicher" Suchstrategien sollte jeweils eine probiert werden (beispielweise S 3 und S 6).
- Verschiedene Heuristiken der Variablenauswahl sind zu verwenden.

In weiteren Untersuchungen ist festzustellen, ob sich unsere Schlußfolgerungen auch auf anderen Probleme übertragen lassen. Außerdem sind weitere Untersuchungen zu Heuristiken der günstigsten Variablenauswahl durchzuführen. Insbesondere ist das Erkennen von Engpässen wichtig. Verschiedene Methoden und Heuristiken sind auch für die anderen Teilprobleme der Stundenplanung zu untersuchen. Neben der vollständigen automatischen Planerzeugung ist für die praktische erfolgreiche Anwendung des Stundenplansystems die interaktive Beeinflussung der Lösungssuche von entscheidender Bedeutung. Beispielsweise ist es möglich einzelne Lehrveranstaltungen einzuplanen oder umzuplanen. Bei der interaktiven Beeinflussung der Lösungssuche erfolgt kein "Backtracking" über Entscheidungen des menschlichen Planers.

Literatur

[1] P. Boizumault, Y. Delon, and L. Peridy. Constraint logic programming for examination timetabling. *J. Logic Programming*, 26(2):217–233, 1996.

[2] E. Burke and M. Carter, editors. *Practice and Theory of Automated Timetabling II*, volume 1408 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1998.

[3] E. Burke and P. Ross, editors. *Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1996.

[4] T. B. Cooper and J. H. Kingston. The complexity of timetable construction problems. In *[3]*, pages 183–295, 1996.

[5] H.-J. Goltz. Reducing domains for search in CLP(FD) and its application to job-shop scheduling. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming – CP'95*, volume 976 of *Lecture Notes in Computer Science*, pages 549–562, Berlin, Heidelberg, 1995. Springer-Verlag.

[6] H.-J. Goltz and D. Matzke. University timetabling using constraint logic programming. In G. Gupta, editor, *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 320–334, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

[7] C. Guéret, N. Jussien, P. Boizumault, and C. Prins. Building university timetables using constraint logic programming. In *[3]*, pages 130–145, 1996.

[8] M. Henz and J. Würtz. Using Oz for college timetabling. In *[3]*, pages 162–177, 1996.

[9] G. Lajos. Complete university modular timetabling using constraint logic programming. In *[3]*, pages 146–161, 1996.

[10] A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567, Centrum voor Wiskunde en Informatica, 1995.

[11] G.M. White and J. Zhang. Generating complete university timetables by combining tabu search with constraint logic. In *[2]*, pages 187–198, 1998.

Implementierung von built-in Constraints für endliche Wertebereiche in Minerva

Georg Ringwelski, Armin Wolf, Ulrich Geske
gr|armin|geske@first.gmd.de
GMD FIRST, Berlin-Adlershof

Kurzfassung

Die logische Programmiersprache Minerva stellt durch ihre Java-Basis die Beziehung zwischen logisch-deklarativen Spezifikationen und internet-basierten Anwendungen her. Minerva erbt dabei von Java den plattformunabhängigen Lösungsansatz. Hier wird die Erweiterung von Minerva um einen Basis-Constraint-Solver beschrieben, durch den kombinatorisch komplexe Probleme nicht nur deklarativer beschrieben, sondern auch verteilt bearbeitet werden können. Minerva wurde dazu um arithmetische Constraints über endlichen Wertebereichen erweitert. Dabei wurde das in `clp(FD)` verwendete und bewährte Konzept der Constraintverarbeitung übernommen. Die Implementierung der low-level Constraints `X in r` erfolgte jedoch nicht in der Zielsprache Java, sondern auf Prolog-Ebene. Es wird lediglich eine eng begrenzte Schnittstelle (attributierte Variablen, vordefinierte Prädikate zur Behandlung von Constraintvariablen) vorausgesetzt und nicht in das System Minerva eingegriffen. Dadurch ist die Unabhängigkeit von Reimplementierungen von Minerva gewährleistet. Außerdem kann das Constraintsystem in anderen logischen Sprachen wiederverwertet werden, wenn dort eine entsprechende Schnittstelle implementiert wird.

1 Motivation

Constraintlogische Programmierung hat in der letzten Zeit weite Verbreitung gefunden, da sie sehr geeignet ist, kombinatorisch komplexe Probleme zu lösen (vgl. [13] oder [10]). Analoges gilt für die Programmiersprache Java, die sich als die Sprache des Internet[1] immer mehr zu etablieren scheint (vgl. [9]). Es liegt nun nahe, diese beiden Konzepte zu kombinieren, um z.B. typische CLP-Problemlösungen im Internet als Dienstleistung zur Verfügung zu stellen. Bisher mußte dazu ein CLP-Programm von einem Server ausgeführt werden, der Ergebnisse an den Internet-Client sendete. Zur Unterstützung von Constraints in Java, sind bisher nur die Implementierungen JCL und JSolver bekannt. Die *Java Constraint Library* (JCL) [2] unterstützt Constraints nur rudimentär. Es sind lediglich Netze binärer Constraints möglich, die Wertebereiche der Variablen sind durch Enumeration festzulegen und die zulässigen Wertepaare eines jeden binären Constraints sind ebenfalls explizit aufzuzählen. Die Formulierung arithmetischer Constraints, z.B. von Gleichungs- und Ungleichungssystemen über ganzen Zahlen wird durch JCL nicht unterstützt. Der JSolver aus [4] liegt bisher in einer β-Version vor und ist eine Java-Klasse, die arithmetische und boolsche Constraints unterstützt. JSolver ist also eine Erweiterung der objektorientierten Sprache Java und keine logisch-deklarative Sprache. Es schien es uns also sinnvoll, die logische Sprache Minerva um einen Constraintlöser

[1] Es werden z.B. eine gewisse Sicherheit für die Benutzer gewährleistet

für endliche Wertebereiche, im Sinne von `clp(FD)` aus [5] zu erweitern. Diese Erweiterung stellt einige built-in Prädikate bzw. Constraints zur Verfügung, die die Verarbeitung von FD-Constraintvariablen erlauben. Die Ausführbarkeit existierender Minerva-Programme für das Internet wird durch diese Erweiterung nicht berührt, da die Implementierung des Constraint-Systems lediglich über sehr wenige Schnittstellen mit dem Kern von Minerva kommuniziert und als logisches Modul an das Gesamtsystem angebunden wird.

2 Minerva

2.1 Allgemeines

Dieser Abschnitt soll, aufbauend auf der Minerva-Beschreibung [12], dazu dienen, die Eigenschaften der Sprache und die damit verbundene Motivation, einen Constraintlöser für diese Sprache zu entwickeln, verdeutlichen.
Minerva ist eine logische Programmiersprache, die zusätzlich zum Prolog-Standard weitere Funktionalität zur Verfügung stellt. Insbesondere steht eine Bibliothek zur Programmierung von GUIs[2] und die Möglichkeit der direkten Kommunikation mit Java-Programmen zur Verfügung. Mit Minerva kann man also auf deklarative Weise (logische Programmierung) direkt Anwendungen für das Internet, wie z.B. *applets* implementieren. Minerva-Programme werden ähnlich wie Java-Programme in einen speziellen Code kompiliert, der z.B. von HTML aus direkt ausgeführt werden kann. Damit sind die Sicherheits- und Kompatibilitätseigenschaften von Java genauso in Minerva vorhanden. Um Minerva als Internet-Client zu benutzen, braucht man lediglich einen Browser (z.B. Netscape4.x), der Java unterstützt. Dies ist z.B. in Oz/Mozart aus [6] nicht direkt möglich, dort muß auf der Client-Seite ein passender Interpreter installiert sein. Laut IF-Computer ist Minerva geeignet zur Implementierung von großen, interaktiven, internetbasierten, dynamischen Systemen und ist bzgl. der Rechengeschwindigkeit so gut wie das unterliegende Java. Minerva und damit auch die Implementierung des Constraintsystems ist also immer dann geeignet, wenn die besonderen Eigenschaften von Java (Sicherheit ...) genutzt werden sollen.

2.2 Die Constraint-Schnittstelle in Minerva

In Minerva stehen ab Version 2.0 Prädikate zur Verfügung, mit denen sich Variablen mit beliebigen Termen (außer Variablen) attributieren lassen. Da Terme Constraints repräsentieren können, eignen sich derartige attributierte Variablen zur Repräsentation von Constraint-Variablen (CV). Attributierte Variablen sind vergleichbar ([11]) mit Metastrukturen aus [14] und stellen eine Erweiterung von Prolog dar. Diese Erweiterung besteht im Binden von Information an Variablen und erfordert eine Beschreibung, wie solche Variablen unifiziert werden können.
Zur Erzeugung solcher attributierter Variablen stellt Minerva[3] das Prädikat `co_add/2` zur Verfügung. Ein Aufruf von `co_add(Var,Constr)`, wobei `Constr` ein Term, aber keine Variable ist, wird dabei unterschiedlich verarbeitet:

1. Ist `Var` eine Variable, so wird eine CV `Var` mit dem Constraint `Constr` erzeugt und der Aufruf ist erfolgreich.

[2] graphische Benutzeroberflächen
[3] Zur Erläuterung der Schnittstelle von Minerva siehe auch Abbildung 4.

2. Ist `Var` eine CV, so wird `Constr` mit dem Constraint, das `Var` bereits beschreibt (`OldConstr`) 'gemischt'. Dieses Mischen ist natürlich abhängig von der Art des Constraints. Es wird daher das zu definierende[4] Prädikat `co_merge/2` mit den Argumenten `Var` und `[Constr,OldConstr]` von Minerva aufgerufen. In diesem Fall ist der Aufruf also genau dann erfolgreich, wenn `co_merge(Var,[Constr,OldConstr])` erfolgreich ist.

3. Ist `Var` keine Variable, so wird das benutzerdefinierte Prädikat `co_check/2` aufgerufen. Das heißt, es wird überprüft, ob `Var`, was in diesem Fall ein Term ist, `Constr` erfüllt. Was es bedeutet, daß ein Term ein Constraint erfüllt, muß ebenfalls vom Benutzer angegeben werden, d.h. `co_check/2` muß ebenfalls definiert werden. Der Aufruf ist also erfolgreich, wenn `co_check(Var,Constr)` erfolgreich ist.

Auf attributierte Variablen kann mit `co_get/2` zugegriffen werden. Ein Aufruf von `co_get(Var, Constr)` ist erfolgreich, wenn `Var` eine CV ist und `Constr` mit dem Term, mit dem `Var` attributiert wurde unifizierbar ist. Andernfalls scheitert `co_get/2`.
Die Verwaltung der CVn wird intern von Minerva durchgeführt.[5] Der Benutzer muß allerdings angeben, wie die Unifikation von zwei CVn miteinander durchgeführt werden soll. Dazu muß er das bereits erwähnte Prädikat `co_merge/2` zur Verfügung stellen, das außerdem verwendet wird, um mehrere Constraints zu einer Variablen zu verarbeiten. Wird eine CV `X` mit einer Variablen `Y` unifiziert `X = Y`, so wird `Y` eine CV mit den gleichen Constraints wie `X`. `Y` ist dann aber abhängig von `X`, d.h. werden die zu `X` gehörige Constraints modifiziert, so verändert sich `Y` in gleicher Weise. Werden zwei CVn unifiziert, so werden ihre Constraints 'gemischt' und das Ergebnis wird in beiden CVn berücksichtigt.[6] Diese kann dann durch `co_add` als CV in den internen Constraint-Speicher eingefügt werden, wobei die Abhängigkeit der CVn erhalten bleibt. Scheitert das 'Mischen' der Constraints, so scheitert auch die Unifikation. Durch die Definition des 'Mischens' wird eine Constrainttheorie über der Signatur der Terme, die sich aus den Attributen ergeben, definiert. Sind die Constraints durch Wertemengen der Variablen gegeben, so handelt es sich beim 'Mischen' um den mengentheoretischen Schnitt dieser Wertemengen.
Das benutzerdefinierte Prädikat `co_check/2` wird immer aufgerufen, wenn eine CV mit einem Wert instantiiert wurde. Es darf nur dann erfolgreich sein, wenn dieser Wert das zur Variable gehörige Constraint erfüllt. Bei der Verarbeitung von Constraints kann also durch dieses Prädikat genau dann Backtracking eingeleitet werden, wenn die Variable instantiiert wurde und die Instantiierung laut `co_check/2` nicht erlaubt ist.
Die gewählte Implementierung von `co_merge/2` und `co_check/2` zur Realisierung von arithmetischen Constraints über endlichen Wertebereichen ist in Abschnitt 3.3 beschrieben.

3 Finite-domain Constraints

3.1 Das 'X in r' -Constraint

Das infix-Prädikat `in/2` definiert Domänen von Variablen. Das heißt, das erste Argument ist eine Variable oder eine CV und das zweite Argument eine Repräsentation der Domäne

[4] Das Prädikat ist in Minerva zwar enthalten, es ist aber nicht durch Regeln semantisch definiert. Die operationale Iterpretation des Prädikats ist natürlich von den implementierten Constraints abhängig und muß daher in jeden Constraintsystem implementiert werden.

[5] IF-Computer empfiehlt auch keine anderen Schnittstellen zur Constraintverarbeitung zu konstruieren.

[6] Die konkrete Realisierung ist Bestandteil des Minerva-Systems.

dieser Variable, die nach dem Aufruf in jedem Fall eine CV ist. Das Prädikat beschreibt also ein Constraint, eine Beschränkung, der Eigenschaften der Variable `X`. Diese besteht in der Beschränkung der möglichen Werte, mit denen die Variable fortan instantiiert werden kann. Die Domäne `r` kann durch Terme zur Signatur `RANGE` aus Abbildung 1 über der leeren Variablenmenge definiert werden[7]. Da CVn nicht nur Platzhalter sind, sondern selbst Information tragen, werden sie selbst als abstrakter Datentyp implementiert (siehe Abbildung 2). Logische Variablen können in der Definition der Domäne nicht verwendet werden[8].

```
RANGE = INT{initial} + CV +
        sorts: range
        opns : .. : int int -> range
```

Abbildung 1: Signatur von `RANGE`, Syntax zur Definition von Domänen. `INT` ist die Signatur der ganzen Zahlen, die semantisch initial interpretiert werden muß.

CVn werden durch ihren Namen und durch die zu ihnen gehörige Domäne beschrieben. Sie werden aus Gründen der Effizienz unterschiedlich repräsentiert. Bei der Erzeugung einer CV durch das Prädikat `in/2` ist die zur CV gehörige Domäne immer ein Intervall ganzer Zahlen. Da die Domänen immer wieder verändert werden können, kann es vorkommen, daß "Lücken" in den Intervallen entstehen, dann wird die Domäne durch eine Liste repräsentiert (`vector` liefert diese Liste). Kann eine Domäne schließlich so eingeschränkt werden, daß sie nur noch genau einen Wert zuläßt, so wird wird ihre Repräsentation ebenfalls umgewandelt (`value` liefert diesen Wert).

```
CV = INT{initial} + LIST<INT> +
     sorts : cv, kind
     opns : min : cv -> int
            max : cv -> int
            len : cv -> int
            which : cv -> kind
            vector : cv -> list<int>
            value : cv -> int
```

Abbildung 2: Signatur des Datentyps `CV`, Syntax einer Constraintvariablen. Verwendet werden ganze Zahlen und der Datentyp List, der Listen von ganzen Zahlen beschreibt.

Da die Datentypen `RANGE` und `CV` nur durch Signaturen, ohne Gleichungen, spezifiziert wurden, kann eine initiale oder denotationale Semantik nicht direkt aus der Syntax abgeleitet[9] werden. Eine solche intendierte Semantik geben wir in Abbildung 3 explizit an, wie es z.B. auch in [5] gemacht wird. Dabei wird nur der "positive Kern" der Operationen beschrieben, es sind syntaktische Konstruktionen möglich, deren Semantik mit der Interpretation aus Abbildung 3 nicht definiert werden kann (z.B. rekursive Definitionen von Domänen). Solche

[7] Zur Erläuterung der Begriffe Signatur und Term, sowie zur Beschreibung der verwendeten Darstellung verweisen wir auf [8].

[8] Dies gilt nur für logische Variablen, die beim Aufruf des Prädikats `X in r` noch nicht instantiiert sind.

[9] Eine Interpretation durch partielle Algebren mit Freier-Funktor-Semantik wie in [18] kann konstruiert werden.

syntaktisch möglichen aber semantisch nicht intendierten Fälle werden durch die implizite Partialität der Interpretationen der Operationen ausgeschlossen, d.h. Terme, die semantisch nicht durch Abbildung 3 interpretiert werden können, haben keine semantische Relevanz. Dies entspricht der Interpretation der Signaturen durch partielle Algebren wie in [15]. Ein expliziter Vergleich mit der operationalen Semantik zur Verifikation der Korrektheit des Systems kann später aus der logischen Interpretation des Systems im "positiven Kern" abgeleitet werden. In den anderen Fällen wird in unserem System die Partialität syntaktisch dadurch abgefangen, daß bestimmte Aufrufe des Prädikats `in/2` nicht durchgeführt werden (vgl. Abschnitt 3.2).

Symbol	Semantik: eine partielle `RANGE`-Algebra A
`INT`	$(\mathbb{Z}, 0, succ, +, -, *)$ Die Algebra der ganzen Zahlen
`range`	$A(\texttt{range}) = \{[A,B] \mid A,B \in \mathbb{Z}\}$, Intervalle ganzer Zahlen[10]
`..`	definiert ein Intervall mit den angegebenen Grenzen: $A(..)((X,Y)) = [X,Y]$ (auch als Infix-Operation)
`LIST<INT>`	Die Algebra der Listen ganzer Zahlen, mit den entsprechenden Operationen: $(P(\mathbb{Z}), concat, ...)$
`cv`	Constraintvariablen, $A(cv) = CV$
`kind`	die Art der Domäne, die Menge $A(\texttt{kind}) = \{inter, vec, val\}$
`which`	ordnet jeder CV ihre Art zu : $A(\texttt{which}) : CV \to \{inter, vec, val\}$ Dabei markieren die Werte folgende Interpretation der Domäne einer CV `X` durch eine Menge: *inter* : Intervalldarstellung: $[A(\texttt{min})(\texttt{X}), A(\texttt{max})(\texttt{X})]$ *vec* : Listendarstellung : $A(\texttt{vector})(\texttt{X})$ *val* : Einzelner Wert : $A(\texttt{value})(\texttt{X})$.
`vector`	falls `which vec` liefert ist `vector` eine Liste ganzer Zahlen, die Werte der Domäne der CV: $A(\texttt{vector}) : \begin{cases} CV \to P(\mathbb{Z}) \text{ falls } A(\texttt{which})(CV) = vec, \\ \text{undefiniert, sonst.} \end{cases}$
`value`	falls `which val` liefert ist `value` der einzige Wert der Domäne der CV: $A(\texttt{value}) : \begin{cases} CV \to \mathbb{Z} \text{ falls } A(\texttt{which})(CV) = val, \\ \text{undefiniert , sonst.} \end{cases}$
`max, min`	in jedem Fall (von `which`) ist `min` der kleinste Wert und `max` der größte Wert der Domäne der CV: $A(\texttt{min}), A(\texttt{max}) : CV \to \mathbb{Z}$
`len`	liefert die Anzahl der Elemente der Domäne der CV: $A(len) : CV \to \mathbb{N}$

Abbildung 3: Intendierte (denotationale) Semantik der Datentypen `RANGE` und `CV`

Das Prädikat `in/2` definiert die Menge aller Constraintvariablen, die definiert werden können. D.h. die angegebene Domäne ist eine Menge von ganzen Zahlen und ist unabhängig von der zu definierenden CV definiert (also nicht rekursiv). Die operationale Semantik von `X in r` besteht aber auch in einer Änderung des Systemzustands, so daß der interne Constraintspeicher das Constraint `X in r` enthält, welches jederzeit semantisch durch $Sem(\texttt{r})\alpha$ interpretiert werden kann. Dabei ordnet $Sem : T(\texttt{RANGE}, \emptyset) \to \alpha \to P(\mathbb{Z})$ dem Intervall $\texttt{r} \in T(\texttt{RANGE}, \emptyset)$ bei einem aktuellen Constraintspeicher α eine Menge von ganzen Zahlen ($\mathbb{Z}$)

zu, wenn diese Menge berechenbar ist.[11]

Definition 1 (Semantik von Wertebereichen) Die Semantik eines Wertebereichs `r` $\in T(\texttt{RANGE}, \emptyset)$ ist gegeben durch einen Constraintspeicher α und eine `RANGE`-Algebra A:
$Sem(X..Y)\alpha = A(..)(eval_\alpha(X), eval_\alpha(Y))$ mit

$$eval_\alpha(X) = \begin{cases} X, \text{ falls } X \in \mathbb{Z}, \\ A(op)(eval_\alpha(Y), eval_\alpha(Z)), \text{ falls } X = op(Y,Z), op \in \{+,-,*\}, \\ A(op)(lookup_\alpha(X)), \text{ falls } X \in CV, op \in \{\texttt{min}, \texttt{max}, \texttt{value}, \texttt{vector}, \texttt{len}\}. \end{cases}$$

Dabei liefert $lookup_\alpha$ die aktuelle Domäne einer CV aus dem Constraintspeicher α.

Der Constraintspeicher muß also immer als Umgebung bei der Berechnung einer Domäne einbezogen werden. Außerdem wird durch `in/2` der Constraintspeicher um das neue Constraint erweitert. Die operationale Semantik von `X in r` ergibt sich also durch eine Veränderung des Constraintspeichers:

Definition 2 (Semantik von `in`) Die operationale Semantik des Prädikats
`in` $= \{(X,r) | Sem(\texttt{r})\alpha \subset \mathbb{Z} \wedge vars(r) \neq X\}$ ist eine Änderung des Constraintspeichers α:
$call(in) : CV \times T(\texttt{RANGE}, \emptyset) \rightarrow CV \times P(\mathbb{Z}) \rightarrow \{true, false\}$, mit
$(X,r) \mapsto (X, Sem(\texttt{r})\alpha) \mapsto Sem^*(\alpha \cup \{\texttt{X in } Sem(\texttt{r})\alpha, \texttt{X in r}\})$.
Wobei *call* den Aufruf von *in* (z.B. als Ziel in Prolog) beschreibt.
Ist $Sem^*(\alpha \cup \{\texttt{X in } Sem(\texttt{r})\alpha, \texttt{X in r}\})$ erfüllbar, so ist die deklarative Semantik des Aufrufs *'true'*, ansonsten *'false'*.

Definition 3 (Semantik des Constraintspeichers) Die semantische Interpretation des Constraintspeichers ist die Konjunktion der semantischen Interpretation aller enthaltenen Constraints:
$Sem^*(\alpha) = Sem^*([X_1 \text{ in } r_1, ..., X_n \text{ in } r_n]) = X_1 \in Sem(r_1)\alpha \wedge ... \wedge X_n \in Sem(r_n)\alpha$.

Beispiel 1 *Als Beispiel wird ein Aufruf des Prädikats* `in/2` *semantisch interpretiert. Sei der Constraintspeicher* $\alpha =$ `[X in 3..8, Y in 1 .. max(X)]` *gegeben. Dieser wird nach Definition 3 durch die Konjunktion* $X \in [3,8] \wedge Y \in [1,8]$ *interpretiert. Ein Aufruf von* `'X in min(Y) .. 6'` *hat dann folgenden Semantik:*
Zunächst wird die Semantik der Domäne berechnet:
$Sem(\texttt{min(Y)..6})\alpha = A(..)(A(min)(lookup_\alpha(Y), 6)) = [1,6]$. *Damit kann entschieden werden, daß das Constraint gültig ist (d.h. endlich, interpretierbar und nicht rekursiv). Also hat der Aufruf* $call(in)$ *eine operationale Semantik im Sinne von Definition 2, die berechnet wird:* $(\texttt{X}, \texttt{min(Y)..6}) \mapsto (\texttt{X}, [1,6]) \mapsto Sem^*(\alpha \cup \{\texttt{X in 1..6, X in min(Y) .. 6}\})$. *Damit hat sich die Semantik des Constrainspeichers verändert:*
$Sem^*(\alpha \cup \{\texttt{X in 1..6, X in min(Y) .. 6}\}) = Sem^*(\texttt{[X in 3..8, Y in 1 .. max(X),X in 1..6,X in min(Y) .. 6]}) = X \in [3,6] \wedge Y \in [1,6]$. *Diese Konjunktion ist erfüllbar, somit ist die deklarative Semantik von* `'X in min(Y) .. 6'` *in* α *'true'. Der Aufruf war also erfolgreich und hat den Constraintspeicher in gewünschter Weise verändert.*

[11] $P(M)$ steht für die Potenzmenge der Menge M. $T(\texttt{RANGE}, \emptyset)$ beschreibt die Menge der Terme über der Signatur `RANGE` mit der leeren Variablenmenge (vgl. [8])

3.2 High-level Constraints

Wie bereits angesprochen wurde, sind (syntaktische) Aufrufe des `in/2`- Prädikats möglich, die keine semantische Relevanz haben und auch nicht haben sollen.[12] Daher werden Prädikate definiert, die sich an den Anforderungen von Constraint-Programmierern orientieren, diese high-level Prädikate werden dann auf Aufrufe des low-level Prädikats `in/2` zurückgeführt. Jedes high-level Prädikat kann die Domäne aller in seinen Argumenten vorkommenden CVn aus der jeweiligen Implementierung neu definieren. Zusammen mit der semantischen Interpretation des Constraintspeichers aus Definition 3 kann sich dann z.B. eine Einschränkung[13] der Domäne einer CV ergeben.

Solche high-level Prädikate werden durch Prolog-Klauseln syntaktisch definiert und durch Implikationen semantisch interpretiert. Dabei ist aber darauf zu achten, daß es bei diesen Prädikaten nur im negativen Fall, wenn also Constraints nicht erfüllbar sind, ausschließlich um das Beweisen von Klauseln geht. Im positiven Fall, wenn also alle Domänen so eingeschränkt werden können, daß sich eine Lösung des durch high-level Constraints beschriebenen Problems ergibt, ist der Effekt dieser Klauseln, außer der Beweisbarkeit, eine Veränderung des aktuellen Constraintspeichers (Seiteneffekt). Durch Änderungen von Constraints zu bestimmten CVn ergeben sich auch Konsequenzen für die Domänen anderer CVn. Ist eine CV durch bestimmte Werte (z.B. `Y in min(X) .. 5`) einer anderen CV definiert, so hat eine Änderung von z.B. `X` auch Auswirkungen auf die Domäne von `Y`. Diese Abhängigkeiten werden aus Effizienzgründen ausschließlich durch Aufrufe von low-level Constraints erfüllt. D.h. nach einer Änderung von z.B. `min(X)` wird das Constraint `Y min(X) .. 5` nochmals aufgerufen, wodurch sich wegen der Interpretation des Speichers durch eine Konjunktion eine Einschränkung der Domäne von `Y` ergeben kann.

Beispiel 2 *Sei der Constraintspeicher* `[X in 5.. 10, Y in 3 .. max(X)]` *gegeben, dieser kann semantisch durch* $X \in [5,10] \wedge Y \in [3,10]$ *interpretiert werden. Sei weiter ein high-level Constraint*

```
'x=y+c'(A,B,C):-
        A in (min(B) + C) .. (max(B) + C), B in (min(A) - C) .. (max(A) - C).
```

gegeben, dann hat der Aufruf des Ziels `'x=y+c'(X,Y,5)` *eine Erweiterung des Constraintspeichers zur Folge:*
`[X in 5 .. 10, Y in 3 .. max(X),X in (min(Y) + 5) .. (max(Y) + 5),`
`Y in (min(X) - 5) .. (max(X) - 5)]`. *Damit ergibt sich eine Änderung der Semantik des Speichers als* $X \in [5,10] \wedge Y \in [3,10] \wedge X \in [8,15] \wedge Y \in [0,5] \equiv X \in [5,8] \wedge Y \in [3,5]$

3.3 Implementierung

Das vorgestellte System ist ein *finite-domain* Constraintsystem, das in der Sprache Minerva für die Benutzung in Minerva entwickelt wird. Es ist in Anlehnung an `clp(FD)` aus [5] entworfen, wobei jetzt, wie beschrieben, eine Grundversion mit den low-level-Prädikaten `in/2` und `remove/2` und einigen high-level-Prädiakten (vgl. Abbildung 5), zur Verfügung steht. Diese

[12] Ein Ausdruck `X in min(X) .. max(X)` scheint trivial, ist aber nicht interpretierbar, da `X` und `min(X)..max(X)` laut Definition 2 nicht in der Relation `in` stehen.

[13] Erweiterungen der Domäne sind nicht möglich, was durch die Monotonieeigenschaft des Schnitts von Mengen (vgl. Def. 3) gesichert ist.

Grundversion ist als Prototyp, der das Konzept der Modularität und den Einsatz in verteilten Systemen erlaubt konstruiert und schon wegen der Verwendung von Java als Zielsprache in der Laufzeit nich mit anderen CLP-Systemen zu vergleichen. Im Unterschied zu `clp(FD)` ist unser System nicht 'hart' kodiert, sondern bisher ausschließlich in der Hochsprache Minerva implementiert. Bis auf die verwendeten Prädikate zur Verwendung von attributierten Variablen (`co_check/2 ...`) entspricht die Implementierung daher dem ISO-13211-1 Prolog-Standard. Die Implementierung stellt also ein logisches Modul im Sinne von [1] dar, das eine klar definierte Import- und Export-Schnittstelle zur Verfügung stellt. Jeder Prolog-Dialekt der, um die Prädikate zur Verarbeitung logischer Variablen erweitert wird, stellt die notwendige Schnittstelle zur Verfügung, so daß das Modul zur Constraintverarbeitung angebunden werden kann. Ein weiterer Vorteil, der sich aus der Implementierung in Minerva ergibt, ist die Möglichkeit, auf logische Variablen in einer beliebigen Programmumgebung zugreifen zu können. Bei der Verwendung des `fd`-Pakets können Programme geschrieben werden, die sowohl logische als auch Constraint-Variablen verwenden, da lediglich die exportierten Prädikate zusätzlich zur Verfügung stehen. Das System besteht, wie es in Abbildung 4 dargestellt ist, aus drei Minerva-Paketen; das Paket `fd_constraint` implementiert die Schnittstelle zu Minerva, d.h. es werden die Prädikate `co_merge/2` und `co_check/2` zur Verfügung gestellt und nur von diesem Paket aus werden Aufrufe der Prädikate `co_add/2` und `co_get/2` durchgeführt. Außerdem wird das Constraint `'X in r'`, das in Abschnitt 3.1 beschrieben wurde, implementiert. Als Export-Schnittstelle dienen die nach außen sichtbaren[14] Prädiakte des Pakets `fd_predicates` bzw. in der zu entwickelnden Version die des Pakets `fd`.

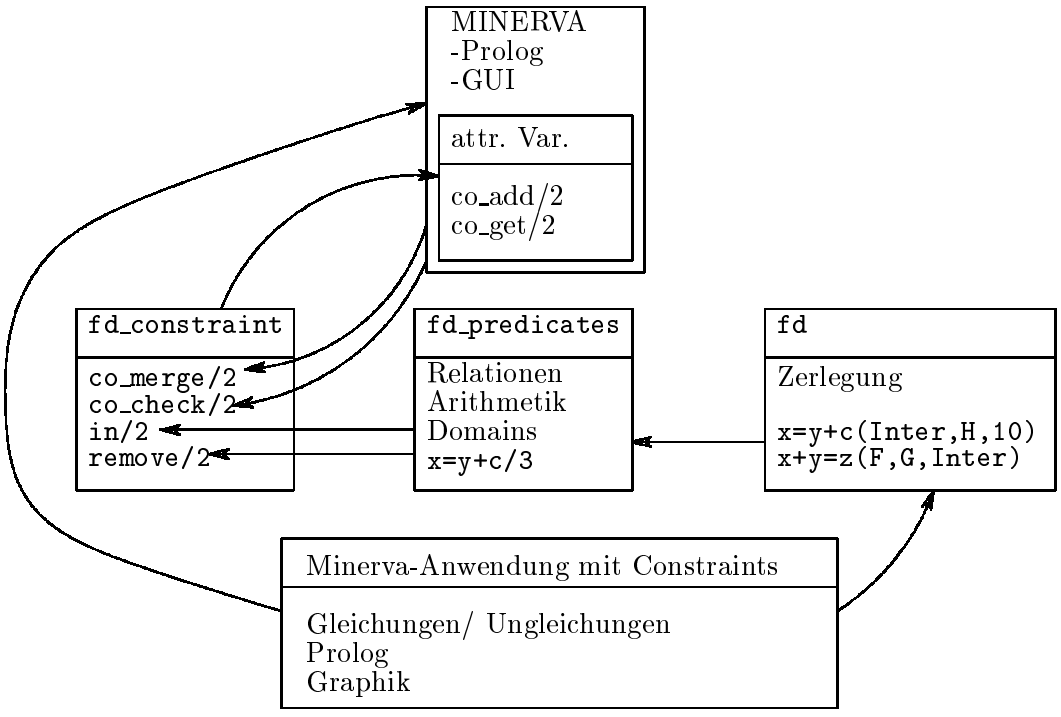


Abbildung 4: Struktur der Implementierung des FD-Constraintsystems

[14] In Minerva ist *information hiding* möglich

Das `fd_constraint`-Paket

Dieses Minerva-Paket enthält die Implementierung des low-level-Constraints `X in r`, so daß seine operationale Semantik der in Abschnitt 3.1 definierten Semantik entspricht. Das Constraint `X in r` wird intern in einen Term übersetzt, der zum einen eine möglichst effiziente Implementierung der Semantik, also der Zuordnung von Variablen zu endlichen Wertebereichen, und zum anderen Propagation erlaubt. Das Constraint `X in r` wird in ein Paar von zwei Termen umgewandelt, die die Abhängigkeiten anderer Constraints (`depend(...)`-Teil) bzw. die Domäne (`domain(...)`-Teil) der CV beschreiben. Diese Paare werden dann mit `co_add/2` in den internen Constraintspeicher von Minerva eingefügt. Bei diesem Einfügen wird dann ggf. das Prädikat `co_merge/2` aufgerufen, das das neue Constraint mit den bekannten Constraints einer Variable verbindet. Ergibt sich dabei eine Änderung der Domäne (sichtbar in `domain(...)`), so muß diese Änderung in den anderen Constraints propagiert werden. Dies wird durch den Teilterm `depend(...)` ermöglicht, der die Abhängigkeiten beschreibt. Die dort enthaltenen Listen von Aufrufen können zur Effizienzsteigerung sortiert werden (Heuristiken), so daß insgesamt weniger Propagation notwendig ist. Dies ist aber in der vorliegenden Version noch nicht implementiert. Die Propagation der Abhängigkeiten ist also 'von Hand' implementiert, da es in Minerva keine `suspend`- und `wake`-Mechanismen wie in ECL^{i}PS^e [16] gibt. Es werden wie in [16] Listen von Constraints verwaltet, die immer dann abgearbeitet (alle Elemente werden mit `call/1` aufgerufen) werden, wenn sich eine Änderung an einer für andere CVn relevanten Domäne ergibt.
In der vorliegenden Implementierung werden die Domänen, wenn möglich, sofort berechnet, denn es ergeben sich immer nur Einschränkungen der möglichen Werte einer CV. Es wird also immer versucht, die Werte `Max,Min,Size` und `Kind` festzulegen. Dies ist auch immer möglich, außer wenn die Grenze eines Intervalls durch den Ausdruck `val(Y)` definiert wurde und die CV `Y` noch nicht instantiiert ist. In diesem Fall wird die Domäne nicht berechnet und das Constraint, das `val(Y)` benutzte in die `C_val`-Liste der CV `Y` eingetragen. Diese Liste wird sofort abgearbeitet, d.h. alle Elemente werden als Ziel aufgerufen, wenn `Y` nur noch einen möglichen Wert hat. Analog werden die Listen `C_max` und `C_min` abgearbeitet, wenn sich die Werte `Max` bzw. `Min` einer CV ändern.
Ein weiteres Prädikat, das in `fd_constraint` zur Verfügung gestellt wird, ist `remove/2`, das es erlaubt, einzelne Werte aus der Domäne von CVn zu entfernen. Dieses Prädikat ist notwendig, um Ungleichungen über CVn implementieren zu können. Wird ein Wert aus einem Intervall gelöscht, so kann es sein, daß das Resultat kein Intervall mehr ist und es muß eine Umwandlung in die Listenrepräsentation vorgenommen werden. Die Implementierung aus [5] durch Bitmaps ist in Minerva oder Java nicht direkt möglich und scheint uns, wegen der begrenzten Anzahl der Elemente[15] jeder Domäne für Minerva nicht geeignet. Alternative denkbare Darstellungen wären Ausschlußlisten oder dünn besetzte Bitmaps [17] in Form von Java-Mengen.
`Kind` wird mit dem Wert `val` instantiiert, wenn nur noch ein Wert in der Domäne der CV enthalten ist, wenn also die Werte von `Min` und `Max` gleich sind. In diesem Fall wird dann auch der Wert `Val` durch diesen einzigen Wert definiert.
Das von Minerva zur Constraintverarbeitung benötigte Prädikat `co_merge/2` wird in die-

[15] In `clp(FD)` können in Bitmaps maximal 128 Elemente verwaltet werden. Dadurch wird das System bei der Verwendung dieser Repräsenatiion unvollständig, wenn eine Domäne mehr als diese 128 Elemente enthält. Um dieses Manko auszugleichen, wird in `clp(FD)` eine Verarbeitung eingeführt, die diese Unvollständigkeiten verwaltet, so daß sie effektiv in einem angewandten System zwar nicht mehr auftreten, aber aus theoretischer Sicht nicht beseitigt werden (vgl. [7]).

sem Paket durch den Durchschnitt der Domänen und die Vereinigung der Abhängigkeiten implementiert. D.h. wenn zwei CVn unifiziert werden, bzw. ein weiteres Constraint einer vorhandenen CV zugeordnet wird, so werden die Repräsentationen der Domänen so zusammengefaßt, daß ihre mathematische Interpretation dem Durchschnitt der repräsentierten Mengen entspricht. Da die Abhängigkeiten anderer Variablen von den unifizierten CVn bestehen bleiben, werden die Listen aus dem `depend(...)`-Teil jeweils konkateniert. Ist eine solche Unifikation nicht möglich, wenn z.B. eine leere Domäne entsteht, so scheitert `co_merge/2` und Backtracking wird eingeleitet.

Das `fd_predicates`-Paket

Das `fd_constraint`-Paket wird vom Paket `fd_predicates` verwendet, das die Constraints zur Verfügung stellt, die die Verbindung zwischen Anwender und Implementierung des Constraints herstellen. Hier wird die Syntax zur Implementierung von Gleichungen und Ungleichungen, sowie einige Prädikate zum Umgang mit Constraints zur Verfügung gestellt. Über dieses Paket ist das `X in r` Constraint selbst nicht aufrufbar, da dieses wie bereits in Abschnitt 3.1 angesprochen wurde, dem Programmierer zu viele syntaktische Möglichkeiten läßt, deren Semantik im vorliegenden Kontext nicht interpretierbar ist. Es werden Prädikate zur Verfügung gestellt, mit denen Domänen von Variablen definiert, bzw. Domänen von CVn eingeschränkt werden können. In Abbildung 5 wird eine Auswahl der zur Verfügung gestellten high-level Constraints dargestellt.

- `'x+y=z'(X,Y,Z)`: Z ist eine Variable oder eine CV, X und Y sind CVn, dann werden die Domänen von `X,Y,Z` so eingeschränkt, daß die Gleichung erfüllt ist
- `'x=y+c'(X,Y,C)`: `C` ist Integer, `X` ist Variable oder CV, `Y` ist eine CV.
- `'fd_geq'(X,Y)`: `X,Y` sind CVn, `X` ist größer oder gleich `Y`
- `'fd_neq'(X,Y)`: `X,Y` sind CVn. Wenn `X` oder `Y` instantiiert wird, wird dieser Wert aus der Domände der anderen CV entfernt
- `label(VarList)`: `VarList` ist eine Liste von CVn. Jedes Element der Liste wird mit einem Wert aus seiner Domäne instantiiert.
- `all_different(VarList)`: `VarList` ist eine Liste von CVn. Alle CVn können nur mit unterschiedlichen Werten instantiiert werden.
- `domain(VarList,X,Y)`: Allen Variablen oder CVn aus `VarList` wird die Domäne [X,Y] zugewiesen.
- Weitere Constraints sind: `fd_eq/2 fd_gne/2, fd_leq/2, fd_lne/2, greatestVal/1, smallestVal/1, fd_var/1, fd_min/2, fd_max/2, fd_dom/2, fd_size/2, indomain/1, max(x,y)=z/3, min(x,y)=z/3,x=c*y/3, x*y=z/3, union/3, intersection/3, co_write/1`.

Abbildung 5: High-level Constraints aus `fd_predicates`

Wie es auch in [5] und [3] empfohlen wird, werden in unserem System arithmetische Constraints mit einer maximalen Anzahl von drei CVn zur Verfügung gestellt. Dieses Vor-

gehen erlaubt eine effizientere Implementierung dieser definierten Prädikate und ist keine Einschränkung der Ausdrucksmöglichkeiten beliebiger Terme über CVn, da komplexe Terme stets in Terme, die sich durch diese Bibliotheksprädikate darstellen lassen, zerlegt werden können (vgl. Beispiel 3). Zu entwickeln ist noch das Paket `fd`, das diese Zerlegung automatisch durchführt, wie es im nächsten Abschnitt beschrieben wird.

3.4 Ausblick: Beliebige Gleichungen

Der nächste Schritt in der Entwicklung des Systems ist die Implementierung des Pakets `fd`, das die Syntax für Gleichungen und Ungleichungen über beliebigen Termen zur Verfügung stellt. Dieses Paket soll im fertigen System die Schnittstelle zum Constraint-Programmierer sein, er gibt Domänen und Gleichungen über bestimmten Variablen ein, die dann als CVn in einem finite-domain Constraintsystem wie z.B. in [7] verarbeitet werden. Die Gleichungen werden auf high-level Constraints durch die Verwendung von CVn, die Zwischenergebnisse repräsentieren zurückgeführt. Dadurch entstehen weniger komplexe Abhängigkeitsbeziehungen zwischen den CVn im Constraintspeicher. Dadurch verringert sich zwar nicht die Gesamtzahl der Abhängigkeiten, aber bei einer Änderung werden weniger überflüssige Aufrufe gemacht, da kontrolliert werden kann, was bereits verändert wurde. Dies wird in [5] verdeutlicht.

Beispiel 3 *Die Gleichung $F+G=H+10$ könnte direkt (inline code) implementiert werden, indem jede CV durch die anderen Werte repräsentiert wird:*

```
main(_):-
     domain([F,G,H],0,20),
     F = H + 10 - G, G = H + 10 - F, H = F + G - 10.
```

Dann müssen bei einer Änderung der Domäne von F die beiden letzen Gleichungen nochmals als Constraint aufgerufen werden. Wird die Gleichung aber durch `fd` *in die Klausel*

```
main(_):-
     domain([F,G,H],0,20),
     'x+y=z'(F,G,Inter), 'x=y+c'(Inter,H,10).
```

übersetzt (library calls), so muß bei einer Änderung nur ein Constraint aufgerufen werden. Und nur wenn diese Änderung einen Effekt auf andere Domänen hat, werden weitere Aufrufe durchgeführt. Wie man sieht, können durch Berechnung von Zwischenvariablen komplexe Abhängigkeitsnetze reduziert werden.

4 Zusammenfassung

Die Implementierung von Constraintsystemen für die verteilte Anwendung im Internet ist bisher nur ansatzweise und direkt in Java (z.B. [2], [4]) möglich. Die logisch-deklarative Sprache Minerva [12] stellt die Verarbeitung von attributierten Variablen wie in [16] zur Verfügung und ist selbst in der Sprache Java implementiert, so daß sie verteilte und besonders sichere Internet-Anwendungen unterstützt. Diese attributierten Variablen werden dazu verwendet, mögliche Wertebereiche an Variablen zu binden, so daß sie als Constraint-Variablen in einem FD-Constraintsystem wie in [5] verwendet werden können. Dieses 'Binden' wird durch das Constraint `X in r` umgesetzt, das die Interpretation einer Domäne `r` in einer bestimmten Umgebung (andere Constraint-Variablen) an die Variable `X` definiert. Dieses low-level-Constraint

wird sowohl konzeptionell (Syntax und Semantik), als auch operational (Implementierung) beschrieben. Das FD-Constraintsystem erlaubt es dem Benutzer, beliebige Gleichungen und Ungleichungen über Constraint-Variablen anzugeben, wobei die Domänen der einzelnen Variablen durch solche Ausdrücke eingeschränkt[16] werden können. Die Gleichungen und Ungleichungen werden via high-level-Constraints auf das implementierte low-level-Constraint zurückgeführt.
Der Constraintlöser ist in der Sprache Minerva und nicht in der unterliegenden Zielsprache Java implementiert, so daß er unabhängig von der vorliegenden Minerva-Implementierung bleibt. Außerdem kann das Modul zur Constraintverarbeitung wegen seiner klaren und eindeutigen Schnittstellen auch zur Erweiterung anderer Prolog-Dialekte verwendet werden, indem dort die entsprechenden Schnittstellen-Prädikate implementiert werden.

Literatur

[1] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transitions on Programming Languages and Systems*, 16(4):1361–1398, July 1994.

[2] Eric Bruchez and Marc Torrens. *Java Constraint Library.* Artificial Intelligence Laboratory, Computer Science Department, Lausanne, `http://liawww.epfl.ch/~torrens/Project/JCL/`, 1996.

[3] B. Carlsson and M. Carlsson. Constraint solving and entailment algorithms for `cc(fd)`. *Research Report SICS*, 1993.

[4] Andy Hon Wai Chun. Constraint programming in java with jsolver. In *PACLP99*, 1999.

[5] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.

[6] Department of Computer Science, Universität des Saarlandes, `http://www.mozart-oz.org/`. *Oz Internet-Pages.*

[7] Daniel Diaz. `clp(FD)` *2.21 User's Manual.* INRIA, 1994.

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, Berlin, Heidelberg, New York, Tokio, 1985.

[9] David Flanagan. *Java in a Nutshell, 2nd Edition.* o'reilly, 1997.

[10] Thom Frühwirth and Slim Abdennadher. *Constraint-Programmierung.* Springer, 1997.

[11] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification - applied for the implemetation of clp languages. Technical report, Austrian Research Institute for Artificial Intelligence, 1992.

[12] IF-Computer. *Minerva Internet Pages.* `http://www.ifcomputer.co.jp/Products/MINERVA/`.

[13] J. Jaffar and L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87*, pages 111–119, 1987.

[14] U. Neumerkel. Extensible unification by metastructures. Technical report, Institut für Praktische Informatik, Technische Universität Wien, 1990.

[15] Horst Reichel. *Structural Induction on Partial Algebras.* Akademie-Verlag Berlin, 1984.

[16] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, UK, 1997.

[17] Armin Wolf. *Adaptive Constraintverarbeitung mit Constraint-Handling-Rules.* PhD thesis, TU Berlin, 1999.

[18] Uwe Wolter. *An Algebraic Approach to Deduction in Equational Partial Horn Theories.* PhD thesis, HU Berlin, 1990.

[16] Es lassen sich insbesondere einzelene Werte inferieren, die Gleichung/Ungleichung lösen.

Using Objects to Build Constraint Databases

Annalisa Di Deo, Dmitri Boulanger
GMD-First
Berlin, Germany
{dmitri, anna}@first.gmd.de

Abstract

Constraint databases exploit a fundamental duality: a constraint (first-order formula) with free variables is interpreted as a set of database tuples that satisfy it. And, conversely, a database object can be viewed as a constraint. This enables to use constraints as basic data types in the underlying DBMS and to enjoy expressiveness of first-order logic and most from constraint programming.

Most of the existing frameworks consider the relational data model as a basic data representation model and use linear constraints to represent complex objects. This is not always efficient. Moreover, modern DBMS offer much more than a simple set of relational tables. Therefore, we focus on the definition and formal background for an implementation of a constraint data model to be built on the top of an object-relational DBMS. As an illustration, our approach is capable to exploit existing spatial and temporal tools (like the one from Oracle8) yielding the declarativeness and expressiveness of constraint programming.

1 Introduction

With the increasing complexity of applications that store and manage spatial and temporal data, Database Management Systems are facing new challenges to efficiently represent and query such data, preserving the declarative and user-friendly features of the query languages. It has been acknowledged for a long time that the relational model fails to handle highly complex data and their operations. For this reason, the existing commercial DBMS, are always more directed to the use of object-oriented models, as they are more suitable to handle with such a kind of data. For instance, the Spatial cartridge of Oracle8 ([1]) no longer manages complex geometrical shapes as sets of points stored in tables, but uses specially designed data types, operations and tools to handle this sort of data efficiently. Due to the extensive range of applications, the problem of developing and unifying data model still remains open, when a facility to manipulate different data in a uniform way is needed.

The recent fields of constraint database, initiated at the beginning of the decade [14], has lead to sound data models and query languages for multi-dimensional data [7, 13, 9]. The big challenge of constraint databases is to introduce constraints as basic data type in databases. The main principles are based on a simple and fundamental duality: given a database with the schema $[A_1, \ldots, A_n]$, a constraint (first-order formula) ϕ with free variables $x_1, \ldots, x_n$ is interpreted as a set of tuples $(d_1, \ldots, d_n)$ that satisfy ϕ; and, conversely, a finitely representable object in $(x_1, \ldots, x_n)$-space can be viewed as a constraint. This enables to manipulate relations in the underlying DMBS in a symbolic way enjoying expressive power of first-order logic. That is, the syntax is constraints, i.e. symbolic expressions; the semantics are the corresponding relations. From this data

modeling perspective, constraints are used as a unifying data type for the representation of different sorts of multi-dimensional data. Most of the existing frameworks rely on linear constraints, first appeared in [10]. They allow to manipulate relations of arbitrary dimension between points in a symbolic manner. In spite of their expressiveness, the existing constraint databases are not always efficient enough for real problems where the size and the complexity of data are high. In particular, spatial data might be so complex, e.g. non-convex geometries with multiple holes, that constraint database models are not suitable to handle them efficiently.

Therefore, we focus on the definition of a constraint data model, built on the top of object-oriented DBMS. This approach is not new, there exist papers and prototypes like [3, 4, 9]. The challenge of our proposal is that the constraint model exploits existing spatial and temporal tools (like the one from Oracle8) preserving the declarativeness of the first-order languages. For these reasons, the model is based on constraints, which express *relations between objects* whose implementation is hidden.

The design of the model is based on two principles. Firstly, constraints must be treated as "first class objects", whose methods are typical logical operations, such as conjunction, disjunction and existential quantification. Secondly, an underlying object-based representation of data must be wrapped as constraints. The core of the approach is the set of rules (or axioms), given by the cylindric algebras. This family of algebras are formed by enhancing Boolean algebras with equality and a family of unary operations called *cylindrification*. Constraint languages can be interpreted under some conditions as cylindric algebras because operations on constraints such as *disjunction, conjunction* and *projection* can be operationally formalized as binary (disjunction and conjunction) and unary (projection) operators of the cylindric algebras. We illustrate the approach using spatial and temporal data.

The remainder of the paper is organized as follows. In section 2 we give an example to motivate our choices and to facilitate the understanding of the following theory. Section 3 recalls basics of first-order logic, constraints and introduces the cylindric algebras with the set of axioms that characterized them. In section 4 we define two domains, respectively of temporal and spatial objects, on which the constraints are defined and show how the approach works. The last section gives some remarks and future works.

2 Motivating Example

Assume that we would like to perform some queries on particular entities in Berlin, for instance:

Q_1 *Give me the buildings that don't exist anymore and the time in which they were present.*

Q_2 *Give me the parcels inside a park along a river.*

In the above queries, different entities are considered, like time, space and thematic features. To answer Q_1 and Q_2 we need a data model that gives the possibility to reason with objects of different nature. In particular, we need to represent parks and parcels as geometric objects with shapes and locations and relationships between them like, intersection, inside etc. and buildings as objects with timestamps.

We suppose to have a database storing information about buildings, parcels, parks and rivers in Berlin as objects. Some of them (temporal objects) have a timestamp, indicating its life time and other objects (spatial objects) have a geometry describing its location and shape. How the timestamps and geometries are stored, is hidden at the DBMS level. Each object is stored in a table with two attributes, namely the name of the object (or identifier) and a timestamp or a geometry (see the fig. 1), depending if the object is a spatial or a temporal object.

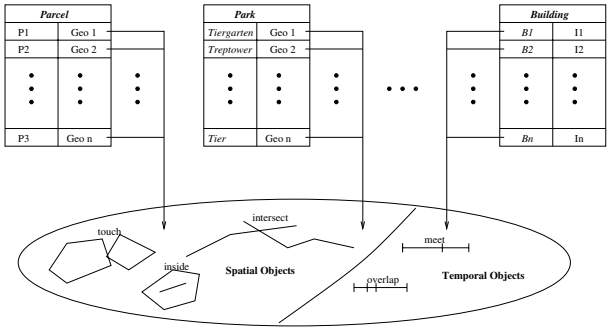


Figure 1: Schema of an object-oriented model for some entities in Berlin

As shown in fig. 1, the second attribute of the tables refers to some objects, namely spatial and temporal objects, whose structure is hidden in the underlying model. What is known is a set of operations (or methods) that the underlying model provides to reason with temporal and spatial objects.

The constraint database model offers *unifying* framework to model this kind of situations. The above queries Q_1, Q_2 can be reformulated in first-order logic as the following formulas:

φ_1: $\exists_{\tau_2}\,(building(name,\,\tau_1) \wedge before(\tau_1,\,\tau_2))$

φ_2: $\exists_{x_2,x_3}(parcel(name,\,x_1) \wedge park(\_,\,x_2) \wedge inside(x_1,\,x_2) \wedge river(\_,\,x_3) \wedge touch(x_2,\,x_3))$

φ_1 defines a set of buildings that lived during an interval ending in some time point. The existence of τ_2, after τ_1, states that τ_1 is not infinite and the corresponding buildings stopped to exists in some time. The meaning of the conjunction is in this case clear. The existential quantifier $\exists_{\tau_2}$ says that we are not interested in the values of the variable τ_2 but only in its existence and therefore can be eliminated. The elimination of τ_2 produces a new formula equivalent to φ_1, which selects only the values for τ_1 that are finite. *building* is a predicate (or a table) in the database that refers to a time interval τ, and $before(\tau_1,\,\tau_2)$ is a relationship between intervals. As the intervals, the relationships are implemented in an underlying model. Similarly we can reason for φ_2. But in this case the objects involved in the formula are spatial ones and an appropriate reasoning has to be made depending on the types of the objects, e.g. polygons, lines, points etc. and the set of possible relationships between them.

The evaluation of the above queries relies on a constraint database model, which is built from the objects *buildings* and *intervals*, for φ_1 and *parcels*, *parks* and *rivers* for φ_2. We assume that an underlying DBMS provides objects manipulation primitives but not the constraints one. A constraint object has to be created as an instance of a predefined class *constraint*. The constraint objects in the class *constraint* have standard constraint operations such as *conjunction*, *disjunction* and *projection*. The *disjunction* operation (*join*) models, in a general way, the "merging" together of information via set union. The *conjunction* (*meet*) plays the important role of collecting information during computation. Finally, the *projection* on a set of variables Y with hidden variables X ($\exists_X$), projects out information about X from the constraint on which the operator is applied.

In the particular case of the queries Q_1, Q_2, the corresponding formulas φ_1, φ_2 are existentially quantified conjunctions of constraints. Two constraint instances must be created with particular references: in the case of φ_1 to the objects *buildings* and *intervals* and the method *before* defined for intervals; in the case of φ_2 to the objects *parcel*, *park*, *river* and geometries and the methods *inside*, *touch* belonging to geometries. Both constraint objects have methods to perform conjunction and projection.

Such a model will be presented in the next sections and an algebra that gives the rules to define the operations of conjunction, disjunction and projection of constraints interpreted on set of structured objects, like intervals and geometries.

3 Wrapping objects as constraints

Fundamental aspects of constraint programming can be separated from the details specific to particular constraint systems. This is exactly what is needed to establish basic principles for building constraint from objects stored in a database. In particular, constraints can be seen as elements of an algebraic structure, providing a uniform treatment of domain-dependent features. We use a widely known algebraic definition of first order logic, namely, *cylindric algebras*. *Constraint systems* are then defined as algebraic structures whose domains represent constraints and whose operations include unification, constraint composition and existential quantification.

3.1 Basics

We start by recalling the needed basic concepts.

An *algebraic structure* is a pair $(\mathcal{D}, \mathcal{Q}_i)_{i \in I}$ where $\mathcal{D}$ is a non-empty set, called the *domain*[1] of the structure and $\mathcal{Q}_i$, $i \in I$ are *functions (operations)* on and to elements of $\mathcal{D}$. The set $\mathcal{Q}_{i \in I}$ is defined by a *vocabulary, or signature*, which consists of a set of function symbols, and of a set of constant symbols, together with an *arity* function that associates a natural number n to each element in the vocabulary.

A *first-order language* has a vocabulary extended with *predicate* symbols. An *interpretation structure* of a first-order language consists of:

- the corresponding algebraic structure $(\mathcal{D}, \mathcal{Q}_i)_{i \in I}$ (its vocabulary is exactly the function symbols and the constants of the language)
- an interpretation of predicate symbols

An interpretation of n-ary predicate is an n-ary *relation* defined as a subset of $\mathcal{A}^n$ (the n-ary Cartesian product $\mathcal{A} \times \cdots \mathcal{A}$).

We distinguish the following syntactic objects of the language:

- a countable set of *variables*
- *terms*, which are built from variables and symbols in the vocabulary of the algebraic structure
- well-defined *formulas*, which are build from variables, terms, predicates, existential ($\exists$) and universal ($\forall$) quantifiers using standard first-order connectives (such as $\vee$, $\wedge$, $\leftrightarrow$, etc.); formulas might contain *free* variables, i.e. variables, which are not bound by the quantifiers
- well-defined *sentences*, which are formulas having *no* free variables

Given an interpretation structure of a first-order language, a *truth value* (i.e. *false* or *true*) can be defined for any sentence in the standard way. A language and its interpretation form *complete logical theory* if there is an algorithm which computes the truth value for *any* sentence.

Let ϕ be a formula with n free variables $\overline{x}$. Its *extensional representation* $ext(\phi)$ is a set of all n-ary tuples of domain elements $\overline{d}$ such that

$$\phi(\overline{d}) \quad \text{is true in the interpretation structure of the language}$$

[1] In the literature can be found other terms: universe, carrier, etc.

Two formulas $\phi(\overline{x})$ and $\psi(\overline{y})$ with $\overline{x}$ and $\overline{y}$ free variables, are said to be *equivalent* iff the sentence

$$(\forall \overline{x}, \overline{y})\ (\phi \leftrightarrow \psi)$$

is true in the interpretation of the language. A language with an interpretation structure form a *decidable logical theory* if there exists an algorithm which decides equivalence for *any* pair of formulas.

One of fundamental problems in first-order logic is *quantifier elimination*: given formula ϕ, find an equivalent formula ψ that has the same free variables *but* contains no quantifiers.

3.2 Flat Constraint Algebras

Given a decidable logical theory, we use term **constraints** to denote the set of formulas having no occurrences of negation and existential quantifier. We give now a formal algebraic specification of **flat** constraints, i.e. constraints having no occurrences of function symbols.

Constraints are constructed using disjunction and conjunction. An algebraic characterization of this process exploits algebraic structures $(\mathcal{C}, \mathcal{Q})$ with $\mathcal{C}$ the domain of the structure. With an abuse of notation, we will denote *distinguished elements* of $\mathcal{C}$ as constant operations on $\mathcal{C}$. In particular, we use special elements $\mathbf{0}$ and $\mathbf{1}$, called *identity elements*. Next, we need a structure $(\mathcal{C}, \mathcal{Q})$, called *monoid*. Its set of operators $\mathcal{Q}$ has a binary function, say $\otimes$, such that $\otimes$ is associative (i.e. $(a \otimes b) \otimes c = a \otimes (b \otimes c)$ for all $a, b, c \in \mathcal{C}$); and there is an identity element, say $\mathbf{1}$, such that $a \otimes \mathbf{1} = \mathbf{1} \otimes a = a$ for any $a \in \mathcal{C}$. Finally, a *semiring* [2, 15] is an algebraic structure $(\mathcal{C}, \otimes, \oplus, \mathbf{1}, \mathbf{0})$ satisfying the following:

R_1. $(\mathcal{C}, \otimes, \mathbf{1})$ and $(\mathcal{C}, \oplus, \mathbf{0})$ are monoids.

R_2. $\oplus$ is commutative and idempotent, i.e. $x \oplus y = y \oplus x$ and $x \oplus x = x$ for any $x, y \in \mathcal{C}$, resp. (Hence, $\oplus$ is associative, commutative and idempotent).

R_3. $\mathbf{0}$ is an *annihilator* for $\otimes$, i.e., for every $c \in \mathcal{C}$, $c \otimes \mathbf{0} = \mathbf{0} \otimes c = \mathbf{0}$.

R_5. $\otimes$ is *left-* and *right-distributive* over applications of $\oplus$, i.e., if a_1, a_2 and c are elements in $\mathcal{C}$, then $c \otimes (a_1 \oplus a_2) = (c \otimes a_1) \oplus (c \otimes a_1)$ and $(a_1 \oplus a_2) \otimes c = (a_1 \otimes c) \oplus (a_2 \otimes c)$

With an abuse of notation, we write $\mathcal{C}$ to denote the semiring $(\mathcal{C}, \otimes, \oplus, \mathbf{1}, \mathbf{0})$ when the meaning is clear from the context.

Semirings are *naturally ordered* by the binary relation $\leq$ such that for any $a, b \in \mathcal{C}$: $a \leq b$ iff $a \oplus c = b$ for some $c \in \mathcal{C}$ ([15]). Since $\oplus$ is idempotent, there exists a natural order relation $\trianglelefteq \subseteq \mathcal{C} \times \mathcal{C}$ for a semiring $\mathcal{C}$:

$c_1 \trianglelefteq c_2$ iff $c_1 \oplus c_2 = c_2$ for any $c_1, c_2 \in \mathcal{C}$,

Modeling the semantics of constraints requires the ability to restrict a constraint to the variables appearing in the query. Another important aspect is unification, substitutions and variable dependencies between constraints. Semirings are too weak to capture these important ingredients.

A family of "hiding" operators models semantics of the first order existential quantification. The intuition here is that given a constraint c, the existential quantification, being considered as an algebraic function $\exists_X(c)$, yields the constraint obtained by "projecting out" from c all information about the variables in S.

Diagonal elements (denoted by $d_{x,y}$ below), model equations of the form $x = y$ with x and y variables. It is a well-known construction in cylindric algebras, which captures semantics of equality directly or indirectly occurring in any constraint system.

A *flat constraint system* $(\mathcal{C}, \otimes, \oplus, \mathbf{1}, \mathbf{0}, \exists_X, d_{x,x'})$, $X \subseteq V$, $x, x' \in V$ is an algebraic structure where

- $\mathcal{C}$ is a set of constraints generated by a given set of *flat atomic constraints* over variables V, i.e. constraints of the form $p(x_1, \ldots, x_n)$ with p a predicate symbol and $x_1, \ldots, x_n \in V$
- $\mathbf{0}, \mathbf{1}, d_{x,x'}$ for each pair $x, x' \in V$, are distinct (atomic) elements of $\mathcal{C}$
- $\otimes, \oplus$ are binary operations on $\mathcal{C}$
- the structure $(\mathcal{C}, \otimes, \oplus, \mathbf{1}, \mathbf{0})$ is a semiring
- $\{\exists_X\}_{X \subseteq V}$ is a family of unary operations on $\mathcal{C}$

such that the following postulates are satisfied for any constraints $c, c' \in \mathcal{C}$; and variables $\{x\}, X, Y \subseteq V$ and $x, x', x'' \in V$:

C_1. $\exists_X \mathbf{0} = \mathbf{0}$

C_2. $c \oplus \exists_X c = \exists_X c$;

C_3. $\exists_X(c \otimes \exists_X c') = \exists_X(\exists_X c \otimes c') = \exists_X c \otimes \exists_X c'$;

C_4. $\exists_X \exists_Y c = \exists_{(X \cup Y)} c$;

C_5. $\exists_X(c \oplus c') = \exists_X c \oplus \exists_X c'$;

D_1. $d_{x,x} = \mathbf{1}$;

D_2. $d_{x,x'} = d_{x',x}$;

D_3. $d_{x',x''} = \exists_{\{x\}}(d_{x',x} \otimes d_{x,x''})$ where $x \neq x', x''$

D_4. $\exists_{\{x\}}(d_{x,x'} \otimes (c \otimes c')) = \exists_{\{x\}}(d_{x,x'} \otimes c) \otimes \exists_{\{x\}}(d_{x,x'} \otimes c')$.

The above suggests an algorithm which generates the set of constraints $\mathcal{C}$ starting from flat *atomic* constraints, which include diagonal elements (equality) and two identity elements. The identity elements $\mathbf{1}$ and $\mathbf{0}$ capture semantics of the fixed first order predicates (constants) *true* and *false*. In the following we distinguish between *constraints* and *simple constraints*. A constraint is any object in $\mathcal{C}$, while a *simple constraint* is an atomic constraint, or the existential quantification of a simple constraint, or a conjunction (i.e., $\otimes$) of simple constraints. Therefore, simple constraints do not contain $\oplus$.

The meaning of existential quantification is given by the axioms from C_1 to C_5, while equality are specified by the axioms from D_1 to D_4. Axioms D_3 and D_4 relate the notion of renaming of the variables with equality.

Variable renaming can be explained with the derived operation

$$\partial_x^y c = \exists_{\{x\}}(d_{x,y} \otimes c) \text{ for pair of variables } x, y \text{ such that } x \neq y$$

Intuitively it changes variable names. As a consequence, notions of "independence" and "occurrence" of variables apply in the obvious way to constraints in $\mathcal{C}$. Let c be a constraint and var be the set of variables occurring in it. Given a variable x, we define:

$$x \; ind \; c \text{ iff } \partial_x^y c = c \text{ for any } y \in V \text{ such that } x \neq y$$

A variable x is *bound* in c iff it is existentially quantified in c; x is *free* in c iff $x \in var(c)$ and x is not bound in c. A *renaming* of c with respect to x is a constraint $\partial_x^y c$ such that $x \neq y$ and $y \; ind \; c$. The set of *free variables* in a constraint c is denoted by $FV(c)$. Thus, we can denote $\exists_{var(c) \setminus X} c$, i.e. hidining all the variables in c except X, as $\exists(c)_X$.

As an illustration, trivial constraint systems can be obtained directly from a semiring $\mathcal{C}$ by letting:

- $d_{x,x'} = \mathbf{1}$ for each pair $x, x' \in V$
- $\exists_X c = c$ for each constraint $c \in \mathcal{C}$ and $X \subseteq V$

This, for instance, yields $\partial_x^y c = c$ for each pair of variables $x, y \in V$, i.e. these constraints don't feel variable dependencies.

3.3 Properties of Constraint Systems

Given a constraint system $\mathcal{C}$, for any constraints $c, c' \in \mathcal{C}$ and variables $x \in V$, $X \subseteq V$ and $y, y', y'' \in V$ such that $x \neq y$, the following properties hold:

P1: $\exists_X \exists_X c = \exists_X c$;
P2: $c \trianglelefteq c' \;\Rightarrow\; \exists_X c \trianglelefteq \exists_X c'$;
P3: $\forall c, c' \in \mathcal{C}: \;\; c' \trianglelefteq \exists_X c \;\Leftrightarrow\; \exists_X c' \trianglelefteq \exists_X c$;
P4: $\forall c, c' \in \mathcal{C}: \;\; c \trianglelefteq c' \;\wedge\; c' \trianglelefteq \exists_X c \;\Rightarrow\; \exists_X c = \exists_X c'$;
P5: $\exists_{\{x\}} c = c$ iff $\exists_{\{x\}} \tilde{c} = c$ for some $\tilde{c} \in \mathcal{C}$;
P6: $\exists_{\{x\}} c = c$ if $x \; ind \; c$ (in particular $\exists_{\{x\}} d_{y',y''} = d_{y',y''}$ when $x \neq y', y''$);
P8: $c \trianglelefteq c' \;\Rightarrow\; \partial_x^y c \trianglelefteq \partial_x^y c'$;
P9: $\partial_x^y \exists_{\{x\}} c = \exists_{\{x\}} c$;
P10: $\partial_x^y \; c = c$ iff $\partial_x^y \tilde{c} = c$ for some $\tilde{c} \in \mathcal{C}$;
P11: $\exists_X \mathbf{1} = \mathbf{1}$; $\exists_X c = \mathbf{0}$ iff $c = \mathbf{0}$;
P12: $\exists_{\{x\}} d_{x,y} = \mathbf{1}$;
P13: $(d_{y,y'} \otimes d_{y',y''}) \oplus d_{y,y''} = d_{y,y''}$ (transitivity).

In particular, from P9 $\partial_x^y c \trianglelefteq \exists_{\{x\}} c$ and if x is bound in c then $x \; ind \; c$. Therefore, if c is a renaming apart of c' with respect to x, then $x \; ind \; c$. The property P6 extends to conjunctions of constraints and describes the interaction of projection with (hidden) variables: for any constraints c and c' and X a set of variables such that $x \; ind \; c$ for every $x \in X$, holds:

$$\exists_X (c \otimes c') = c \otimes \exists_X (c').$$

Three is an important relation between existential quantification (hiding variables) and renaming apart of constraints with "fresh" variables. Namely, given constraints c and c', holds:

$$c \otimes \exists_{\{x\}} c' = \exists_{\{y\}} (c \otimes \partial_x^y c'), \text{ for } y \; ind \; c, c' \text{ and } y \neq x$$

4 Application to Temporal and Spatial Objects

In this section we present an application of the machinery introduced in the section 3 above. Namely, we develop a constraint system which wraps temporal and spatial data. To model them we exploit two classes of objects, a temporal and spatial one, as subclasses of the class "constraint".

The operational semantics of the class "constraint" has been explained in section 3. Below we use the standard Java object model as an object-oriented framework [8]. In particular, all objects, which can be used in our constraint database **must** extend the *abstract* class "constraint" (see fig. 2). Our aim is to develop a programming technology facilitating development of complex constraint systems correctly. In particular, if all objects extend the class "constraint", then the standard compiler can be used to ensure that the constraint system is well-defined in the sense of section 3.

Below we consider time intervals and spatial objects as domains of the constraint system. Predicates express relationships between the objects. They are the bricks used to built up our constraints. Next, we define two binary operations, namely $\wedge$, $\vee$, to compose our constraints and an unary operator $\exists_X$ to hide variables. The result must satisfy the definition of constraint system in 3.

In this way in the next subsection, we define a temporal constraint system, based on Allen's interval algebra [11, 12], and a spatial constraint system, based on Egenhofer spatial relationships between two-dimensional objects. As soon as the both kinds of objects are wrapped as constraint systems, they can be used in a uniform constraint system.

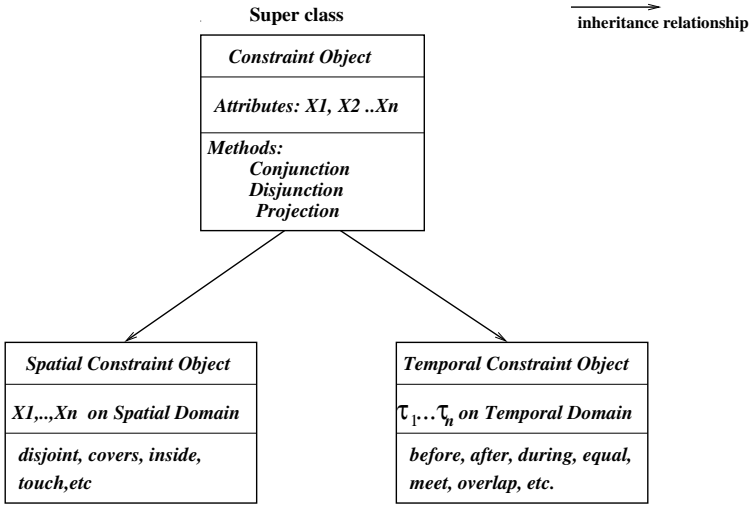


Figure 2: Spatial and temporal classes as subclasses of *constraint*

4.1 Temporal Constraint System

The temporal constraint system is built up from the predicates corresponding to Allen's algebra relations. Constraints are interpreted over the predefined interval domain $\mathcal{I}$.

4.1.1 Temporal Domain and Predicates

The domain $\mathcal{I}$ is the set of well-defined time intervals. A well-defined interval i is a closed interval whose starting and ending points $sp(i)$ and $ep(i)$ are finite and $sp(i) < ep(i)$. Also, we use the unique open interval, with starting and ending points $-\infty$ and $+\infty$, resp. Below we denote with $i, i_1, i_2, \ldots, i_n$ closed elements of $\mathcal{I}$, with $time$, the open interval $(-\infty, +\infty)$.

Allen [11] defines a set of binary relations between time intervals, which cover all possible binary relations between intervals. We need the corresponding set of predicates and an interpretation of such predicates over the domain $\mathcal{I}$. Namely, for all intervals $i, i_1, i_2 \in \mathcal{I}$, we define the following predicates:

p_0. $equals(i_1, i_2) = true \Leftrightarrow sp(i_1) = sp(i_2) \wedge ep(i_1) = ep(i_2)$

p_1. $before(i_1, i_2) = true \Leftrightarrow ep(i_1) < sp(i_2)$

p_2. $meets(i_1, i_2) = true \Leftrightarrow ep(i_1) = sp(i_2)$

p_3. $overlaps(i_1, i_2) = true \Leftrightarrow sp(i_1) < sp(i_2) \wedge ep(i_1) > sp(i_2) \wedge ep(i_1) < ep(i_2)$

p_4. $during(i_1, i_2) = true \Leftrightarrow sp(i_1) > sp(i_2) \wedge ep(i_1) < ep(i_2)$

p_5. $starts(i_1, i_2) = true \Leftrightarrow sp(i_1) = sp(i_2) \wedge ep(i_1) < ep(i_2)$

p_6. $finishes(i_1, i_2) = true \Leftrightarrow sp(i_1) > sp(i_2) \wedge ep(i_1) = ep(i_2)$

The above set of predicates corresponds to a subset of all Allen's relations. Remaining predicates are obvious as it can be seen in the figure 3 which shows the complete set.

To build the constrain system correctly, we extend the above set of predicates adding four more unary predicates on $\mathcal{I}$ having the following interpretation:

q_1. $time(i) = true \Leftrightarrow i = time$

q_2. $not\_time(i) = true \Leftrightarrow i \neq time$

q_3. $not\_point(i) = true \Leftrightarrow sp(i) = ep(i)$

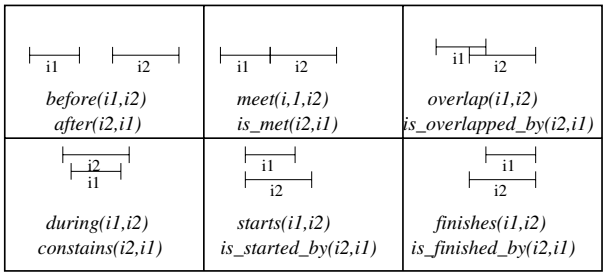


Figure 3: The complete set of Allen's relations between intervals

$q_4.\ not\_point(i) = true \Leftrightarrow sp(i) \neq ep(i)$

Consider the set of all well-formed formulas built from the above predicates $p_1 - p_6, \hat{p_1} - \hat{p_6}$ and $q_1 - q_4$ such that they have no occurrences of the universal quantifier and negation, i.e. formulas are built up only using connectives $\{\wedge, \vee, \exists\}$. Also, there are two special constraints, the tautology **true** and the inconsistent formula **false**. Two formulas are equivalent is they have the same extensional representation (cf. sec.3.1). Let $\mathcal{P}_{time}$ be the corresponding quotinet set. This, for instance, enables a well-defined algebraic characterization of the two logical connectives $\wedge$ and $\vee$ defining them as binary operators from elements of $\mathcal{P}_{time}$ to elements of $\mathcal{P}_{time}$, such that $\forall \varphi_1, \varphi_2 \in \mathcal{P}_{time}$:

$$\wedge, \vee : \mathcal{P}_{time} \times \mathcal{P}_{time} \rightarrow \mathcal{P}_{time}$$

$$\varphi_1 \wedge (\vee) \varphi_2 = \begin{cases} \varphi & : \ \varphi \in \mathcal{P}_{time} \ and \ \varphi \equiv \varphi_1 \wedge (\vee) \varphi_2 \\ \textbf{true} & \\ \textbf{false} & \end{cases}$$

It is not difficult to see that the algebraic structure $(\mathcal{P}_{time}, \wedge, \vee, \textbf{true}, \textbf{false})$ is a semiring (cf. sec. 3.2).

4.1.2 Cylindrification of Temporal Constraints

We construct a family of unary operators, which models the semantics of the first-order existential quantification. Let $\{x\}, X, Y \subseteq \mathcal{V}$ be sets of variables. The existential quantification is an algebraic function, that applied to a constraint φ generates another constraint $\exists_X \varphi$ projecting out from φ all the information about the variables X.

We define a cylindrification operator on the atoms of the Allen language and show that it satisfies the axioms of the cylindrification operation of the cylindric algebra (cf. sec.3.2). Let $q(x)$ be the unary predicates and $p(x, y)$ the binary predicates defined in 4.1.1. The projection of such predicates is defined as the following:

$EQ_1.\ \exists_x q(x) = \textbf{true}$

$$EQ_2.\ \exists_x p(x,y) = \begin{cases} not\_time(y) & iff \quad p \in \{before, after, meets, is\_met\_by, \\ & \quad\ contains, started\_by, finished\_by\} \\ not\_point(y) & iff \quad p \in \{during\} \\ not\_point(y) \wedge not\_time(y) & iff \quad p \in \{overlap, overlapped\_by, starts, \\ & \quad\ finishes\} \end{cases}$$

EQ_1 and EQ_2 define the existential quantifier for the atomic formulas of $\mathcal{P}_{time}$. To have a complete definition of the existential quantifier, consider constraints in $\mathcal{P}_{time}$. Since the operators $\wedge$, $\vee$ satisfy the properties of associativity and distributivity, a constraint in $\mathcal{P}_{time}$ can be transformed into the disjunctive normal form (a set of conjunctions). Since the existential quantifier distributes on disjunction, the complete definition

of the operator $\exists_X$ is obtained only considering it for conjuntions of predicates. We do it inductively, first considering a conjunction of two predicates.

We use the transitivity table defined by Allen in [11]. We denote the table with $T_{time}[i,j]$, where i and j correspond respectively to the rows and columns of the table. $T_{time}[i,j]$ has twelve rows and twelve columns, each of them corresponds to one of the predicates described in 4.1.1. Each entry $T_{time}[i,j]$ defines a formula equivalent to $p_i(x,y) \wedge p_j(y,z)$, where new relations between x and z are deduced.

Example 4.1 The entry $T_{time}[1,1]$ of the transitivity table defines an equivalent formula for $(before(y_1,x) \wedge before(x,y_2))$ such that:

$$T_{time}[1,1] = (before(y_1,x) \wedge before(x,y_2)) = before(y_1,y_2)$$

In the resulting formula the variable x is eliminated. □

Conjunctions of two formulas in the transitivity table can be used to define formulas that are equivalent to existentially quantified formulas but the variables in the scope of quantifier are be eliminated. Let $p_1, p_2, \ldots p_{12} \in \mathcal{P}$, where the $p_i s$ are the two-place predicates in 4.1.1, the following equivalences hold:

$$\exists_x (p_i(x,\, y_1) \wedge p_j(x,\, y_2)) = \exists_x p_i(x,\, y_1) \wedge \exists_x p_j(x,\, y_2) \wedge T_{time}[i,j]$$

where $\exists_x p_2(x,y_2)$ are defined by EQ_1 and EQ_2.

Therefore, for an arbitrary sequence of conjunctions, whose predicate share the same variable x, we have:

$$EQ_3.\ \exists_x(\bigwedge_{i=1}^{n} p_i(x,y_i)) = \bigwedge_{i=1}^{n} \exists_x p_i(x,y_i) \wedge \bigwedge_{k=1}^{\binom{n}{2}} (T_{time}[h,j])_k$$

where $h, j \in \{1, \ldots, n\}$.

4.2 Spatial Constraint System

In this section we define a constraint system for spatial relationships between spatial objects, namely *binary topological relationships* from [5]. Constraints are interpreted over a predefined domain of spatial objects $\mathcal{O}$.

4.2.1 Spatial Domain and Predicates

Let $\mathcal{O}$ be a set of well-defined spatial objects such as points, lines and polygons. We don't not make for the beginning any assumption on the representation of the objects. We assume that two functions *bound* and *int* are defined, that return respectively the boundary and the interior of the object. Furthermore, assume that:

$bound, int : \mathcal{O} \rightarrow \mathcal{T}$

$\mathcal{T} = \{(x,y) : x, y \in \mathcal{R}\}$

on $\mathcal{T}$ the operation $\cap$ intersection is defined.

In the follow, we denote with $o_1, o_2, \ldots, o_n$ elements of $\mathcal{O}$.

We use the set of binary topological relationships between the objects in $\mathcal{O}$ as it is defined in [6, 5], and the corresponding interpretation of them over $\mathcal{O}$. All the relations are shown in the fig. 4.

s_1. $equal(o_1,o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) \neq \emptyset \wedge bound(o_1) \cap int(o_2) = int(o_1) \cap bound(o_2) = \emptyset \wedge int(o_1) \cap int(o_2) \neq \emptyset$

s_2. $disjoint(o_1,o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) = bound(o_1) \cap int(o_2) = int(o_1) \cap bound(o_2) = int(o_1) \cap int(o_2) = \emptyset$

s_3. $touch(o_1, o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) \neq \emptyset \wedge bound(o_1) \cap int(o_2) = int(o_1) \cap bound(o_2) = int(o_1) \cap int(o_2) = \emptyset$

s_4. $inside(o_1, o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) = \emptyset \wedge bound(o_1) \cap int(o_2) \neq \emptyset \wedge int(o_1) \cap bound(o_2) = \emptyset \wedge int(o_1) \cap int(o_2) \neq \emptyset$

s_5. $contains(o_1, o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) = \emptyset \wedge bound(o_1) \cap int(o_2) = \emptyset \wedge int(o_1) \cap bound(o_2) \neq \emptyset \wedge int(o_1) \cap int(o_2) \neq \emptyset$

s_6. $covered\_by(o_1, o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) \neq \emptyset \wedge bound(o_1) \cap int(o_2) \neq \emptyset \wedge int(o_1) \cap bound(o_2) \neq \emptyset \wedge int(o_1) \cap int(o_2) \neq \emptyset$

s_7. $covers(o_1, o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) \neq \emptyset \wedge bound(o_1) \cap int(o_2) = \emptyset \wedge int(o_1) \cap bound(o_2) \neq \emptyset \wedge int(o_1) \cap int(o_2) \neq \emptyset$

s_8. $intersect(o_1, o_2) = true \Leftrightarrow bound(o_1) \cap bound(o_2) \neq \emptyset \wedge bound(o_1) \cap int(o_2) \neq \emptyset \wedge int(o_1) \cap bound(o_2) \neq \emptyset \wedge int(o_1) \cap int(o_2) \neq \emptyset$

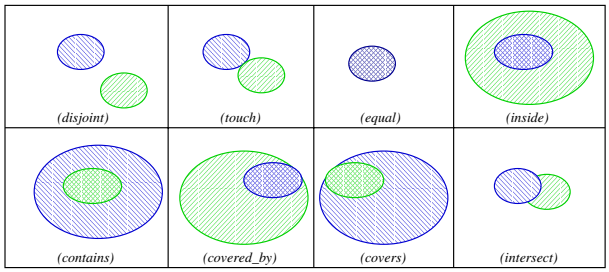


Figure 4: Basic topological relationships between spatial objects embedded in $\mathcal{R}^2$

We need some unary predicates to complete our language:

s_9. $point(o) = true \Leftrightarrow int(o) = \emptyset$

s_{10}. $not\_point(o) = true \Leftrightarrow int(o) \neq \emptyset$

s_{11}. $space(o) = true \Leftrightarrow bound(o) = \emptyset$

s_{12}. $not\_space(o) = true \Leftrightarrow bound(o) \neq \emptyset$

Consider the set of all well-formed formulas which are built up from the above predicates and the connectives $\{\wedge, \vee, \exists\}$. The set contains the tautology **true** and the inconsistent formula **false**. Let $\mathcal{P}_{space}$ be such a set. Like in the temporal case in 4.1.1, we define the quotinet $\mathcal{P}_{space}$. The algebraic structure ($\mathcal{P}_{space}$, $\wedge$, $\vee$, **true**, **false**) forms a semiring.

4.2.2 Cylindrification of Spatial Formulas

We define in the following the cylindrification operator on the atoms of $\mathcal{P}_{space}$. Let $q(x)$ be the unary predicates and $p(x, y)$ the binary predicates defined in 4.2.1, we define the projection of such predicates as the following:

SQ_1. $\exists_x q(x) = \mathbf{true}$

SQ_2. $\exists_x p(x, y) = \begin{cases} not\_space(y) & iff \quad p \in \{disjoint, touch, contains\} \\ not\_point(y) & iff \quad p \in \{inside\} \\ not\_point(y) \wedge not\_space(y) & iff \quad p \in \{covered, covers, intersect\} \end{cases}$

The following table $T_{space}[i, j]$ is the transitivity table defined for the predicates $s_1 - s_8$ described in 4.2.1. p_i and p_j are one of the binary predicates and x, y, z are objects of $\mathcal{O}$. $p_i(x, y)$ is the predicate on the raws and $p_j(y, z)$ the one of the columns. The conjunction $p_i(x, y) \wedge p_j(y, z)$ infers new conditions on the objects x, z as described below:

$p_1(x,y) \wedge$ $p_2(y,z)$	$disjoint$	$touch$	$inside$	$contain$	$covered\_by$	$cover$	$intersect$
$disjoint$ (d)	all	d,t,i cov_by,int	d,t,i cov_by,int	d	d,t,i cov_by,int	d	d,t,i cov_by,int
$touch$ (t)	d,t,con cov,int	d,t,cov cov_by,int	i,cov_by int	d	i,cov_by int	d,t	d,t,i cov_by,int
$inside$ (i)	d	d	i	d,t,i,con cov_by,cov,int	i	d,t,i cov_by,int	d,t,i cov_by,int
$contain$ (con)	d,t,con cov,int	con,cov int	i,con,cov_by cov,int	con	con,cov int	con,cov int	con,cov int
$covered\_by$ (cov_by)	d	d,t	i	d,t,con cov,int	i,cov_by	d,t,cov_by cov,int	d,t,i cov_by,int
$cover$ (cov)	d,t,con cov,int	t,con cov,int	i,cov_by int	con	cov_by,cov int	con,cov	con,cov int
$intersect$ (int)	d,t,con cov,int	d,t,con cov,int	i,cov_by int	d,t,con cov,int	i,cov_by int	d,t,con cov,int	d,t,i,con cov_by,cov,int

Table 1: Transitivity table $T_{space}[i,j]$ for Egenhofer relationships.

The existential quantifier is defined in the same way as in 4.1.2, but in this case we use the table 1. Therefore, for an arbitrary sequence of conjunctions, whose predicate share the same variable x, we have:

$$SQ_3.\ \exists_x(\bigwedge_{i=1}^{n} p_i(x,y_i)) = \bigwedge_{i=1}^{n} \exists_x p_i(x,y_i) \wedge \bigwedge_{k=1}^{\binom{n}{2}} (T_{space}[h,j])_k$$

where $h,j \in \{1,\ldots,n\}$.

It can be seen that the operators $\exists_X$ defined above and in sec. 4.1.2 satisfy the axioms $C_1 - C_5$, where $\oplus = \vee$ and $\otimes = \wedge$. For instance, C_2 states that the constraint $\exists_X \varphi$ is weaker than φ. In the Allen constraint system, this means that the set of intervals satisfying $\exists_X \varphi$ contains all intervals satisfying φ. Since both our constraint systems are logic-based and the formula $\varphi \rightarrow \exists_X \varphi$ is the well-known tautology, the property C_2 holds in our constraint systems.

5 Conclusion

We have introduced a formal background for a component-based programming tool, which enables to use constraints as basic data types in an object-relational DBMS. We have used cylindric algebras as a backbone since it allows to formalize the notion of "constraint objects". It preserves most of important ingredients of constraint programming. We believe that a suitable Java-based implementation of the cylindric algebra can be done as an abstract super class of all constraints in the system. This would allow to guide an implementation of particular constraints. In particular, type checking facilities of the Java compiler can be used. In other words, out ultimate goal is a component-based programming environment for developing different constraint databases. Basic elements of such an environment have been illustrated using temporal and spatial objects.

Most of existing constraint databases exploit the same idea: a restricted class of first order formulas can be given a reasonably efficient operational semantics. For instance, it is used to obtain the so-called constraint algebras and then to compile them into extended relational operators. However, in our case, we have a different goal: given an object-relational database schema, construct its light-weight wrapping as a set of constraints. The "light-weight wrapping" means that an interpretation of constraint-based data manipulation statements should be done directly in the DBMS. Our implementation

exploits Java-based database programming tools (such as SQLJ and JPub). These tools enable us to extend classes of available persistent objects. As it can be seen from sec. 4, the wrapping operations are not very complex (except the quantifier elimination, which, in our settings is not obligatory). Therefore, we hope that our wrapping technology will not undermine efficiency of the underlying DBMS.

References

[1] http://technet.oracle.com/doc.

[2] A.V.Aho, J.E.Hopcroft, and J.D.Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley Publishing Company, 1974.

[3] A. Brodsky and Y. Kornatzky. The l_{yric} language: Querying constraint objects. In *Int. Conf. on Management of Data*, ACM SIGMOD, 1995.

[4] A. Brodsky and V.E. Segal. The c^3 constraint object-oriented database system: An overview. In Gaede et al., editor, *Constraint Databases and Applications*, volume 1191 of *LCNS*, pages 134–159, Delphi, Greece, January 1997. Springer-Verlag.

[5] M. Egenhofer. A model for detailed binary topological relationships. *Geomatica*, 47(3 and 4):261–273, 1993.

[6] M. Egenhofer and R. Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5(2):161–174, 1991.

[7] D.Q. Goldin and P.C. Kanellakis. Constraint query algebras. *Constraint Journal, 1st issue*, pages 45–83, 1996.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] S. Grumbach, P. Rigaux, and L. Segoufin. The *dedale* system for complex spatial queries. In *Intl. Conf. on the Management of Data*, SIGMOD, pages 213–224, 1998.

[10] S. Grumbach, J. Su, and C. Tollu. Linear constraint query languages: Expressive power and complexity. In D. Leivant, editor, *Logic abd Computational Complexity*, Indianapolis, 1994. Springer Verlag. LNCS 960.

[11] Allen J.F. Maintaining knowledge about temporal intervals. In *Communication of the ACM*, volume 26 (11), pages 832–843. November 1983.

[12] Allen J.F. Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6(4):341–356, July 1991.

[13] J. Paradaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 279–288, 1994.

[14] P.C.Kanellakis, G.M.Kuper, and P.Z.Revesz. Constraint query languages. In *Proc. 9th ACM PODS*, pages 299–313, 1990.

[15] S.Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.

On the Complexity of Generalized Horn Clause Intuitionistic Logic

Clemens Beckstein and Manfred Rahneberg

Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik,
Institut für Informatik, D-07740 Jena, Germany,
Email: beckstein@informatik.uni-jena.de and Manfred.Rahneberg@softlab.de

Abstract There is an interesting syntactical generalization of Horn clauses—the so called generalized Horn clauses. Generalized Horn clauses are obtained from regular Horn clauses by permitting conditional goals in queries and clause bodies. Logic programming with generalized Horn clauses can be understood as logic programming within intuitionistic logic. This raises the question whether intuitionistic consequence is decidable in the propositional case and, if this is the case, then at what cost. For regular Horn clauses the answer to this question is well known: logical consequence for propositional Horn clauses can be decided in linear time. In this paper we will sketch a procedure that will deterministically decide intuitionistic consequence for propositional generalized Horn clauses in polynomial space but exponential time. We will also show in detail that the corresponding decision problem itself is PSPACE complete.

1 Introduction

Logic programming traditionally is based on the logic of Horn clauses. Combining the logic of Horn clauses with negation as failure reasoning resulted in the development of the logic programming PROLOG—a very successful language of the fifth generation.

But the modest expressivity of Horn clauses nevertheless makes it hard to naturally support many programming techniques which are desirable for logic programming. One example of a useful technique which is not directly supported by Horn clause logic is that of experimenting with a theory [2]. A user of a logic programming system might ask a question like 'Will this query have a positive answer if I add this clause to my program?' by performing three step experiments of the following type:

1. modify the program by adding or removing certain clauses
2. use the resulting program (in the following also called the context of the query) to answer the query (this may result in the deduction of lemmas), and,
3. once the answer is known, undo all the changes from step 1 and 2 along with their consequences[1].

In traditional (Horn clause) logic programming environments step 1 and step 3 are done using meta-predicates like *assert* and *retract*. With these predicates undoing the changes from step 1 is comparably easy. But it is much more difficult to undo the changes that the program resulting from step 1 makes to itself during the evaluation of the query in step 2. Here the programmer is on its own. The system won't give him any help in identifying the right clauses to add or remove.

[1] Step 3 ensures that experiments don't irreversibly change the program so another experiment can be conducted with the original program.

Shortcomings of this type have raised interest in more expressive logics for logic programming which nevertheless preserve most of the computational simplicity of Horn clause logic. One type of logic that has been considered for logic programming is intuitionistic logic. In intuitionistic logic experiments with a theory are directly supported via implications that may be embedded in goals and in the bodies of clauses: attempting to prove a goal of the form $D \to G$ from the context Γ will then automatically result in an attempt to prove G in the (extended) context $\Gamma \cup \{D\}$.

One instance of an intuitionistic logic supporting this type of reasoning is that of hereditory Harrop formulas [13, 11]. Another one is the subset of hereditory Harrop formulas which we call generalized Horn clauses [15, 17]. Both logics extend the logic of Horn clauses by permitting embedded implications and the first one even allows to have other than universally quantified variables in formulas. For both of these logics logic programming languages were developed: a language based on hereditory Harrop formulas is λProlog [14, 12, 13, 11, 10] and one based on generalized Horn clauses is Risc [1, 4, 3] which itself was inspired by (Q)NProlog [8, 6].

In this paper we will address the following complexity problem for intuitionistic logic: *Given a set P of propositional generalized Horn clauses and an atomic formula G, what is the complexity class of the decision problem whether G is an intuitionistic consequence of P?*

For our answer to this question we will restrict ourselves to the generalized Horn clauses subset of the hereditory Harrop formulas. Logic programs consisting of generalized Horn clauses allow for the use of a comparably simple query answering procedure whose complexity we will determine in this paper as well.

In the following section 2 we will introduce intuitionistic logic and several definitions that are necessary for the following sections. Section 3 will be devoted to the exposition of a decision procedure for propositional generalized Horn clause theories. The central complexity result will then be presented in section 4.

2 Logic programming in intuitionistic logic

Deciding logical consequence for Horn clause theories containing function symbols is known to be undecidable. Regular Horn clauses are already expressive enough in order to be able to simulate the working of a universal Turing machine. It can therefore be expected that logical consequence for any superclass of Horn clauses is also undecidable.

Logical consequence for propositional Horn clause theories (Datalog) on the other hand is even decidable in linear time—as was shown e.g. in [5, 16]. The question we are interested in this paper is whether a similar result holds for propositional generalized Horn clause theories.

Following Gabbay [7], intuitionistic consequence in the propositional case can be characterized as follows: Let P be a set of formulas. The smallest relation

$$\models_I \subseteq 2^P \times P,$$

satisfying condition 1–4 is called an intuitionistic consequence relation of P:

1. If $A \in P$, then $P \models_I A$.
2. If $P \models_I A$, then $P \cup \{B\} \models_I A$.
3. If $P \models_I A$ and $P \cup \{A\} \models_I B$, then $P \models_I B$
4. $P \cup \{A\} \models_I B$ iff $P \models_I B \leftarrow A$

Note that adding to this definition the principle of the *excluded middle*

$$\models_I A \vee \neg A,$$

or the principle of the *double negation*,

$$\neg\neg A \models_I A,$$

transforms it into a definition of classical logic consequence.

Nadathur has shown in [13] how a special class of fomulas — the first order Hereditory Harrop formulas — can be used to do intuitionistic logic programming. Hereditory Harrop formulas are either G-formulas (goals) or D-formulas (data). They can be defined inductively by the following two syntactic rules

$$G \equiv A \mid G \wedge G \mid G \vee G \mid D \to A \tag{1}$$
$$D \equiv A \mid G \to A \mid D \wedge D. \tag{2}$$

where A stands for an atomic formula.

In this paper we focus on the subset of the set of hereditory Harrop formulas which consists of the so called generalized Horn clauses:

Definition 1 (Generalized Horn clauses). *Let A be an atomic formula, then G-formulas and D-formulas are defined by the following two syntactic rules:*

$$G \equiv A \mid G \wedge G \mid D \to A$$
$$D \equiv A \mid G \to A. \tag{3}$$

Hence, D-formulas (clauses of a program) are generalized Horn clauses and G-formulas (goals for a program) are conjunctions of generalized Horn clauses.

Definition 2 (Intuitionistic query). *A pair $\langle G, P\rangle$ where G is a G-formula and P a finite set of D-formulas is called an intuitionistic query with goal G and context P.*

Using these definitions the calculus of hereditory Harrop formulas from [13] can be tailored for propositional generalized Horn clauses in the following way: A set $\mathcal{G}$ of intuitionistic queries can be understood as the state of a proof procedure for generalized Horn clause logic that is just evaluating the queries in $\mathcal{G}$. A derivation within intuitionistic logic can therefore be modeled as a sequence of state transformations that follows certain rules which ensure soundness:

Definition 3 (Direct derivability for intuitionistic logic). *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two sets of intuitionistic queries. $\mathcal{G}_2$ is intuitionistically derivable from $\mathcal{G}_1$ (written as $\mathcal{G}_1 \vdash_I \mathcal{G}_2$), if one of the following holds:*

1. $\langle G_1 \wedge G_2, P\rangle \in \mathcal{G}_1$ *and*

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle G_1 \wedge G_2, P\rangle\}) \cup \{\langle G_1, P\rangle, \langle G_2, P\rangle\}$$

2. $\langle D \to G, P\rangle \in \mathcal{G}_1$ *and*

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle D \to G, P\rangle\}) \cup \{\langle G, P \cup \{D\}\rangle\}$$

3. $\langle A, P\rangle \in \mathcal{G}_1$*; A Atom and $A \in P$, then*

$$\mathcal{G}_2 = \mathcal{G}_1 - \{\langle A, P\rangle\}$$

4. $\langle A, P\rangle \in \mathcal{G}_1$*; A Atom and $G \to A \in P$ and*

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle A, P\rangle\}) \cup \{\langle G, P\rangle\}$$

Definition 4 (Derivability in intuitionistic logic). *A sequence $\mathcal{G}_1, \ldots, \mathcal{G}_n$ is called a* derivation *for generalized Horn clauses, if $\mathcal{G}_{i+1}$ is directly intuitionistic derivable from $\mathcal{G}_i$ for each $i \in \{1, \ldots, n-1\}$. The derivation terminates if there is no state which can be derived from $\mathcal{G}_n$ i.e. if none of the four rules in definition 3 is applicable. It terminates successfully if $\mathcal{G}_n = \emptyset$.*

Let G be a G-formula, P a set of D-formulas and $\mathcal{G}_1 = \{\langle G, P\rangle\}$. Then a successful derivation $\mathcal{G}_1, \ldots, \mathcal{G}_n$ is called a derivation for G from P, written

$$P \vdash_I G.$$

Searching for a proof for a goal formula G from a program P of D-formulas can therefore be understood as a search for a derivation for G from P.

Example 1. The atomic query A to the program P ($P \vdash_I A?$) of generalized Horn clauses

$$\begin{aligned} &(B \to C) \to A \\ &B \to C \end{aligned} \tag{4}$$

can be answered positively by the following derivation[2]

$$\begin{aligned} (A, P) &\vdash_I (B \to C, P) \\ &\vdash_I (C, P \cup \{B\}) && (5)\\ &\vdash_I (B, P \cup \{B\}) && (6)\\ &\vdash_I \emptyset. && (7) \end{aligned}$$

The following theorem was proven in [15]:

Theorem 1. *Let P be a set of generalized Horn clauses and A an atomic goal formula. Then*

$$P \models_I A \quad \textit{iff} \quad P \vdash_I A.$$

We will now look closer at the problem of effectively deciding whether an atomic query intuitionistically follows from a set of generalized Horn clauses.

3 Deciding intuitionistic consequence for generalized Horn clauses

An interesting subclass of the generalized Horn clauses are the so called G_1-formulas—generalized Horn clauses with no more than one embedded implication:

Definition 5 (G_1-formulas). *A G_1-formula is either a definite clause or a clause of the form $(B \to D) \longrightarrow A$, where A, B, and D are atomic formulas.*

For G_1-formulas the following proposition holds (a proof can be found in [17, 15]) if the size $|P|$ of a set P of generalized Horn clauses is measured as the number of variables in P (multiple occurrences of variables counted multiply):

Proposition 1. *It is possible to effectively transform an arbitrary set P of generalized Horn clauses into an intuitionistically equivalent set P' of G_1-formulas. This transformation can be done*

1. *in time proportional to $|P|$ and*
2. *with $|P'| \leq 3 \times |P|$.*

[2] When it is clear from the context we abbreviate states containing just one pair (G, P) by just this pair.

Definition 6 (Direct derivability for G_1-formulas). *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two sets of intuitionistic queries where the contexts consist just of G_1-formulas. $\mathcal{G}_2$ is intuitionistically G_1-derivable from $\mathcal{G}_1$ (written as $\mathcal{G}_1 \vdash_{I_1} \mathcal{G}_2$), if the rules from definition 3 holds where rule 2 is replaced by the following rule:*

2'. $\langle D \to G, P\rangle \in \mathcal{G}_1$ *and*

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle D \to G, P\rangle\}) \cup \{\langle G, P' \cup \{D\}\rangle\}$$

where

$$P' = P - \{(D \to G') \to G'' \mid G', G'' \, atoms\} \cup \{G' \to G'' \mid (D \to G') \to G'' \in P\}.$$

In analogy to definition 4 we use the term "intuitionistic G_1-derivability" for the transitive closure of "direct G_1-derivability".

For this type of derivability apparently the following holds:

Proposition 2. *An atom is intuitionistically G_1-derivable from a set P of G_1-formulas iff it is intuitionistically derivable from P.*

Propositions 1 and 2 together imply that we can—without loss of generality—restrict our attention to the class of G_1-formulas: any interesting complexity or decision result that holds for generalized Horn clauses also must hold for G_1-formulas and vice versa.

Theorem 2. *If P is a finite set of propositional generalized Horn clauses and A a propositional atom, then the problem $P \models_I A?$ is decidable.*

Proof. According to [5, 16] the problem $P \models A?$ is decidable if P just contains (regular) Horn clauses. In the following we will only sketch the proof of the corresponding result for generalized Horn clauses—its exact details can be found in [15] where a generalization of the decision procedure in [5, 16] to G_1-formulas is described and analyzed.

Because of theorem 1 the problem $P \models_I A?$ is decidable iff $P \vdash_I A?$ is decidable, i.e. if there is a search procedure that will find an intuitionistic derivation of A from P iff A is intuitionistically derivable from the generalized Horn clause program P.

On the other hand, due to proposition 2, $P \vdash_I A?$ is decidable iff $P \vdash_{I_1} A?$ is decidable. So let us look at derivations starting with the state $\{\langle A, P\rangle\}$ which are allowed by definition 6 but not forbidden by the loop preventing marking techniques in [15].

Let m be the number of different variables occurring in P. Then any G_1-derivation from P can contain no more than m applications of rule 2' from definition 6 (the contexts of queries later in the derivation must consist only of regular Horn clauses). These applications of rule 2' are interleaved by sequences of regular Horn clause derivation steps (rules 1,3, and 4). The marking techniques in [5, 16, 15] ensure that these sequences are aperiodic and therefore of finite and a priori bounded length.

Hence, any G_1-derivation permitted by the mentioned marking techniques must terminate after finitely many steps and only a finite and a priori bounded number of them does exist. We can therefore recursively enumerate all of them and effectively decide whether one of them terminates in the empty state $\emptyset$ which is equivalent to the problem $P \vdash_I A?$. Therefore $P \models_I A?$ is decidable as well. □

Note that if a program P of G_1-formulas contains m embedded implications then P can be extended by applications of rule 2 of definition 3 (or rule 2' of definition 6) in no more than $\binom{m}{i}$ different ways. There are therefore up to 2^m ways to generate contexts from P.

It is sometimes necessary to visit all contexts which could possibly be encountered in a derivation starting from a generalized Horn clause theory:

Example 2. Let A be an atom and P be the following set

$$\begin{aligned} a_1 \wedge \ldots \wedge a_m &\rightarrow A, \\ (B_1 \rightarrow A) &\rightarrow a_1, \\ B_1 \wedge a_1 &\rightarrow A, \\ &\vdots \\ (B_m \rightarrow A) &\rightarrow a_m, \\ B_m \wedge a_m &\rightarrow A. \end{aligned}$$

of G_1-formulas. Any query processor for intuitionistic logic is forced to examine derivations containing all the 2^m possible contexts before it safely can know that the query $P \vdash_I A$? eventually must fail.

This suggests that there are generalized Horn clause theories and queries where intuitionistic consequence probably cannot be decided in polynomial time. In the next section we will look closer at this conjecture.

4 The complexity class of the decision problem belonging to generalized Horn clauses

Before we can formally state our complexity result we first must introduce some well known notation and several useful complexity results:

Let K be the set of closed quantified Boolean formulas and $3K \subseteq K$ the set of closed quantified Boolean formulas with kernel in 3KNF (i.e. with a matrix having the form of a clause with exactly three literals). It is well known (see e.g. [9]) that the decision problems belonging to

$$QBF := \{F \in K \mid \text{F is true}\}.$$

and

$$3QBF := \{F \in 3K \mid \text{F is true}\}.$$

both are PSPACE complete.

Using these results we will now have a closer look at the decision problem

$$\begin{aligned} \text{GHORN-SAT} := \{(\text{P,A}) \mid\ & P \text{ is a set of generalized Horn clauses} \\ & \text{and } A \text{ is an atomic formula such that } P \models_I A\}. \end{aligned}$$

Proposition 3. *GHORN-SAT $\in$ PSPACE.*

Proof. Because of theorem 1 we know that $P \models_I A$ holds iff $P \vdash_I A$. Hence GHORN-SAT $\in$ PSPACE iff intuitionistic derivability is decidable in polynomial space.

The fact that intuitionistic derivability is polynomially (wrt. time and space) reducable to intuitionistic G_1-derivability (propositions 1 and 2) implies that GHORN-SAT is in PSPACE iff P' is a G_1-transform of P according to proposition 1 and $P' \vdash_{I_1} A$ is decidable in polynomial space.

Intuitionistic G_1-derivability can be decided in polynomial space with the deterministic procedure in [15] that we sketched in the proof of theorem 2. GHORN-SAT therefore is in PSPACE. □

Proposition 4. *3QBF $\leq_m^P$ GHORN-SAT.*

Proof. We will construct a polynomial reduction of the PSPACE complete problem class 3QBF to GHORN-SAT:

Let F be a closed quantified Boolean formula

$$F = (Q_1 x_1)(Q_2 x_2) \ldots (Q_k x_k) F'(x_1, x_2, \ldots, x_k)$$

where F' denotes a Boolean formula in 3KNF with its free variables $x_1, x_2, \ldots, x_k$ and $Q_i \in \{\forall \exists\}$. F' has the form $D_1 \wedge \ldots \wedge D_s$ with $D_i = l_{i1} \vee l_{i2} \vee l_{i3}$ and $l_{ij} \in \{x_1, x_2, \ldots, x_k, \neg x_1, \neg x_2, \ldots, \neg x_k\}$ $(1 \leq i \leq s, 1 \leq j \leq 3)$.

We define a function f on F as follows:

$$f(F) = (P, c_0)$$

where

$$P = P_1 \cup \ldots \cup P_k \cup \{d_1 \wedge \ldots \wedge d_k \rightarrow c_k\} \cup \{q_{ij} \mid 1 \leq i \leq s, 1 \leq j \leq 3\}$$

with

$$P_i = \begin{cases} \{(a_i \rightarrow c_i) \wedge (b_i \rightarrow c_i) \rightarrow c_{i-1}\} & \text{if } Q_i = \forall, \\ \{(a_i \rightarrow c_i) \rightarrow c_{i-1}, (b_i \rightarrow c_i) \rightarrow c_{i-1}\} & \text{if } Q_i = \exists \end{cases}$$

and

$$q_{ij} = \begin{cases} a_t \rightarrow d_i & \text{if } l_{ij} = x_t, \\ b_t \rightarrow d_i & \text{if } l_{ij} = \neg x_t. \end{cases}$$

$a_i, \ldots, d_i$ are atomic formulas. Apparently f can be computed in polynomial time. We will now show the following proposition:

$$F \in \text{3QBF} \quad \text{iff} \quad f(F) \in \text{GHORN-SAT},$$

i.e.

$$F \in \text{3QBF} \quad \text{iff} \quad P \models_I c_0.$$

f is defined in such a way that P consists only of generalized Horn clauses. It is therefore sufficient to show

$$F \in \text{3QBF} \quad \text{iff} \quad P \vdash_I c_0.$$

We will show just the case where the prefix $Q_1 Q_2 \ldots Q_k$ of F has the form $\exists \forall \exists \forall \ldots$ because each other case can be handled in exactly the same way.

1. If $F \in$ 3QBF then $P \vdash_I c_0$:

If $F \in$ 3QBF then there is a tree of valuations, such that each path defines values for $x_1, x_2, \ldots, x_k$ satisfying F'.

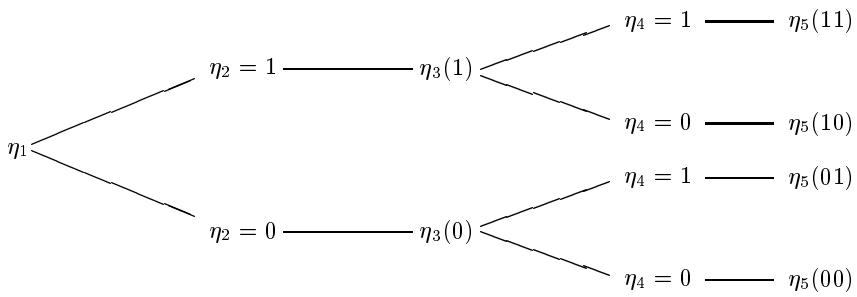


Figure1.

This valuation tree suggests a derivation with the following first step:

$$(c_0, P) \vdash_I (e_1 \to c_1, P)$$

where

$$e_1 = \begin{cases} a_1 & \text{if } \eta_1 = 1, \\ b_1 & \text{otherwise.} \end{cases}$$

The following steps are

$$\begin{array}{l} \vdash_I (c_1, P \cup \{e_1\}) \\ \vdash_I ((a_2 \to c_2) \wedge (b_2 \to c_2), P \cup \{e_1\}) \\ \vdash_I \{(a_2 \to c_2, P \cup \{e_1\}), (b_2 \to c_2, P \cup \{e_1\})\} \\ \vdash_I \{(c_2, P \cup \{e_1, a_2\}), (c_2, P \cup \{e_1, b_2\})\}. \end{array}$$

If

$$e_{2j+1}(e_2 e_4 \ldots e_{2j}) = \begin{cases} a_{2j+1} & \text{if } \eta_{2j+1}(\eta_2 \eta_4 \ldots \eta_{2j}) = 1, \\ b_{2j+1} & \text{otherwise} \end{cases}$$

and

$$P_{e_2 e_4 \ldots e_{k-1}} = P \cup \{e_1, e_2, e_3(e_2), e_4, e_5(e_2, e_4), \ldots, e_k(e_2, e_4, \ldots, e_{k-1})\}$$

then eventually we will get

$$\vdash_I \{(d_i, P_{e_2 e_4 \ldots e_{k-1}}) \mid 1 \leq i \leq s \wedge e_2 \in \{a_2, b_2\} \wedge \ldots \wedge e_{k-1} \in \{a_{k-1}, b_{k-1}\}\}.$$

Each extended context $P_{e_2 e_4 \ldots e_{k-1}}$ corresponds to a path S in the valuation tree: $P_{e_2 e_4 \ldots e_{k-1}}$ was constructed, so that the pair $(d_i, P_{e_2 e_4 \ldots e_{k-1}})$ can be deleted from the derivation if S (interpreted as values for $x_1, x_2, \ldots, x_k$) satisfies F'.

All paths in the tree satisfy F'. Hence, all pairs $(d_i, P_{e_2 e_4 \ldots e_{k-1}})$ can be removed from the derivation and eventually $\emptyset$ can be derived.

2. If $P \vdash_I c_0$ then $F \in$ 3QBF:

If $P \vdash_I c_0$ then there is a derivation

$$(c_0, P) \vdash_I \ldots \vdash_I \emptyset.$$

Let us look closer at one such derivation α. The first step of α is either

$$(c_0, P) \vdash_I (a_1 \to c_1, P)$$

or

$$(c_0, P) \vdash_I (b_1 \to c_1, P).$$

In the first case we set $e_1 = a_1$ and $\eta_1 = 1$, in the latter $e_1 = b_1$ and $\eta_1 = 0$. Now, the next steps are

$$\begin{array}{ll} (c_0, P) & \vdash_I (e_1 \to c_1, P) \\ & \vdash_I (c_1, P \cup \{e_1\}) \\ & \vdash_I ((a_2 \to c_2) \wedge (b_2 \to c_2), P \cup \{e_1\}) \\ & \vdash_I \{(a_2 \to c_2, P \cup \{e_1\}), (b_2 \to c_2, P \cup \{e_1\})\} \\ & \vdash_I \{(c_2, P \cup \{e_1, a_2\}), (c_2, P \cup \{e_1, b_2\})\}. \end{array}$$

Reversing the argument in the first part of our proof shows that α determines

$$e_1, e_2, e_3(e_2), e_4, e_5(e_2, e_4), \ldots, e_k(e_2, e_4, \ldots, e_{k-1})$$

and

$$\eta_1, \eta_2, \eta_3(\eta_2), \eta_4, \eta_5(\eta_2, \eta_4), \ldots, \eta_k(\eta_2, e_4, \ldots, \eta_{k-1}),$$

resulting in a valuation tree T like in the first part of the proof.

At last, using the same notation as above, α delivers

$$\vdash_I \{(d_i, P_{e_2 e_4 \ldots e_{k-1}}) \mid 1 \leq i \leq s \wedge e_2 \in \{a_2, b_2\} \wedge \ldots \wedge e_{k-1} \in \{a_{k-1}, b_{k-1}\}\}.$$

Since α is a successful derivation terminating in $\emptyset$ all pairs $(d_i, P_{e_2 e_4 \ldots e_{k-1}})$ with fixed $e_2 e_4 \ldots e_{k-1}$ can be eliminated from α. But this implies that for each d_i there is a value u in the set

$$\{e_1, e_2, e_3(e_2), e_4, e_5(e_2, e_4), \ldots, e_k(e_2, e_4, \ldots, e_{k-1})\}$$

belonging to $(d_i, P_{e_2 e_4 \ldots e_{k-1}})$ which can be used to eliminate $(d_i, P_{e_2 e_4 \ldots e_{k-1}})$ from the derivation α:

$$(d_i, P_{e_2 e_4 \ldots e_{k-1}}) \vdash_I (u, P_{e_2 e_4 \ldots e_{k-1}}) \vdash_I \emptyset.$$

It is therefore possible to satisfy each d_i if values for the variables are chosen wrt. the corresponding path in T:

$$\eta_{2j+1}(\eta_2, \eta_4 \ldots \eta_{2j}) = \begin{cases} 1 & \text{if } e_{2j+1}(e_2 e_4 \ldots e_{2j}) = a_{2j+1}, \\ 0 & \text{otherwise.} \end{cases}$$

The same valuation apparently also satisfies F'. Since this is true for every path in T it follows that $F \in$ 3QBF. □

Propositions 3 and 4 together immediately entail the central theorem of this paper:

Theorem 3. *GHORN-SAT is PSPACE complete.*

Hence, GHORN-SAT really is a very hard decision problem.

5 Conclusion

We have investigated a syntactical generalization of Horn clauses that we called generalized Horn clauses. Generalized Horn clauses are obtained from regular Horn clauses by permitting conditional goals in queries and clause bodies. Logic programming with generalized Horn clauses can be understood as logic programming within intuitionistic logic. We have shown that intuitionistic consequence for theories consisting of propositional generalized Horn clauses is deterministically decidable in polynomial space and that the corresponding decision problem is PSPACE complete.

As a consequence of this result and the well known inclusion NP $\subseteq$ PSPACE any deterministic derivation seeking procedures for logic programming languages which include propositional generalized Horn clause intuitionistic logic probably will exhibit a worst case exponential time complexity.

This especially applies to the query processors of the logic programming languages RISC and λPROLOG. Both languages therefore are already complex enough to make it impossible to design tractable query processors for their propositional specializations.

Allowing nested implications therefore only prima facie looks like a small syntactic change to Horn Clause logic. But from a computational point of view it really means all the difference between interesting in theory and usable in practice.

6 Achnowledgements

We would like to say thank you to Prof. Gerd Wechsung for his valuable comments on an early draft of this paper—they helped a lot to streamline the proof of theorem 3.

References

[1] Beckstein, C.; Klausner, J.: *An Abstract Machine for the Compilation of Logic Programs that Can Guess.* In: *Proceedings of the Workshop on Sequential Implementation Technologies for Logic Programming Languages (held in association with the 1995 International Logic Programming Symposium, ILPS-95)*, Portland, Oregon, December 8th 1995.

[2] Beckstein, C.; Tobermann, G.: *Evolutionary Logic-Programming with* RISC. In: *Proc. of the Fourth International Workshop on Logic Programming Environments (in conjunction with the Joint International Conference and Symposium on Logic Programming, JICSLP 92)*, pages 16–21, Washington, D.C., November 1992. Proceedings published as TR 92-143 from the Center for Automation and Intelligent Systems Research at Case Western Reserve University.

[3] Beckstein, C.; Tobermann, G.: *Algorithmic Debugging and Hypothetical Reasoning. Journal of Automated Software Engineering*, 4:151–178, 1997.

[4] Beckstein, C.; Tobermann, G.; Stolle, R.: *Meta-Programming for Generalized Horn Clause Logic.* In: *Proceedings of the Workshop on Metaprogramming and Metareasoning in Logic, META96 (held in association with the 1996 Joint International Conference and Symposium on Logic Programming, JICSLP-96)*, Bonn, September 2nd–6th 1996.

[5] Dowling, W. F.; Gallier, J. H.: *Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. Journal of Logic Programming*, 3:267–284, 1984.

[6] Gabbay, D. M.: N-PROLOG*: An Extension of Prolog with Hypothetical Implication II. Logical Foundations, and Negation as Failure. The Journal of Logic Programming*, 2(4):251–284, December 1985.

[7] Gabbay, D. M.: *Elements of Algorithmic Proof.* In Abramsky, S.; Gabbay, D. M.; Maibaum, T. S. E., editors: *Handbook of Logic in Computer Science*, pages 331–414. Clarendon Press, Oxford, 1992.

[8] Gabbay, D. M.; Reyle, U.: N-PROLOG*: An Extension of* PROLOG *with Hypothetical Implications I. The Journal of Logic Programming*, 1(4):319–355, April 1984.

[9] Garey, M. R.; Johnson, D. S.: *Computers and Intractability.* Freeman, 1979.

[10] Hodas, J. S.; Miller, D.: *Logic Programming in a Fragment of Intuitionistic Linear Logic. Journal of Information and Computation*, 1994.

[11] Miller, D.: *Abstractions in Logic Programs.* In Odifreddi, O., editor: *Logic and Computer Science.* Academic Press, February 1993.

[12] Miller, D.; Nadathur, G.; Pfennig, F.; Scedrov, A.: *Uniform Proofs as a Foundation for Logic Programming. Annals of Pure and Applied Logic, 51*, 1991.

[13] Nadathur, G.: *A Proof Procedure for the Logic of Hereditary Harrop Formulas.* Technical Report CS-1992-17, Department of Computer Science, Duke University, Durham, North-Carolina, Movember 1992.

[14] Nadathur, G.; Miller, D.: *An Overview of λProlog.* In Bowen, K. A.; Kowalski, R. A., editors: *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, 1988.

[15] Rahneberg, M.: *Beweisverfahren für die intuitionistische Aussagenlogik verallgemeinerter Hornklauseln.* Diplomarbeit, Institut für Informatik, Universität Jena, 1998.

[16] Scutella, M. G.: *A Note on Dowling and Gallier's Top-Down Algorithm for Propositional Horn Satisfiability. The Journal of Logic Programming*, 8:265–273, 1990.

[17] Tobermann, G.: *Verallgemeinerte Hornklauselllogik: vom logischen Kalkül zum Logik-Programmiersystem.* Dissertation, Universität Erlangen-Nürnberg, 1994.

Using the dlv System for Planning and Diagnostic Reasoning*

Thomas Eiter Wolfgang Faber† Nicola Leone Gerald Pfeifer
Axel Polleres

Institut für Informationssysteme, TU Wien
A-1040 Wien, Austria

{eiter,faber}@kr.tuwien.ac.at,{leone,pfeifer,polleres}@dbai.tuwien.ac.at

Abstract

dlv is a state-of-the-art system for disjunctive logic programming, which may be used as an implementation bed for solving declarative knowledge representation problems. In this paper, we review the use of the dlv system for solving diagnostic problems, and contrast this with solving planning problems in dlv. This is possible using a declarative "guess and check" approach: The power of disjunction allows for nondeterministic generation of plans, which are verified in a step by step manner by applying state transition rules. Efficient processing of these phases, in an interleaved mode, will be provided through the algorithms incorporated in the dlv engine. Based on these ideas, we are currently developing a front-end for deductive planning in dlv, which will be included in future releases of dlv.

1 Introduction

Planning and diagnostic reasonings are very important AI tasks. Both fields have been studied quite extensively before, for an overview we refer to [4] (for diagnosis) and to [17, 22] (for planning).

In this paper, we compare the two domains and review the use of the dlv system for solving these problems. We show that both problems exhibit a goal-oriented "guess and check" structure, which allows for a declarative representation in disjunctive datalog, dlv's language.

In particular, disjunctive rules are used for guessing a solution candidate, which are then checked for validity. We would like to point out that dlv does not really perform a naïve approach like this; rather it uses efficient algorithms to interleave the guessing and checking phases as good as possible, such that the generation of candidates can be pruned effectively, see e.g. [8, 12].

Let us first focus on the planning and diagnostic reasoning domains.

*This work was supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.
†Please address correspondence to this author.

1.1 Planning

In planning, given a description of a world, an initial situation, and a desired situation, the goal is to find a sequence of actions (which can change the situations), such that the desired situation is reached from the initial one. In addition, not all actions are applicable in every situations. We formulate our model of planning accordingly:

Definition 1 *A* planning problem *(PP) $\mathcal{P}$ is a quadruple $\langle F, A, I, G\rangle$, where:*

- *F is a set of* fluents*, which characterize the situations*
- *A is a set of* actions*, with a definition of their respective preconditions and effects*
- *I is a set of fluents describing the initial situation*
- *G is a set of fluents describing the goal situation*

□

We define fixed-length solutions to a PP as follows:

Definition 2 *Given a PP $\mathcal{P} = \langle F, A, I, G\rangle$ and an integer n, a* plan *for PP is a sequence $A = a_1 \cdots a_n$ such that there are $n+1$ situations $S_0, \ldots, S_n$, such that for each a_i in A, S_{i-1} is consistent with a_i's preconditions, and S_i is modified from S_{i-1} by exactly the effects of a_i. To make the plan valid for the given situations, S_0 must be entailed by I, and G must be entailed by S_n.* □

Consider the planning problem $\mathcal{P}_{switch} = \langle\{light\_on\}, \{toggle\}, \{\neg light\_on\}, \{light\_on\}\rangle$, where the action *toggle* has no precondition and the effect is that if *light_on* holds, $\neg$*light_on* holds after the action and vice versa. Then there exists a plan of length 1, *toggle*. In fact every sequence of *toggle* actions, whose length is an odd number, is a valid plan.

In this definition (and the example) we did not go into any details concerning the language involved in specifying the problem. We refer to [15, 3, 16, 18] for definitions of action languages, which are suited for this purpose.

1.2 Diagnosis

In diagnosis, given a theory and some observations, the goal is to find a set of hypotheses which can explain the observations by means of the theory. The notion of "explanation" is not a priori clear, therefore we define two notions of diagnosis, *abductive* and *consistency-based* diagnosis, following [4].

In the *abductive* model of diagnosis [1, 5, 20], the theory is a function-free disjunctive logic program, whose semantics is given by the set of its stable models [14]. This form of diagnostic reasoning has been recently proved to be highly expressive, as it is able to represent even problems located at the third level of the polynomial hierarchy [6].

Definition 3 *We define an* abductive diagnostic problem (ADP) *$\mathcal{P}$ as a triple $\langle H, T, O\rangle$, where:*

- *H is a set of ground atoms and is referred to as* hypotheses*.*

- *T is a disjunctive datalog program and is referred to as* theory.
- *O is a set of ground literals and is referred to as* observations.

□

Definition 4 *Given an ADP $\mathcal{P} = \langle H, T, O \rangle$, a*

- generic diagnosis *is a set $\Delta \subseteq H$, such that $T \cup \Delta \models O$ holds.*
- single error *diagnosis Δ is a generic diagnosis, for which $|\Delta| = 1$ holds additionally.*
- subset minimal *diagnosis Δ is a generic diagnosis and all subsets $\Delta^* \subset \Delta$ are not generic diagnoses.*

□

On the other hand, the *consistency-based* diagnosis model refers to theories of first-order logic under classical semantics, similar to the original definition by Reiter [21].

Definition 5 *A* consistency-based diagnostic problem (CDP) *$\mathcal{P}$ is a triple $\langle H, T, O \rangle$, where:*

- *H is a set of ground atoms with predicate name ab (standing for* abnormal*) and is referred to as* hypotheses.
- *T is a set of first-order sentences and is referred to as* theory.
- *O is a set of ground literals and is referred to as* observations.

□

We assume that these sentences are written in clausal form so that they can be represented by datalog programs. H is comprised of atoms of the form `ab(c)`, meaning that the component `c` behaves abnormally.

Definition 6 *Given a CDP $\mathcal{P} = \langle H, T, O \rangle$, a*

- generic diagnosis *is a set $\Delta \subseteq H$, such that $T \cup O \cup \Delta \cup \{\neg h | h \in H - \Delta\}$ is consistent in the classical sense.*
- single error *diagnosis Δ is a generic diagnosis, for which $|\Delta| = 1$ holds additionally.*
- subset minimal *diagnosis Δ is a generic diagnosis and all subsets $\Delta^* \subset \Delta$ are not generic diagnoses.*

□

Examples for diagnostic reasoning can be found in Section 2.2 and in [4].

1.3 Planning versus Diagnostic Reasoning

Let us now consider the relationship between the two domains: First of all, it can be observed that both tasks are goal-oriented. In the case of diagnosis, the goal is to find explanations for the observations. For planning, the goal is to find a plan which leads to the desired situation. In a less procedural view, the observations and the desired situation actually are the goals.

In both diagnosis and planning, a choice must be made out of a pool of atomic items (hypotheses and actions), which forms a solution candidate. The difference is that the solutions are ordered in planning, while they are an unordered set for diagnosis.

For checking the validity of solution candidates, the theory is used in diagnosis, while in planning there are more implicit conditions, such as checking the admissibility (by means of the preconditions) and the correct execution (by means of effects and inertial laws) of actions. Of course, these checks also have to guarantee the satisfaction of the goals.

The analysis above shows that these problems follow a guess and check pattern involving a goal. As motivated in [7], a disjunctive logic programming approach is particularly suitable for representing and solving problems following such a pattern.

2 Representing Diagnosis and Planning Problems Using `dlv`

2.1 The `dlv` System

`dlv` is a powerful logic programming and non-monotonic reasoning (LPNMR) system with advanced knowledge representation mechanisms and interfaces to classic relational database systems [8, 9].

Its core language is disjunctive datalog (function-free logic programming) under the Answer Set semantics [14] with integrity constraints, strong (or explicit) negation, and queries. Integer arithmetics and various built-in predicates are also included in the core language.

In addition to this base language, `dlv` has several frontends, namely brave and cautious reasoning, abductive diagnosis, consistency-based diagnosis and a subset of SQL3. For up-to-date information on the system and a full manual please refer to the project homepage [13]. `dlv` is also available for download from that page.

Syntactically, `dlv` programs consist of rules and constraints, possibly also queries. A rule has the form

$$a_1 \texttt{ v } \ldots \texttt{ v } a_m \texttt{ :- } b_1\texttt{, } \ldots\texttt{, } b_k\texttt{, not } b_{k+1}\texttt{, } \ldots\texttt{, not } b_n\texttt{.}$$

where $m \geq 1$ and $n \geq 0$.

$a_1, \ldots, a_m, b_1, \ldots, b_n$ are *strong literals*, i.e., atoms $a(t_1, \ldots, t_f)$ or strongly negated atoms $\texttt{-}a(t_1, \ldots, t_f)$, where a is the predicate name of the atom and each t_i $(1 \leq i \leq f)$ is either a constant, a variable, or an anonymous variable. Note that in the actual implementation the order of the body literals is immaterial.

The part to the left of `:-` is called the *head* of the rule, the part to the right is the *body*. A constraint is a rule with an empty head ($m = 0$). A rule with $m = 1$ and an empty body is called a *fact*, the `:-` can be omitted in this case.

Constants are strings starting with a lower-case letter, while variables are strings starting with an upper-case letter. Anonymous variables are denoted by "_". Each occurrence of an anonymous variable stands for a variable that is different from all others in the entire rule.

Semantically, each program can have no, one or several consistent Answer Sets, each of which is also a minimal model of that program.

A nice example for the knowledge modeling power of `dlv` is *Graph 3-Colorability* – given an undirected graph, represented by facts of the form `edge(_,_)`, assign each node one of three colors such that no two adjacent nodes have the same color. The simple program in Figure 1 computes the legal 3-colorings of the graph.

```
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).

% Each node is either red, green, or blue.
colored(X,red) v colored(X,green) v colored(X,blue) :- node(X).

% Two nodes on the same edge must not have the same color.
:- edge(X,Y), colored(X,C), colored(Y,C).
```

Figure 1: Graph 3-Coloring

As another example, take the program in Figure 2, which computes the *Hamilton paths* of a directed graph. (Recall that a Hamilton path of a directed graph is a path through that graph that reaches each node exactly once.) In our example, the graph is given by a relation `arc(X,Y)` denoting its arcs and a starting node `start(X)`.

```
% start(X), the starting node, is reached by definition.
% Any other node is reached, if it is the destination
% of an arc that is part of the current path.
reached(X) :- start(X).
reached(X) :- inPath(_,X).

% Each arc starting in an already reached node is either
% in the path, or it is not.
inPath(X,Y) v -inPath(X,Y) :- reached(X), arc(X,Y).

% No two arcs with the same source resp. destination
% nodes may be included in the path.
% != is a built-in predicate denoting inequality.
:- inPath(X,Y), inPath(X,Y1), Y != Y1.
:- inPath(X,Y), inPath(X1,Y), X != X1.

% Each node has to be reached.
:- arc(X,_), not reached(X).
```

Figure 2: Finding Hamilton Paths

2.2 Diagnostic Reasoning in `dlv`

`dlv` provides two kinds of diagnosis frontends: one for *abductive diagnosis*, and one for *consistency based diagnosis*, as shown in Section 1.

Both frontends support three different modes: general diagnosis, where all diagnoses are computed, subset minimal diagnosis[1], and single failure diagnosis.

The diagnostic theory obeys the basic syntax described in Section 2.1. Hypotheses (resp. observations) are lists of atoms (resp. literals) followed by a dot (`.`) and are stored in files whose names carry the extension `.hyp` (resp. `.obs`). Note that hypotheses and positive observations are syntactically equal to facts, but their semantics is different.

In the case of abductive diagnosis, these modes are invoked by command line options `-FD`, `-FDmin` and `-FDsingle`, respectively, while for *consistency-based diagnosis* the corresponding command line options are `-FR`, `-FRmin` and `-FRsingle`.

For detailed information on the theoretical foundations of the implementation of the diagnosis frontends please refer to [4].

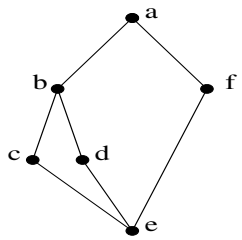


Figure 3: A computer network

As an example, consider the computer network depicted in Figure 3. Suppose that machine a is online in this network but we observe it cannot reach machine e. Which machines are offline? This can be easily modeled as a diagnosis problem, where the theory is:

```
reaches(X,X) :- node(X), not offline(X).
reaches(X,Z) :- reaches(X,Y), connected(Y,Z), not offline(Z).
```

the observations are:

```
not offline(a). not reaches(a,e).
```

and the hypotheses are:

```
offline(a). offline(b). offline(c).
offline(d). offline(e). offline(f).
```

The diagnoses in this case are D_1 ={`offline(b)`, `offline(f)`},
D_2 ={`offline(c)`, `offline(d)`, `offline(f)`}, D_3 ={`offline(e)`}, and all supersets of D_1, D_2, or D_3.

[1] For positive non-disjunctive theories only.

2.3 Planning in `dlv`

In the literature the relation between planning, reasoning about actions, and non-monotonic logic programming has been studied quite extensively [15, 23, 2]. However, most of this work does not deal with disjunctive logic programming. Exceptions are [10, 19], where several planning problems are encoded using disjunctive datalog.

Due to lack of space, we omit the step of describing an action language and the associated translation to disjunctive datalog (this will be the task performed by a planning front-end, which we are currently implementing), and directly show the encoding of an example planning domain in disjunctive datalog.

The following encoding is similar to the ones presented in [10, 18], but there is a main difference: In our approach, we view situations as states of knowledge, rather than complete states. Our approach can be thought of as modelling an autonomous robot, which usually is not omniscient, hence his knowledge may be incomplete.

We show by example of the well-known blocksworld domain, how such an encoding can be achieved in disjunctive datalog. The objects in the blocksworld are one table and an arbitrary number of labeled cubic blocks. They are referred to as `locations`:

```
location(table).   location(L) :- block(L).
```

Since the number of blocks varies in different problem instances, the blocks are defined together with these instances.

We have not described the time yet: We use integer built-in predicates and constants to model time. The first time is always `0`, and the last time `k` is specified when invoking `dlv` via the command-line option `-N=k`. `#int(T)` represents all numbers in [0..k], `#succ(T,T1)` defines the successor relation in the same interval, and the constant `#maxint` is substituted by `k`. There is also an inequality operator `<>`, which is defined over these numbers. We define `actiontime`, which represents all times at which actions may be initiated (all but the last).

```
actiontime(T) :- #int(T), T <> #maxint.
```

The state of the blocksworld at a particular time can be described by the relation `on(B,L,T)`, which specifies that block `B` resides on location `L` at time `T`.

There is one action, which is moving a block from one location to another location. A move is started at one point in time, and it is completed before the next time. At any time `T` which allows for initiating an action, the action of moving a block `B` to location `L` (where `B` must be different from `L`) may be taken (`move(B,L,T)`) or not (`-move(B,L,T)`). This single rule serves as the guessing part of our program.

```
move(B,L,T) v -move(B,L,T) :- block(B), location(L),
                              actiontime(T), B <> L.
```

In order to move a block, some preconditions must hold. The first is that no other block may be on the moved block.

```
:- move(B,L,T), on(B1,B,T).
```

Similar, if a block should be moved onto another block, no block may be on the latter. Note that an arbitrary number of blocks may be moved onto the table.

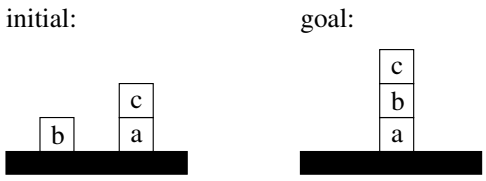


Figure 4: The Sussman anomaly.

```
% Define the involved blocks.
block(a).  block(b).  block(c).

% Specify the initial and goal situations.
on(a,table,0).  on(b,table,0).  on(c,a,0).
on(a,table,#maxint), on(b,a,#maxint), on(c,b,#maxint)?
```

Figure 5: The Sussman anomaly in disjunctive datalog.

```
:- block(B1), move(B,B1,T), on(B2,B1,T).
```

Next, the effects of an action are that the moved block is no longer on its previous location and that it is on the new location at the next point in time.

```
on(B,L,T1) :- move(B,L,T), #succ(T,T1).
-on(B,L,T1) :- move(B,_,T), on(B,L,T), #succ(T,T1).
```

It is required that at most one action is initiated at one time:

```
:- move(B,L,T), move(B1,L1,T), B != B1.
:- move(B,L,T), move(B1,L1,T), L != L1.
```

The `on` relation is inertial. This means that unless it is changed by an action, it remains stable over the time.

```
on(B,L,T1) :- on(B,L,T), #succ(T,T1), not -on(B,L,T1).
```

For a problem instance, the involved blocks, the initial state (describing where each block is located) must be specified by facts. The task is to find a sequence of moves (a *plan*) such that the goal state (which is given by a query) is reached by applying the action sequence to the initial state.

One classical instance is the so-called Sussman anomaly[2] [24], depicted in Figure 4, and its representation in disjunctive datalog is shown in Figure 5. When we look for a plan of length 3 (by specifying `-N=3` on the commandline) for this instance, exactly one is found:

```
move(c,table,0), move(b,a,1), move(c,b,2)
```

Indeed, this is the only valid plan.

[2]It is called anomaly, because at the time it was formulated (and also quite some time after that) planners could not handle it correctly.

3 Conclusion and Further Work

We have discussed some relationships between two important AI tasks, namely Diagnosis and Planning. The way in which they are related, namely by showing a guess and check pattern, shows that disjunctive datalog is particularly suitable for representing and solving these types of problems.

The `dlv` system is a state-of-the-art system for solving problems represented in disjunctive datalog, and can thus serve as the computational engine for solving both diagnostic and planning problems. To ease the use of the system, frontends to `dlv` are developed, which transform problems written in a particular domain language to disjunctive datalog automatically. They are an integral part of `dlv`.

For diagnostic reasoning we have already developed such a frontend by using techniques which are described in detail in [4].

For planning, we have shown by example how such problems can be represented. A frontend which can transform planning problems specified in an action language to disjunctive datalog is currently under development and will be included in one of the next releases of `dlv`.

To get an idea about performance, we refer to the webpage [11] at the University of Texas, where you can find the report [10], which contains benchmark data for `dlv` and other systems.

References

[1] L. Console, D. Theseider Dupré, and P. Torasso. On the Relationship Between Abduction and Deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[2] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *Proceedings of the European Conference on Planning 1997 (ECP-97)*, pages 169–181. Springer Verlag, 1997.

[3] P. M. Dung. Representing Actions in Logic Programming and Its Applications in Database Updates. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 222–238, 1993.

[4] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the `dlv` System. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.

[5] T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. In *Theoretical Computer Science* [6], pages 129–177.

[6] T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

[7] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):315–363, September 1997.

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Nonmonotonic Reasoning. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Berlin, 1997. Springer.

[9] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[10] E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft, 1999.

[11] E. Erdem. Website for applications of logic programming to planning: Computational experiments, since 1998. <URL:http://www.cs.utexas.edu/users/esra/experiments/experiments.html>.

[12] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, Lecture Notes in AI (LNAI), El Paso, Texas, USA, December 1999. Springer Verlag.

[13] W. Faber and G. Pfeifer. dlv homepage, since 1996. <URL:http://www.dbai.tuwien.ac.at/proj/dlv/>.

[14] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[15] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[16] E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pages 623–630, 1998.

[17] H. Kautz and B. Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, 1992.

[18] V. Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm - A 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[19] V. Lifschitz. Answer set planning. In *Proceedings of the 16th International Conference on Logic Programming – ICLP'99*, 1999.

[20] D. Poole. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. *Computational Intelligence*, 5(1):97–110, 1989.

[21] R. Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32:57–95, 1987.

[22] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[23] V.S. Subrahmanian and C. Zaniolo. Relating Stable Models and AI Planning Domains. In *Proceedings of the 12th International Conference on Logic Programming*, pages 233–247, 1995.

[24] G. J. Sussman. The Virtuous Nature of Bugs. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, chapter 3, pages 111–117. Morgan Kaufmann Publishers, Inc., 1990. originally written 1974.

Type-based Nondeterminism Checking in Functional Logic Programs*

Michael Hanus and Frank Steiner

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

Abstract

Functional logic languages combine nondeterministic search facilities of logic languages with features of functional languages, e.g., monadic I/O to provide a declarative method to deal with I/O actions. Unfortunately, monadic I/O cannot be used in programs which split the computation due to nondeterministic reductions. This problem can be avoided if nondeterministic computations are encapsulated by search operators which are available, for instance, in the multi-paradigm language Curry. To support the programmer in identifying nondeterministic parts of a program, we develop a method based on a type and effect system that will find every possible source of nondeterminism. Additionally, such information can be exploited in compilers to optimize deterministically reducible parts of a program.

1 Introduction

An important feature of logic languages is the ability to deal with nondeterministic computations to compute solutions for partially instantiated goals. This can lead to problems when using I/O operations, because these are usually not backtrackable. For instance in Prolog, output made by a failing branch of the search tree will remain on the screen and disturb the output of a possibly successful computation. In languages supporting more flexible search strategies instead of backtracking, like Curry [13] or Oz [24], the problem becomes more serious since different branches of a nondeterministic computation might be evaluated concurrently. Thus, different branches of the search tree would compete for the input and output devices.

To provide a clean and declarative method of I/O, one can use the monadic I/O concept [26] which was developed for Haskell [23] and adapted in Curry. In this concept, I/O operations are seen as transformations that act on the outside world, which contains the file system, the Internet etc. This world can of course not be copied. Thus, nondeterminism in combination with monadic I/O is not allowed and leads to a run-time error in Curry. To avoid this problem, Curry allows to encapsulate nondeterministic computations [12, 24], thus increasing program stability and safety.

The remaining problem is to detect all possible sources for nondeterminism in a program. This can be very difficult even for small programs and often also depends on the form of

*The research described in this paper has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1.

queries the user may ask. Thus, our aim is to develop a method to detect all possible sources of nondeterminism.

Additionally, the information computed by our program analysis can be used for compiler optimizations. For instance, instructions for checking function arguments and deciding if the actual call of this function reduces deterministically or not can be eleminated for functions which are proven to reduce deterministically.

If larger program parts or the complete program do not cause nondeterminism, code for handling nondeterminism, for instance for spawning new computation branches, can be dropped (*dead code elimination*). Such optimizations can be applied for instance to the Curry2Java compiler [11] which is part of our Curry system PACS [14] and compiles Curry programs into code for an abstract machine implemented in Java.

Our analysis is based on a type and effect system (see [20] for an overview), which can be seen as an extension of classical type systems known from functional languages. The basic idea is to annotate function types with some effect to describe the run-time behavior of the function. In our case we annotate the names of those functions as an effect that might cause nondeterministic computations. Then our analysis returns all function names that could split the computation when applied during the evaluation of a certain expression (i.e., the goal to be solved).

Note that existing determinism analyses for (functional) logic languages cannot be directly adapted to Curry because the same function might reduce deterministically or not, depending on its arguments. Thus, we need to derive groundness information for arguments in function calls which is not trivial due to the lazy evaluation mechanism in Curry. Therefore, analyses for languages like Prolog [25, 4], Mercury [15], or HAL [5] do not apply because they do not deal with lazy evaluation. In contrast, analyses proposed for narrowing-based functional logic languages dealing with lazy evaluation cannot handle residuation, which additionally exists in Curry, and rely on the non-ambiguity condition [18] which is too restrictive for Curry programs. Furthermore, these analyses are either applied during run time (like in Babel [18] and partially in K-Leaf [17]), or are unable to derive groundness information for function calls in arguments (like in K-Leaf).

In the next section we will briefly introduce the language Curry which is the object language of our analysis. Note that the analysis itself should be adaptable also to other (functional) logic languages that suffer from similar problems. In Section 3 the type and effect system will be described together with some examples. Section 4 contains our conclusions and discusses some future work. Due to lack of space, some detailed definitions and the proofs of the results are omitted but will be presented in a forthcoming technical report.

2 Overview of Curry

Curry [8, 13] is a multi-paradigm language combining in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). It also amalgamates the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [7] for a survey on functional logic programming). Thus, various operational models developed for declarative programs can be seen as restrictions of Curry's computation model (see [8] for a detailed

discussion).

A Curry program specifies the semantics of expressions, where goals, which occur in logic programming, are particular constraint expressions. Executing a Curry program means simplifying an expression until a value (or solution) is computed. To distinguish between values and reducible expressions, Curry has a strict distinction between (data) *constructors* and *operations* or *defined functions* on these data. Hence, a Curry program consists of a set of type and function declarations. The type declarations define the computational domains (constructors) and the function declarations the operations on these domains.

Curry combines various features known in declarative programming, like higher-order functions, constraints and the possibility to use constraint solvers for various domains, encapsulated search [12, 24], a Hindley/Milner-like polymorphic type system [3], monadic I/O [26] and features for communication and distributed programming [9]. A detailed description of these features can be found in [13]. In the following we will only outline those which are necessary to understand the ideas of our analysis.

2.1 Basic Features of Curry

Values in Curry are, similarly to functional or logic languages, *data terms* constructed from constants and data constructors. For instance, the datatype declarations

```
data Bool = True | False
data List a = [] | a : List a
```

introduce the datatype `Bool` with the 0-ary constructors (*constants*) `True` and `False`, and the polymorphic type "`List a`" of lists. Natural numbers, which we will use in the examples, are represented in Curry by constants (`0`, `1`, `2`,...) of type `Int`.

A *data term* is a well-typed expression containing variables, constants and data constructors, e.g., `1:2:xs`. *Functions* operate on data terms. Their meaning is specified by *rules* (or *equations*) of the form "l `|` c `=` r" (the condition part "`|` c" is optional) where l is a *pattern*, i.e., l has the form $f\ t_1 \ldots t_n$ with f being a function symbol, $t_1, \ldots, t_n$ data terms and each variable occurs only once, and r is a well-formed *expression* containing function calls, constants, data constructors and variables from l and c. The *condition* c is a *constraint* which optionally contains a list of locally declared variables, i.e., a constraint can have the form `let` $v_1, \ldots, v_k$ `free in` *con* where the variables v_i are only visible in the constraint *con*. Basic constraints are (strict) equations of the form e_1 `=:=` e_2 which are solvable if e_1 and e_2 are reducible to unifiable data terms. Constraints can be composed by the concurrent conjunction operator `&`, i.e., c_1 `&` c_2 will be evaluated by concurrently evaluating c_1 and c_2. If a local variable v of a condition should be visible also in the right-hand side, the rule is written as l `|` c `=` r `where` v `free`. A rule can be applied if its condition is solvable. A *head normal form* is a variable, a constant, or an expression of the form $c\ e_1 \ldots e_n$ where c is a data constructor. A *Curry program* is a set of data type declarations and equations.

Example 1 The following rules define the concatenation of lists, a function for computing the last element of a list and a (partial) square function `sq`, which we will use in later examples:

```
append []     ys = ys
append (x:xs) ys = x : append xs ys

last l | let xs free in append xs [x] =:= l  = x  where x free
```

```
sq 1 = 1
sq 2 = 4
```

If the equation "`append xs [x] =:= l`" is solvable, then `x` is the last element of the list `l`. □

From a functional point of view, we are interested in computing the *value* of an expression, i.e., a data term which is equivalent (w.r.t. the program rules) to the initial expression. In logic languages, we want to solve goals, i.e., compute bindings for free variables in an initial expression. Since Curry integrates these two paradigms, it computes *answer pairs* consisting of a substitution and an expression. Due to the nondeterministic features of Curry, an expression may reduce to more than one answer pair, i.e., a reduction step has the general form[1]

$$e \;\Rightarrow\; \sigma_1, e_1 \;\mid \cdots \mid\; \sigma_n, e_n$$

where $n \geq 0$, e, e_1,...,e_n are expressions, $\sigma_1, \ldots, \sigma_n$ are substitutions on the free variables in e, and "`|`" joins different alternatives to a disjunction. We call the evaluation step *deterministic* if $n = 1$ and *nondeterministic* if $n > 1$. The case $n = 0$ corresponds to a *failure*.

For selecting the next reducible function call (so called *redex*) in an expression that must be evaluated, Curry uses a combination of residuation and *needed narrowing* [1, 8]. This is, roughly speaking, the combination of lazy evaluation with bindings of uninstantiated variables as demanded by the patterns of left-hand sides in the rules. For instance, consider the function `sq` from Example 1. The function call "`sq 2`" is reduced to the value `4` like in any functional language. However, if the argument is an uninstantiated variable, there are two possibilities:

1. If we evaluate `sq` by residuation (in this case `sq` is called *rigid*), the call `sq x` will suspend until `x` is bound to some constructor term that will allow to choose one of the rules for reduction. This is possible by the concurrent evaluation of constraints where a different thread can bind `x` to some value.

2. If we evaluate `sq` by narrowing, as it will be done in all following examples (in this case `sq` is called *flexible*), we bind `x` to all possible patterns of the left-hand sides and continue with all different computation branches independently:

   ```
   sq x ⇒* {x=1} 1 | {x=2} 4
   ```

   Thus, the call `sq x` causes a nondeterministic step in our computation.

In Curry, constraints are evaluated by narrowing (since they correspond to predicates in logic languages), while non-constraint functions are computed using residuation. This behavior can be easily changed by annotations [13].

To make the pattern matching and the rigid/flexible status of functions explicit, we assume that all functions are defined by one rule whose left-hand side contains only variables as arguments and the right-hand side contains case-expressions for pattern matching. Thus, expressions have the following form (we assume that lambda abstractions and local declarations are eliminated by lambda lifting [16]):

$$\begin{array}{rcl} e & ::= & x \mid f\ e_1 \ldots e_n \mid \texttt{let}\ x\ \texttt{free in}\ e \mid \texttt{case}\ e\ \texttt{of}\ p_1 : e_1; \ldots; p_n : e_n \mid \\ & & \texttt{fcase}\ e\ \texttt{of}\ p_1 : e_1; \ldots; p_n : e_n \mid e_1\ \texttt{or}\ e_2 \end{array}$$

[1] See [13] for a definition of the one step relation $\Rightarrow$.

f is a constructor or defined function and p_i are flat patterns of the form $C\ x_1 \ldots x_n$ where C is an n-ary data constructor. `case` and `fcase` are the rigid and flexible case distinctions, respectively, and `or` denotes a don't know alternative between two expressions.

The assumption that each function f is defined by a single rule with left-hand side $f\ x_1 \ldots x_n$ does not restrict the class of Curry programs we can handle, since all function definitions can be transformed into this form (higher-order features can be translated into first-order using Warren's method [27]). For instance, the `sq` function can be rewritten as

```
sq x = (f)case x of 1:1; 2:4
```

where `case` is used if `sq` is defined rigid (i.e., evaluated by residuation), and `fcase` if it is flexible (i.e., evaluated by narrowing). A precise definition of this transformation can be found in [10].

The binary `or` operator is used to translate functions with overlapping left-hand sides (we omit further explanations here due to lack of space).

2.2 Nondeterminism and I/O

Nondeterministic computations are problematic if they appear in programs that use I/O. A declarative treatment of input/output, as implemented in Curry, can be obtained by the *monadic I/O* concept [26]. In this concept, an interactive program is considered as a function computing a sequence of actions which are applied to the outside world. An *action* changes the state of the world and possibly returns a result (e.g., a character read from the terminal). For instance, `getChar` reads a character from the standard input whenever it is executed, i.e., applied to a world. Several I/O actions can be composed, e.g., `getChar` can be composed with the action `putChar` (which writes a character to the terminal) by the sequential composition operator `>>=`, i.e.,

```
getChar >>= putChar
```

is a composed action which prints the character typed in the keyboard to the screen (see [26] for more details).

Since the world cannot be copied (note that the world contains at least the complete file system or the complete Internet in web applications), an interactive program having a disjunction as a result makes no sense. For instance consider the program

```
nonsense file x | sq x =:= y  = writeFile file y
                                where y free
```

where `writeFile f e` is an action that writes the value of `e` to the file `f`. If we call `nonsense "dummy" x`, a nondeterministic splitting during the evaluation of `sq x =:= y` would result in two independent computation branches, both trying to write different values to the same file. This must obviously be avoided because it could for instance lead to an inconsistent file state. Thus, a Curry program that uses I/O actions will result in a run-time error whenever a nondeterministic computation is detected. The goal of this paper is to provide a method to detect such kind of programming errors at compile time, so that they can be avoided by encapsulating all possible search between I/O operations (see [12, 24] for a more detailed description of encapsulated search).

3 The Nondeterminism Analysis

In this section, we develop a method based on a non-standard type system to check expressions for possible nondeterministic evaluation steps. First, we sketch the ideas behind our nondeterminism analysis before we provide and explain the typing rules in detail.

As we have seen in the examples in Section 2, uninstantiated variables as arguments may cause nondeterministic steps when using narrowing. Hence, we propose a type and effect system that allows us to identify the form of arguments, i.e., if they are ground terms or not. This is a non-trivial task because the arguments can be function calls which are evaluated lazily in Curry. Thus, the type of the arguments cannot be derived from their *actual* form but the analysis must take into account their reduction behavior. We will collect the names of the functions that might split the computation as an *effect* of this computation.

In a first step, the program will be typed, either by hand or by a type inferencer. In a second step, we will analyse an expression with respect to the program and obtain a result like "the expression reduces deterministically" or "the reduction of the expression raises a nondeterminism caused by the evaluation of the function f." For instance, the function definition

```
f x = sq x
```

could be given the following type:

$$A \xrightarrow{\{\texttt{f},\texttt{sq}\}} G$$

This type expresses that `f` takes **A**ny argument and returns a **G**round term but during the application of `f` nondeterministic steps may be raised by the functions `f` and `sq`. This is signalized by the effect $\{\texttt{f},\texttt{sq}\}$ which is annotated above the arrow. We will explain this in more detail in the next sections.

3.1 The Types and Effects

Type and effect systems can be seen as an extension of classical type systems known from functional languages (see [19, 20] for details). The basic idea is to extend the type annotation for a function with an effect that may occur during the application of this function. For every expression a type and the effect of its reduction can be approximated. Type and effect systems provide for powerful analyses like exception analysis [22], side effect analysis [19] or communication analysis [20, 21].

Since nondeterminism is mostly caused by uninstantiated variables in arguments of flexible functions, we use a non-standard type system to distinguish between ground terms and any terms. Thus, we define *type expressions* as follows:

$$\tau \;=\; G \;\mid\; A \;\mid\; \tau_1 \ldots \tau_n \to \tau$$

G denotes a ground term and A denotes any term. Since a ground term is also any term, we have subtyping [2], i.e., $G < A$ where $\tau_1 < \tau_2$ means that any term of type τ_1 has also type τ_2. For functions it is important to note that covariance holds for the result type but contravariance for the argument types, i.e., $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ iff $\tau_1' \leq \tau_1, \tau_2 \leq \tau_2'$. This is easy to understand if we think of smaller types as more precise information. The type $A \to G$ is smaller than $G \to G$ and it is more precise, because it provides information about a larger set of inputs. Thus, the larger the input type, the more precise the information, i.e.,

the smaller the function type. The opposite holds for the output type, i.e., $A \to G$ is smaller than $A \to A$, because the information about the target set is more precise.

Our type system allows several types for one expression. For instance, for a function defined by `f x = x` the two types $G \to G$ and $A \to A$ are correct but not related by subtyping.

As an *effect* we want to collect the names of functions whose application can cause nondeterministic computations. Therefore, we define the effect φ by

$$\varphi \subseteq \mathcal{F} \cup \{case, or\}$$

where $\mathcal{F}$ is the set of function symbols of a program P. The effect annotated for a function declaration must contain all function names that might reduce in a nondeterministic way during the application of this function. For instance, the type annotation for a function f could be

$$f :: A \xrightarrow{\{f,g,h\}} A$$

if in the rhs of f a call to the function g causes nondeterminism, whereas g splits the computation because it calls the function h in its body which raises the nondeterminism.

The aim of our analysis is to calculate type judgements

$$E \vdash e :: \tau/\varphi$$

for an expression e with respect to a program P, where E is a type environment, i.e., a set of type annotations for function and constructor symbols from P and for variables. $E \vdash e :: \tau/\varphi$ means that the expression e has type τ, and during the reduction of e the effect φ may occur.

The soundness of our approximation, which will be formalized at the end of Section 3.2, ensures that we find every function which can raise a nondeterminism and that terms identified as ground by our analysis will actually reduce to ground terms. Similarly to most program analyses, we cannot compute precise information for all program parts but only approximations of the real program behavior. In our case this means that the computed type might be too imprecise, e.g., A instead of G, and the effect might be too large, i.e., it might contain functions that will never raise a nondeterminism. This imprecision can be reduced by refining the type domain (which is future work).

3.2 The Typing Rules

Figure 1 shows the typing rules for our analysis which define the type judgements $E \vdash e :: \tau/\varphi$.

The DECL rule defines when a program rule r is correctly typed w.r.t. a type annotation $\mathcal{A}$ (i.e., $E \vdash_{\mathcal{A}} r$): The type annotation $\mathcal{A}$ for the function f must be given in the environment E, and with the corresponding types for the arguments x_i added to the environment[2] the derived type for the right-hand side must match the type of f. For the effect, two cases are distinguished: If $\varphi \neq \emptyset$ then the right-hand side e possibly raises a nondeterministic computation step. Thus, f could also reduce nondeterministically, and so the annotated effect must include φ and $\{f\}$. Otherwise, if $\varphi = \emptyset$, ψ is completely unrestricted. In both cases ψ can contain arbitrary function symbols from the program, because, due to subeffecting (see below), it is still a safe approximation to annotate a larger effect.

[2] $E[x_1 :: \tau_1, \ldots, x_n :: \tau_n]$ denotes the type environment obtained from E by deleting all existing type annoations for $x_1, \ldots, x_n$ and adding the new type annotations in the square brackets.

Typing of rules:

DECL $\dfrac{E[x_1 :: \tau_1, \ldots, x_n :: \tau_n] \vdash e :: \tau/\varphi}{E \vdash_{\mathcal{A}} f\ x_1 \ldots x_n\ =\ e}$ if $\mathcal{A} = f :: \tau_1 \ldots \tau_n \overset{\psi \cup \varphi}{\to} \tau \in E$, and $f \in \psi$ if $\varphi \neq \emptyset$

Typing of expressions:

VAR $\dfrac{}{E \vdash x :: \tau/\emptyset}$ if $x :: \tau \in E$

NEWVAR $\dfrac{E[x_{fresh} :: A] \vdash e[x/x_{fresh}] :: \tau/\varphi}{E \vdash \texttt{let}\ x\ \texttt{free in}\ e\ ::\ \tau/\varphi}$

APP $\dfrac{E \vdash e_1 :: \tau_1/\varphi_1 \ldots e_n :: \tau_n/\varphi_n \qquad E \vdash f :: \tau_1 \ldots \tau_n \overset{\varphi}{\to} \tau/\emptyset}{E \vdash f\ e_1 \ldots e_n\ ::\ \tau/\bigcup_i \varphi_i \cup \varphi}$

FCASE $\dfrac{E \vdash e :: \tau/\varphi \qquad E[\overline{x_{1m}} :: \tau] \vdash e_1 :: \tau_1/\varphi_1\ \ldots\ E[\overline{x_{nm}} :: \tau] \vdash e_n :: \tau_n/\varphi_n}{E \vdash \texttt{fcase}\ e\ \texttt{of}\ p_1(\overline{x_{1m}}) : e_1; \ldots p_n(\overline{x_{nm}}) : e_n\ ::\ max_i(\tau_i)/\bigcup_i \varphi_i \cup \varphi \cup \varphi'}$

where $\varphi' = \{case\}$ if $\tau = A$ and $n > 1$
$= \emptyset$ otherwise

CASE $\dfrac{E \vdash e :: \tau/\varphi \qquad E[\overline{x_{1m}} :: \tau] \vdash e_1 :: \tau_1/\varphi_1\ \ldots\ E[\overline{x_{nm}} :: \tau] \vdash e_n :: \tau_n/\varphi_n}{E \vdash \texttt{case}\ e\ \texttt{of}\ p_1(\overline{x_{1m}}) : e_1; \ldots p_n(\overline{x_{nm}}) : e_n\ ::\ max_i(\tau_i)/\bigcup_i \varphi_i \cup \varphi}$

OR $\dfrac{E \vdash e_1 :: \tau_1/\varphi_1 \qquad E \vdash e_2 :: \tau_2/\varphi_2}{E \vdash \texttt{or}(e_1, e_2)\ ::\ max(\tau_1, \tau_2)/\varphi_1 \cup \varphi_2 \cup \{or\}}$

SUB $\dfrac{E \vdash e :: \tau/\varphi}{E \vdash e :: \tau'/\varphi'}$ if $\tau \leq \tau', \varphi \subseteq \varphi'$

Note: $\overline{x_{ij}}$ denotes the sequence $x_{i1}, \ldots, x_{ij_i}$.

Figure 1: Typing rules

The VAR rule is common. If a type is given for x in the environment, this type and an empty effect can be derived. Note that x can be a variable, a function symbol, or a constructor symbol.

The NEWVAR rule handles local (existentially quantified) variables. The newly introduced variable x_{fresh} is uninstantiated, and thus must be given the type A for analysing e. By $e[x/t]$ we denote the replacement of all free occurrences of x in e by t. An occurrence of a variable x in e is free if it does not occur inside a subterm e' of e with $e' = \texttt{let}\ x\ \texttt{free in}\ \tilde{e}$. A variable x occurs free in e if at least one occurrence of x in e is free.

When applying a function f to some e_i, the effect φ annotated for f might occur. Thus, in the APP rule, φ is returned together with the effects derived for the arguments.

The FCASE rule handles nondeterminism caused by narrowing. If the type A is derived for e and more than two case branches exist, a non-empty effect must be returned to signalize the potential nondeterminism. This is ensured by adding the keyword *case* (which could be indexed by the function name from the left-hand side, as we will do it in subsequent examples) to the effect. Additionally, all effects from the right-hand sides must be collected. Note that

for analysing the right-hand sides, the pattern variables are given the type of e: If e reduces to a ground term, so will the terms that the pattern variables are instantiated to. Otherwise, they could be uninstantiated in the subsequent computation and therefore they have the type A. If only one right-hand side has type A, the entire fcase-expression might return this type. Thus, the maximum type of all right-hand sides (w.r.t. ordering $G < A$) is the type of the fcase-expression.

The CASE rule analyses rigid functions. Thus, the case-expression itself will never split the computation and no additional effect is returned.

Nondeterminism which is caused by overlapping left-hand sides is handled by the OR rule, where the keyword *or* is returned. Like in the (F)CASE rules, the effects of the right-hand sides are collected and the maximal type is returned. Note that by using *case* and *or* keywords, nondeterminism caused by narrowing and by overlapping left-hand sides can be distinguished.

The last rule, SUB, defines subtyping and subeffecting [20]. For our type system, the rule expresses that any expression of type G is also of type A, and that every effect φ can be enlarged without loosing the safety condition. The approximation just becomes more imprecise.

In order to show the correctness of the typing rules, we need the definition of a *correct type environment*.

Definition 1 *Let P be a Curry program and E a type environment which contains at least one type annotation for each function and constructor occurring in P.*

1. *Let $r \in P$ be a program rule with left-hand side $f\ x_1 \ldots x_n$ and $E_f \subseteq E$ the set of all type annotations for f in E. r is correctly typed w.r.t. E, denoted by $E \vdash r$, iff $E \vdash_{\mathcal{A}} r$ for all $\mathcal{A} \in E_f$.*

2. *E is a correct type environment for P iff $E \vdash r$ for all $r \in P$, and for all type annotations $c :: \tau_1 \ldots \tau_n \xrightarrow{\varphi} \tau \in E$, where c is an n-ary constructor, $\tau_i = A$ implies $\tau = A$.*

Thus, a correct type environment E for a program must contain at least one type for each defined function and constructor. Otherwise, the rules that use these constructors and functions cannot be correctly typed. Note that for a constructor c its result type is always the maximum of its input types, and it is always correct to annotate $\varphi = \emptyset$ because a constructor itself is not reduced (only its arguments) and so does never raise a nondeterministic reduction step.

In the following we use E_e as an abbreviation for $E\,[x_1 :: A, \ldots, x_n :: A]$ where $\{x_1, \ldots, x_n\}$ is the set of variables which occur free in the expression e.

The following lemma states the correctness of the typing rules w.r.t. a single reduction step.

Lemma 1 (Subject reduction) *Let E be a correct type environment for a Curry program and e an expression. If $E_e \vdash e :: \tau/\varphi$ and there is a reduction step $e \Rightarrow \sigma_1, e_1 \mid \cdots \mid \sigma_n, e_n$ with $n > 0$, then $E_{e_i} \vdash e_i :: \tau/\varphi$.*

Thus, the type and effect of an expression is invariant under reduction steps. We can easily prove the following correctness results for our framework with this important property:[3]

[3] The reduction semantics of Curry can be found in [13] and (concerning case-expressions) in [10].

Theorem 1 (Correctness of typing rules) *Let E be a correct type environment for a Curry program, e an expression and $E_e \vdash e :: \tau/\varphi$.*

1. *If $\tau = G$ and e reduces in finitely many steps to a* value *v (i.e., a term without defined function symbols), then v is a ground term.*
2. *If e reduces in finitely many steps to an expression $\tilde{e}$ where $\tilde{e}_{|_o}$ is the next redex with $\tilde{e}_{|_o} = \texttt{fcase}\ldots$ ($\tilde{e}_{|_o} = \ldots\texttt{or}\ldots$) and $\tilde{e} \Rightarrow\ \sigma_1, e_1\ \texttt{|}\ \cdots\ \texttt{|}\ \sigma_n, e_n$ with $n > 1$, then case $\in \varphi$ (or $\in \varphi$).*

From the second property we can directly conclude that e reduces deterministically if $\varphi = \emptyset$.

Thus, we know that every function that might raise a nondeterministic computation step during the reduction of e will be collected in the effect φ by our typing rules, i.e., we will never miss such a function. This is the meaning of a *safe* approximation for our analysis. The same holds for the type, i.e., a term analysed as ground will indeed reduce to a ground term (if its reduction terminates).

3.3 Examples

We want to clarify the ideas of our analysis by providing some simple examples. If the type environment E contains only one type annotation $\mathcal{A}$ for the function in the left-hand side of the rule r, we simply write $E \vdash r$ instead of $E \vdash_{\mathcal{A}} r$.

Example 2 We want to show that the rule

```
f :: A -(∅)-> G
f x = 0
```

is correctly typed. The type expresses that `f` will reduce deterministically and return a ground term regardless of its input argument. It should be obvious that this type is correct. The initial type environment contains the type for `f` and the constructor `0`:

$$E = \{\texttt{f} :: A \xrightarrow{\emptyset} G,\ \texttt{0} :: G\}$$

The correctness of the type is proven by the following derivation:

$$\dfrac{\dfrac{}{E[\texttt{x} :: A] \vdash \texttt{0} :: G/\emptyset}\ \text{VAR}}{E \vdash \texttt{f x = 0}}\ \text{DECL}$$

If we extend E by $\texttt{x} :: A$ according to the type annotation for `f`, the analysis of the right-hand side returns an empty effect which matches the empty effect annotated with `f`. Note that due to subtyping and subeffecting, there are more correct types for `f`, e.g., $A \xrightarrow{\emptyset} A$, $G \xrightarrow{\emptyset} G$, $G \xrightarrow{\emptyset} A$ and any of these types with every non-empty effect. However, all these types are larger than the most precise type $A \xrightarrow{\emptyset} G$. □

Example 3 If we anotate the type $A \xrightarrow{\emptyset} G$ from the former example for the identity function `id x = x`, the analysis should detect this type to be wrong, because `id` applied to a non-ground term will not return a ground term. Indeed, with $E = \{\texttt{id} :: G \xrightarrow{\emptyset} G\}$, the analysis fails to analyse `id` as being correctly typed:

$$\dfrac{\dfrac{}{E[\mathtt{x} :: A] \not\vdash \mathtt{x} :: G/\emptyset}\ \text{VAR}}{E \not\vdash \mathtt{id\ x = x}}\ \text{DECL}$$

With E extended by $\mathtt{x} :: A$, we cannot derive the required type G for the right-hand side `x`. □

Next we will study an example where nondeterminism occurs.

Example 4 We use the function `sq` which was defined in Section 2:

$$\mathtt{sq} :: \mathtt{G} \xrightarrow{\emptyset} G$$
$$\mathtt{sq} :: \mathtt{A} \xrightarrow{\{\mathtt{sq}, case_{\mathtt{sq}}\}} G$$
```
sq x = fcase x of 1:1; 2:4
```

Two types are defined for `sq`, claiming that the function will reduce deterministically when applied to a ground term, but might raise a nondeterministic step otherwise. The initial environment contains these two types together with the types for the constructors (we identify the two types by $\mathcal{A}_1$ and $\mathcal{A}_2$):

$$E = \{\mathcal{A}_1 :\ \mathtt{sq} :: G \xrightarrow{\emptyset} G,\ \mathcal{A}_2 :\ \mathtt{sq} :: A \xrightarrow{\{\mathtt{sq}, case_{\mathtt{sq}}\}} G,\ \mathtt{1} :: G,\ \mathtt{2} :: G,\ \mathtt{4} :: G\}$$

Since we have two types for `sq`, we must verify two cases of the DECL rule, one for each type (we drop the VAR derivation for the constructor `4` because it is the same as for `1`):

$$\dfrac{\dfrac{\dfrac{}{E[\mathtt{x} :: G] \vdash \mathtt{x} :: G/\emptyset}\ \text{VAR} \quad \dfrac{}{E[\mathtt{x} :: G] \vdash \mathtt{1} :: G/\emptyset}\ \text{VAR}}{E[\mathtt{x} :: G] \vdash \mathtt{fcase\ x\ of\ 1:1;2:4} :: G/\emptyset}\ \text{FCASE}}{E \vdash_{\mathcal{A}_1} \mathtt{sq\ x = fcase\ x\ of\ 1:1;\ 2:4}}\ \text{DECL}$$

$$\dfrac{\dfrac{\dfrac{}{E[\mathtt{x} :: A] \vdash \mathtt{x} :: A/\emptyset}\ \text{VAR} \quad \dfrac{}{E[\mathtt{x} :: A] \vdash \mathtt{1} :: G/\emptyset}\ \text{VAR}}{E[\mathtt{x} :: A] \vdash \mathtt{fcase\ x\ of\ 1:1;2:4} :: G/\{case_{\mathtt{sq}}\}}\ \text{FCASE}}{E \vdash_{\mathcal{A}_2} \mathtt{sq\ x = fcase\ x\ of\ 1:1;\ 2:4}}\ \text{DECL}$$

With the first type of `sq`, `x` has type G and therefore the FCASE rule returns an empty effect for the right-hand side, thus matching the type of `sq`. With the second type, E is extended by $\mathtt{x} :: A$, causing the FCASE rule to return a non-empty effect. Therefore, the DECL rule demands the effect annotated with `sq` to contain `sq` as well as the effect from the right-hand side, which is satisfied. Thus, the rule is correctly typed because it is correctly typed w.r.t. to both type annotations. □

The first step in the analysis will always be the typing of a given program. In the second step, we type an expression to be evaluated w.r.t. the analysed program. For instance, if we would like to evaluate `sq 1` the analysis returns $E \vdash \mathtt{sq\ 1} :: G/\emptyset$, signalizing that this expression will reduce deterministically. In contrast, for `sq x` the result is $E[\mathtt{x} :: A] \vdash \mathtt{sq\ x} :: G/\{\mathtt{sq}, case_{\mathtt{sq}}\}$. Actually, the computation will split while evaluating `sq x`.

4 Conclusions and Future Work

We have proposed a method to analyse functional logic programs in order to identify expressions that might raise nondeterministic computations during their evaluation. By the results

obtained from such an analysis, the programmer will be able to change the program to make it more robust. For instance, to avoid splitting computations she might remove nondeterminism completely if it was not necessary or a result of a programming error. Otherwise, she can encapsulate the affected program parts, if the nondeterminism is necessary for computing the result. This will avoid run-time errors in programs that use I/O actions (which is the case for almost every larger program) and thus increase program stability. Additionally, compilers can exploit the analysis results to optimize the code for deterministically reducible parts of a program. Although we have presented the typing rules and some examples for the functional logic language Curry, the analysis should be easily adaptable to other functional logic languages (or just logic languages).

For future work we will study how the current set of typing rules needs to be extended to cover features like external functions or search operators. This should not be difficult but has just not been considered yet. A type and effect inference algorithm has already been developed, but could not be sketched here due to lack of space. Thus, another important point for future work is its efficient implementation since the goal of this work is the development of a fully automatic nondeterminism analysis so that the user is not forced to specify the type and effect annotations by hand (in contrast to mode-based languages like Mercury). Last but not least, we will consider to refine the type domain in order to compute more precise groundness information.

References

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[2] L. Cardelli. Type systems. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] S. K. Debray and D. S. Warren. Detection and optimization of functional computations in Prolog. In *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 490–504. Springer-Verlag, 1986.

[5] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. Stuckey. Herbrand constraint solving in HAL. To appear in Proc. ICLP'99.

[6] J.C. Gonzáles-Moreno, M.T. Hortalá-Gonzáles, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996.

[7] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[8] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[9] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[10] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. In *Journal of Functional Programming*, 9(1):33–75, 1999.

[11] M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, 1999(6).

[12] 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.

[13] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1998.

[14] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PACS: The Portland Aachen Curry System. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/pacs/`, 1999.

[15] F. Henderson and T. Somogyi, Z. Conway. Determinism analysis in the Mercury compiler. In *Proc. of the Nineteenth Australasian Computer Science Conference*, pages 337–346, 1996.

[16] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Functions. In *Functional Programming Languages and Computer Architecture*, pp. 190–203. Springer LNCS 201, 1985.

[17] F. Liu. Towards lazy evaluation, sharing and non-determinism in resolution based functional logic languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 201–209, New York, NY, USA, 1993. ACM Press.

[18] R. Loogen and S. Winkler. Dynamic detection of determinism in functional logic languages. *Theoretical Computer Science*, 142(1):59–87, 1995.

[19] J.M. Lucassen and D.K. Gifford. Polymorphic Effect Systems. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[20] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[21] H. R. Nielson and F. Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer-Verlag, 1997.

[22] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 276–290. ACM Press, 1999.

[23] J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.

[24] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[25] P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *Proc. of the TAPSOFT '87*, pages 111–125. Springer LNCS 250, 1987.

[26] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3)240–263, 1997.

[27] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.

TkCurry: A Declarative Approach to GUI Programming

Michael Hanus*

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

We show how the features of modern integrated functional logic programming languages can be exploited to implement graphical user interfaces (GUIs) in a high-level declarative style. For this purpose, we have developed a GUI library in Curry, a multi-paradigm language amalgamating functional, logic, and concurrent programming principles. The functional features of Curry are exploited to define the graphical structure of an interface and to implement new graphical abstractions, and the logic features of Curry are used to specify the logical dependencies of an interface. Moreover, the concurrent and distributed features of Curry support the easy implementation of GUIs to distributed systems.

1 Introduction

The implementation of graphical user interfaces for application programs is a non-trivial task which is usually supported by specific libraries. Although it is clear that any serious programming language must have a library for implementing GUIs, there are many different approaches to structure those libraries. In this paper we propose a GUI library for integrated functional logic languages (see [5] for a survey) and show how the features of such integrated languages can be exploited to provide a nice structure for the implementation of GUIs.

In this paper, we consider the language Curry [6, 10], a modern multi-paradigm declarative language which integrates functional, logic, and concurrent programming paradigms. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), concurrent programming (concurrent evaluation of expressions with synchronization on logical variables), supports programming-in-the-large with specific features (types, modules, encapsulated search).

In order to avoid reinventing the wheel, our GUI library is based on Tcl/Tk [13]. The main purpose of this contribution is to provide a suitable structure to access the components of Tcl/Tk in a high-level way from Curry programs. We will see that the functional *and* logic features of Curry supports together a good structure to describe GUIs.

*This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1 and by the DAAD under the PROCOPE programme. An extended version will appear in the Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00), Springer LNCS 1753.

```
runWidget "Hello"
      (TkCol [] [TkLabel [TkText "Hello world!"],
                 TkButton tkExit [TkText "Stop"]])
```

Figure 1: A simple "Hello world" GUI

```
TkCol [] [
   TkEntry [TkRef val, TkText "0"],
   TkRow [] [TkButton (tkUpdate incrText val) [TkText "Increment"],
             TkButton (tkSetValue val "0")    [TkText "Reset"],
             TkButton tkExit                  [TkText "Stop"]]]
   where val free
```

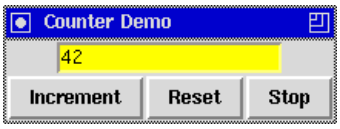


Figure 2: A specification of a counter GUI

In order to get an impression of the proposed structure of GUI implementations, Fig. 1 shows a simple but complete implementation of a "Hello world" GUI based on our library. The GUI is started by the I/O action `runWidget` which takes a string (the title of the main window) and a specification of a GUI as an argument. This specification is basically a description of the hierarchical layout of the various GUI elements. In this simple example, the GUI is a column (`TkCol`) of two elements: a label element (`TkLabel`) containing a text and a button (`TkButton`) which terminates the GUI by the action `tkExit` when the button is pressed.

Beyond the hierarchical layout structure, GUIs have also a logical structure which connects the different elements of a GUI. For instance, different buttons refer to the manipulation of particular entry fields in a GUI. As a simple example, consider the counter GUI shown in Fig. 2. Since clicking the increment button should increase the value of the entry field by one, there is a connection between the action of the "Increment" button and the value shown in the entry field (and similar for the "Reset" button). Many GUI libraries (e.g., [13, 16]) solve this problem by forcing the programmer to assign explicit names (strings) to the different GUI elements which are subsequently used as references to them. Since these names are strings, often no consistency checks are done so that runtime errors can occur when a name is referred but does not exist. Moreover, new graphical abstractions which combine several elements are difficult to define since the necessary names can clash with other existing names. To avoid these problems, we use logical variables ("fixed but unknown widget references") to refer to the different GUI elements. If a reference to some GUI element is necessary, we introduce for this purpose a logical variable ("`TkRef val`" in Fig. 2) which can be used in actions like `tkSetValue` or `tkUpdate` to manipulate these elements. Thus, a GUI in our framework is a partially instantiated data structure[1] where multiple occurrences of a logical variable denotes the logical dependencies inside the GUI. In Fig. 2 the entry field showing the current value of the counter is referred by the logical variable `val`. Clicking the "Increment" button causes the invocation of the event handler "`tkUpdate incrText val`" that applies

[1]Since the GUI library does not export any constructor for the argument type of `TkRef`, the type system ensures that no ground values can be inserted as arguments of `TkRef`.

the function `incrText` (which increments the textual representation of a number by one) to the string shown in the entry element referred by `val`. Similarly, this field is set to the string `"0"` by pressing the "Reset" button.

2 Basic Elements of Curry

This section provides a brief overview of Curry as necessary to understand our approach to GUI programming. More details about Curry's computation model and a complete description of all language features can be found in [6, 10].

From a syntactic point of view, a Curry program is a functional program[2] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of functions and the data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with uninstantiated arguments (logical variables). The behavior of such function calls depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1 The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc eval flex
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs   = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type `List a` is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[3] Since `conc` is explicitly defined as flexible[4] (by "`eval flex`"), the equation "`conc ys [x] =:= xs`" can be solved by instantiating the first argument `ys` to the list `xs`

[2] Curry has a Haskell-like syntax [14], i.e., (type) variables and function names 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 ("$f\ e$").

[3] Curry uses curried function types where α->β denotes the type of all functions mapping elements of type α into elements of type β.

[4] As a default, all non-constraint functions are rigid.

without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form "l `|` c `=`e `where` vs `free`" where l has the form $f\,t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, and vs is the list of *free variables* that occur in c and e but not in l (the condition and the `where` parts can be omitted if c and vs are empty, respectively). The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Constraint`. Each Curry system must support at least equational constraints of the form e_1 `=:=`e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms (i.e., terms without defined function symbols). However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints for applications in operation research problems, as in the PACS implementation [8]. *Expressions* are of the following form:

e	::=	c	(constants like numbers or identifiers)
		x	(variables x)
		`(`e_0 $e_1 \ldots e_n$`)`	(application)
		`if` b `then` e_1 `else` e_2	(conditional)
		e_1`=:=`e_2	(equational constraint)
		e_1 `&` e_2	(concurrent conjunction of constraints)
		e_1 `&>` e_2	(sequential conjunction of constraints)
		`let` $x_1, \ldots, x_n$ `free in` e	(existential quantification)

Curry has also a polymorphic type system which ensures that the expressions e, e_1, e_2 in the last three alternatives are always constraints.

Monadic I/O: Since the implementation of GUIs in a declarative language requires some knowledge about performing I/O in a declarative manner, we sketch the I/O concept of Curry which is identical to the monadic I/O concept of Haskell [17]. In the monadic approach to I/O, an interactive program computes a sequence of actions which are applied to the outside world. *Actions* have type "`IO` α" which means that they return a result of type α whenever they are applied to (and change) the outside world. For instance, `getChar` of type `IO Char` is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. Actions can only be sequentially composed. For instance, the action `getChar` can be composed with the action `putChar` (which has type `Char -> IO ()` and writes a character to the terminal) by the sequential composition operator `>>=` (which has type `IO` α `->` (`(`α `->` `IO` β`)` `->` `IO` β`)`, i.e., "`getChar >>= putChar`" is a composed action which prints the next character of the input stream on the screen. The second composition operator `>>` is like `>>=` but ignores the result of the first action. Furthermore, `done` is the "empty" action which does nothing (see [17] for more details). To avoid an ad-hoc combination of disjunctive computations (search) with I/O actions, all non-deterministic computations must be encapsulated between I/O operations (see [9] for details). Thus, GUIs (which are created via I/O operations) will not be duplicated due to non-deterministic computations.

3 Object-Oriented and Distributed Programming

GUI programming as proposed in this paper is based on the techniques for object-oriented and distributed programming in Curry. Therefore, we sketch these features in this section. More details and examples can be found in [7].

It is well known [15] that concurrent logic programming languages provide a simple way to implement (concurrent) objects. An object can be seen as a constraint or predicate processing a stream of incoming messages. The local state of the object is a parameter which may change in recursive calls when a message is processed. Thus, the general type of an object o is

o `::` st `->` `[`mt`]` `->` `Constraint`

where st is the type of the local state and mt is the type of the messages which can be sent to the object. For instance, a simple counter object which understands the messages `Inc`, `Get` v, and `Stop` can be implemented in Curry as follows (the predefined type `Int` denotes the type of all integer values and `success` denotes the always satisfiable constraint):

```
data CounterMessage = Inc | Get Int | Stop

counter :: Int -> [CounterMessage] -> Constraint
counter eval rigid

counter n (Inc   : ms)  =  counter (n+1) ms
counter n (Get v : ms)  =  v=:=n & counter n ms
counter _ (Stop  : ms)  =  success
```

The type declaration for `counter` (which can be omitted since types are reconstructed in Curry by a type inference algorithm) specifies that a `counter` object keeps an integer as local state and understands messages of type `CounterMessage`. Since `counter` is declared as a rigid function, an expression "`counter` n s" can reduce only if s is a bound variable.

The evaluation of the constraint "`counter 0 s`" creates a new counter object with initial value `0` where messages are sent by constraining the variable `s` to hold the desired messages. For instance, the constraint

```
let s free in  counter 0 s  &  s=:=[Inc, Inc, Get x, Stop]
```

is successfully evaluated by binding `x` to the value `2`.

In realistic applications, the stream of messages is not instantiated at once but incrementally constrained by various other objects (message senders). In order to allow a dynamic extension of senders and to ensure the sending of messages in constant time, Janson et al. [11] proposed the use of port constraints which have been generalized in Curry to provide a high-level approach to implement distributed systems [7]. In principle, a *port* can be considered as a multiset (of messages) where the individual elements are not directly accessible. There are two primitive constraints on ports, where "`Port a`" denotes the type of a port to which messages of type `a` can be sent:

```
openPort :: Port a -> [a] -> Constraint
send     :: a  ->  Port a -> Constraint
```

The evaluation of "`openPort p s`" where `p` and `s` are uninstantiated variables establishes a *port constraint* which is satisfied iff all elements in the port `p` also occur in the message stream `s` and vice versa. A message `m` is sent to the port `p` by evaluating the constraint "`send m p`" which constrains (in constant time) `p` and the corresponding stream `s` to hold the element `m`. From a logic programming point of view, `p` and `s` are partially instantiated

variables that are more and more constrained by solving the constraint "`send m p`". To avoid a communication overhead, communication is strict, i.e., the message `m` is reduced to a data term before sending it.
With the use of ports, we can define a generic constraint `new`

```
new :: (st -> [mt] -> Constraint) -> st -> Port mt -> Constraint
new obj st p = let s free in  openPort p s &> obj st s
```

to create new objects with initial state `st` and communication port `p`. Thus,

```
let cp free in  new counter 0 cp  &  client1 cp  &  client2 cp
```

creates a counter with two different clients. Each client can increment the counter by solving the constraint "`send Inc cp`". The current state of the counter can be asked by "`send (Get x) cp`" so that `x` is unified with the current counter value. Thus, free variables in messages provide an elegant method to return values to the sender without explicitly creating reply channels. To support the distributed applications, ports can be declared as *external* so that they are accessible from any machine connected to the Internet (see [7] more details).

4 A Functional Logic GUI Library

A main objective of our GUI library is a design which smoothly interacts with the features of the base language Curry. Therefore, GUIs are always created by the new primitive I/O action

```
openWish :: String -> IO (Port TkMsg) .
```

"`openWish t`" creates a new GUI window with title `t` and returns a *communication port* for this GUI. The (abstract) data type `TkMsg`[5] is the type of possible messages for GUI communication. These are only used in the implementation of the GUI library but not visible to the user of the library. The important design issue is the fact that a GUI communication port is always external and created by such an I/O action, which avoids the potential duplication of GUIs in different disjunctive branches of a computation (compare Sect. 2). Nevertheless, the non-deterministic features of the base language can be used inside a GUI if the search computations are encapsulated.

After the creation of a GUI communication port `gp`, we can run a GUI specification `gs` (like the one shown in Fig. 2) by the constraint "`runWidgetOnPort gs gp`". Basically, `runWidgetOnPort` communicates with the port `gp` by translating the GUI specification `gs` into appropriate Tcl commands. The I/O action `runWidget` (see Fig. 1) composes the functionality of `openWish` and `runWidgetOnPort`: it creates a new GUI communication port and runs the GUI specification on this port. Note that `runWidget` executes a GUI as an I/O action whereas `runWidgetOnPort` executes a GUI as a (concurrent) constraint. Therefore, `runWidget` is usually applied when one GUI is executed as the main program (Fig. 1), whereas `runWidgetOnPort` is applied when one GUI should be executed concurrently to other activities (see Sect. 5 for an example).

Layout structure of a GUI: A GUI specification is a description of the hierarchical layout structure of the GUI together with the actions that should be performed when, for instance,

[5] Most of the identifiers defined in the GUI library are prefixed by `Tk` since the library is based on the Tcl/Tk toolkit. Similarly, the name `openWish` refers to the fact that the windowing shell `wish` is used for the communication with the Tk toolkit.

a GUI button is pressed. To be more precise, a GUI specification is a term of the following data type:

```
data TkWidget a = TkButton (Port TkMsg -> a) [TkConfItem a]
                | TkCheckButton              [TkConfItem a]
                | TkEntry                    [TkConfItem a]
                | TkLabel                    [TkConfItem a]
                ⋮
                | TkRow [TkCollectionOption] [TkWidget a]
                | TkCol [TkCollectionOption] [TkWidget a]
```

Thus, a GUI specification is a simple widget (like a button or entry), a row (`TkRow`) or a column (`TkCol`) of widgets. The first parameter of `TkRow`/`TkCol` specifies additional options for the geometric alignment for widget composition, like centering, left alignment, expanding subwidgets if extra space is available:

```
data TkCollectionOption = TkCenter | TkLeft | ... | TkExpand
```

A button widget (`TkButton`) is intended to perform an action whenever the user presses this button. Therefore, an event handler is associated to each button widget (first parameter). Other widgets can also contain event handlers but they are optionally associated in the list of configuration items (see below). Since these event handlers are responsible for an event of a specific GUI, *event handlers* have type "`Port TkMsg -> a`" where `a` is the result type of the event handler which is either `Constraint` (for GUIs executed concurrently to other objects) or `IO ()` (for GUIs executed as an I/O action). Consequently, this type variable is also a parameter for the entire GUI structure.

Logical structure of a GUI: Before discussing event handlers in more detail, we must understand the concept to describe the logical structure of GUIs. As mentioned in the introduction, GUIs have a layout structure *and* a logical structure. While the layout structure is simply described by composing simple widgets into widget collections (`TkRow` and `TkCol`), the logical structure contains dependencies between different widgets and their event handlers. For instance, pressing some button usually results (after some computation) in the update of one or more other widgets. Although many GUI libraries (e.g., [13, 16]) are based on user-selected strings to identify the different widgets, we propose to use logical variables to refer to individual widgets which avoids many programming errors and provides for better abstractions. For this purpose, each primitive widget can have a number of items to configure the widget, like[6]

```
data TkConfItem a =
         TkRef TkRefType          -- a reference to this widget
       | TkText String            -- an initial text contents
       | TkWidth Int              -- the width of a widget
       | TkBackground String      -- the background color
       | TkCmd (Port tkMsg -> a)  -- an associated event handler
       ⋮
```

Most of these configuration items directly correspond to similar options in the Tk toolkit with the exception of `TkRef`. Since `TkRefType`, the type of all widget references, is abstract,

[6] Note that not all configuration items are meaningful for all widgets. This is checked at run time in our library, but in can be also checked at compile time with a more sophisticated type system, as proposed in [2].

i.e., no constructors of this data type are available to the user of the GUI library, the only reasonable way to use the `TkRef` item is with a free logical variable as shown in Fig. 2. If we run a GUI specification on a concrete port, this variable will be instantiated to a unique widget reference. The important point is that this variable can also be used in event handlers for other widgets in the same GUI. For this purpose, there are the following primitives to construct event handlers for GUIs:

```
tkExit     :: Port TkMsg -> IO ()
tkGetValue :: TkRefType -> Port TkMsg -> IO String
tkSetValue :: TkRefType -> String -> Port TkMsg -> IO ()
tkUpdate   :: (String->String) -> TkRefType -> Port TkMsg -> IO ()
```

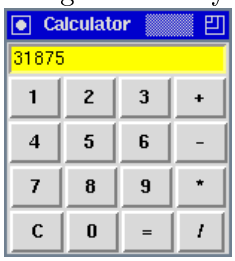
`tkExit` terminates the GUI, `tkGetValue` gets the (String) value currently stored in the widget referred by its first argument, `tkSetValue` sets the value stored in the referred widget, and `tkUpdateValue` updates the value according to an update function. The same set of primitives is also available for GUIs executed as a concurrent constraint:

```
tkCExit     :: Port TkMsg -> Constraint
tkCGetValue :: TkRefType -> Port TkMsg -> String -> Constraint
...
```

The event handlers attached to some widget are automatically invoked with the current GUI communication port whenever a GUI specification is executed by `runWidgetOnPort` (or `runWidget`), see also the examples in Fig. 1 and 2. Thus, a GUI specification is executed by sending commands that create the widget layout through the communication port followed by a scheduler which invokes the corresponding event handlers whenever the user performs some action on the GUI.

5 Example: A Calculator

We have already seen in Fig. 1 and 2 two specifications of simple GUIs using our library. However, many interactive applications contain a state which is shown and modified by a GUI. To demonstrate the implementation of these kinds of applications with our GUI concept, we present in the following the implementation of a simple calculator GUI as shown to the right. We model the calculator as an object which accepts the following messages:




```
data CalcMsg = Button Char | Display String
```

The message "`Button c`" is sent whenever the button `c` (e.g., `'1'`, `'2'`,...,`'+'`, `'*'`,...) is pressed. The message "`Display s`" is sent to get the current value of the operand, i.e., the argument `s` (which is usually an unbound variable) is instantiated with the current operand of the calculator. The calculator's local state is a pair `(d,f)` with the current operand `d` and an accumulator function `f` to be applied to `d` (this idea is due to [16]). With the techniques sketched in Sect. 3, we can implement calculator objects as follows (the rigid Boolean function `==` tests the equality of two ground expressions, i.e., e_1`==`e_2 reduces to `True` if both e_1 and e_2 are reducible to identical ground data terms; (e *op*) denotes the partial application of the operator *op* to the left argument e):

```
calcMgr :: (Int,Int->Int) -> [CalcMsg] -> Constraint
calcMgr eval rigid
calcMgr (d,f) (Display s : ms) = s=:=(show d) &> calcMgr(d,f) ms
```

```
calcMgr (d,f) (Button b  : ms)
 | isDigit b = calcMgr (10*d + ord b - ord'0', f) ms
 | b=='+'    = calcMgr (0, ((f d) +)) ms
 | b=='-'    = calcMgr (0, ((f d) -)) ms
 | b=='*'    = calcMgr (0, ((f d) *)) ms
 | b=='/'    = calcMgr (0, ((f d) `div`)) ms
 | b=='='    = calcMgr (f d, id) ms
 | b=='C'    = calcMgr (0, id) ms
```

Since the GUI needs a reference to the calculator object, we add it as a parameter `cm` to the GUI:

```
calc_GUI cm = TkCol [] [TkEntry [TkRef display, TkText "0"],
                        TkRow [] (map cbutton ['1','2','3','+']),
                        TkRow [] (map cbutton ['4','5','6','-']),
                        TkRow [] (map cbutton ['7','8','9','*']),
                        TkRow [] (map cbutton ['C','0','=','/'])]
  where display free
        cbutton c = TkButton (button_pressed c) [TkText [c]]
        button_pressed c gp = let d free in
                                  send (Button c) cm &>
                                  send (Display d) cm &>
                                  tkCSetValue display d gp
```

Here we exploit the higher-order features of the base language: To create the individual buttons, we use a generic function `cbutton` which is mapped on the particular lists of characters. The event handler `button_pressed` for each button sends a corresponding `Button` message to the calculator and shows the new operand of the calculator in the `display` widget. A new calculator application on a given GUI communication port `gp` is created by the following function:

```
runCalcOnPort gp | let cm free in
                     new calcMgr (0,id) cm & runWidgetOnPort (calc_GUI cm) gp
                 = done
```

Now the complete application is started by: `openWish "Calculator" >>= runCalcOnPort`

This implementation is modular similarly to the classical model-view-controller paradigm of Smalltalk-80 [12]. The application (represented by `calcMgr`) is completely independent to the user interface. All the programming techniques of the base language (laziness, higher-order functions, constraints, search etc.) can be used to implement the application. Due to the independence of the user interface and the application, it is also possible to have several GUIs (which represents the applications in different ways) for one application. In our implementation above, this is easily possible by changing the function `runCalcOnPort` to start one application together with several concurrent GUIs. This feature of our GUI design is also useful for developing GUIs for distributed applications where the GUI shows and manipulates different components of a distributed system. For instance, we have implemented a GUI for sending emails where the email address can be inserted by querying an address server running on some other machine. Due to lack of space, we omit a concrete example for this, but from the previous example it should be obvious how to use the distributed features of Curry (see Sect. 3 and [7]) in GUIs. Furthermore, we want to remark (due to lack of space, without a

concrete example) that the functional dimension of the base language is very useful to define new application-oriented graphical elements for GUIs, like radio buttons, groups of widgets connected by constraints etc.

6 Conclusions and Related Work

We have presented a library for implementing GUIs in the functional logic language Curry. We have exploited the functional as well as the logic features of Curry for the design of the library. The functional features are used to define the layout structure of GUIs and to built new application-oriented graphical abstractions. The logic features (logical variables) are used to specify the logic dependencies inside a GUI. This allows a compact and readable specification of GUIs as expressions of a particular data type rather than a sequence of actions to built the GUI. This is the reason why we consider this Tk library for Curry a *declarative* approach to GUI programming. As far as we know, this is the first approach to design a functional logic GUI library. Nevertheless, we want to relate it to some other approaches for GUI programming in declarative languages.

TkGofer [2] extends Gofer, a lazy functional language similar to Haskell, with a library for GUI programming. Similarly to our approach, the implementation is based on Tcl/Tk. TkGofer uses a monadic approach for GUI programming which forces the user to an imperative style. The different widgets are defined in a flat and sequential order and are later composed to a hierarchical layout structure by `pack` operations, similarly to Tcl/Tk scripts. This has the disadvantage that the layout structure is not defined together with the individual widgets and each widget must be given a name—even if they are used only once like the labels or buttons in Fig. 1 and 2. Furthermore, TkGofer is based on a sequential functional language. Thus, logic programming techniques like constraint solving as well as features for concurrent and distributed programming are not available.

The same holds for Fudgets [1], a GUI concept for lazy functional languages. Fudgets are processes accepting messages (for manipulating the state) and delivering messages (for sending information about an event). Primitive fudgets, like buttons, text inputs etc., are composed to more complex entities by connecting the input and output streams of the different fudgets. This approach makes it necessary to pass the GUI events via streams to the corresponding widgets to be manipulated by these events, which leads to less intuitive GUI specifications than using direct references to the corresponding widgets as in our approach.

Haggis [4] is a further GUI framework for Haskell. It is based on monadic I/O and uses concurrent processes to handle events. Instead of specifying a handler to be invoked when an event occurs, a new process is created that waits for this event. This has the drawback that widgets have two handlers in Haggis (one for the layout and one for the event handling).

Our proposal for GUI programming is not just an adaptation of existing GUI library designs to an integrated functional logic language, but it exploits the features of such an integrated language to support simple and readable GUI specifications. Since the logic programming features of the language are used to express the logical dependencies inside a GUI, an imperative style for constructing GUIs as in other approaches is avoided. Moreover, the features of the base language like higher-order functions, constraints, and concurrency can be used to build new application-oriented graphical abstractions in a simple way and to support a modular connection of the application program to the user interface. In particular, the distribution features of Curry largely simplify the implementation of user interfaces for

distributed systems.

The current definition of Curry has also some limitations which restricts the design of our GUI library. Currently, Curry has a Hindley-Milner like polymorphic type system [3]. It has been shown in [2] that a richer type system including type classes can improve the structure of a GUI library so that more errors can be caught at compile time. Since this is independent on the design issues discussed in this paper, we plan for the future to refine the design of our GUI library with a more sophisticated type system. Another topic for future work is to add the possibility to dynamically change the layout structure of GUIs which is currently not supported.

References

[1] M. Carlsson and T. Hallgren. Fudgets - A Graphical User Interface in a Lazy Functional Language. In *Conference on Functional Programming and Computer Architecture (FPCA'93)*. ACM Press, 1993.

[2] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proc. of the International Conference on Functional Programming (ICFP'97)*, pp. 251–262. ACM SIGPLAN Notices Vol. 32, No. 8, 1997.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[4] S. Finne and S. Peyton Jones. Composing Haggis. In *Proc. of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*. Springer, 1995.

[5] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[6] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

[7] M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.

[8] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PACS: The Portland Aachen Curry System. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/pacs/`, 1999.

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

[10] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.5). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.

[11] S. Janson, J. Montelius, and S. Haridi. Ports for Objects in Concurrent Logic Programs. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[12] G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface in Smalltalk-80. *Journal of Object-Oriented Programming*, Vol. 1, No. 3, pp. 26–49, 1988.

[13] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison Wesley, 1994.

[14] J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.

[15] E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pp. 251–273. MIT Press, 1987.

[16] T. Vullinghs, D. Tuijnman, and W. Schulte. Lightweight GUIs for Functional Programming. In *Proc. of the 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pp. 341–356. Springer LNCS 982, 1995.

[17] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Challenge Problems for the Integration of Logic and Connectionist Systems (Extended Abstract)

Steffen Hölldobler

Artificial Intelligence Institute
Department of Computer Science
Dresden University of Technology
D–01062 Dresden
sh@inf.tu-dresden.de

Keywords: Logic Programming, Recurrent Connectionist Networks

1 Motivation

Connectionist systems[1] exhibit many desirable properties of intelligent systems like, for example, being massively parallel, context–sensitive, adaptable and robust (see eg. [10]). It is strongly believed that intelligent systems must also be able to represent and reason about structured objects and structure–sensitive processes (see eg. [12, 29]). Unfortunately, we are unaware of any connectionist system which can handle structured objects and structure–sensitive processes in a satisfying way. Logic systems were designed to cope with such objects and processes and, consequently, it is a long–standing research goal to combine the advantages of connectionist and logic systems in a single system.

There have been many results on such a combination which involves propositional logic (c.f. [27, 32]). In [18] we have shown that a three–layered feed forward network of binary threshold units can be used to compute the meaning function of a propositional logic program. The input and output layer of such a network consists of a vector of units, each representing a propositional letter. The activation pattern of these layers represent an interpretation I with the understanding that the unit representing the propositional letter p is active iff p is true under I.

For certain classes of logic programs it is well–known that they admit a least model, and that this model can be computed as the least fixed point of the program's meaning function applied to an arbitrary initial interpretation [2, 11]. To compute the least fixed point in such cases, the feed forward network mentioned in the previous paragraph is turned into a recurrent one by connecting each unit in the output layer to the corresponding unit in the input layer. We were able to show — among other results — that such so-called RNNs, ie. recursive neural networks with a feed forward *kernel* as shown in Figure 1, converge to a stable state which represents the least model of the corresponding logic program. Moreover,

[1] A brief introduction to connectionist models is given in the Appendix.

in [8, 7] it was shown that the binary threshold units in the hidden layer of the kernel can be replaced by units with sigmoidal activation function. Consequently, the networks can be trained by backpropagation and after training new refined program clauses (or rules) can be extracted using, for example, the techniques presented in [42]. Altogether, this is a good example of how the properties of logic systems can be combined with the inherent properties of connectionist system.

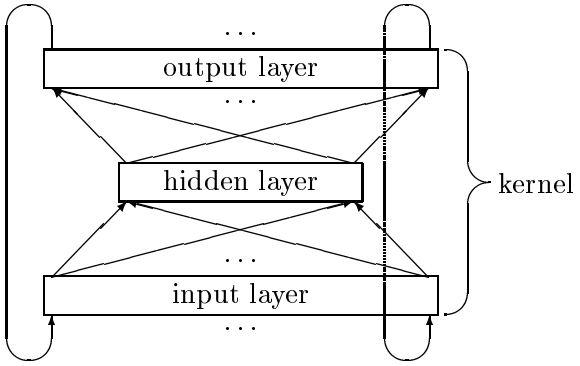


Figure 1: Schematic view of an RNN. The activation and output functions of the units may vary. Input and output layer are identical and each unit in the output layer is connected to its counterpart in the input layer with weight 1. The kernel is a fully connected feed forward network which may have one or more hidden layers.

Unfortunately, this does not solve the aforementioned problem because structured objects and structure–sensitive processes cannot be modeled within propositional logic but only within first- and higher–order logics. For these logics, however, similar results combining connectionist and logic systems are not known. One of the problems is that as soon as the underlying alphabet contains a single non–nullary function symbol and a single constant, then there are infinitely many ground atoms which cannot be represented locally in a connectionist network.

Interpretations for first–order logic programs can nevertheless be represented in connectionist systems if they are mapped to real numbers or vectors of real numbers as suggested in [20]: Let $\mathcal{P}$ be an acyclic logic programs with injective level mapping and $B_\mathcal{P}$ its Herbrand base. Each interpretation $I \in 2^{B_\mathcal{P}}$ is mapped to an $r \in \mathbb{R}$ by an invertible representation function $R : 2^{B_\mathcal{P}} \to \mathbb{R}$. Furthermore, for such programs a function $f_\mathcal{P} : \mathbb{R} \to \mathbb{R}$ is defined such that $R(T_\mathcal{P}(I)) = f_\mathcal{P}(R(I))$ and $f_\mathcal{P}$ is continuous. The latter property allows the application of a well–known result in the area of feed forward networks, namely that $f_\mathcal{P}$ can be approximated to any desired degree of accuracy by a feed forward network with sigmoidal activation function for the hidden layer units [14]. This leads to the following result:

Theorem 1 [20] *Let $\mathcal{P}$ be an acyclic logic program with an injective level mapping, $T_\mathcal{P}$ the meaning function associated with $\mathcal{P}$ and $f_\mathcal{P}$ the continuous real valued function corresponding to $T_\mathcal{P}$. Then for an arbitrary $\varepsilon > 0$, there exists a feed forward network with sigmoidal activation function for the hidden layer units and linear activation functions for the input*

and output layer units computing a function $\tilde{f}_{\mathcal{P}}$ *which satisfies*

$$\max_{x\in[-1,1]} |f_{\mathcal{P}}(x) - \tilde{f}_{\mathcal{P}}| < \varepsilon.$$

Using the mapping R^{-1} we obtain a feed forward network capable of approximating the meaning function $T_{\mathcal{P}}$ for an acyclic logic program $\mathcal{P}$ with injective level mapping. But what precisely is the approximation of such a meaning function in terms of interpretations? Let $\mathcal{P}$ be a logic program and $|\ |$ a level mapping for $\mathcal{P}$. An interpretation I *approximates* an interpretation J to a degree $N \in \mathbb{N}$ with respect to the level mapping $|\ |$, if for all atoms $A \in B_{\mathcal{P}}$ with $|A| < N$ we find $A \in I$ iff $A \in J$. Equivalently, I approximates J to a degree N iff $d_{\mathcal{P}}(I,J) \leq 2^{-N}$, where $d_{\mathcal{P}}(I,J) = 0$ if $I = J$ and $d_{\mathcal{P}}(I,J) = 2^n$ if I and J differ on some atom A of level n but agree on all atoms of lower level. $(2^{B_{\mathcal{P}}}, d_{\mathcal{P}})$ is a complete metric space [11] and if $\mathcal{P}$ is acyclic, then $T_{\mathcal{P}}$ is a contraction on this space [20, 36].

But we are not just concerned with the approximation of the meaning function $T_{\mathcal{P}}$, but are mainly interested in the approximation of the least fixed point of $T_{\mathcal{P}}$. Given a feed forward network according to Theorem 1 we can turn it into an RNN by adding recurrent connections with weight 1 between corresponding units in the output and input layer of the feed forward network. Does such a recursive network approximate the least fixed point of $T_{\mathcal{P}}$? Whereas Theorem 1 tells us the activation value $\tilde{f}_{\mathcal{P}}(r)$ is within ε of $f_{\mathcal{P}}(r)$ after the first iteration of the recursive network, this small difference could lead to a bigger difference in each step. Fortunately, this is not the case with the chosen encoding R:

Theorem 2 [20] *Let* $\mathcal{P}$ *be an acyclic logic program with an injective level mapping,* $T_{\mathcal{P}}$ *the meaning function associated with* $\mathcal{P}$ *and* $M_{\mathcal{P}}$ *the least fixed point of* $T_{\mathcal{P}}$. *For an arbitrary* $N \in \mathbb{N}$ *there exists a recursive network with sigmoidal activation function for the hidden layer units and linear activation functions for the input and output layer units computing a function* $\tilde{f}_{\mathcal{P}}$ *such that there exists an* $n_0 \in \mathbb{N}$ *such that for all* $n \geq n_0$ *and for all* $x \in [-1,1]$ *we find*

$$d_{\mathcal{P}}(R^{-1}(\tilde{f}^n_{\mathcal{P}}(x)), M_{\mathcal{P}}) \leq 2^{-N}.$$

This theorem tells us that for any initial value $x \in [-1,1]$ the interpretation represented by $\tilde{f}^n_{\mathcal{P}}(x)$, ie. the activation pattern of the output layer after an n–fold iteration through the recurrent network, and the least fixed point $M_{\mathcal{P}}$ of $T_{\mathcal{P}}$ agree in all atoms with a level less than N. In other words, we can approximate the least fixed point of the meaning function $T_{\mathcal{P}}$ of an acyclic logic program $\mathcal{P}$ with an injective level mapping arbitrarily well with a recurrent neural network.

Unfortunately, Theorems 1 and 2 are purely theoretical and we have not yet developed a real connectionist system for a truly first–order logic program based on these results. In [21] we have specified a logic system for model generation in a first–order calculus with constants and multi–place relations (datalogic) and have specified a recurrent neural network implementing this system. The logic system is based on the connection method [5] using reduction techniques which link logic systems to database systems. We have defined a calculus called BUR (for bottom–up reductions) which has the following properties: For unary relation symbols it is sound and complete, whereas for relation symbols with an arity larger than one it is complete but not necessarily sound; we have developed a criteria which guarantees that the result achieved in the latter case are sound; computations require only linear parallel time and linear parallel space.

Furthermore, we have extended the feed forward neural networks developed in [18] by turning the units into phase–sensitive ones. Thus, the variable binding problem is solved by phase–coding as first proposed in SHRUTI [38]. We have formally shown that the BUR calculus can be implemented in these networks. Compared to SHRUTI our networks consist only of two types of phase–sensitive units. The connection structure is an RNN with a four–layered feed forward neural network as kernel and, thus, is considerable simpler than the connection structure of SHRUTI. Besides giving a rigorous formal treatment this line of research may also lead to networks for reflexive reasoning, which can be trained using standard techniques like backpropagation.

But the BUR system does also not solve the main problem at hand, viz. to integrate the full power of logic systems within a purely connectionist setting. Although this is not the place to give a complete account of the current state of the art, we are unaware of a single connectionist inference system, whose ability to handle structured objects and structure sensitive processes comes even close to the degree that is achieved in conventional, first–order inference systems.

2 Challenge Problems

How can first–order terms be represented and manipulated in a connectionist system?

This is the main question that needs to be answered in order to built systems based on the results achieved in Theorems 1 and 2. The proposals made so far do not give a satisfying answer to this question: Structured connectionist networks as used in [16] are completely local. The unification and matching operations can directly be implemented in this networks. However, the number of units is quadratic and the number of connections even cubic wrt the size of the terms. It is not obvious at all how such networks can be learned. The structure is by far too complex for current learning algorithms based on the recruitment paradigm [9].

Vectors of fixed length are used to represent terms in the recursive auto–associative memory [34], the labeling recursive auto–associative memory [39] or in the memory based on holographic reduced representations [33]. Unfortunately, in extensive tests none of these proposals has led to satisfying results: The systems could not safely store and recall terms of depth larger than five [24].

In hybrid systems terms are represented and manipulated in a conventional way. But this is not a kind of integration I am hoping for because in this case results from connectionist systems cannot be applied to the conventional part.

The phase–coding mechanism suggested in SHRUTI and used in the BUR system restricts the first–order language to contain only constants and multi–place relation symbols.

We definitely need new ideas to solve this challenge problem! Recently, a new connectionist encodings for conventional data structures like counters and stacks [44, 19, 25] has been proposed. Although the current models are still quite simple, they appear to be promising candidates for a connectionist representation of terms.

How can first–order rules be extracted from a connectionist network?

To the best of my knowledge all rule extraction techniques for connectionist networks are propositional in the sense that they only generate propositional rules. For example, we have already mentioned that the propositional RNNs in [18] were slightly modified in [8, 7] such

that backpropagation could have been applied to the kernels and standard rule extraction techniques could have been used to extract new revised rules.

Theorems 1 and 2 guarantee the existence of RNNs with a feed forward kernel to approximate the least model of a first–order program. Backpropagation can again be used to train these kernels. But it is by no means obvious of how the rule extraction techniques known so far can be generalized such that first–order rules are extracted from these kernels.

On the other hand, first–order logic programs can be learned using techniques developed in the field of inductive logic programming [28]. I strongly believe that the combination of these techniques with propositional rule extraction techniques will lead to a solution for this challenge problem.

How can multiple instances of first–order rules be represented in a connectionist system?

One of the properties of first–order reasoning is that it cannot be determined in advance how many copies of a rule are needed to answer a given query or, equivalently, to prove a theorem. In local connectionist systems like CHCL [22] this problem is defined away by simply assuming that each rule is used only once. A similar assumption is made in SHRUTI, where each relation may be instantiated only a fixed number of times in one reasoning episode. The connectionist implementation is rather crude using so–called multiple instantiation switches. Basically, the networks or parts thereof are simply copied. This is not a general solution since even for datalogic programs exponentially many copies may be needed. The BUR system does not provide multiple copies, which is the reason for the fact that the system may be unsound if multiplace relation symbols are involved.

I propose to follow two lines of research in order to tackle this challenge problem: Firstly, work on growing cell structures [13] suggests that connectionist networks can be dynamically extended. I would like to see experiments which indicate that these techniques can be successfully applied in connectionist inference systems as well.

Secondly, Theorems 1 and 2 suggest that the problem of generating new instances of a rule can be mapped on the problem of obtaining a better approximation of the least model of a logic problem in the following sense: The level assigned to ground atoms occurring in the nth iteration of the meaning function for the first time should be higher than the level assigned to ground atoms which occur at earlier stages. If the accuracy of an approximation can then be correlated to the available hardware resources as in [19], we obtain a solution for this challenge problem.

How does a theory for the integration of logic and connectionist systems look like?

The results achieved so far on connectionist inference systems are more or less unrelated to each other. Different logics are mapped onto different connectionist systems and very often not much effort has been spent on (i) formally showing properties of the system, (ii) formally relating the logic to the system and (iii) formally relating the various systems to each other. There are some exceptions though, eg. in [16] it was proven that the presented connectionist system really solves the unification and matching problem or in [4] we have given a rigorous logical reconstruction of the backward reasoning version of SHRUTI.

I would like to see a theory where in various layers of increased expressiveness logics, their corresponding connectionist models, their time and space complexities, their properties concerning learning and rule extraction as well as learning and rule extraction algorithms are specified. Such a theory could be developed along the lines proposed in [18], the BUR system

and Theorems 1 and 2: In each layer the logic would be defined by a certain class of logic programs and the corresponding connectionist systems would be RNNs.

For example, if the logic programs are propositional, then interpretations are represented locally. The units in the corresponding RNN are logical threshold units. We have formally shown in [18] (among other results) that the kernel is an optimal parallel implementation of the meaning function $T_{\mathcal{P}}$ of the logic program $\mathcal{P}$. If learning shall be applied, then the threshold units in the hidden layer must be replaced by sigmoidal ones. If the programs are datalogic programs, then the corresponding RNN must be able to bind variables to constants which can be done by using phase coding as in SHRUTI and BUR. If the logic programs are full first order, then interpretations shall be represented by vectors of real numbers and models are only approximated, etc.

But the picture is by far not complete. Eg. if we really want to make use of the fact that connectionist computations are massively parallel, and achieve a considerable speedup in the computation time, then we must look for logics such that the corresponding entailment problems are in the class $\mathcal{NC}$. But already full propositional logic is unlikely to be in this class and, consequently, we have to look for less expressive logics for which optimal or efficient (in the sense of [26]) parallel algorithms for solving the entailment problem can be encoded in connectionist settings. First steps in this direction were taken eg. in [41].

The logics in such a theory shall not only be the standard monotonic ones, but we should also consider nonmonotonic ones. The results achieved in [18, 8, 7] are for normal logic programs with stable model semantics. By the way, nonmonotonic reasoning was originally proposed as a technique for "jumping to a conclusion". Nowadays conventional nonmonotonic reasoning systems have time and space complexities which are not at all in accordance with the original goal. It may well be that connectionist techniques may help to put the research in nonmonotonic reasoning techniques back on track.

A general theory for the integration of logic and connectionist systems could also be developed for symmetric networks [23]. Pinkas has shown that the problem of finding a model for a propositional logic formula is equivalent to finding a global minima in an energy function [32]. He has extended his results to nonmonotonic [31] and first–order logics [30]. Although a symmetric network is sequential in nature, it was shown in [40] how several units may switch at the same time without loosing the convergence property. Again, the picture is far from being complete.

Recently, techniques for greedy satisfiability testing (GSAT) in propositional logics [37] have been very successfully applied in conventional AI systems. It was shown that even first–order entailment problems can be solved using such techniques (see eg. [43]). Since GSAT is closely related to simulated annealing [3, 17] I expect that connectionist systems would greatly benefit from the new findings in greedy satisfiability testing.

Can such a theory be applied in a real domain outperforming conventional approaches?

All applications of connectionist inference systems that I have seen so far are toy examples. We have to come up with applications in real domains which outperform conventional approaches. This can only be done if we use hardware which exploits the massive parallelism inherent in connectionist networks. If we are reasoning in a logic whose entailment problem is in $\mathcal{NC}$ and an efficient or optimal parallel algorithm for deciding this problem is known, then it does not suffice to simulate this algorithm on a computer with just one or a few processors.

Because the general theory for integrating logic and connectionist systems shall be layered, the application I am looking for should have a similar structure. I propose to look for such an

application in the area of integrating the low–level control of a real robot with the high–level control developed in the area of cognitive robotics. Just recently it was proposed in [1] to model Brooks' subsumption architecture [6] by a logic based layered architecture. The first experiments are very promising. The approach appears to be feasible because the logic used in the first layers is so simple that the entailment problem can be easily solved. Such a simple logic is in $\mathcal{NC}$ and, hence, a good candidate for exploiting its inherent parallelism.

Such an application would also be a good showcase for learning and rule extraction: Even if such a robot is initialized with some knowledge, it must learn to behave in its environment, and this learning never stops. Consequently, the knowledge is constantly updated.

References

[1] E. Amir and P. Maynard-Reid II. Logic–based subsumption architecture. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 147–152, 1999.

[2] K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29:841–862, 1982.

[3] A. Beringer, G. Aschemann, H. Hoos, M. Metzger, and A. Weiß. GSAT versus Simulated Annealing. In A. G. Cohn, editor, *Proceedings of the European Conference on Artificial Intelligence*, pages 130–134. John Wiley & Sons, 1994.

[4] A. Beringer and S. Hölldobler. On the adequateness of the connection method. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 9–14, 1993.

[5] W. Bibel. On matrices with connections. *Journal of the ACM*, 28:633–645, 1981.

[6] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE J. Robotics and Automation*, RA-2(1):14–23, 1986.

[7] A.S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11:69–77, 1999.

[8] A.S. d'Avila Garcez, G. Zaverucha, and L.A.V. de Carvalho. Logic programming and inductive learning in artificial neural networks. In Ch. Herrmann, F. Reine, and A. Strohmaier, editors, *Knowledge Representation in Neural Networks*, pages 33–46, Berlin, 1997. Logos Verlag.

[9] J. A. Feldman. Memory and change in connection networks. Technical Report TR96, Computer Science Department, University of Rochester, 1981.

[10] J. A. Feldman and D. H. Ballard. Connectionist models and their properties. *Cognitive Science*, 6(3):205–254, 1982.

[11] M. Fitting. Metric methods – three examples and a theorem. *Journal of Logic Programming*, 21(3):113–127, 1994.

[12] J. A. Fodor and Z. W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. In Pinker and Mehler, editors, *Connections and Symbols*, pages 3–71. MIT Press, 1988.

[13] B. Fritzke. *Vektorbasierte Neuronale Netze*. Shaker Verlag, 1998.

[14] K.-I. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.

[15] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, 1991.

[16] S. Hölldobler. A structured connectionist unification algorithm. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 587–593, 1990.

[17] S. Hölldobler, H. Hoos, A. Strohmaier, and A. Weiß. The GSAT/SA–familiy — relating greedy satisifability testing to simulated annealing. Technical Report AIDA–94–17, Intellektik, Informatik, TU Darmstadt, 1994.

[18] S. Hölldobler and Y. Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[19] S. Hölldobler, Y. Kalinke, and H. Lehmann. Designing a counter: Another case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the KI97: Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Artificial Intelligence*, pages 313–324. Springer, 1997.

[20] S. Hölldobler, Y. Kalinke, and H.-P. Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–59, 1999.

[21] S. Hölldobler, Y. Kalinke, and J. Wunderlich. A recursive neural network for reflexive reasoning. In S. Wermter and R. Sun, editors, *Hybrid Neural Symbolic Integration*, LNAI. Springer, 1999. (to appear).

[22] S. Hölldobler and F. Kurfess. CHCL – A connectionist inference system. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, pages 318 – 342. Springer, LNAI *590*, 1992.

[23] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences USA*, pages 2554 – 2558, 1982.

[24] Y. Kalinke. Using connectionist term representation for first–order deduction – a critical view. In F. Maire, R. Hayward, and J. Diederich, editors, *Connectionist Systems for Knowledge Representation Deduction*. Queensland University of Technology, 1997. CADE–14 Workshop, Townsville, Australia.

[25] Y. Kalinke and H. Lehmann. Computations in recurrent neural networks: From counters to iterated function systems. In G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence*, volume 1502 of *LNAI*, Berlin/Heidelberg, 1998. Proceedings of the 11th Australian Joint Conference on Artificial Intelligence (AI'98), Springer–Verlag.

[26] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 17, pages 869–941. Elsevier Science Publishers B.V., New York, 1990.

[27] W. S. McCulloch and W. Pitts. A logical calculus and the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[28] S. Muggleton. *Inductive Logic Programming.* Academic Press, London, 1992.

[29] A. Newell. Physical symbol systems. *Cognitive Science*, 4:135–183, 1980.

[30] G. Pinkas. Expressing first-order logic in symmetric connectionist networks. In L. N. Kanal and C. B. Suttner, editors, *Informal Proceedings of the International Workshop on Parallel Processing for AI*, pages 155–160, Sydney, Australia, August 1991 1991.

[31] G. Pinkas. Propositional non-monotonic reasoning and inconsistency in symmetrical neural networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 525–530, 1991.

[32] G. Pinkas. Symmetric neural networks and logic satisfiability. *Neural Computation*, 3:282–291, 1991.

[33] T. A. Plate. Holographic reduced representations. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 30–35, 1991.

[34] J. B. Pollack. Recursive auto-associative memory: Devising compositional distributed representations. In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 33–39, 1988.

[35] Raúl Rojas. *Theorie der neuronalen Netze.* Springer, 1993.

[36] A.K. Seda and P. Hitzler. Topology and iterates in computational logic. In *Proceedings of the Twelveth Summer Conference on General Topology and its Applications: Special Session on Topology in Computer Science*, 1997.

[37] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 440–446, 1992.

[38] L. Shastri and V. Ajjanagadde. From associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioural and Brain Sciences*, 16(3):417–494, September 1993.

[39] A. Sperduti. Labeling RAAM. Technical Report TR-93-029, International Computer Science Institute, Berkeley, CA, 1993.

[40] A. Strohmaier. Multi–flip networks: Parallelizing gensat. In G. Brewka, C. Habel, and B. Nebel, editors, *KI–97: Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Artificial Intelligence*, pages 349–360. Springer, 1997.

[41] Antje Strohmaier. *Logisches Schließen mit massiv parallelen Methoden*, volume 169 of *DISKI.* infix Verlag, Sankt Augustin, Germany, 1997.

[42] G.G. Towell and J.W. Shavlik. Extracting refined rules from knowledge–based neural networks. *Machine Learning*, 131:71–101, 1993.

[43] D. S. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.

[44] J. Wiles and J. Elman. Learning to count with a counter: A case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the Annual Conference of the Cognitive Science Society*, 1995.

Appendix: Connectionist Models

This appendix contains a brief introduction to connectionist models containing all notions used in the paper. For a more extensive introduction the reader is referred to the literature (see e.g. [10, 15, 35]).

A *connectionist network* consists of a set of units and connections between these units. A *unit* U is characterized by its *potential* $p \in \mathbb{R}$, its *value* v, and its *input vector* $(i_1, \ldots, i_n)$. The units are connected via a set of directed and weighted connections. If there is a connection from unit U_j to unit U_k, then w_{kj} denotes the weight associated with this connection and $w_{kj}v_j$ denotes the input i_j received by U_k from U_j. The potential p_k and value v_k of the unit U_k are computed wrt to an *activation* and an *output* function respectively. An activation function $\phi : \mathbb{R} \to [0, 1]$ is called *sigmoidal* if it is non–decreasing, $\lim_{\lambda\to\infty}(\phi(\lambda)) = 1$, and $\lim_{\lambda\to-\infty}(\phi(\lambda)) = 0$. A unit is said to be a *binary threshold* unit if its potential and output are computed as follows, where j iterates through the set of units which have a connection to U_k, $\theta_k \in \mathbb{R}$ is the *threshold* of U_k and t denotes linear time.

$$p_k(t) = \textstyle\sum_j w_{kj}v_j(t-1) \qquad v_k(t) = \begin{cases} 1 & \text{if } p_k(t) > \theta_k \\ 0 & \text{if } p_k(t) < \theta_k \\ v_k(t-1) & \text{if } p_k(t) = \theta_k \end{cases}$$

The re–computation of a unit's potential and output is called an *update*. Units can be updated synchronously or asynchronously.

A *layer* is a vector of units. An *n–layered feed–forward network* consists of an *input* layer, $n-2$ *hidden* layers, and an *output* layer, where $n \geq 2$. Each unit occurring in the i–th layer is connected to each unit occurring in the $i+1$–st layer, $1 \leq i < n$. All units are updated synchronously. Let r and s be the number of units occurring in the input and output layer, respectively. A multi–layered feed–forward network computes a function $f : \{0, 1\}^r \to \{0, 1\}^s$ as follows. The input vector is presented to the input layer at time t_1 and propagated through the hidden layers to the output layer. At each time point all units update their potential and value. At time t_n the output vector is read off the output layer.

An *n–layered recurrent network* consists of an n–layered feed–forward network called *kernel* such that the number of units in the input and output layer are identical. Furthermore, each unit in the k–th position of the output layer is connected with weight 1 to the unit in the k–th position of the input layer, where $1 \leq k \leq n$ and n is the number of units in the output (or input) layer. Figure 1 shows a 3–layered recurrent network.

Deliberate Agents Reconcile Reactive and Goal-Directed Agents

Pierre Bonzon

HEC, University of Lausanne
1015 Lausanne, Switzerland
pierre.bonzon@hec.unil.ch

Abstract

We propose a logical construction of deliberate agents that reconcile reactive and goal-directed agents. In a first simple model, agents are situated objects which respond to messages by invoking methods. By extending these objects with a special method, we then reproduce the behaviour of deliberate reactive agents. By further enhancing this model with practical reasoning rules and a process of goal reductions, we finally obtain a model of deliberate BD agents. Using a theoretical result by Hindrik & al., it is argued that these agents subsume BDI agents. The operational semantics of these various models is given under the form of logic programming procedures.

1. Introduction

Agents are sometimes described as (re)active objects which also exhibit pro-active (or goal directed) behaviours. Accordingly, we would like to illustrate, using a logical construction, that agents = reactive objects + pro-active extensions. In a first simple model, agents are situated objects which respond to messages by invoking methods. By extending these objects with a special method, we can then reproduce the behaviour of deliberate *reactive* agents. By further enhancing this model with practical reasoning *rules* and a process of *goal* reductions taken from [Hindriks & al. 97], we finally obtain a model of deliberate BD agents. Conspicuously absent in this model is the representation of *intentions*, which together with beliefs and goals (or desires) define the core of BDI agents. Recent results [Hindriks & al. 98] have shown however that the history of intentions can be represented by complex goal structures (i.e. goals consisting of lists of goals). As a result, there is no need to represent intentions as separate lists of plans waiting to be executed. This allows us to argue that deliberate BD agents subsume BDI agents.

This paper is organized as follows. In section 2 we present *AgentClass(0),* our simple model of situated objects. In section 3, we develop *AgentClass(1)*, a model that extends *AgentClass(0)* with the functionality of deliberate reactive agents as defined in [Wooldridge 99]. In section 4, we introduce *AgentClass(2)*, our enhanced model of deliberate BD agents.

We present the operational semantics of these various models under the form of logic programs. We thus inherit the whole machinery of logic programs interpretation, and avoid the explicit manipulation of unifiers. At the same time we get *directly executable* specifications. This can prove useful for prototype experimentation, as the corresponding interpreter (or even compiler) will take care of both the usual aspects of object-oriented programming (including agent creation and message interpretation) and the less conventional aspect of agent-oriented programming (such as goal reduction and rule-based planning).

In full generality, our model definitions rely on a few concepts related to *contextual deductions* presented in Appendix I. For practical purposes, some subtleties associated with the theory of contexts can be ignored, with the result that deduction in context, represented by the usual *ist(C,P)* modality meaning "P is true in context C", can be put to work via the classical "vanilla" meta-interpreter of logic programming (see the end of Appendix I). All model definitions and implementations rely on an abstract data type for *contexts* defined in Appendix II.

2. AgentClass(0): a simple model of agents as situated objects

In this section, we define *AgentClass(0)*, a simple model of agents which introduces our approach for implementing and experimenting with objects.

This first model can be described informally as follows:

- agents are "situated objects" encapsulating a set of methods and a set of beliefs
- actions performed by a agent result in updating this agent's beliefs
- as classical objects, agents respond to messages invoking methods

Formally, we propose the following definitions:

Definition (agents): agents are objects encapsulating two contexts, i.e. a context *methods(Agent)* containing methods to be invoked by messages and a context *beliefs(Agent)* containing explicit beliefs; these can be any contextual fact and/or implications (see Appendix I).

Definition (methods and messages): methods have the format *method(Agent.Call,Body),* where *Body* is a list of predefined procedure calls and/or user defined messages*;* messages sent to an agent have the format *Agent.Call* ; messages are interpreted by the procedure

```
Agent.Call :- instance(methods(Agent)),method(Agent.Call,Body)),
              execute(Body).
```

with

```
execute([]).
execute([H|T]):- call(H), execute(T).
```

Agent are created with the predefined procedure *newAgent(Agent),* interpreted as follows:

```
newAgent(Agent):- new(methods(Agent)),
                  new(beliefs(Agent)),
                  addList(methods(Agent),template(methods(Agent))).
```

This procedure involves the creation of new contexts followed by the instantiation, within context `methods(Agent)`, of methods from the following "class template"

```
template(methods(Agent)) :
         [method(Agent.assume(Slot,List), [addList(Slot(Agent),List)]),
          method(Agent.add(Slot,P), [add(Slot(Agent),P)]),
          method(Agent.delete(Slot,P), [delete(Slot(Agent),P)]),
          method(Agent.bel(P), [ist(beliefs(Agent),P)])].
```

According to this template, the *assume* method will insert a list of formulas in any given context. The instance *add* and *delete* methods will insert or remove individual formulas. The instance method `bel(P)` allows an `Agent` to deduce facts `P` which are derivable in the context `beliefs(Agent)`: by definition, any such `P` will be called an *implicit belief* of `Agent`.

Example

Let us consider a variant of the so-called "vacuum cleaner world" introduced in [Russel & Norvig 95] and developed in [Wooldridge 99]. The following context contains three methods representing the basic actions of a vacuum cleaner robot:

```
moves:[method(Agent.suck,[Agent.bel(in(X,Y)),
                          Agent.delete(beliefs,dirt(X,Y)),
                          write(Agent.suck)]),

       method(Agent.forward,[Agent.bel(in(X,Y)),
                             Agent.bel(facing(F)),
                             Agent.delete(beliefs,in(X,Y)),
                             (F=north, X1=X,Y1 is Y+1;
                              F=east,  X1 is X+1,Y1=Y;
                              F=south, X1=X,Y1 is Y-1;
                              F=west,  X1 is X-1, Y1=Y),
                             Agent.add(beliefs,in(X1,Y1)),
                             write(Agent.forward)]),

       method(Agent.turn,[Agent.bel(facing(F)),
                          Agent.delete(beliefs,facing(F)),
                          (F=north, F1=east;
                           F=east,  F1=south;
                           F=south, F1=west;
                           F=west,  F1=north),
                          Agent.add(beliefs,facing(F)),
                          write(Agent.turn])].
```

In addition to being able to suck up dirt, the robot can move forward one step or turn right 90^0. Any action results in updating the robot's beliefs (the connection with sensors and motors reflecting the actual physical effect of actions is out of the scope of this work; instead, a written trace of the sequence of actions is produced). The following context assertion defines the initial state of the world:

```
room:   [in(0,0),facing(north), dirt(0,2)].
```

The sequence of messages needed to create agent `robot(1)` is as follows:

```
newAgent(robot(1)).
robot(1).assume(methods,moves).
robot(1).assume(beliefs,room).
```

The following message will instantiate all `robot(1)` beliefs

```
?- robot(1).bel(P).
 P = in(0,0) ;
 P = facing(north) ;
 P = dirt(0,2)
```

The following message will instruct `robot(1)` to move forward

```
?- robot(1).forward.
 robot(1).forward
```

This action will be reflected in the following new belief instantiation

```
?- robot(1).bel(in(X,Y)).
 X=0
 Y=1
```

If instructed to suck, robot(1) will obey this order in any position (as `retractall/2`, which is used to implement `delete`, always succeeds).

3. *AgentClass(1)*: a model of deliberate reactive agents

In this section, we introduce AgentClass(1), a model that extends *AgentClass(0)* with the functionality of deliberate *reactive agents*, as defined in [Wooldridge 99]. This model can be intuitively characterised as follows:

in addition to being able to respond to messages, deliberate reactive agents respond to changes in their environment (including changes resulting from their own actions); towards this end, they successively

- deduce an appropriate reaction
- invoke the corresponding method.

Formally, we have the following definition:

Definition: deliberate reactive agents are situated objects which have explicit beliefs `do(Action)`
and/or
`Q -> do(Action)`
and whose behaviour is determined by a calling a method `react(Action)` defined as

```
method(Agent.react(Action), [Agent.bel(do(Action)),
                             Agent.Action])
```

Indeed, when called, this method will first deduce an implicit belief of the form `do(Action)`
representing an appropriate reaction, and then invoke the corresponding method with `Agent.Action`.

Example

In order to define a reactive vacuum cleaner, let us consider the following context `reactions` containing the "hardwired navigation algorithm" [Wooldridge 99] of a robot continuously moving up and down on a 3X3 grid; when reaching the top right corner (2,2), it will directly head back home to (0,0), and start again.

```
reactions:
[(in(X,Y),dirt(X,Y))                          -> do(suck),
 (in(X,Y),not dirt(X,Y),facing(north),Y<2)    -> do(forward),
 (in(X,Y),not dirt(X,Y),facing(south),Y>0)    -> do(forward),
 (in(X,2),not dirt(X,2),facing(north),X<2)    -> do(turndown),
 (in(X,0),not dirt(X,0),facing(south),X<2)    -> do(turnup),
 (in(2,2),not dirt(2,2),facing(north))        -> do(turnback),
 (in(2,0),not dirt(2,0),facing(south))        -> do(turn),
 (in(X,0),not dirt(X,0),facing(west),X>0)     -> do(forward),
 (in(0,0),not dirt(0,0),facing(west))         -> do(turn)].
```

In addition to the basic actions previously defined in the `moves` context, this navigation involves sequences of basic moves defined in the following context:

```
sequences:
[method(Agent.turnback,[Agent.turn, Agent.turn]),
 method(Agent.turndown,[Agent.turn, Agent.forward, Agent.turn]),
 method(Agent.turnup,[Agent.turn, Agent.turn, Agent.turn, Agent.forward,
                      Agent.turn, Agent.turn, Agent.turn])].
```

The sequence of messages to create an instance `robot(1)` must be extended with

```
robot(1).assume(beliefs,reactions).
robot(1).assume(methods,sequences).
```

Successive invocations of `react(Action)` calls, e.g. under the form of a Prolog loop

```
run :- repeat, robot(1).react(Action).
```

will result in the following sequence of moves showing a truly reactive behaviour

```
?- run.
robot(1) . forward
robot(1) . forward
robot(1) . suck
robot(1) . turn
robot(1) . forward
robot(1) . turn
robot(1) . forward
robot(1) . forward
...
```

N.B. In order to ensure single steps within each reaction, define the loop as

```
run :- repeat, call((robot(1).react(Action),!)).
```

4. *AgentClass(2)*: a model of deliberate BD agents

In this section we implement a model that extend *AgentClass(1)* with the functionality of *AgentSpeak(2),* the BD agent model shown by Hindriks & al. to be at least as expressive as *AgentSpeak(L),* itself a standard reference for BDI agent models. This model can be characterised as follows:

- in addition to *methods* and *beliefs*, agents further encapsulate *rules* and *goals*
- rules have a *head*, a *guard* and a *body*; a rule whose head matches an agent's goal and whose guard elements are implicit agent's beliefs can substitute its body for the agent's matching goal
- agents behaviour is determined by a process of goal reductions

Formally, we propose the following definitions:

Definition (BD agents): deliberate BD agents are given by four contexts, i.e. *methods(Agent)*, *rules(Agent)*, *beliefs(Agent)* and *goals(Agent).*

Definition (methods, messages): the definition of section 2 still applies.

Definition (rules): rules have the format *rule(Agent.Call,Guard,Body),* where *Guard* is a list of contextual facts, and *Body* a list of goals.

Definition (goals): a goal can be either *simple* or *complex*; a simple goal is a message invoking a rule or a method; a complex goal is a list of simple or complex goals; goals are reduced using a agent method defined as follows

```
method(Agent.reduce(Goal), [instance(goals(Agent),Goal),
                            transition(Goal,NewGoals),
                            delete(goals(Agent),Goal),
                            (NewGoals=[]-> true;
                             add(goals(Agent),NewGoals))]).
```

Transitions for simple goals invoking a method are defined as

```
transition(Agent.Call,[]):- Agent.Call.
```

Transitions for simple goals invoking a rule are defined as

```
transition(Agent.Call,Body) :- instance(rules(Agent),
                                        rule(Agent.Call,Guard,Body)),
                               check(Agent,Guard).
```

with

```
check(_,[]).
check(Agent,[Head|Tail]) :-  Agent.bel(Head),
                             check(Agent,Tail).
```

Transitions for complex goals are defined as

```
transition([Head|Tail],NewGoal) :- transition(Head,NewHead),
                                   (NewHead=[] -> NewGoal=Tail;
                                    (Tail=[] -> NewGoals=NewHead;
                                     NewGoal=[NewHead|Tail])).
```

We have the following result.

Proposition : any deliberate reactive agent can be defined as a deliberate BD agent, i.e. *AgentClass(2)* agents subsume *AgentClass(1)* agents.

Proof: let us consider a context *plans* containing a single rule

```
plans:[rule(Agent.do(Action),[do(Action)],
                             [Agent.Action,
                              Agent.do(NewAction)])].
```

together with the following "generic" messages to be used when creating any `Agent`

```
Agent.assume(rules,plans),
Agent.add(goals,Agent.do(Action))
```

Calling `Agent.reduce(Goal)` will retrieve the goal `Agent.do(Action)`. A first transition will in turn retrieve the rule `do(Action)`. Similarly to calling `react(Action)`, checking this rule's guard will lead to deduce an implicit belief `do(Action)`

This first transition will then establish a new complex goal (sometimes called a *plan*) given by the rule's body. A subsequent `Agent.reduce(Goal)` will lead to a complex transition. This in turn will produce a third transition (for a simple goal). Similarly again to `react(Action)`, this last transition will invoke the corresponding method `Agent.Action`

At the same time, the second transition (for the complex goal) will set up the remaining goal `Agent.do(NewAction)` as a single goal identical to the initial goal. Were are thus back were we started and, similarly to `react(Action)`, ready for a new deliberative reaction .

Example

Let us extend the sequence of messages to create an instance `robot(1)` with

```
robot(1).assume(rules,plans).
robot(1).add(goals,robot(1).do(Action)).
```

Successive invocations of `reduce(Goal)`, e.g. under the form of a loop

```
run :- repeat, call((robot(1).reduce(Goal),!)).
```

will result in the same sequence of moves as before

```
?- run.
robot(1) . forward
robot(1) . forward
robot(1) . suck
robot(1) . turn
robot(1) . forward
...
```

As convincingly argued by [Hindriks & al. 97], rules can be used both to specify the means to achieve a particular goal, and to revise goals: in this last case, rules encode the reflective capabilities of an agent. In other words, deliberative BD agents are not only capable of *fixed*, reactive behaviour, but can also exhibit *flexible* behaviours. To quote form these authors, modifying complex goals can lead to express things like*: "if the agent has adopted a plan P (to achieve a goal G), and believes S, then agent should (consider) revising plan P and substitute it with a new plan P' or goal G'* "
N.B. As noted above, *plans* are complex goals introduced by rule bodies.

In order to illustrate this, let us consider a vacuum cleaner robot that could decide, at any time, either to *work*, i.e. to move up and down (including returning home to start again) , or *retreat* back home and stop. These two different behaviours will give rise to two separate set of rules replacing the single `do(Action)` rule given before. These

will in turn correspond to two possible goals, i.e. `do(work(Action))` and `do(retreat(Action))`. These rules are defined as follows:

```
plans: [rule(Agent.do(work(Action)),    [work, not retreat
                                         do(work(Action))],
                                        [Agent.Action,
                                         Agent.do(work(NewAction))]),
        rule(Agent.do(work(switch)),    [work,retreat],
                                        [Agent.delete(beliefs,work),
                                         Agent.do(retreat(NewAction))]),
        rule(Agent.do(retreat(Action)),[not work,retreat,
                                         do(retreat(Action))],
                                        [Agent.Action,
                                         Agent.do(retreat(NewAction))]),
        rule(Agent.do(retreat(switch)),[work,retreat],
                                        [Agent.delete(beliefs,retreat),
                                         Agent.do(work(NewAction))])].
```

In short, if the current belief is to work, then the robot will continue to do so until a new belief (i.e. *retreat*) leads him to switch behaviour (i.e. adopt a different goal) and vice-versa. The navigation algorithm must accordingly be modified as follows

```
reactions:
 [(in(X,Y),dirt(X,Y))                          -> do(work(suck)),
  (in(X,Y),not dirt(X,Y),facing(north),Y<2)    -> do(work(forward)),
   . . .
  (in(0,0),not dirt(0,0),facing(west))         -> do(work(turn)),
  (in(X,Y),facing(north),not (X=0,Y=0))        -> do(retreat(turnback)),
  (in(X,Y),facing(south),Y>0)                  -> do(retreat(forward)),
  (in(X,0),facing(south))                      -> do(retreat(turn)),
  (in(X,0),facing(west),X>0)                   -> do(retreat(forward)),
  (in(0,0),facing(west))                       -> do(retreat(turn))].
```

This flexible behaviour can be simulated with an interruptible run defined as

```
run :- repeat, call((robot(1).reduce(Goal),!)), until(grab(_)).
```

leading to following results

```
?- robot(1).add(beliefs, work), run.
robot(1) . forward
robot(1) . forward
robot(1) . suck
robot(1) . turn
robot(1) . forward
robot(1) . turn
robot(1) . forward
robot(1) . forward
yes
```

Interrupt occurred here after hitting any key; the robot is in (1,0),facing south.

```
?- robot(1).add(beliefs, retreat),run.
robot(1).turn
robot(1).forward
robot(1).turn
```

The robot is back home and has no more goals: `reduce(Goal)` no more succeeds.

N.B. In order to interrupt this last run, redefine `reduce(Goal)` to always succeed.

An important result by [Hindriks & al 98] is that *AgentSpeak(2)*, a model of deliberate BD agents, can be used to simulate, without loss of expressive power, *AgentSpeak(L)* [Rao 96], a standard model for BDI agents. In short, events and intentions, which are *AgentSpeak(L)* model constituents, can both be represented by complex goal structures. Consequently, there is no need to represent intentions as separate lists of plans waiting to be executed.

This allows us to argue in turn that deliberate BD agents also subsumes BDI agents.

Proposition: *AgentClass(2)* agents subsume *AgentSpeak(L)* agents.

***Proof*:** no formal proof will be given; the informal argument goes as follows: by transition, if *AgentSpeak(2)* has at least the same expressive power as *AgentSpeak(L)*, then this leaves us with the task to prove that *AgentClass(2)* has at least the same expressive power as *AgentSpeak(2)*.

A close look first reveals that the syntax of *AgentSpeak(2)* and *AgentClass(2)* are syntactic variants: in particular, *P?* is represented by the method `bel(P)`. As for their semantics, the operational semantics of *AgentSpeak(2)* is given by a transition system which consists of a set of *transition rules.* These rules rely on a function specifying the update semantics of basic actions. In contrast, the operational semantics of *AgentClass(2)* is given by a set of (pure) logic programming procedures. These procedures themselves rely on a abstract data structure for contexts (see Appendix I and II) which is similarly used to specify the update semantics of basic actions.

A close look further reveals that to each *AgentSpeak(2)* transition rule corresponds an "equivalent" *AgentClass(2)* procedure: `reduce(goal)` corresponds to the transition for a goal base, `transition(Agent,Call,Body)` to the transition for rule application, `transition(Agent,Call,[])` to the execution rule for action, and `transition([Head|Tail],NewGoal)` to the execution rule for sequential decomposition. As for the remaining execution rule for first-order test, it is taken care of by the fact that *P?* and `bel(P)` have the same interpretation, i.e. first order test for provability (or truth) in the "context" of an agent's beliefs.

On the other way around, it should be noted that the *AgentSpeak(2)* counterparts of the general methods available to *AgentClass(2)* agents are limited to basic actions. This could be an indication that *AgentClass(2)* has strictly more expressive power than *AgentSpeak(2)* and, by transition, than *AgentSpeak(L)* also. One could object that this ability relies on Prolog, not on a clearly delineated and defined set of constructs belonging to a (new) programming paradigm. However, if AgentClass(2) agents behave as situated objects, it is not by the virtue of Prolog, but because of their ability to respond to messages, a process which happens to be defined in Prolog, but could also (actually not so easily) be defined in an other formalism.

5. Related work

Following the *S*tructural approach to *O*perational *S*emantics, [Hindriks & al 98] use SOS-transition rules to specify the meaning of each programming construct in their language. Actually, there is not much difference between writing

a) SOS-transition rules, which rely on an abstract function specifying the update semantics of basic actions (as they do)

b) Prolog (or more generally logic programming) transition procedures, which rely on an abstract data type specifying the same basic actions (as we do).

In both cases, transitions are inference rules, and the agent model is a *deduction system* "grounded" on the semantics of basic actions. Because they don't bother with actual implementations, SOS-transitions (similarly to other formalisms based on abstract structures) lend themselves easily to extended modes of operation, including non deterministic choice, concurrency and parallelism. On the other hand, concrete list manipulations in Prolog lead to executable specifications, and Prolog extensions would be needed to accommodate concurrency and/or parallelism.

A popular logical approach to specify agent models (exemplified by the GOLOG system [Levesque & al. 97]), is to define extended versions of the *situation calculus.* The situation calculus is a model for representing the effect of actions. It relies on an *implicit* state description (the situation variable) based on axioms, rather than on an *explicit* data base of facts (i.e. the beliefs), this later approach being followed by AgentSpeak(i) and AgentClass(i). The situation variable allows agents to reason about actions and changes. Given a description of the world and a particular goal, GOLOG can verify whether a *predefined* program is applicable. If successful, it will then deliver a sequence of basic actions achieving this goal. It is however questionable whether this approach captures the essence and potentiality of deliberate agents.

There is another distinction to be made here, i.e. whether one is using logic to specify
. a first order theory, i.e. a set of formulas closed under a given deduction relation
. an arbitrary deduction system.
In the first case (an example of which is the situation calculus), logic is used strictly as an object language to define non-logical axioms. If these are made to rely on full first order logic (including negation, existential quantifiers, disjunctive formulas, and so on), then unfortunately the resulting specifications become quite unreadable, and often intractable.

As noted above, AgentClass(i) is an example of the second case. It uses logic both as an object language (e.g. to specify actions an beliefs) and as a *meta language* to define new inference rules. In this case, it is advisable to use a restricted subset of first order logic, like the one used in logic programming (i.e. definite Horn clauses), both for the object and meta language, because it is easier to read and offers more effective ways to deduce true formulas.

6. Conclusion and future work

Besides reactivity and pro-activeness, a third desirable capability usually associated with intelligent agents is their capacity to interact with other agents. Unfortunately, simple models of interacting agents are not easy to develop. Not surprisingly, this dimension was not taken into account in our preceding developments. Among other problems, the following issues, at least, should be dealt with: agents should
- have their own thread of control
- enjoy distributed beliefs
- communicate.

The first issue is related to the theory and implementation of concurrent objects, for which there is already a large body of models and technical solutions. The second issue confronts us with the somewhat new problem of generalising contextual deduction in order to take into account the beliefs that agents can have about the beliefs of other agents. These would include the representation of situations in which "agent A believes that agent B does not believe that he, agent A, believes that …". As for the last issue, it calls first for clear semantic models of communicative actions. A promising approach towards this goal is offered by the recent proposal by [Hindriks & al. 99] of two pairs of synchronisation primitives. The first pair, i.e. *tell* and *ask* can be used to synchronise an information exchange. The second one, i.e. *req* and *offer* similarly allows for an information request. Their introduction within an extended AgentClass(3) model is currently being investigated.

Appendix I: a model of contextual deduction

We assume a minimal background in computational logic, including the definition of a substitution θ, of an instance eθ and of a most general unifier. We represent variables with capital letters, ordinary variables (e.g. `X,Y`) standing for terms, and meta variables (e.g. `P,Q`) standing for arbitrary formulas.

In order to define a language ***L*** of *contextual implications,* let us consider a two-sorted predicate calculus, and denote by ***C*** the set of terms of the context sort, and by ***A*** the set of atomic formulas of the discourse sort [Buvac 96]. ***L*** is then defined as the least set satisfying the equation $\boldsymbol{L}=\boldsymbol{A}\cup\boldsymbol{L}\rightarrow\boldsymbol{L}\cup ist(\boldsymbol{C},\boldsymbol{L})$, with modality *ist(c,p)* meaning "formula $p\in\boldsymbol{L}$ is true in context $c\in\boldsymbol{C}$". The subset $\boldsymbol{F}\subset\boldsymbol{L}$ whose formulas do not contain any implication defines the language of *contextual facts* .

In order to define theories, i.e. sets of formula closed under contextual deduction, or, equivalently, deductible in a formal system of logical axioms and inference rules augmented with context-dependant non-logical axioms, let us assume that non-logical axioms are contained within *context assertions*.

Definition (context assertion): A context assertion is a formula $\alpha:\Phi$ where α is a term denoting a context and Φ is a set of contextual implications, i.e. $\Phi\subset\boldsymbol{L}$, with all variables universally quantified

Given $\alpha:\Phi$ and a context *C*, we will say that $\alpha:\Phi$ *relates to C* if *C* can be unified with α .

Definition (axiom instance): If $\alpha:\Phi$ relates to *C*, with $C\theta=\alpha\theta$, then any formula *P* which can be unified with $\phi\theta\in\Phi\theta$ is called a (non-logical) *axiom instance* for *C*.

Let *instance(C,P)* denote "P is an axiom instance for context C".

In order to achieve *contextual deduction*, let us consider the following axiom system including *a link to the non-logical axioms* of a theory contained within context assertions:

logical axioms

$ist(C, P\rightarrow Q)\rightarrow ist(C, P)\rightarrow ist(C,Q)$	*distribution axiom (or axiom K)*
$ist(C,P)\rightarrow ist(C_1,ist(C,P))$	*semi-flatness axiom*

inference rules

from instance(C,P) *infer* ist(C,P)	*link to non-logical axioms*
from P *and* $P\rightarrow Q$ *infer* Q	*modus ponens*

This axiomatic system, which is equivalent to a system of natural deduction in context [Bonzon 97] **,** can also be represented by the following Prolog program :

`ist(C,P->Q) -> ist(C,P) -> ist(C,Q).`	*distribution axiom*
`ist(C,P) -> ist(C1,ist(C,P)).`	*semi-flatness axiom*
`ist(C,P):- instance(C,P).`	*link to non-logical axioms*
`Q:- P->Q,P.`	*modus ponens*

We conclude this review of contextual deduction with the definition of a *contextual deduction relation*.

Definition (contextual deduction relation $\vdash$): A contextual fact *P* is derivable in context *C*, noted $C\vdash P$, iff $\exists$ a derivation (in the usual sense) of *ist(C,P).*

A Simplified Model

For practical purposes, semi-flatness properties of contexts can be ignored, and implication chains can be reduced using the derived conjunctive operator "," defined as

$P_1\rightarrow P_2\ \ldots\rightarrow P_{n-1}\rightarrow P_n\equiv(P_1,P_2\ldots,P_{n-1})\rightarrow P_n$

As a result, the program of figure 1 reduces to the "vanilla" meta-interpreter of logic programming

```
ist(C,P)      :-  instance(C,P).
ist(C,(P,Q))  :-  ist(C,P),
                  ist(C,Q).
ist(C,Q)      :-  instance(C,P->Q), ist(C,P).
```

This metainterpreter can be easily extended to include negation, disjunctive conditions, arithmetic, and so on.

Appendix II: an abstract data type for contexts

Contexts are made explicit through context assertions. Any assertion α:Φ related to context α partially defines α. In order to access the total extension of a given context (i.e. to consider all context assertions related to this context), let us define an abstract data type with the following signatures:

Basic types

C : the set of terms of the context sort
L : the language of contextual implications
$List_L$: the set of lists of formulas of L

Operations

instance:	C x L	→	boolean	true if context contains an instance of a formula
new:		→	C	creates a new empty context
add:	C x L	→	C	add a formula to a given context
delete:	C x L	→	C	remove from context all instances of a formula
assume:	C x $List_L$	→	C	add to context all formulas from a list of formulas

Implementation

```
new(Context)              :- retractall(instance(Context,_)).
add(Context,P)            :- assert(instance(Context,P)).
delete(Context,P)         :- retractall(instance(Context,P)).
addList(Context,List)     :- forall((List:Elements,member(P,Elements)),
                                    add(Context,P)).
```

Remarks

The above implementation relies on the following assumptions:

- all formulas `P` belonging to a context assertion related to `Context` have been asserted as unit clauses `instance(Context,P)`
- list of formulas are given as context assertions, i.e. asserted as unit clauses of the form `List:Elements`, where `List` is a Prolog term *naming* the list and `Elements` is a Prolog list containing the formulas.

References

[Bonzon 97] P. Bonzon, A Reflective Proof System for Reasoning in Contexts, Proc. 14th Natl. Conf. on Artificial Intelligence (AAAI97)

[Buvac 96] S. Buvac, Quantificational Logic of Context, Proc. 13th Natl. Conf. on Artificial Intelligence (AAAI96)

[Hindriks & al. 97] K.V. Hindriks, F.S. de Boer, W.van der Hoek and J-J. Ch. Meyer, Formal Semantics for an Abstract Agent Programming Language, in: M.P. Singh, A. Rao and J. Wooldridge (eds.), *Intelligent Agents IV*, LNAI (vol. 1365), Springer Verlag

[Hindriks & al. 98] K.V. Hindriks, F.S. de Boer, W.van der Hoek and J-J. Ch. Meyer, A Formal Embedding of AgentSpeak(L) in 3APL, in: G. Antoniou & J. Slaney (eds), Advanced Topics in AI, 11th Australian Conference in AI, LNAI (vol.1502) , Springer Verlag

[Hindriks & al. 99] K.V. Hindriks, F.S. de Boer, W.van der Hoek and J-J. Ch. Meyer, Semantics of Communicating Agents Based on Deduction and Abduction (submitted)

Levesque & al. 97] H.J. Levesque, R. Reiter, Y. Lespérance, F.Lin and R.Scherl, GOLOG: A logic programming language for dynamic domains, Journal of Logic Programming, 31

[Rao 96] A.S. Rao, AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in: *Agents Breaking Away* (MAAMAW '96), W. Van der Velde and J.W. Perram (eds), LNAI (vol 1038), Springer Verlag

[Russel & Norvig 95] S. Russel and P. Norvig, Artificial Intelligence , Prentice-Hall

[Wooldridge 99] M. Wooldridge, Intelligent Agents, in: G. Weiss (ed.), Multiagent Systems, MIT Press

Debugging Prolog Using Annotations

M. Kulaš
FernUniversität Hagen, FB Informatik, D-58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract

We propose an annotation language well suited for rendering aspects of Prolog execution. Our annotations are special Prolog goals that act as executable comments, performing debugging at run-time. We do not place any restrictions upon the object language, the concern being verification of (full) Standard Prolog programs. Several illustrations of the merits of the approach are given. All the examples presented here are actual runs of our system Nope.

1 Introduction

We start by briefly surveying previous research in debugging of Prolog programs. Then we summarize our approach, introduced in [8]. Sec. 4 allows a close look into the merits of the approach, by means of extensive examples. Sec. 5 rounds off the survey of related work.

2 Debugging Prolog

In the area of Prolog debugging and validation different approaches have been put forward.

The very influential approach of declarative debugging (algorithmic debugging, declarative diagnosing) pioneered by [17] has an inherent limitation of assuming declarativeness of the debugged predicates. Namely, the predicates have to be instantiation-stable (monotone) in the sense that the properties of success / failure hold under substitutions: if a goal succeeds/fails, then any instance of it shall succeed/fail as well. This clearly does not have to hold if var/1 or nonvar/1 is allowed, which is the case in full Prolog.

Another very interesting declarative approach, LPTP [18], proposes a first-order language for rendering procedural properties of Prolog (success, failure and universal termination of goals) in a declarative way, together with a powerful calculus (formalizing a suitable execution model) for interactively proving these properties. The inherent limitation here is akin to the previous approach: only a logical subset of Prolog (no cut, var or assert/retract allowed) can be treated.

The traditional verification paradigm of pre/post conditions has first been adopted for full Prolog in [5] and further promoted especially by [7], in their generic framework CIAO for program development and debugging. In addition to pre/post conditions, their annotation language offers the so-called comp annotations, like in :- comp qsort(A,B) : (list(A), var(B)) + does_not_fail, expressing some properties of the computation (here: non-failure of the query qsort(A,B)). Behind the somewhat patchworky syntax is a more serious issue of expressive power: pre/post conditions alone do not suffice to express every property of execution, e. g. the non-failure. As stated in [7], "no property which refers to (a sequence of) intermediate states in the computation *(...)* can be easily expressed using calls

and success assertions only". This shortcoming of the pre/post conditions is here being compensated for by handing the more tricky properties over to Prolog.

For the benefit of the user, as well as for automation of debugging, it is advantageous to start from an explicit execution model. The model should ideally be simple and complete.

One such model happens to be generally known. The 4-port model [2], which is the basis of the standard Prolog debugger, is a complete execution model of Prolog, in the sense of entailing every aspect of Prolog execution of necessity for verification, and therefore all the information one might need. This model maps a query Q to its *standard trace* $\mathcal{T}$(Q), which is a sequence of *events* of the form Port F(T_1, .., T_N), where Port may be one of {call, exit, fail, redo}, F/N are predicates, T_i are terms, representing the atomary steps of the Prolog interpreter during the execution of the query Q (as defined in [2], [19]).

A consequence of such expressive power is also that the amount of information generated by this model be overwhelming, so that its straightforward use as in the standard 4-port debugger is often considered as too low level.

One line of research addressing this problem is *trace analysis* of [6], and it advocates building (essentially) the standard trace and filtering the non-interesting events out of it. The approach bases upon primitives for sequential forward and backward traversal of the standard trace, which can be aggregated into arbitrarily high-level filters. The user starts from an erroneous query Q and interactively conceives and discards hypotheses about the culprit by selecting and assessing different subsets of the standard trace $\mathcal{T}$(Q). These subsets can be arbitrary ('abstract views of the execution'), and are built by filtering out. Starting from one subset the user can specialize it, or generalize, or replace altogether, until reaching the end of trace. This performs best as exhaustive checking with dynamically changing focus.

An alternative line of research based on the 4-port model is pioneered by [13], and it advocates tracing only the interesting events. The user starts typically from some hypotheses and hopes to hit upon any queries that might violate them. In some cases the hypotheses can be statically proved as valid. We call this paradigm *event transforming* because of its typical implementation. As opposed to a trace-analysis system like Opium [6], where the execution of the user's program Π will be sampled at all ports of all predicates, in a transformation system like Advice [13] only the execution of some selected goals will be re-routed, so virtually, not the user's original program Π but a *transformed* program Π' will be executed.

Our own approach [8], realized in the system Nope, is event transforming.

None of the two paradigms (trace analysis vs. event transforming) is a special case of the other. In a transformation approach, one can as well start from an erroneous query Q and use trivial 'invariants' that only serve to construct a selected subset of $\mathcal{T}$(Q), hereby emulating trace analysis. However, here one cannot change the subset mid-query and is stuck with it, for better or for worse, till the end of the query execution. Analogously, one could emulate event transforming within trace analysis by using the invariant as the filter. For example, at the call port of \+ Q filter out everything which is ground. If this produces a non-empty set, then the invariant is violated. However, this is clearly far more inefficient than using a dedicated transformation (checking only \+ Q), due to the extreme waste of overhead. Apart from this, in case of several simultaneously present invariants there would probably have to be several trace traversals [6], as opposed to only one in the other paradigm, since in general it isn't clear in advance which event is going to happen first (so the first event may never happen, which is bad luck for the second one).

As a rule of thumb one might say that trace analysis is better suited for dynamic checking of expensive queries, where as much information as possible shall be tapped in order to be rolled back and forth many times using different abstract views. One of its main assets lies in the completeness of the

standard trace, permitting exhaustive exploration of the bugs, especially combined with trace recording. Further, freely abstracting over the sequence of events, compare the quote from [7] on page 123. Finally, the ability to modify the checking 'on the fly' (during the execution of the query): depending on the outcome of the current filter, a new one may be used in the sequel. Event transformers cannot modify their checks mid-query.

On the other hand, event transformation, having a very low runtime-overhead and allowing for quite useful annotation languages (as we hope to show in the following), is better suited for quick prototyping and monitoring. The invariants serve also as valuable program documentation. Some classes of them can be checked statically.

3 Nope annotation language (summary)

Annotations describe properties of predicates. Such properties can be pre or post conditions, which must hold true when a predicate is called or exited, respectively. Our concept transcends pre/post conditions: we introduce two more kinds of annotations, fail and redo annotations, hence incorporating a whole model of Prolog execution into our language. This enables natural rendering of many procedural properties of Prolog which cannot be expressed with only pre/post conditions, like non-failure. There are four more novelties in our approach. First, any annotation can be "narrowed down" to a subset of calls, via templates and contexts, giving much more flexible assertions. Notably the novel idea of calling context adds significant expressive power, giving access to arbitrary program points. The annotations are defined simply as Prolog goals, making them fully parametric and therefore very comfortable for debugging. Finally, the annotations are applied via a general kind of matching instead of unification, enabling the use of local variables.

3.1 Call annotations

Motivated by the need to express more precisely the intended calling patterns of a predicate, we introduced in [8] call annotations, with the following syntactic variants.

call AtomicGoal ⇒ MustHold
call AtomicGoal with Template ⇒ MustHold
call AtomicGoal { with Template } within Context ⇒ MustHold

An example for the first variant is `call append2(X,Y) => nonvar(X); Y=[]`. There will be diverse examples in this paper, each tested with Nope. Nope is an (almost standard) Prolog module that recognizes our annotations and checks them at run-time.

The second variant is an enhancement of the goal specification in Prolog, namely a template allows us to specify the calling pattern more precisely than with head unifications alone. In fact, arbitrarily precise, since a template can be any Prolog goal. The default template is true.

The third variant introduces the idea of *calling context*, a way of specifying sets of program points. The contexts are partially ordered (*context subsumption*). The default context is _, which subsumes any other context, so an annotation without a context part applies in any context. In case of annotation failure, Nope will issue a warning. The default warning in Nope shows the violated annotation, with its arrow broken, e. g. `/* NOPE call append(A,[1,2]) ⇏ nonvar(A); [1,2]=[] */` This can be overridden via the else/2 predicate (see page 130): `call append(X,Y) => nonvar(X); Y=[] else write(query_diverges)` produces the warning `/* NOPE query_diverges */`.

Definition 1 (call annotation) *Let* P *be a predicate of arity* n*. A call annotation for* P/n *is a Prolog goal*

call Premiss ⇒ Constraint

where Premiss ::= Goal { with Template } { within Context }*. Here* Goal *is a Prolog goal for* P/n*, i. e. a Prolog term of the form* $P(S_1, .., S_n)$*.* Context *is a context (see Sec. 3.1.2).* Template, Constraint *are arbitrary Prolog goals.*

Technical note 1: We show our annotations mostly in the traditional, declaration form `:- Annotation.` to make them stick out, but please note that they are simply *goals* and can be called like any other Prolog goals, amounting to *parametric* annotations. (In case you wonder which predicate is here being called, it's.. right, the arrow ⇒/2.)

Technical note 2: Our annotation language is implemented by means of *goal replacement*. Therefore, in principle there is nothing to prevent us from annotating *arbitrary goals* and not just *'atomary'* ones (as opposed to composite predications like conjunction, disjunction, if-then-else or negation). For example, it is just a question of slowing down the parser a notch to allow arbitrary negated goals, making it possible to express safe negation in its most general form: `call \+ X => ground(X).` The default parser of our implementation will recognize annotated negation, but conjunction, disjunction and if-then-else will be transparent, i. e. any annotations for these built-in 'predicates' of Standard Prolog will be ignored, as opposed to all other built-ins (and of course all user-defined predicates).

We shall assume every call annotation in the form call Goal with Template within Context ⇒ Constraint, which we may do owing to the defaults for Template and Context.

The annotations shall not be applied via unification, since we consider matching (subsumption) to be the more natural choice for our purpose. Recall that Pattern subsumes Object if Object is an instance of Pattern, i. e. if Object can be unified with Pattern without further instantiating any variables in Object. We generalize this so as to accommodate not only equality but arbitrary goals in Template: everything goes but there is an invariant, the user's goal, which may not be changed.

Definition 2 (succeeds soft) *Let* Goal *and* Invariant *be arbitrary Prolog goals. We say that* Goal *succeeds soft on* Invariant *if* Goal *succeeds without changing* Invariant*.*

More formally: If Goal succeeds with an answer substitution σ, then σ may only rename Invariant, written as Invariant $\cong \sigma$(Invariant).

In terms of this definition, Pattern subsumes Object iff Pattern = Object succeeds soft on Object. Soft succeeding goals are not only of the kind \+ \+ G. Nope provides a predicate call_soft(G, J) that succeeds only softly on J for any G and any J. The advantage over \+ \+ G is that call_soft allows for *local variables*, because the variables of G not occurring in J may get instantiated.

In Def. 7 we define applicability of an annotation via soft success. Owing to this we may have arbitrary premisses, like p(X) with X=f(Y), or p(X,Y) with X=Y, although these would wreak havoc if applied via unification.

3.1.1 Simple context

It is useful to be able to access the immediate ancestor (parent) of Q, i. e. the goal which called Q.

Definition 3 (parent of a goal) *The head of a clause is the parent of each body goal. If a goal is issued at the top-level of a Prolog interpreter, we will say that its parent is the goal* false*.*

To capture the notion of the parent, we introduce *contexts* in the annotations.

A context shall represent the parent of the annotated goal. In the simplest case, it can be any atomary Prolog goal. So the meaning of an annotation for Q within Context shall be to check on Q occurring anywhere in the clause bodies of Context. One can view contexts as specifying *sets of program points*, namely those where the annotated goal might occur (in this first simple case: all points in the bodies of Context).

Furthermore, sometimes it is useful to exclude contexts. For example, if we only want to check the non-recursive calls of a predicate P, we need to express "check P within any context but P". For such purposes we allow *negative contexts*, represented as dash-prefixed atomary Prolog goals. The meaning of -Context is the complement of the set of program points represented by Context, i. e. this can be any point in the program (plus top-level, 'defining' false/0) outside of the definition of Context.

In the following we shall further refine the idea of a context. But first we have some questions for our contexts so far. One key question about contexts is their inheritance, i. e. when is an annotation for Q within X pertinent to Q within Y? Also, when is an annotation bound to a context CA pertinent to a goal within a context CQ? To settle these questions, we define a partial ordering for contexts.

We start from the natural lattice of first-order atomary formulas (augmented by a 'universal' and a 'null' formulas) modulo renaming, as introduced by [16]. The partial ordering $\preceq$ being "is instance of", the greatest lower bound of two atoms being their most general unifier (or the special bottom element *null formula*) and the least upper bound being their most specific generalizer (or the special top element *universal formula*).

Next we enhance the language. Our language is the following set $\mathcal{C}$ of atomary and non-atomary formulas (plus the universal and the null formula): the first-order language – modulo renaming – made of standard Prolog variables, function and predicate symbols, and a unary operator -. Atoms Q of this language we call *positive contexts*, and non-atoms -Q we call *negative (negated) contexts*. The universal formula is represented here by _, and the null formula is represented by -_.

It remains to enhance the partial ordering.

Definition 4 (partial ordering of contexts) *For positive contexts* X *and* Y *we define:* $\text{-X} \preceq \_$, $\text{X} \preceq \text{Y}$ *if* X *is an instance of* Y, $\text{X} \prec \text{-Y}$ *if* X *is not unifiable with* Y, $\text{-X} \preceq \text{-Y}$ *if* $\text{Y} \preceq \text{X}$.

If we assign to every atom Q from $\mathcal{C}$ the set of its ground instances, and to -Q all the other ground atoms in the Herbrand base $\mathsf{B}_\mathcal{C}$ for $\mathcal{C}$, we obtain an isomorphism between $(\mathcal{C}, \preceq)$ and a sub-poset of $(\mathcal{P}(\mathsf{B}_\mathcal{C}), \subseteq)$.

Due to this isomorphism, $(\mathcal{C}, \preceq)$ is a poset and the seeming asymmetry of the definition actually is a perfect duality: $\text{X} \prec \text{-Y}$ iff $\text{Y} \prec \text{-X}$ iff X,Y are not unifiable.

Definition 5 (context subsumes) *If* $\text{CQ} \preceq \text{CA}$*, then we say: the context* CA *subsumes the context* CQ.

We chose the names CA and CQ to suggest *annotation context* as opposed to *query context*. The interesting case is namely when an annotation context subsumes the actual query context, meaning the annotation may fire.

Definition 6 (context inheritance) *If* $\text{X} \prec \text{Y}$*, then an annotation for* Q *within* Y *shall be inherited to* X*, i. e. it is also an annotation for* Q *within* X.

In examples of Sec. 4.3 we will advocate usefulness of contexts.

3.1.2 (Generalized) Context

Assertions can be divided into predicate assertions, stating properties of a predicate, and program-point assertions, stating properties that shall hold when the code at a given point in the program is about to be executed. Traditionally, the two classes have different syntax, predicate assertions being written as declarations, and program-point assertions being scattered all over the program code (like in [7]). Our annotations are predicate assertions. However, due to contexts, as defined so far, we can capture some sets of program points. For illustrations see Sec. 4.3.

There is only a small step necessary to be able to capture also individual program points. For example, say we have a program-point annotation $\mathcal{A}$ placed as follows:

$$\mathsf{H} :\!\text{-}\ \mathsf{B}_1, .., \mathsf{B}_i,\ \mathcal{A},\ \mathsf{B}_{i+1}, .., \mathsf{B}_n. \qquad \%\ \text{k-th clause for the predicate of H}$$

then we can simulate it in Nope as

$$\mathsf{call}\ \mathsf{B}_{i+1}\ \mathsf{within}\ \mathsf{H}^*\mathsf{pp(k,i)} \Rightarrow \mathcal{A}$$

In other words, we enhance the context specification by an optional part, program-point, linked to the simple context by an asterisk. A non-asterisk term Context is regarded as an abbreviation of Context * _ meaning "any program point within the simple context Context". The program-point is a Datalog term representing the intended program point(s) like pp(K,J), pp(K,J,N), pp(K,J,N,M) where K is the serial number of the clause (counting top-down) in the definition of the parent goal, J is the serial number of the body subgoal (counting each conjunct/disjunct/implication-part from left to right). The last two items N, M represent the totals of body subgoals and bodies. Each of K,J,N,M may be a variable or a number.

The partial ordering $\preceq$ of simple contexts shall be lifted in the obvious way to generalized contexts.

3.1.3 Calling premiss

For the premiss of an annotation we will also use the more evocative name *calling premiss*.

Definition 7 (calling premiss subsumes goal) *We say that a calling premiss* Goal with Template within Context *subsumes the call port of* Q *(for short: subsumes* Q*), if at the call port of* Q Q=Goal, Template *succeeds soft on* Q*, and then* Context *subsumes the parent of* Q*.*

More formally: Let Parent be the parent goal of Q. The following three conditions must hold. If σ is the answer substitution for Q=Goal, then σ(Q)≅Q. If θ is an answer for σ(Template), then θ(Q)≅Q. Lastly, σ(Parent) $\preceq$ σ(Context).

A logical reading of this definition shall be the following (we omit the contexts for simplicity):

$$\forall\ \mathcal{V}(\mathsf{Q})\ \exists\ \mathcal{V}(\mathsf{Template})\backslash\mathcal{V}(\mathsf{Q}) \quad \mathsf{X}_1 = \mathsf{Q}_1, .., \mathsf{X}_n = \mathsf{Q}_n \vdash \mathsf{X}_1 = \mathsf{Goal}_1, .., \mathsf{X}_n = \mathsf{Goal}_n, \mathsf{Template}$$

Here $\mathsf{Q}_1, .., \mathsf{Q}_n$ are respectively the first, .., nth arguments of Q, i. e. the actual calling substitution, and similarly $\mathsf{Goal}_1, .., \mathsf{Goal}_n$ are the arguments of Goal, i. e. the calling substitution monitored by the annotation. $\mathsf{X}_1, .., \mathsf{X}_n$ are new variables. The monitored calling substitution together with Template constitutes the full (i. e. normalized) monitored calling template. $\mathcal{V}$(Term) is the set of variables in Term. So the actual calling substitution shall imply the monitored calling template.

Observe that this allows local variables, namely variables that don't occur in Goal can be used to create and propagate values from the calling premiss throught to the constraint.

Now we can define what does it mean to apply a call annotation.

Definition 8 (applicable) *An annotation is applicable on a goal* Q *if its calling premiss subsumes* Q.

Definition 9 (call annotation satisfied/violated) *An applicable call annotation is satisfied/violated for a goal* Q*, if its constraint succeeds/fails at the call port of* Q.

Definition 10 (correctness wrt annotations and queries) *An annotated program is said to be correct if, during the computation of arbitrary queries (from the given set of allowed queries), all the applicable annotations are satisfied.*

3.2 Following the execution model

In the previous subsection we suggested annotating the call port of a goal, in order to grasp more precisely the intended calling patterns. So what about the other three ports? Here we refer to the classic model of Prolog execution [2], abstracting a goal to a black box with four ports, known from the trace/debug sessions.

Following this model, we propose a language of annotations as a step towards capturing procedural aspects of Prolog. This language has the advantage of there being a workable consent among Prolog users on what the box trace model of Prolog execution means. We consider it a head-start for an annotation language to be based upon standard concepts.

Definition 11 (port annotation) *Let* P *be a predicate of arity* n. *A* Port *annotation for* P/n *is a Prolog goal of the form*

$$\mathsf{Event} \Rightarrow \mathsf{Constraint}$$

where

Event ::= Port Premiss
Port ::= call | exit | fail | redo
Premiss ::= Goal { with Template } { within Context }

Here Goal *is a Prolog goal for* P/n*, i. e. a Prolog term of the form* $P(S_1, .., S_n)$. Context *is a context.* Template, Constraint *are arbitrary Prolog goals.*

As in [5], we deal only with *partial correctness*, because we do not claim whether goals actually succeed, fail or redo. Or even get called. Also, as the transformational semantics below reveals, our annotation language provides for accessing the values of variables on call and on exit from a query, similar to [5]. Thus, our language is also a language of *binary assertions*. The actual calling pattern is accessible on the left of $\Rightarrow$, and, in case of exit and redo annotations, the exit pattern is on the right. This circumvents the need for language primitives to access the call/exit values.

Applicability of a port annotation is defined as in Def. 8. To define the meaning of port annotations, we can adopt Def. 9. But observe that a premiss shall always match the *call* port of the goal (therefore we dubbed it 'calling premiss').

In this paper we do not define the concept of a port. But still, we do give a precise semantics of our annotations, via a program transformation. Namely, we show the essentials of our implementation of the run-time checking of annotations. The implementation bases upon the following program transformation. Each annotation for a predicate P imposes a virtual transformation on the user program: each goal $\mathsf{P}(\mathsf{T}_1, .., \mathsf{T}_n)$, be it a top-level query or in a body of a clause, will be computed as if changed according to the following table (here we abstract from contexts and bookkeeping to obtain a simpler picture). The labels on the arrows indicate the respective Port of the annotation.

$P(T_1,..,T_n) \overset{CALL}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$
(Constraint else Warning), $P(T_1,..,T_n)$
; $P(T_1,..,T_n)$

$P(T_1,..,T_n) \overset{EXIT}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$
$P(T_1,..,T_n)$, (Constraint else Warning)
; $P(T_1,..,T_n)$

$P(T_1,..,T_n) \overset{FAIL}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$
($P(T_1,..,T_n)$; (Constraint else Warning), fail)
; $P(T_1,..,T_n)$

$P(T_1,..,T_n) \overset{REDO}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$
$P(T_1,..,T_n)$, (true; (Constraint else Warning), fail)
; $P(T_1,..,T_n)$

Figure 1: The program transformation of Nope

Note that a call annotation hence corresponds to a (global) precondition, and an exit annotation to a (global) postcondition. The usual coupling is achieved via an own (local) precondition to a postcondition.

Lemma 1 (non-interference of port annotations) *Port annotations in Nope are not interfering with the success set, i. e. a query about the user's program* Π *computes the same answers and in the same sequence when posed to* $\Pi+\Delta$*, where* Δ *denotes arbitrary port annotations with converging and pure templates and constraints.*

Annotations with a diverging template or constraint are clearly a bad idea. But otherwise, this lemma imposes the too strong restriction of purity upon templates and constraints. In practice, users profit from annotations with side-effects. Like [7], we guarantee that no amount of annotation tweaking will change any variables of a user's query. As for the rest of dangers, we simply count on users to be sensible in their choice of side-effects. Lemma 1 follows from the program transformation of Nope (Fig. 1) and the implementations of call_soft/2 and else/2:

```
call_soft(Test, Invariant) :-
  copy_term(Test,TestCopy), TestCopy, !, % Test succeeds, but is it harmless on Invariant?
  \+ \+ (numbervars(Invariant,0,_), TestCopy=Test),
  Test=TestCopy. % Other variables may be propagated.

else(Constraint, Warning) :- % Constraint else Warning
  \+ Constraint -> Warning; true.
```

Lemma 2 (compositionality of port annotations) *Port annotations in Nope are compositional, i. e. if* $\mathsf{Premiss}_k$ *of a stated annotation* A_k *matches the call port of* Q*, then* $\mathsf{Constraint}_k$ *is going to be computed at the* Port_k *of* Q*, no matter how many annotations, for what ports and in what order, were stated until the call.*

This lemma follows from the program transformation of Nope.

4 Illustrations

Here we give several illustrations in support of our claims that the port annotations are a versatile and an intuitively appealing means to specify general mode information, to express and support the

verification of hypothetical invariants (by creating demons), to write customized tracers, to alert to unwanted events, and generally enhance the prototyping and self-reflective capacities of Prolog.

4.1 Mergesort and its partial specification

As a first illustration of our debugging concept, we show a buggy implementation of the predicate merge/3, which expects as inputs two sorted lists of integers and merges them into a sorted output list.

```
:- ensure_loaded(library(nope)).  % how to activate NOPE

% Precondition: the inputs should make sense
:- call merge(X, Y, Z) => sorted(X), sorted(Y) else format('inputs ~w and ~w must be sorted', [X,Y]).

% Postcondition
:- exit merge(X, Y, Z) with sorted(X), sorted(Y) => sorted(Z).

merge([X|Xs], [Y|Ys], [X|Zs]) :- X>Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,X|Zs]) :- X =:= Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X<Y, merge([X|Xs], Ys, Zs).
merge([], [X|Xs], [X|Xs]).
merge(Xs, [], Xs).

sorted([]). sorted([_X]). sorted([X,Y|L]) :- X =< Y, sorted([Y|L]).
```

After loading this program, we can have the following interaction

```
| ?- merge([3,1], [], [3,1]).
/* NOPE inputs [3,1] and [] must be sorted */

yes
| ?- merge([1], [2], M).
/* NOPE exit merge([1],[2],[2,1])with sorted([1]),sorted([2]) ⇏ sorted([2,1]) */

M = [2,1] ?
```

The second message is warning us that the postcondition is violated, namely the merging algorithm produces unsorted lists (the first and the third clause for merge/3 indeed have swapped comparisons).

An avid reader might object that we do not need the template sorted(X), sorted(Y) in the postcondition, since this is already being taken care of by the precondition. Well this is not quite true.

Note that there can be arbitrarily many annotations for a predicate, even for the same port of it, and they will be composed via conjunction. But still, each annotation is regarded as an independent entity, in the sense of not having to take into account any other annotations. So our exit annotations actually do not assume that any call annotations have to be satisfied. For example, if we omit the template above and prove the following two annotations to hold

```
:- call merge(X,Y,Z) => sorted(X), sorted(Y).
:- exit merge(X,Y,Z) => sorted(Z).
```

with respect to the given program and a closed set of top-level queries, then we know that merge/3 is only going to be called with sorted inputs, and that the output is also going to be sorted.

But if the call annotation fails for a certain goal, the exit annotation is still going to be checked in Nope, requiring too much of the predicate (namely, that merge/3 is always going to deliver a sorted output, regardless of inputs), compared to its definition.

So the call annotations are 'global' preconditions, not bound to any postcondition. On the other hand, we do have a way of *binding* a certain precondition to a certain postcondition: via the premiss of an exit annotation.

4.2 Some modes

To express that \+ does not alter its argument, we can use the following mode declaration

```
:- exit \+ X with groundcopy(X,X0) => X=X0.
```

The predicate groundcopy(Term, GroundTerm) supplies the ground form, e. g. groundcopy(p(f(X),Y), G) would compute G = p(f('$VAR'(0)), '$VAR'(1)), so that the unification in X=X0 amounts to identity. This example shows how to access the call/exit values of variables.

The constraint that the first argument of append/2 from [14] be a *complete list*[1] can be expressed (and documented) through the following two annotations.

```
:- call append(X,Y) with var(X), \+ Y=[] => false else write(divergence).
:- call append(X,Y) with var(X), \+ \+ Y=[] => false else write('trivial answers').
```

Note that, by using the default context, we do not need to actually check the list completeness, but only the var property, since each recursive call of append/2 will be checked. Note also how the constraint false serves to alert, at runtime, to unwanted events like ill-defined modes.

The following annotations relate the two list-type predicates mentioned above to the classic *list*[2]

```
:- exit list(X) => complete_list(X).
:- exit list(X) with copy_term(X,X0) => complete_list(X0); incomplete_list(X0).
```

claiming that a successful list/1 query has been called with a complete or an incomplete list, and will succeed with a complete list.

4.3 Use of contexts, parametricity and uniform conditioning

Correctness of the original Byrd's trace algorithm [2] depends essentially on the claim that only one flipflop/0 predicate, defined as alternatively succeeding and failing, suffices to handle arbitrarily many predicates. Of the annotation languages known to us, only [7] can express such a claim, by resorting to program-point assertions. Contexts make this expressible in a predicate assertion.

```
:- dynamic tracing/0, notracing/0.
flipflop :- retract(tracing), assert(notracing).
flipflop :- retract(notracing), assert(tracing), fail.
tracing.

% Byrd's trace invariant
:- call call(Goal) within break(Goal) => notracing. % for any Goal
```

[1] a list with the ultimate tail [ ], as opposed to a list whose ultimate tail is a variable, called an *incomplete list*.

[2] list([ ]). list([_|T]) :- list(T).

```
break(Goal) :-
  trace(call, Goal), (call(Goal); trace(fail, Goal), fail),
  (trace(exit, Goal); trace(redo, Goal), fail).

trace(Port, Goal) :- tab(1), write(Port:Goal).

% Example. For every traced predicate a new first clause will be asserta'ed.
p(X) :- flipflop, !, break(p(X)).
p(1) :- p(2), p(3).
p(2) :- p(4).
p(4).
```

Due to parametricity and contexts, our annotations can be seen as macro definitions for program-point annotations. For example,

```
lr_alert(Q) :-
  functor(Q,F,N), functor(Q0,F,N),
  (call Q within Q0 => \+ subsumes_noc(Q,Q0) else bark, write(lr:Q)).

subsumes_noc(X,Y) :- groundcopy(Y,Yg), X=Yg.

:- is_my_predicate(X), lr_alert(X), fail; true.
```

monitors leftmost (assuming left-to-right execution) recursive calls of all my predicates, like this

```
| ?- append2(X,[Y]).
/* NOPE lr:append2(_347,[_94]) */
Prolog interruption (h for help)? c
/* NOPE lr:append2(_758,[_94]) */
Prolog interruption (h for help)? a
Execution aborted
```

The utility bark/0 tunes the warning handler of Nope to interrupt Prolog after one warning. The warning handler provides uniform and graded (and fully customizable) means of handling special events like 'unkosher' modes.

As a hint on the merits of conditioning even the call annotations (in order to express dependencies between the input arguments), see

```
:- call functor(T,F,N) with var(T) => nonvar(F), nonvar(N).
```

Observe that here we can get rid of the template, via a less readable but equivalent

```
:- call functor(T,F,N) => var(T) -> nonvar(F), nonvar(N); true.
```

However, this does not hold in general. Namely, the *call* values of variables are normally not identical with their exit values. Thus, templates are not redundant, even in case of pure logic programs.[3] For programs with side-effects just more so, because (in case of exit, fail and redo) on the left of the arrow we have the state *before*, and on the right of the arrow: the state *after* the computation of Q.

[3] another reason is the subsumption on the left vs unification on the right of the arrow, but that is not a vital design decision of the language, the way the call/exit values are

4.4 Tracing

The Byrd tracer of the previous example is a special case of annotation composition. We don't have to bear with all four ports of a general goal pattern if we only want a specific port of a specific goal pattern traced.

```
fail_alert(X) :- fail X => write('Fail: '), write(X), nl.
:- fail_alert(p(X)).

p(1) :- p(2), p(3).
p(2) :- p(4).
p(4).

| ?- p(K).
Fail: p(3)
Fail: p(4)
Fail: p(2)

K = 2 ?
```

4.5 Finite failure

The fail-port is capturing 'no (more)' rather than 'no'. Often we are interested only in genuine 'no', the finite failure. We can derive the *whole* of this new event from the basic four events, even as simple-mindedly as

```
ff(( Q with T within Ctx => C )) :-
  (call Q with T within Ctx => retractall(lemma(Q))),
  (exit Q with T within Ctx => asserta(lemma(Q))),
  (fail Q with T within Ctx => \+ lemma(Q) -> C; true).
```

Alternatively, we can provide finite failure as a regular annotation, say ff, on a par with call/exit/fail/redo. This is a matter of only adding a new entry in the transformation table of Nope, amounting to two new unit clauses, and allows for superior code. Either way, we may now write

```
ff_alert(X) :- ff X => write(finitely_fails:X), nl.
:- ff_alert(p(_)).
```

and obtain the desired behaviour:

```
| ?- p(K).
finitely_fails: p(3)

K = 2 ?
```

Non-failure, which is not expressible using pre/post-conditions alone, can now be expressed as

```
:- ff Premiss => false.
```

So the subset assertions of [7] or examples of [11] would here look like

```
:- ff qsort([2,1],[1,2]) => false.
```

Similarly to finite failure, we can *derive* the event of building a resolvent (the unify port), or *build* it *in*.

5 Related work (coda)

In Sec. 2 we started a comparison of debugging methods for Prolog, which we now round off with a few technical questions. For some more details and references about the declarative work in the area of Prolog verification, especially the annotation languages, see [8].

The idea of inserting checks at strategic places in a program Π ('instrumenting' it into a program Π') is a traditional debugging aid which is being continually refined upon. For C, the library macro *assert(·)* serves the purpose of aborting a program after issuing a message, in case the given expression is zero at the given point in the program.

Often it is tiresome to insert the checks manually. Automating this by a tool which instruments the program without the user's having to meddle with the sources is an attractive prospect. Such a tool has to be *non-interfering*, i. e. must not change the meaning of the program (the program must 'feel' the same).

One such kind of tool are tracers, doing no checking but just sampling the execution at some predefined points. A further step towards automated verification is coupling some actual checks with some points of execution. Traditionally pre/post conditions are being employed for this purpose (e. g. design by contract in Eiffel). In Prolog, one could observe several directions of refining this idea.

On acceptancy grounds, a smooth integration of the debugging tool in the Prolog mode of operation is recommended. Recall that Prolog operates as a read-interpret-write loop. So we can modify the Prolog reader, persuading it to read in not our program Π but the instrumented program Π' (this is what Nope does). Or we can even modify the program after it has been read in (this is what the original Advice does, hence its limitation on modifiable predicates). Or we can modify the Prolog interpreter, persuading it to compute with our program Π a bit differently, like it were Π'. This is what happens after trace or debug is issued, and also in the Advice re-implementation in Quintus Prolog, which we here refer to as AdviceQ, as well as in Opium.

The alternative to these 'smooth' methods is to write an own Prolog interpreter, i. e. a meta-interpreter, which is an efficiency problem. For practice, the program 'feeling the same' with and without the debugging tool means also, that the program shall not be slowed down too much. However, the main concern of preserving the program's meaning may not be compromised for efficiency.

Meta-interpretation we won't discuss here. Modifying Π is employed in Advice, Nope and CIAO. The more 'surgical' method of modifying Prolog is employed in the standard debugger, Opium and AdviceQ. The original Advice as well as Nope do not modify Prolog (we call them *non-invasive* upon Prolog). The standard debugger, AdviceQ and Opium do not modify Π.

In AdviceQ the advised predicates don't get modified at all, except calls to them get trapped by a mechanism that checks for and applies the relevant piece of advice at the various procedure box ports [3]. In Opium, all of the calls get trapped. The same low-level mechanism as used by the standard debugger. This accounts for the basic difference in runtime efficiency between the two paradigms, of Advice and Opium.

There is one technical issue to consider if a debugging tool is to be in daily use. For annotations which are currently un-checked, or for predicates without any annotations, the user expects that there be no perceptible decline in performance, so that the user doesn't have to abandon Π' in order to do 'some serious work' with Π. Recall the NDEBUG switch for *assert(·)*. This is fulfilled in Advice, AdviceQ and Nope, having idle overhead linear in the number of annotated predicates (e. g. in Nope this means three resolutions per annotated predicate).

One further criterion, discriminating between the transformation tools, is whether the transformation is performed on a goal basis or on the predicate basis. Advice or CIAO transform predicates, i. e. annotated predicates will be replaced by their 'instrumentations'. Nope transforms goals, at only the

program points specified in the contexts. This further saves runtime overhead. The program points can be in the program or at top-level.

Finally, since Advice is the most closely related effort to our own, let us summarize the differences. Advice[13] differs from Nope in several ways, the main being: it has unary assertions; uses unification instead of matching; has no contexts; negated occurences of a predicate cannot be checked on their own (as no built-ins can be checked). Regarding the first two items, it is possible to enhance the goal replacement of Advice so that the calling pattern is being captured and made available to the advice checker.[4] The third and fourth item seem [3] to remain true restrictions of Advice.

6 Outlook

Annotation language Since [8], the annotation language of Nope has grown one natural bit of expressivity. Now it is possible to express *arbitrary program-point* assertions as *predicate* assertions of Nope, as shown in Sec. 3.1.2.

We envisage much further work on annotation languages for describing procedural properties of Prolog, especially in view of practical relevance. A competitive instance of Prolog has to accommodate foreign languages and programming paradigms like modules, constraints or agents, getting ever more complex on the way.

Runtime handling Nope is rather minimally invasive upon Π, and none upon Prolog, i. e. it won't modify the low-level of its host Prolog interpreter. Also, it is an (almost) standard Prolog program. The only essential facility outside of the ISO Prolog standard but necessary for Nope is term expansion[5]. The program transformation Nope is performing is goal replacement, at precisely the program points specified in the premisses. This is achieved by term expansion in a non-interfering way (in the sense of Lemma 1) and with small runtime overhead.

Nevertheless, the pragmatic division of program analysis between static and dynamic phases, as promoted by [4] and [7], places severe efficiency demands and a host of other technical issues upon runtime handling of annotations, so Nope might undergo another rewriting [15].

Acknowledgments

Many thanks for valuable comments upon previous drafts of this paper and other bits are due to C. Beierle, G. Meyer, R. Stärk, I. Đurđanović, M. Hermenegildo, U. Neumerkel, M. Carlsson, Vuk and the anonymous referees. This research was supported by the DFG within "Schwerpunktprogramm Deduktion" (Be 1700/3-1).

References

[1] Edinburgh University A. I. Applications Institute. Edinburgh Prolog Tools. ftp://sunsite.doc.ic.ac.uk/-packages/prolog-pd-software/tools.tar.Z, 1988.

[2] Lawrence Byrd. Understanding the control flow of Prolog programs. In S. A. Tärnlund, editor, *Proc. of the Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, 1980. Also as D. A. I. Research Paper No. 151.

[4] This way, the first item will be an advantage of Advice over Nope, since matching can be emulated by unification but not vice versa.

[5] From [12]: "It is important to realise that a Prolog compiler that *only* supports vanillia Prolog without DCG and term_expansion/2 support is about as useful as a C compiler that lacks any preprocessing facilities and half its libraries."

[3] M. Carlsson. Personal communication., Sept. 1999.

[4] P. W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *J. of Logic Programming*, pages 31–69, 1992.

[5] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[6] M. Ducassé. Opium: An extendable trace analyzer for Prolog. *J. of Logic Programming*, 39:177–223, 1999.

[7] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 1999.

[8] M. Kulaš. Annotations for Prolog – A concept and runtime handling. In *LOPSTR'99: 9th Internat. Workshop on Logic-based Program Synthesis and Transformation*, pages 39–48. Universita' Ca' Foscari di Venezia, Rapporto di Ricerca CS-99-16, Sept. 1999.

[9] M. Kulaš. Debugging Prolog using annotations. In *WLPE'99: 10th Workshop on Logic Programming Environments*, pages 18–34, Las Cruces, NM, Nov. 1999.

[10] Gregor Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Informatik Berichte 200, FernUniversität Hagen, 1996. Also tutorial on ICLP'97. Source distribution http://www.fernuni-hagen.de/pi8/typical.

[11] U. Neumerkel. A programming course for declarative programming with Prolog. http://www.complang.-tuwien.ac.at/ulrich/gupu/material/1997-gupu.ps.gz. Also tutorial on ICLP, 1997.

[12] R. A. O'Keefe. Re: parser in C. Thread in comp.lang.prolog, Sept. 1994.

[13] Richard A. O'Keefe. *advice.pl*. In DEC-10 Prolog Library [1]. Interlisp-like advice package, 1984.

[14] Quintus Corp., Palo Alto, CA. *Quintus Prolog Language and Library*, 1991. Release 3.1. Also library sources.

[15] P. B. Reintjes. Prolog for software engineering. In *Internat. Conference on the Practical Applications of Prolog*, 1994. Tutorial. Also as http://www.logic-programming.org/people/Reintjes_Peter/se94.html.

[16] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*. Edinburgh University Press, 1970.

[17] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[18] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/personal/staerk/lptp.html. Release 1.05.

[19] G. Tobermann and C. Beckstein. What's in a trace: The box model revisited. In *Proc. of the First Internat. Workshop on Automated and Algorithmic Debugging, Linköping, Sweden*, volume 749 of *LNCS*. Springer-Verlag, 1993.

New Missing Solution Method for Trace Trees

Gabriella Kókai[1]
[1] IMMD-II, Friedrich-Alexander University of Erlangen-Nürnberg
Martensstr. 3. D-91058 Erlangen, Germany
e-mail: kokai@informatik.uni-erlangen.de

Abstract

Shapiro developed an algorithm to diagnose the error in PROLOG programs where output is missing. The original algorithm can handle only *pure* logic programs and it has some other disadvantages In this paper a new approach is presented. This solution based on the *trace tree* (kind of SLD tree) which is built up from the trace generated by the PROLOG interpreter. We also introduced a new types of existence questions. [1]

1 Introduction

Research on the topic debugging PROLOG programs started with the development of Shapiro's APD-System ([16]). The algorithmic program debugging method introduced by Shapiro can isolate an erroneous procedure if a program together with an input leading to an incorrect result is given. Shapiro's model was originally applied to PROLOG programs to diagnose the three types of errors: termination with incorrect output, termination with missing output and nontermination. A major drawback of this debugging method is the great number of queries to be answered by the user on the correctness of intermediate results of clause calls. Frequently they are *yes/no* questions about the correctness of a given instance of a clause. Sometimes the allocation of the variables in an instance of a clause also has to be given.

Shapiro's algorithms traverse the proof tree of a program in various ways (*bottom-up*, *top-down* and *divide and query*) asking the user about the expected behaviour of each resolved goal.

In this paper a new approach is presented for the problem to find the error in a PROLOG program in the case of missing solution. We engage with the question: How the error can be found if it is situated in the *redo* branch of the derivation for a given goal. Our idea is to use an *trace tree* (kind of SLD tree) generated directly from the PROLOG trace. This tree contains the whole computation of a goal. Moreover this solution allows to handle programs which use *full* PROLOG and the extra-logical features of the language. We introduced other types of existence questions. It contains not only one atom but also the descendants of a node which contains the asked atom. This improvement helps the user to answer the questions correctly.

We integrate the solution described in this paper in an Interactive Diagnosing, Testing and Slicing System (*IDTS*, [7]) which is an integrated debugging and testing programming environment for PROLOG programs. The *IDTS*-System combines Shapiro's algorithmic program debugging method with functional testing (*CPM*) and with programslicing. The goal is to develop an effective system for debugging and testing of programs written in Prolog.

Other debugging techniques and tools of PROLOG programs were developed APROPOS [10], EDS [6], DED [9] and ADAss [4]) (PRESET [15], RD [14], OPIUM [3], PTP [5] NU-PROLOG debugging environment [11]). The main disadvantage of these systems is that they can debug programs which are written using only the *standard* PROLOG syntax and

[1]This work was supported by the grants of Bayerischer-Habilitationsförderpreis 1999

semantic. Besides these systems investigate only the proof tree. This means that these systems cannot be applied for *realistic* programs.

In the remainder of this paper a theoretical description of the original missing solution method is given in Section 2. It follows an introduction to the extension of this method in Section 3. In Section 3 also an example is given to demonstrate the completion of our method. Finally a summary concludes the paper.

2 Shapiro's Method for Termination with missing Output

Shapiro developed an algorithm to diagnose the error where output is missing. This kind of error can appear only if the programming language is nondeterministic. As we know Prolog belongs to this set of languages. Although the computation of a clause C in the program $\mathbf{P}$ ($C \in \mathbf{P}$) with input x terminated and the computed outputs (if it exists) is correct in an interpretation $\mathcal{I}$ ($\in \mathcal{I}$), there is nevertheless no derivation which computes the output y. Such a behaviour of a program is called *finitely fail* and need a special treatment (see [1], [2]). To explain Shapiro's method we need some definitions.

Definition 1 A program $\mathbf{P}$ (*finitely fails*) on $< C, x, y >$ if all computations of C with the input x terminate and the output is correct in $\mathcal{I}$ but no computation returns with y.

Definition 2 A *clause* C is *complete* concerning $\mathcal{I}$, if for all triples $< C, x, y >$ in $\mathcal{I}$ the program $\mathbf{P}$ covers $< C, x, y >$ concerning $\mathcal{I}$.

Definition 3 A *program* $\mathbf{P}$ is *complete* in $\mathcal{I}$, if for every triple $< C, x, y > \in \mathcal{I}$ exists a computation of $C \in \mathbf{P}$ with the input x which gives as output y. If a program finitely fails then it is not complete in $\mathcal{I}$.

If a program $\mathbf{P}$ with a given goal G in $\mathcal{I}$ finitely fails, then an other goal G_1 exists in $\mathcal{I}$, in which $\mathbf{P}$ with G_1 *immediately fails*. Immediately fail denotes that no clause $A \leftarrow B$ in $\mathbf{P}$ exists that A is unifiable with G_1.

The procedure *incomplete-procedure* (see Algorithm 1) works as follows: It traverses the proof tree which leads to *fail* assumed that the goal is true and finitely fails. The error can be found in two different manners. Either the procedure does not find any other clause, then the computation cannot be continued or the algorithm *satisfiable* (see Algorithmus 2) delivers *fail*. But if *satisfiable* delivers *true* then with the recursively calling the procedure *incomplete_procedure* the underlying level of the proof tree will be searched.

Algorithm 1 To find an incomplete clause

Input: $< C, x, y >$ in $\mathcal{I}$, where C finitely fail
Output: $< C_j, x_j, y_j >$ in $\mathcal{I}$, where C_j is not covered
procedure *incomplete_procedure*$(< C, x, y >, < C_j, x_j, y_j >)$
begin
found:=false; $i := 1$
let $t = \{< C_1, x_1, y_1 >, \ldots < C_n, x_n, y_n >\}$ be the top level trace of $< C, x, y >$,
where $C : A_0 \leftarrow A_1, \ldots A_n$ and $\forall i\ 1 \leq i \leq n\ A_i = C_i\theta$
if $\exists\ t$ **then** *satisfiable*(t , success, Instances)
if success = false **then** $< C_j, x_j, y_j >:=< C, x, y >$
else while $(i \leq n)$ and (not found) **do**
if call$(< C_i, x_i, y_i >)$ = fail
then *incomplete_procedure*$(< C_i, x_i, y_i >, < C_j, x_j, y_j >)$; found := true
else i:= i+1
endif
endwhile
endif
else $< C_j, x_j, y_j >:=< C, x, y >$ {finitely fail}
endif
end

The main algorithm calls the additional procedure *satisfiable*. This procedure investigates the top level trace given as input to determine whether the derivation can be continued. The user has to verify each element by answering existence questions.

Algorithm 2 Search for an instance of the clause if this exists

Input: top level trace t
Output: success: *false*, if no instance exists; if it is *true*, then it delivers the instance
procedure *satisfiable*(t,success,Instance)
begin
$i := 1$; Instance := {}; n:=length(t)
while $(i \leq n)$ **and** (Query $(< C_i, x_i, y_i > \in t)$ = yes) **do**
Instance := Instance $\cup\ y_i$; $i := i + 1$
endwhile
if $i > n$ **then** success := true
else success := false
endif
end

The function *Query* (see Algorithm 3) is called with the triple $< C, x, y >$. $< C, x, y >$ may contain any variables. The result of calling *Query* is a boolean value which specifies whether $< C, x, y >$ is in $\mathcal{I}$ or not. Should be $< C, x, y >$ in $\mathcal{I}$ and contains any unbound variables then the task of the user is to give the values of the variables.

Algorithm 3 Ask question about the existence

Input: $< C, x, y >$ is atom, which also can contain unbound variables
Output: *yes* and $< C, x, y >$ will be bound, if $< C, x, y > \in \mathcal{I}$ *no*, if $< C, x, y > \notin \mathcal{I}$
function *Query* $< C, x, y >$):{*yes* and *Instance no*}
begin
  answer := "Is the goal $< C, x, y >$ correct? (y/n)"
  if answer = false **then return**(answer)
  else
    if variable($< C, x, y >$) **then**
      write('Which value');
      Instance = list of values of the variables;
      return(answer); bind variables in $< C, x, y >$ with Instance;
    else return(answer)
    endif
  endif
end

The original algorithm has some disadvantages:

- Variables can appear in the derivation. Shapiro's algorithm substitutes the representation of the variable identifier in PROLOG with variable names from a predefined list instead of keeping the original names applied in the program. If the method finds a variable in the current atom it replaces this variable with the first element of the list and so on. Because the replacing process begins always ahead it can happen that the variables in the proof tree are not forwarded but the same variable is identified with an other name.
- The algorithm asks existence questions which are a pair $< C, x >$. The answers on these questions in an interpretation $\mathcal{I}$ are a finite set $\{y | < C, x, y >$ ist in $\mathcal{I}\}$. So first the method collects the variables and expects from the user that he/she binds these variables with values. The original algorithm can treat only one variable per atom.
- This method as generally Shapiro's algorithms can handle only *pure* logic programs.
- In the case of a missing solution error Shapiro's method can only recognize a set of clauses with the same predicate on the left side as suspicious clauses to contain the error.

To demonstrate how the missing solution method works the modified version of the *isort* example, borrowed from [16] is chosen. The program is called with the goal `isort([3,2,1],X).` and it delivers *fail*.

Example 1 The `isort` example

```
isort([X|Xs],Ys) :- isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs])   :-  Y > X, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :-  X =< Y.
insert(X,[],[X]).
```

The error is in the clause:
`insert(X,[Y|Ys],[Y|Zs]) :- Y > X, insert(X,Ys,Zs).`
The correct version would be:
`insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).`

The proof tree of the program *isort* with the goal `isort([3,2,1],X)` can be seen in the Figure 1. The Algorithm 3 asks the following questions:

```
This fact fails. Is it correct or not? (y/n/c) n

error: missing solution isort([3,2,1],X). diagnosing...
Query:isort([3,2,1],X)?y.
Which X?[1,2,3].
Asserting: sol(isort([3,2,1],_10271),[isort([3,2,1],[1,2,3])]).
Query:isort([2,1],X)?y.
Which X?[1,2].
Asserting: sol(isort([2,1],_10477),[isort([2,1],[1,2])]).
Query:isort([1],X)?y.
Which X?[1].
Asserting: sol(isort([1],_10597),[isort([1],[1])]).
Query:insert(2,[1],[1,2])?y.
Asserting: sol(insert(2,[1],[1,2]),[insert(2,[1],[1,2])]).

[Error diagnosed:,insert(2,[1],[1,2]), is uncovered.]
[Listing of ,insert(X,Y,Z),:]
insert(X,[Y|Z],[Y|U]):-Y>X,insert(X,Z,U).
insert(X,[Y|Z],[X,Y|Z]):-X=<Y.
insert(X,[],[X]):-true.
```

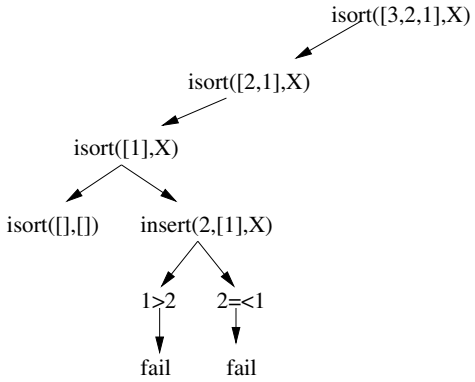


Figure 1: The proof tree with the goal `isort([3,2,1],X)`.

How these disadvantages can be adjusted and other improvements are described in the Section 3.

3 Improvement of the Algorithm using Trace Trees

As it was mentioned in Section 2 the Shapiro's algorithm has some disadvantages, which had to be adjusted. Our main idea is that we introduce a solution based on *trace tree*. If the computation terminates without any output it is very difficult to predict where the error is located. It seems to be advisable to proof all possibilities. For this reason we generate a trace tree from the trace delivered by the PROLOG system itself. This tree contains not only the success derivation but also all *redo* branches. It can be considered as a version of the SLD-tree. The differenc is the following:

- The SLD tree of a goal G with respect to a program $\mathbf{P}$ can be described as follows: The root of the tree is G. Nodes of the tree are (possibly conjunktive) goals with one goal selected. There is an edge leading form node N for each clause in the program whose head unifies with the selected goal. Each branch in the tree from the root is a computation of G by $\mathbf{P}$. Leaves of the tree are succes nodes where the empty goal has been reached or failure nodes where the selected goal at the node cannot be further reduced. Succes nodes correspond to solutions of the root of the tree,
- The trace tree is a part of the SLD tree. It contains all *redo* branches (derivation of failure nodes) generated before the first succes node is computed.
- The proof tree contains only the computation of the first succes node.

Let introduce a small example to demonstrate the different between proof tree, SLD-tree and trace tree The example is borrowed from Lloyd ([8]). The program is called with the goal `p(X,b)`. The different type of trees can be seen in the Figure 2.

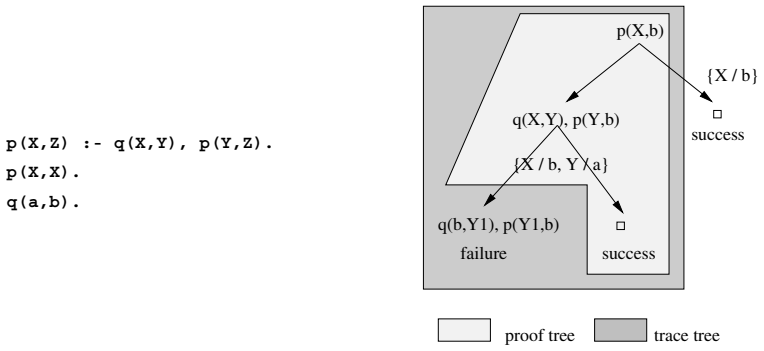


Figure 2: The difference between proof tree, SLD-tree and trace tree

After the generation to find the missing solution this tree can be traversed by all methods used by the incorrect output case.

Moreover in the new version of the *IDTS* system it is implemented that the different traversing methods (top-down, bottom-up and divide and query) are treated similary and work on the generated *trace tree*.

As we mentioned the trace tree is generated using the PROLOG trace. It means that we can handle PROLOG programs which contain control and extra logical facilities. But the application of the trace causes that in certain cases the form of the program is also changed.

For example if a clause in the program contains an *if-statement* then the trace generates from it two different clauses where the first one interprets the condition as true and executes the then-branch. In the second one the condition delivers fail so it contains the else-branch.

For handling built-in predicates by each predicate is examined whether it is a system predicate or not. We assume that the system predicates are always correct. It means that if a node in the proof tree contains a system predicate the system does not ask the user but answers the question with *yes*.

The other improvement is that if the asked atom contains variables the original method collects the variables and expects from the user that he/she binds these with values. On one hand Shapiro's algorithm can treat only one variable per atom. On the other hand this algorithm takes the name of the variables from a fixed list independent which ones are used in the original program. It can lead to misunderstandings and makes it complicated to answer the questions. In our solution arbitrary variables can be labelled with the original names from the program, through which the questions asked by the user are treated and unified.

Using the trace tree the form of the questions has to be also modified. Shapiro asks only questions in the form:

Is the goal $predicate_i(arg_{i1}, \ldots, arg_{in})$ correct? (y/n)

But now the trace tree is used. It means that the system asks the user wether it is intended that a *redo* branch delivers *fail* or not. Accordingly a new type of question is introduced:

Is it intended that the goal $predicate_i(arg_{i1}, \ldots, arg_{i1})$ fails? (y/n)

To answer this question is a hard task without having some information about the nodes derived as descendant of the nodes which keeps the asked predicates. For this reason we improved both types of questions and asks not only whether the predicate is correct or it is intended that it delivers *fail* but we extend these questions and ask whether the predicate with a given right side is correct or it is intended that the computation delivers fail. But this right hand side is generated from the trace delivered by the PROLOG interpreter. This means that if the program contains some extra logical features (as for example cut) they cannot be displayed in the questions because they are worked up by the interpreter.

Is the goal $predicate_{i_0}(arg_{i_0 n_1}, \ldots, arg_{i_0 n_0})$:-
$predicate_{i_1}(arg_{i_1 n_1}, \ldots, arg_{i_1 n_1})$, ..., $predicate_{i_l}(arg_{i_l n_1}, \ldots, arg_{i_l n_l})$ correct? (y/n)

Is it intended that the goal $predicate_{k_0}(arg_{k_0 1}, \ldots, arg_{k_0 n_0})$:-
$predicate_{k_1}(arg_{k_1 n_1}, \ldots, arg_{k_1 n_1})$, ..., $predicate_{k_m}(arg_{k_m n_1}, \ldots, arg_{k_m n_l})$ fails? (y/n)

Of course in the case of fail we can display only so far the right side of a given clause that some predicates deliver fail, because after it the computation makes redo without completely executing the right side and search for an other possibility to derivate the given goal. It is the so called *Backtracking* mechanism.

At the end of the debugging our implementation can display as the erroneous clause the original clause from the program containing the whole right side and additionally the unification of the variables at the moment when *IDTS* found the error. The nonidentified variables are substituted with the original variable identifiers applied in the program.

Our solution delivers the correct clause which is the erroneous clause with the unified arguments by which the program delivers fail.

The conventional advantage of the improvements described above is that the modified method needs much fewer unification given by the user as Shapiro's method because under the building up of the trace tree some variables can be unified by the system.

In the rest of this section the PROLOG program Eratosthenes' Sieve is introduced as example to illustrate how the method works. The program is borrowed from [13].

Example 2 The erroneous program Eratosthenes' Sieve

```
compute(N):-N1 is N-1,  primes(N1).

primes(N1):-numbers(2,N1,Initial_list),  print(Initial_list), nl,
            sieve(Initial_list,Primes),  print(Primes).

numbers(M,N,List):-(M>N ->List=[]; M1 is M+1, numbers(M1,N,Rest),List =[M|Rest]).

sieve([], []).
sieve([Prime|Rest],[Prime|Primes]):-delete_mult(Rest,Prime,Next),sieve(Next,Primes).

delete_mult(List,M,Result):- delete_start(List, M, M, Result).

delete_start([], _M, _X, []).
delete_start([X|Rest],M,X,Ys)     :- !, XpM is X+M, delete_start(Rest,M,XpM,Ys).
delete_start([Z|Rest],M,X,[Z|Ys]) :- Z>X, !, XpM is X+M, delete_start(Rest,M,XpM,[Z|Ys]).
delete_start([Z|Rest],M,X,[Z|Ys]) :- delete_start(Rest,M,X,Ys).
```

The predicate `compute` delivers a list containing all primes less than the given argument. It uses Eratosthenes' sieve method to do it. In the predicate numbers(M,N,List) List contains all numbers between M and N, inclusive. The predicate sieve(Numbers, Primes) computes the result of performing an Eratosthenes sieve sifting operation on Numbers. delete_mult(List, M, Result) deletes from (sorted) List all multiples of M, giving Result and delete_start(List, M, X, Result) deletes from (sorted) List all multiples of M starting with X, giving Result.
Let us introduce the following buggy version of clause delete_start:

delete_start([Z|Rest],M,X,[Z|Ys]):- Z>X, !, XpM is X+M, delete_start(Rest,M, XpM, [Z|Ys]).

the correct version is:

delete_start([Z|Rest],M, X, [Z|Ys]):- Z>X, !, XpM is X+M, delete_start(Rest, M, XpM, Ys).

The program computes with the goal compute(10) *fail*. Of course it is incorrect. So we try to find the error in the program.

As the first step the trace tree is generated (see Figure 3). Then the improved algorithm asks the following questions.

Example 3 The questions asked by the improved algorithm

```
Is it intended that the goal primes(9):- numbers(2,9,[2,3,4,5,6,7,8,9]),
    print([2,3,4,5,6,7,8,9]), nl, sieve([2,3,4,5,6,7,8,9],[2|Primes])fails? (y/n) n
```

The user is asked whether it is intended or not that the predicate `primes` with the number 9 as argument delivers *fail*. Of course it is incorrect so the answer is *no*.

```
Is it intended that the goal numbers(2,9,[2,3,4,5,6,7,8,9]):-2>9  fails? (y/n) y
```

In this case the question is whether it is intended that the predicate `numbers` cannot compute any result. It is intended because on the right side the condition `2>9` correctly delivers *fail* and it causes that the left side also *fails*. This question does not intend to experience whether it is correct that the condition delivers fail. Of course it can be decided

by the system itself. The question does not refer the operator or the call but it ask the user whether the condition is correctly placed or not. It can happen that the programmers made the error writing > instead of <.

```
 Is the goal numbers(2,9,[2,3,4,5,6,7,8,9]):-  3 is 2+1,
     numbers(3,9,[3,4,5,6,7,8,9]),[2,3,4,5,6,7,8,9]=[2,3,4,5,6,7,8,9] correct? (y/n) y
```

Now because we use the trace tree the *else* branch of the clause with the left side `number` is asked. It computes a correct result so the answer is *yes*.

```
Is it intended that the goal [sieve([2,3,4,5,6,7,8,9],[2|Primes]):-
     delete_mult([3,4,5,6,7,8,9],2,Result) fails? (y/n) n
```

Our missing solution method can use any traversing method of Shapiro. Here the top-down method is applied which means that because the question above answered with *yes* the underlying tree is not visited and the next predicate in the top-level-trace is asked. It is the predicate `sieve` which computes *fail*. This result is incorrect so the trace tree under this node has to be reviewed.

```
 Is it intended that the goal [delete_mult([3,4,5,6,7,8,9],2,Result):-
      delete_start([3,4,5,6,7,8,9],2,2,[3|Ys])  fails? (y/n) n
```

The first node of the trace tree with the root `sieve` contains the predicate `delete_mult` which also delivers *fail*. It is also not correct, so we visit the underlying trace tree.

```
Is it intended that the goal delete_start([3,4,5,6,7,8,9],2,2,[3|Ys]):-
   3>2,4 is 2+2,delete_start([4,5,6,7,8,9],2,4,[3|_479881])  fails? (y/n) n
```

Because the predicate `delete_mult` calls `delete_ start` it is asked whether the result that the predicate `delete_ start` does not compute any result is intended or not. It is not intended so the answer is *no*

```
Is it intended that the goal [delete_start([4,5,6,7,8,9],2,4,[3|_481512]):-
  6 is 4+2,delete_start([5,6,7,8,9],2,6,[3|_481405])  fails? (y/n) y
```

The debugger visits the nodes derived from `delete_ start` but it cannot identify the next node with any left side from the program. This is the reason why the question contains variables in the PROLOG representation form `_481512`. It signalizes that something is wrong because the unification cannot proceed. The answer of the user is *yes*: It is intended that the the program delivers *fail* if it cannot unify the given goal with any left hand side from the program.

```
The false clause:
43:delete_start([3,4,5,6,7,8,9],2,2,[3|Ys]) :- 3>2, !, XpM is 2+2,
                                                delete_start([4,5,6,7,8,9],2,XpM,[3|Ys]) .
```

The false clause is identified and the original form of the clause from the Eratosthene's Sieve example is displayed.

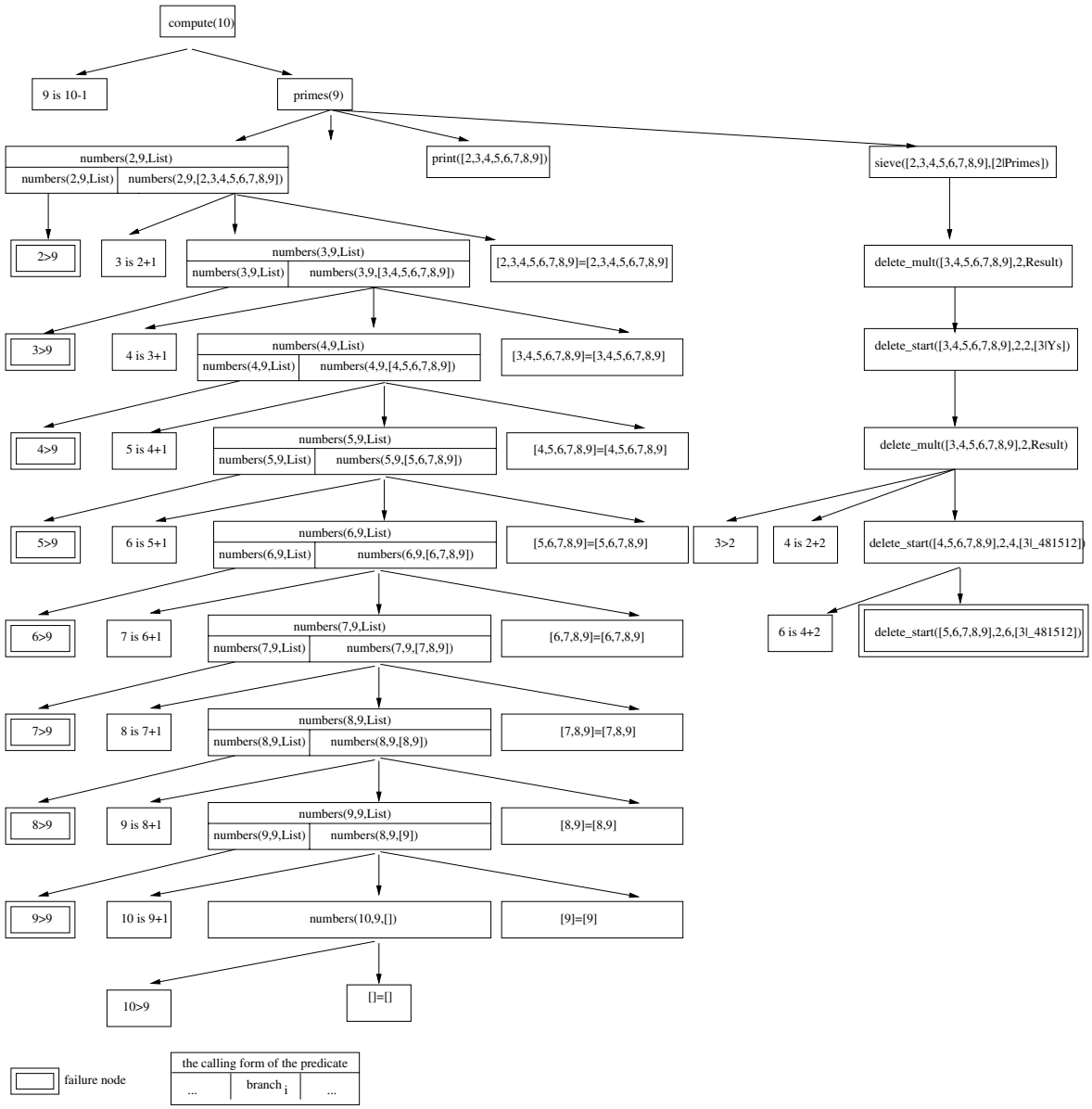


Figure 3: The trace tree with the goal `compute(10)`

4 Conclusion, Further Work

In this paper a new method for the search of a missing solution is presented. This solution based on the trace tree which is built up from the trace generated by the PROLOG interpreter. Moreover we can handle programs which use *full* PROLOG and the extra-logical features of the language. Other types of existence questions are also introduced.

It is clear that it is a hard problem to answer each question correctly. Only that person who wrote the program has a chance to do it. So at moment this solution is suitable for so called *white box testing*. An other problem is what can be done if the error is not recognised. In the future we engage to develop solution for these difficulties. Possibly the graphical

representation of the trace tree in the the scope of a graphical enviroment for *IDTS* can help to support this debugging form. Of course we have to test the method on on real problems with complex PROLOG programs.

References

[1] Apt, K.R., van Emden M.H. : *Contribution to the Theory of Logic Programming* Journal of ACM 29, 841-862, 1982

[2] Clark, K.L *An Introduction to Logic Programming*, In Introductions Reading in Expert Systems, New York, 1982, 93-112

[3] Ducass, M.: *Opium - An Extensible Tracer for Prolog*, Technical Report Lp14, January 1987, 203-210

[4] Drabent, W., Nadjm-Tehrani, S., Małuszyński, J.: *Algorithmic Debugging with Assertions*, In Proc Meta-Programming in Logic Programming, 1988, 375-378,

[5] Eisenstadt, M.: *Retrospective Zooming: A Knowledge Based Tracing and Debugging Methodology for Logic Programming.* In Proc of the 9.th IJCAI, 1985

[6] Ferrand, Gérard:, *Error Diagnosis in Logic Programming an Adaptation of E.Y. Shapiro's Method*, Journal of Logic Programming, 1987, 177-198

[7] Kókai, G.: *Debugging und maschinelles Lernen von PROLOG-Programmen* Arbeitsberichte des Instituts fr Mathematische Maschinen und Datenverarbeitung (Dissertation), Vol. 31, No. 8, Universitt Erlangen-Nrnberg, September 1998.

[8] Lloyd J. W.: Foundation of Logic Programming Springer Verlag, 1985

[9] Lloyd J. W.: *Declarative Error Diagnosis*, New Generation Computing, 1987, 133-154, OHMSHA LTD and Springer Verlag

[10] Chee-Kit Looi: *Analysing Novices' Programs in Prolog Intelligent Teaching System*, In Proc of the European Conference on Artificial Intelligence, Munich 1988, 314-319,

[11] Naish L., Dart, P. W., Zobel, J.: *The NU-Prolog debugging environment* In Proc of the Sixth International Conference on Logic Programming, Lisboa, Portugal, June 1989, 521–536

[12] Nute, D.: *A programming solution to certain problems with loops in Prolog*, ACM SIGPLAN Notices Vol 20, No. 8, 1985,

[13] Salus, Peter H.: *Handbook of Programming Languages, Volume IV, Functional and logic programming languages*, Macmillan Techn. Publ. 1998

[14] Pereira, L. M. : *Rational Debugging in Logic Programming* In Proc of the Third Logic Programming Conference, London, England, July 1986, 203-210

[15] Takahashi, H., Shibayama, E.: *PRESET - A Debugging Environment for Prolog* , Proc of the International Conference on Logic Programming, Tokyo, 1985

[16] Shapiro, E. Y.: Algorithmic Program Debugging MIT Press (1983)

Progress Report on Attempto Controlled English

Norbert E. Fuchs
Institut für Informatik
Universität Zürich
fuchs@ifi.unizh.ch

Abstract

In this progress report we describe version 3 of Attempto Controlled English, a subset of full English that can be unambiguously translated into a logic language and thus can serve as a formal specification and documentation language.

Motivation

Both formal and informal specification languages have advantages and disadvantages. Formal specification languages feature a well-defined syntax and unambiguous semantics and support the automatic analysis of specifications, i.e. verification, validation and even execution. They are, however, for many people incomprehensible and introduce a conceptual distance between the application domain and the domain of formal methods. Informal specification languages – in this context we consider only natural language – are easy to use and to understand and allow users to directly express the concepts of their application domain. On the other hand, it is well known that natural language can lead to ambiguous, incomplete and even inconsistent specifications.

Our goal is to combine the advantages of formal specification languages with the advantages of natural language, and to avoid the disadvantages of both. Our solution is Attempto Controlled English.

Natural Language as a Formal Language

Controlled natural languages – English, French, German etc. – have been used for technical documentation since many years. These controlled languages consist of subsets of the full natural language with a restricted lexicon, a restricted grammar and a large number of style recommendations. The ensuing reduction of ambiguity and complexity improves the comprehensibility of technical texts. Recently, there are even computer-supported style and grammar checking.

Attempto Controlled English (ACE) is a controlled natural language that we developed specifically for software specifications. ACE consists of a precisely and constructively defined subset of English that is computer processable and unambiguously translatable into logic languages. Thus ACE may seem informal, but is in effect a formal language that needs to be learned. Arguably, ACE is easier to learn, use and understand than formal languages that are visibly based on mathematics.

We designed ACE after a careful analysis of natural language specifications found in industry and in the literature. Our analysis identified a number of standard language constructs as candidates for ACE. Further input came from the requirement that the target logic language must be fully and adequately represented in ACE. Surprisingly, our analysis and the representation requirement left ACE underspecified and allowed us to introduce language constructs systematically, constructively and parsimoniously. The result is a small number of construction and interpretation rules that define ACE. Many rules come in pairs that allow users to express the alternative meanings of a sentence.

Attempto Controlled English (ACE)

Vocabulary. The vocabulary of ACE consists of predefined function words (articles, quantifiers, prepositions, conjunctions, pronouns, ...) and user-defined content words (nouns, verbs, adjectives, adverbs). Content words delineate the pertinent application domain and are entered by users with the help of a lexical editor requiring only basic grammar skills.

Construction Rules. Construction rules allow users to constructively build up an ACE specification. They exclude many imprecise language constructs. Here are some construction rules.

- Simple sentences are built according to the schema

subject +	*predicate* +	*(complements)* +	{*adjuncts*}
A customer	*enters*	*a card*	*into a slot.*

- Composite sentences are built from simpler sentences with predefined coordinators (*and*, *or*) and subordinators (*if ... then*, *who*, *which*, *that*), e.g.

  A customer enters a card and a code that is valid.

- Verbs are only used in indicative mood and active voice, only in simple present tense, and only in third person singular or plural.

- Modal verbs (*may*, *can*, *must* etc.) and intensional verbs (*believe*, *suggest* etc.) are not allowed.

- Modal adverbs (*possibly*, *probably* etc.) are not available.

- Coordination is possible between sentences and between phrases of the same syntactic type.

  A customer enters a card and enters a code.

- Coordination between verbal phrases can be simplified by omitting the repeated verb.

  A customer enters a card and a code.

Interpretation Rules. Interpretation rules control the semantic interpretation of grammatically correct sentences, resolve ambiguities and anaphora, and allow users to express a restricted form of ellipsis.

- Verbs denote events or states that are associated with a time.
- The textual order of verbs determines their default temporal order.

  A customer enters a card and types a code.

- The default temporal order can be overridden by predefined adjuncts.

  A customer enters a card and types a code at the same time.

- Prepositional phrases modify the verb, not the noun.

  A customer {*enters a card with a code*}.

- Relative phrases modify the immediately preceding noun.

  A customer enters {*a card that carries a code*}.

- The textual occurrence of a quantifier (*a*, *there is a*, *every*, *for every*) opens its scope that extends to the end of the sentence.

  Every customer enters a card.

- The antecedent of an anaphoric reference is the most recent accessible noun phrase that agrees in number and gender.

  A customer owns a card. She enters it.

Example Specification. Here is an excerpt of the two-page ACE specification of SimpleMat – a simple automated teller machine.

> *A customer enters a card C and a numeric personal code.*
> *SimpleMat checks the code.*
> *If the code is valid then SM accepts the card.*
> *If the code is not valid then SM rejects C.*

The example specification uses

- simple sentences (*a customer enters a card*),
- composite sentences (*and*, *if ... then*, *not*),
- distribution of verb phrases in coordination (*and [enters] a numeric personal code*),
- dynamic names as apposition (*a card C*),
- compounds (*personal code*),
- anaphoric references (definite noun phrase *the code* $\rightarrow$ *a numeric personal code*; definite noun phrase *the card* $\rightarrow$ *a card C*; dynamic name *C* $\rightarrow$ *a card C*),
- synonyms (*code/personal code*; *SM/SimpleMat*).

From ACE to Logic Specifications

The construction and interpretation rules are implemented as a unification-based phrase structure grammar enhanced with feature structures. The grammar is used by a deterministic chart parser that generates concurrently

- a discourse representation structure (DRS) as formal representation of an ACE specification,
- a syntax tree,
- a paraphrase.

Optionally, the ACE text is translated into the standard and the clausal forms of first-order predicate logic.

Discourse Representation Structures. Discourse Representation Theory allows the encoding of interrelations, e.g. anaphoric references, between the sentences of a text. The formal representation of the text is a discourse representation structure (DRS) that uses a syntactic variant of first-order predicate logic. A discourse representation structure (U, C) consists of a set U of discourse referents, i.e. quantified variables representing the objects being talked about, and a set C of simple or complex conditions for the discourse referents.

Example DRS. The ACE text

> *A customer enters a card and a numeric personal code.*
> *If it is not valid then SM rejects the card.*

is translated sentence by sentence into the DRS

```
[A,B,C,D,E,F]
customer(A)
card(B)
event(C,enter(A,B))
numeric(D)
personal_code(D)
event(E,enter(A,D))
named(F,'SimpleMat')
IF
   []
   NOT
      [G]
      state(G,valid(D)
   THEN
      [H]
      event(H,reject(F,B))
```

Translating the second sentence in the context of the first one allows the Attempto system to resolve the anaphoric references (pronoun *it* → *numeric personal code*; definite noun phrase *the card* → *a card*).

Optional Translations. The DRS can also be translated into the standard form of first-order predicate logic, i.e.

```
exists(A, exists(B, exists(C, exists(D, exists(E,
exists(F, customer(A) & card(B) & event(C,enter(A,B))
& numeric(D) & personal_code(D) & event(E,enter(A,D)) &
named(F,SimpleMat) & (-(exists(G, state(G,valid(D)))) => exists(H,
event(H,reject(F,B)))))))))
```

and into the clausal form, i.e.

```
customer(sk1):-true
card(sk2):-true
event(sk3,enter(sk1,sk2)):-true
numeric(sk4):-true
personal_code(sk4):-true
event(sk5,enter(sk1,sk4)):-true
named(sk6,SimpleMat):-true
state(sk7,valid(sk4)),event(sk8,reject(sk6,sk2)):-true
```

Very often the clausal form constitutes a Prolog program.

Example Paraphrase. Translating an ACE specification generates a paraphrase that shows all the substitutions and interpretations done by the parser. For the input

> *The customer enters a card and a numeric personal code.*
> *SimpleMat checks the code.*
> *If the code is valid then SM accepts the card.*
> *If the code is not valid then SM rejects the card.*

the parser generates the paraphrase

> *The customer enters a card and [enters] a numeric personal code.*
> *SimpleMat checks [the numeric personal code].*
> *If [the numeric personal code] is valid then [SimpleMat] accepts [the card].*
> *If [the numeric personal code] is not valid then [SimpleMat] rejects [the card].*

The paraphrase optionally shows interpretation rules, e.g.

> *A customer enters {a card that carries a code}.*

A paraphrase without parentheses is a valid ACE text, which reinforces the learning of ACE.

Constraining Ambiguity

The Attempto system uses three simple means to constrain ambiguity - the curse of natural language processing.

- Some ambiguous language constructs are not part of ACE, e.g. the condition *not* in

If the card is valid then ....
If the code is correct then ....
If not then ....

- Since not all ambiguous language constructs can be eliminated, Attempto parses deterministically according to the interpretation rules.

  Input 1:
  A customer inserts a card that is valid and opens an account.
  Paraphrase 1:
  A customer inserts {a card that is valid} and opens an account.
  Input 2:
  A customer inserts a card that is valid and that opens an account.
  Paraphrase 2:
  A customer inserts {a card that is valid and that opens an account}.

- Users accept the interpretation shown in the paraphrase, or they must reformulate the input

  Input:
  A customer enters a card with a code.
  Paraphrase:
  A customer {enters a card with a code}.
  Reformulated input:
  A customer enters a card that carries a code.

Deductions from ACE Specifications

ACE specifications can be queried and executed to verify and validate them in domain concepts. Query answering and execution are performed in ACE and are realised as logical deduction of DRSs.

A first deduction system performed query answering via translation into Prolog and execution of the specification via a DRS metainterpreter. The deduction system that we currently implement is based on the DRS deduction rules of Gabbay and Reyle. Since DRSs can be translated into the standard form of first-order predicate logic, it may also be interesting to base the deduction system on standard theorem provers.

Querying an ACE Specification. Let an ACE specification S be translated into the DRS DS and an ACE query Q into the DRS DQ. Answering the query Q from S is realised as the deduction

$$DS \vdash DQ$$

The proof binds variables in DS that complete the answer to Q.

There are two types of queries: *yes/no* queries and *wh-queries* (*who*, *what*, *how*, *when*, *where*, ...).

Given the specification

A customer enters a card and a numeric personal code.
SimpleMat checks the code.
If the code is valid then SM accepts the card.
If the code is not valid then SM rejects the card.

we can query it in ACE and get the answers in ACE

Query 1:
Does SM check the code?
Answer 1:
Yes.
Query 2:
What does the customer enter?
Answers 2:
The customer enters [a card].
The customer enters a [numeric personal code].

Executing an ACE Specification. Let an ACE specification S be translated into the DRS DS and an ACE test case T into the DRS DT. Executing T in the context of S is realised as the tracing of the deduction

$$S \vdash T$$

Given the specification

A customer enters a card and a numeric personal code.
SimpleMat checks the code.
If the code is valid then SM accepts the card.
If the code is not valid then SM rejects the card.

and the test case

A customer enters a card.

the execution trace is represented as a dialog with the user who provides missing information. In the first implementation of the Attempto system the dialog looks like

user: *John is a customer*
user: *Visa is a card*
event: *John enters Visa*

while the currently implemented Attempto system will generate an execution trace using events and states as skeleton

event: *A enters B*
A: *customer*
B: *card*

We are also considering an execution trace by highlighting the ACE text used in the current proof step.

Applications

ACE is a tool to acquire and formalise specifications in the concepts of the respective application domain. The Attempto system contains no a priori knowledge or ontology, and makes no assumptions concerning application domains or software engineering methods.

Completed applications. We have completed the specifications of an automated teller machine, of Kemmerer's library database, of Schubert's Steamroller, of Kowalski's subway example, and of a set of data base integrity constraints. Furthermore, we used ACE as the interface language to the model generator EP Tableaux.

Planned applications. Currently, we are extending ACE to serve as the interface language to a synthesiser of logic programs. In addition, it is planned to use ACE to describe diseases and their courses, and to teach logic.

Conclusions and Open Problems

ACE is not just another logic specification language since it

- combines familiar natural language with the rigour of formal methods,
- bridges the conceptual gap between application domains and formal methods,
- resolves the conflict between restricted domain-specific specification languages and powerful abstract specification languages,
- can be defined by a small set of construction and interpretation rules and can, arguably, be easily learned.

Currently we are working on

- extending the coverage of ACE by plurality,
- structuring large specifications,
- introducing complementary notations for graphics and algorithms.

References

More information and references to our own and related publications can be found at `http://www.ifi.unizh.ch/~fuchs/`.

PACS: The Portland Aachen Curry System

– System Demonstration –

Michael Hanus*

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

1 Curry

The development of the higher-order concurrent constraint functional logic programming language Curry is an international initiative intended to provide a common platform for the research, teaching [1] and application of declarative multi-paradigm languages. Curry's computation model, first described in [2], is based on lazy evaluation of expressions together with the demand-driven instantiation of logical variables and the concurrent evaluation of constraints. Thus, Curry combines in a seamless way features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent constraint programming (concurrent evaluation of constraints with synchronization on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions).

Since the functional kernel of Curry is a higher-order lazy functional language, Curry uses a Haskell-like syntax. Actually, Curry can be seen as a conservative extension of Haskell where features for logical variables, constraint solving, and concurrent and distributed programming have been added. Compared to Prolog, the advantage of a higher-order lazy functional base language is the use of an optimal (demand-driven) evaluation strategy and the availability of functions and constraints as first-class values. The latter supports reuse of code and powerful abstractions. To show a small example, consider the following definition of a function to concatenate two lists (`[]` denotes the empty list and ":" is the infix list constructor):

```
conc [] ys     = ys
conc (x:xs) ys = x : conc xs ys
```

This definition can be used to compute the concatenation of two known lists as well as to compute solutions to constraints over functional expressions. Equational constraints have the form "e_1`=:=`e_2" and are solvable if both sides e_1 and e_2 are evaluable to unifiable data terms. Thus, the computed solution to the constraint

```
conc xs [x] =:= [1,2,3]
```

*This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1, the DAAD under the PROCOPE programme, and a grant from Portland State University.

instantiates x with the last element of the list on the right-hand side. Actually, Curry distinguishes between functions that instantiate their arguments ("free") and functions that suspend until their arguments are sufficiently instantiated ("rigid"). As a default (which can be easily changed), constraints are flexible and non-constraint functions are rigid. Thus, purely logic programs (where predicates correspond to constraints) behave as in Prolog, and purely functional programs are executed as in Haskell. Beyond this basic computation model, Curry offers a number of further features:

Concurrency: Constraints can be executed concurrently by composing them with a concurrent conjunction operator.

Constraints: Although the kernel of Curry defines only equational constraints over functional expressions (see above) as basic constraints, different implementations provide also other constraint structures (see, for instance, PACS below).

Input/Output and search: Adapted from Haskell, Curry has a declarative I/O concept (monadic I/O, adapted from Haskell). To encapsulate don't know non-deterministic computations between between I/O actions, Curry has a primitive to control arbitrary don't know non-deterministic steps [5] which can be used to implement different search strategies in Curry.

Programming in the large: To support the development of larger applications, Curry has a module system and a polymorphic type system together with a type inferencer.

Distributed programming: Curry offers port constraints [3] to support the development of reactive and distributed systems. Ports can be declared as external so that they can be accessed by clients via the Internet. The use of logical variables in messages supports an elegant way to return values without complicated communication structures like reply channels.

2 PACS

PACS (the Portland Aachen Curry System) is an implementation of Curry jointly developed by the Aachen University of Technology and Portland State University. It has a common front end and two different back ends: one compiles Curry programs into Java [4] (where Java's concurrency features are exploited) and the other compiles Curry programs into Prolog (and requires a Sicstus-Prolog system). The latter has also an interactive user interface to support the development of larger applications. Beyond the implementation of the basic features of Curry, it supports the following extensions:

- arithmetic constraints (equations, inequations) over reals
- finite domain constraints
- port constraints for distributed programming, e.g., logic programming on the Internet
- library for meta programming
- library for GUI programming (based on Tk)

In this **system demonstration**, we will present the PACS programming environment, the features of Curry and some applications implemented with PACS using the extensions listed above.

More details about Curry: `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/`

More details about PACS and download:
`http://www-i2.informatik.rwth-aachen.de/~hanus/pacs/`

References

[1] 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)*, pp. 335–350. Springer LNCS 1292, 1997.

[2] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

[3] M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.

[4] M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Vol. 1999, No. 6, 1999.

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

Implementing Default Reasoning Using Quantified Boolean Formulae*

(System Description)

Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran

Technische Universität Wien
Abt. Wissensbasierte Systeme 184/3
and
Ludwig Wittgenstein Labor für Informationssysteme
Favoritenstraße 9–11, A–1040 Wien, Austria
e-mail: [uwe,eiter,tompits,stefan]@kr.tuwien.ac.at

In this abstract, we outline the default logic module of the system QUIP, an automated inference tool which provides a uniform implementation for several nonmonotonic reasoning formalisms. The theoretical basis of QUIP is derived from well-known results about the computational complexity of nonmonotonic logics and exploits a representation of the different reasoning tasks in terms of *quantified boolean formulae* (QBFs).

1 Background

A major challenge in designing theorem provers for nonmonotonic reasoning formalisms is the observation that these logics have in general a higher worst-case complexity than classical reasoning. In particular, it has been shown [6, 4, 5] that the main reasoning tasks for almost all propositional nonmonotonic logics are either Σ_2^p-complete or Π_2^p-complete, i.e., they are located at the second level of the polynomial hierarchy, whereas classical reasoning resides at the first level of the polynomial hierarchy.

Important constituents in establishing the Σ_2^p-completeness results are so-called *quantified boolean formulae* (QBFs), which generalize ordinary propositional formulae by the admission of quantifiers ranging over propositional variables. Hence, quantified boolean formulae belong to the language of *second-order logic*. The reason for the consideration of QBFs is the fact that the problem of deciding the satisfiability of QBFs being in prenex normal-form whose leading quantifiers are ordered like $\exists P \forall Q$ (P, Q are lists of propositional variables) is complete for Σ_2^p. Thus, in some sense, this problem is "prototypical" for Σ_2^p. Moreover, it generalizes the situation that the problem of determining the satisfiability of a given propositional formula is complete for nondeterministic polynomial time (i.e., NP-complete). In general, for any class Σ_i^p ($i \geq 1$), there is a corresponding decision problem, QSAT_i, whose task is to check the satisfiability of QBFs of the form $\exists P_1 \forall P_2 \ldots \exists P_{n-1} \forall P_n \, \Phi$, where $P_1, \ldots, P_n$ are lists of propositional variables and Φ is a propositional formula, which is complete for Σ_i^p.

*This work was partially supported by the Austrian Science Fund Project N Z29-INF.

Returning to the discussion of establishing the Σ_2^p-completeness of propositional nonmonotonic decision problems, the hardness parts of these proofs are established by encoding the problem QSAT_2 into the given nonmonotonic decision problem (call it "NMP"). Now, once the membership of NMP in the class Σ_2^p has been shown, from the Σ_2^p-completeness of QSAT_2, it follows immediately that *there is a transformation mapping* NMP *into* QSAT_2 *such that yes-instances of* NMP *correspond to yes-instances of* QSAT_2. The basic idea of the system QUIP, then, is to implement such a translation and to evaluate a given nonmonotonic decision problem in terms of the resultant instance of QSAT_2, by using existing sophisticated theorem provers for quantified boolean formulae.

In its current state, QUIP handles the following nonmonotonic approaches:

- default reasoning;
- disjunctive stable model semantics;
- autoepistemic logic;
- circumscription;
- abduction.

In the next section, we sketch some details concerning the implementation of the default logic part of QUIP.

2 Implementation Details

The overall architecture of QUIP consists of three parts, namely an input filter, a QBF-evaluator, and an output interpreter. The input filter translates the given nonmonotonic decision problem into a quantified boolean formula, which is then sent to the QBF-evaluator. In the present implementation, the latter evaluation task is handled by a commonly available propositional theorem prover, called *boole*, which is based on so-called *binary decision diagrams* (BDDs) [2, 1] (for a comparison between different BDD packages, cf. [7]). We chose to use this particular tool because its design and performance both are well-suited for our purposes. Moreover, the program, together with its source code, is freely available. Other theorem provers which can handle QBFs (like, e.g., [3, 9]) can alternatively be used. In order to obtain accessible answers, the output interpreter of QUIP processes the returned data structures of boole into meaningful tokens, in accordance to the chosen nonmonotonic formalism. Finally, all parts of QUIP are written in C using standard tools like LEX and YACC which make them easily portable to various platforms.

The particular translation used by QUIP to handle default-logic queries is structured as follows. Given a default theory T, the associated QBF Φ encodes the problem whether T possesses an extension, i.e., Φ is true iff T has an extension. The QBF Φ is an existentially quantified formula, comprised of several subformulae, each of which expresses a different property that a set of formulae must obey in order to serve as an extension of T.

Several alternate translations are possible—also with differing semantical meanings, e.g., for the purpose of checking whether a particular formulae is in at least one extension of T, or in all

extensions of T—our translation is based on a *proof-theoretical* characterization of extensions, following [8].

An advantage of our approach is the straightforward extensibility to other nonclassical logics, like, e.g., various modal logics. In fact, since the problem of checking the satisfiability of an *arbitrary* QBF is PSPACE-complete (in contrast to the Σ_i^p-completeness of the problem QSAT_i), *any* logic with a corresponding decision problem within PSPACE can in principle be handled by QUIP, provided a proper encoding has been implemented. Moreover, the modular architecture of QUIP allows an easy scalability and parallelization, by using, e.g., several QBF-provers simultaneously, each of which with its own optimization method.

References

[1] bddlib. http://www.cs.cmu.edu/~modelcheck/bdd.html.

[2] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] M. Schaerf M. Cadoli, A. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proc. of AAAI-98*, 1998.

[4] T. Eiter and G. Gottlob. Propositional Circumscription and Extended Closed World Reasoning are Π_2^P-complete. *Journal of Theoretical Computer Science*, 114(2):231–245, 1993. Addendum in vol. 118, p. 315, 1993.

[5] T. Eiter and G. Gottlob. The Complexity of Logic-Based Abduction. *Journal of the Association of Computing Machinery*, 42(1):3–42, 1995.

[6] G. Gottlob. Complexity Results for Nonmonotonic Logics. *Journal of Logic and Computation*, 2:397–425, 1992.

[7] David E. Long. The Design of a Cache-Friendly BDD Library. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 639–645, San Jose, California, 1998.

[8] W. Marek and M. Truszczyński. *Nonmonotonic Logic: Context-Dependent Reasoning.* Springer Verlag, Berlin, 1993.

[9] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1192–1197. Morgan Kaufmann Publishers, 1999.

Constraintbasierte interaktive und automatische Stundenplanung

– Systembeschreibung –

Hans-Joachim Goltz
GMD – Forschungszentrum Informationstechnik GmbH
GMD-FIRST, Kekuléstr. 7, 12489 Berlin
e-mail: goltz@first.gmd.de

1 Einleitung

Die Entwicklung von Methoden, Verfahren und Konzepten für eine interaktive, automatische Stundenplanung, die in Universitäten und Schulen angewendet werden können, ist ein Bestandteil unserer Forschungsarbeiten. Im Rahmen unserer Forschungen entwickeln wir ein System für die interaktive, automatische Stundenplanung an der Medizinischen Fakultät Charité der Humboldt Universität zu Berlin. Die Stundenplanung der Charité wird seit Sommersemester 1998 mit unserem System erzeugt. Die Vorteile einer kombinierten interaktiven und automatischen Stundenplanerzeugung konnten eindeutig nachgewiesen werden. Die erzeugten Pläne werden als HTML-Dateien ausgegeben und sind im Internet unter `http://www.first.gmd.de/plan/charite` zu finden.
Die Kombination zwischen automatischer und interaktiver Arbeitsweise ist die wichtigste Komponente unseres Systems. Seit 1998 wurde das System kontinuierlich weiterentwickelt. Insbesondere wurde in der letzten Zeit die Lösungssuche und die grafische Unterstützung des Anwenders bei der interaktiven Arbeit verbessert. Das System wird in [3] und [1] genauer beschrieben. Als Implementationssprache wurde die constraintlogische Programmiersprache CHIP (Version 5.2) der Firma COSYTEC aus Frankreich gewählt. Auch die grafischen Schnittstellen wurden in CHIP implementiert. Neben den globalen Constraints erwies sich die objektorientierte Komponente von CHIP besonders vorteilhaft für die Implementation.

2 Problemcharakterisierung

In der medizinischen Ausbildung müssen die Studenten relativ viele Pflichtveranstaltungen absolvieren. Neben den Vorlesungen eines Semesters finden Seminare, Untersuchungskurse und Praktika statt, die in Gruppen bis zu 20 Studenten durchgeführt werden. Dazu werden die Studenten eines Semesters an der Charité in Seminargruppen unterteilt.
Die zwei Standorte der Fakultät befinden sich in verschiedenen Stadtbezirken. Einige Lehrveranstaltungen werden auch in anderen Berliner Kliniken und Krankenhäuser durchgeführt. Die Wege zwischen den möglicherweise verschiedenen Veranstaltungsorten der Lehrveranstaltungen sind zu berücksichtigen. Die Lehrveranstaltungen können zu jeder Viertelstunde beginnen und können unterschiedliche, vorgegebene Längen haben. Einige Lehrveranstaltungen finden nur in bestimmten Wochen statt. Den Vorlesungen sind im Stundenplan auch Räume konfliktfrei zuzuordnen. Viele Vorlesungen können dabei nur in bestimmten Räumen stattfinden. Die Anzahl der Seminare, Untersuchungskurse und Praktika, die eine Klinik gleichzeitig durchführen kann, ist beschränkt, wobei die Schranke auch zeitabhängig sein kann. Die Wunschzeiten und Raumwünsche für Vorlesungen sind möglichst zu berücksichtigen.

3 Modellierung und Lösungssuche

Der Constraintlöser über endlichen Domänen von CHIP bildet die Grundlage unserer Problemmodellierung. Die gewählte Modellierung besitzt einen großen Einfluß auf die Propagationseigenschaften des Constraintlösers und damit auf die Effizienz der Lösungssuche. Die in CHIP enthaltenen globalen Constraints haben sich für unsere Modellierung als sehr wertvoll erwiesen. Durch das Hinzufügen von redundanten Constraints konnte die Effizienz der Lösungssuche wesentlich verbessert werden. Mit Ausnahme der Berücksichtigung der Wunschzeiten konnten alle Bedingungen direkt als Constraints formuliert werden und vor der Lösungssuche erzeugt werden. Die Berücksichtigung der Wunschzeiten ist ein weiches Constraint. Innerhalb der Lösungssuche wird versucht, die Wunschzeiten möglichst zu erfüllen.
Für die Erzeugung einer Lösung ist i. allg. Suche erforderlich, die oft durch *"Labelling"* realisiert wird. Dabei werden schrittweise den Constraintvariablen Werte aus ihren Domänen zugeordnet. Als Modifizierung dieser Suchmethode haben wir in [2] vorgeschlagen, daß die Wertzuweisung durch eine Einschränkung der Domäne der ausgewählten Variable ersetzt wird. Diese Idee der backtrackbaren Domänenreduzierung wurde auch hier verwendet.
In vielen Fällen unserer Versuche konnte eine Lösung entweder in wenigen Suchschritten (Backtrackingschritten) gefunden werden oder es wurde eine außerordentlich große Anzahl von Suchschritten benötigt, falls das Problem überhaupt lösbar war. Deswegen wählten wir die folgende grundsätzliche Vorgehensweise bei der Lösungssuche: die Anzahl der erlaubten Suchschritte wird stark eingeschränkt und es werden verschiedene Heuristiken der Variablenauswahl bzw. der Domänenreduzierung verwendet. Somit wird ein Backtracking über verschiedene Heuristiken durchgeführt.
Neben der vollständigen automatischen Planerzeugung kann die Lösungssuche über eine grafische Repräsentation interaktiv beeinflußt werden. Folgende Möglichkeiten der interaktiven Planerzeugung, die beliebig kombinierbar sind, wurden realisiert: Lehrveranstaltungen einzeln einplanen (nur widerspruchsfreie Zeiten können zugeordnet werden), markierte Lehrveranstaltungen automatisch einplanen, markierte geplante Lehrveranstaltungen ausplanen, alle noch nicht geplante Lehrveranstaltungen automatisch einplanen. Die interaktive Einzelplanung wird durch Darstellung der auswählbaren Werte unterstützt. Bei der interaktiven Beeinflussung der Lösungssuche erfolgt kein "Backtracking" über Entscheidungen des menschlichen Planers. Durch die interaktive Komponente können spezielle Lösungen erzeugt und vorhandene Pläne modifiziert werden. In einigen Fällen war auch eine schrittweise Planerzeugung notwendig, um überhaupt einen Plan in kurzer Zeit zu erzeugen. Nur durch diese interaktive Komponente konnte sich dieses System der Stundenplanung in der Praxis so gut bewähren.

Literatur

[1] H.-J. Goltz. Über Methoden des constraintbasierten Lösens von Problemen der Stundenplanung. In Proc. 14. Workshop on Logic Programming.

[2] H.-J. Goltz. Reducing domains for search in CLP(FD) and its application to job-shop scheduling. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming – CP'95*, volume 976 of *Lecture Notes in Computer Science*, pages 549–562, Berlin, Heidelberg, 1995. Springer-Verlag.

[3] H.-J. Goltz and D. Matzke. University timetabling using constraint logic programming. In G. Gupta, editor, *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 320–334, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

Constraintbasierte Raumplanung für Universitäten
— Systembeschreibung —

Slim Abdennadher, Matthias Saft, Sebastian Will
Institut für Informatik, Ludwig-Maximilians-Universität
Oettingenstraße 67, 80538 München
{abdennad,saft,wills}@informatik.uni-muenchen.de

Die Aufgabe „Raumplanung“ an einer Universität besteht im Allgemeinen darin, die vorhandenen Räume zu den stattfindenden Veranstaltungen zuzuordnen. In vielen Universitäten haben die verschiedenen Fakultäten keine eigenen Räume zur Verfügung. Deshalb muß die Raumplanung von einer zentralen Stelle durchgeführt werden. Bei genauerer Betrachtung stellt sich heraus, daß diese Aufgabe sehr aufwendig und zeitraubend ist, da zum einen sehr viele Veranstaltungen und Räume zu verwalten sind, und zum anderen zugeordnete Räume die für die Veranstaltung benötigte technische Ausstattung besitzen müssen und nicht doppelt belegt sein dürfen. Um die Raumplanung an der Ludwig-Maximilians-Universität in München unterstützen zu können, wurde ein System, genannt *Raumplaner*, entwickelt, das die folgenden Funktionalitäten anbietet:

1. Erstellung eines Raumplanes: Hier wird die eigentliche Zuordnung von Veranstaltungen an Räume unter Beachtung oben genannter Bedingungen (technische Ausstattung, keine Doppelbelegung) vorgenommen.

2. Visualisierung des Raumplans und Möglichkeit der Änderung (Hinzufügen, Löschen, Ändern von Daten) des Plans.

Bei der Generierung eines Raumplans müssen unterschiedliche Constraints berücksichtigt werden. „Harte“ Constraints sind Bedingungen, die unbedingt erfüllt werden müssen, und „weiche“ Constraints sind Bedingungen, die nach Möglichkeit erfüllt werden sollen. Ein neuer Ansatz, solche Optimierungsprobleme anzugehen, ist die Constraint-Programmierung [4, 6, 3, 5]. Der *Raumplaner* verwendet diesen Ansatz, um sowohl harte als auch weiche Constraints zu behandeln.

Dafür haben wir einen Constraintlöser für endliche Bereiche, der in der deklarativen Constraintsprache *Constraint Handling Rules* (CHR) [1, 2] implementiert ist, erweitert. Mit dem neuen Constraintlöser wird die Berechnung der Kosten für eine Lösung, in diesem Fall für einen Raumplan, während der Propagierung der Constraints stattfinden und nicht wie im klassischen Ansatz während der Enumerationsphase. Kosten für eine Lösung entstehen, wenn weiche Constraints verletzt werden.

Der *Raumplaner* erhält als Eingabe eine Liste der vorhandenen Räume zusammen mit ihrer technischen Ausstattung (Sitzplätze, vorhandene Geräte), sowie einer Liste der stattfindenden Veranstaltungen, mit den für jede Veranstaltung benötigten Raumwünschen, Sitzplätzen und Geräten. Ausgabe dieses Programms ist ein Raumplan, der oben genannten Bedingungen genügt.

Nachdem ein Raumplan generiert wird, kann er visualisiert und verändert werden. Dazu wurde eine graphische Benutzeroberfläche zur Ausgabe des Raumplans, Hinzufügen von Veranstaltungen oder Räumen, und zur Änderung von Veranstaltungen (Veranstaltungszeitpunkt, benötigte Technik) und der Raumausstattung, entwickelt. Besonderen Wert wurde dabei auf die Einfachheit der Eingabe gelegt, die mittels einer für alle Fälle (Hinzufügen, Ändern) gleichen Eingabemaske erfolgt. Die Ausgabe geschieht in Form eines graphischen Raumplans, wie er bisher auch auf Papier erstellt wurde.

Beim Hinzufügen oder Ändern von Veranstaltungen werden Konflikte (z.B. doppelte Raumbelegungen oder nicht ausreichende Technik) vom *Raumplaner* erkannt. Tritt solch ein Fall auf, so erhält der Benutzer die Möglichkeit, einen *passenden* Raum suchen zu lassen. Ein Hörsaalplan kann automatisch generiert, und ausgedruckt werden. Die Implementierung erfolgte in Java. Zur Verwaltung aller benötigten Daten (Räume und Veranstaltungen mit jeweiliger vorhandener oder benötigter Technik) wurde ein Datenbank-Schema für das Datenbanksystem mSQL entwickelt.

Literatur

[1] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995.

[2] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.

[3] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung: Grundlagen und Anwendungen.* Springer-Verlag, September 1997.

[4] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20, 1994.

[5] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[6] M. Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1,2), September 1996.

Terminierungsanalyse in dem statischen Codeanalysesystem CAPP

Gabriella Kókai[1]

[1] IMMD-II, Friedrich-Alexander University of Erlangen-Nürnberg
Martensstr. 3. D-91058 Erlangen, Germany
e-mail: kokai@informatik.uni-erlangen.de

Zusammenfassung

Code Analysis of PROLOG Programs *CAPP* erkennt einfache Fehlerklassen allgemeiner Art. Aber der wesentliche Teil des *CAPP* Systems wurde im PROLOG implementiert und es übeprüft automatisch die sogenannte Links-Termination eines Logikprogramms: Gegeben ein Programm und eine Abfrage, antwortet das System entweder, dass die Abfrage terminiert, oder dass Nictterminierung vorkommen kann. Der Algorithmus, der durch das System eingesetzt wird, besteht aus drei Hauptteilen: Instantiierungsanalyse, Erzeugung der Regelgraphen und Aufbau der Ableitungspaare, entsprechend dem Programm und der Abfrage. Der Algorithmus wurde an vielen in der Literatur vorkommenden Beispielen getestet und in das IDTS-System integriert.[1]

1 Einleitung

Das Ziel der Entwicklung von *CAPP* (Code Analysis of PROLOG Programs) war die Erstellung eines Systems zur statischen Codeanalyse von PROLOG-Programmen zur Ergänzung und Unterstützung des IDTS-Systems ([7]). *IDTS* ist ein interaktives System zum dynamischen Testen und Debuggen von PROLOG-Programmen.

CAPP untersucht und erkennt einfache Fehlerklassen allgemeiner Art. Außerdem beinhaltet *CAPP* ein Instantiierungs- und Terminierungsanalysemodul. Die Instantiierungsanalyse ermittelt Modus-Informationen, die angeben, welche Instantiierungen eines Prädikats möglich sind. Die Terminierungsanalyse versucht die Terminierung eines PROLOG-Programms aus einer gegebenen Startanfrage nachzuweisen. Die wesentlichen Strukturen dieser Analyse bilden dabei die Regelgraphen, sowie die daraus ableitbaren Abbildungspaare. Anhand eines Terminierungstests, der für alle Abbildungspaare durchgeführt wird, kann entschieden werden, ob ein gegebenes Programm terminiert oder ob möglicherweise Nichtterminierung vorliegt.

Die Analyseverfahren basieren auf der Arbeit von Lindenstrauss und Sagiv ([8]) aber beide können verbessert werden: Durch ein top-down-Verfahren, bei dem die Information, gegeben durch das Instantiierungsmuster der Startanfrage, mitverwendet wird, kann die Instantiierungsanlyse bezüglich der Effizienz verbessert werden. Ein Effizienzgewinn der Terminierungsanalyse kann durch die Einschränkung der zu erzeugenden Abbildungspaare auf rekursive Abbildungspaare erreicht werden.

Die Entwicklung des Systems ist noch nicht abgeschlossen. Erste Tests werden in diesem Artikel schon präsentiert, aber wir brauchen ausführlichere Tests, um über die Mächtigkeit und Geschwindigkeit im Vergleich mit anderen Systemen etwas sagen zu können. Ein resultat der Tests ist, dass bei Unterprogrammen, bei denen die Herleitung von Größenbeziehungen innerhalb der Atome (Inference of Constraints) durchgeführt werden muss, die Terminierungsanalyse zu keinem Ergebnis kommt.

In Kapitel 2 wird zunächst das Verfahren vorgestellt, mit dem Programme auf Terminierung hin überprüft werden können Im darauffolgenden Kapitel 3 wird unser Verbesserungsvorschlag beschrieben. In Kapitel 4 werden einige Textergebnisse vorgestellt. Die Komplexität und frühere Arbeiten werden im Kapitel 5 behandelt. Das abschließende Kapitel 6 fasst die wesentlichen Ergebnisse dieser Arbeit zusammen.

[1] Diese Arbeit wurde von dem Stipendium Bayerischer-Habilitationsförderpreis1999 unterstützt.

2 Terminierungsanalyse

Dieses Kapitel beschreibt ein Terminierungsverfahren, das bei gegebenem Programm **P** und Startanfrage Q die Terminierung von **P** für Q nachweist, oder ausgibt, dass **P** für Q möglicherweise nicht terminiert. Die grundsätzliche Idee dieses Terminierungsverfahrens basiert auf dem Nachweis, dass gewisse Argumente, die ausreichend instantiiert sind, bezüglich einer gegebenen Norm größenmäßig abnehmen. Mit den verschiedenen Möglichkeiten, wie man eine Norm definieren kann, beschäftigen wir uns nicht näher, wir verwenden die von Lindenstrauss und Sagiv definierte Term-Norm. Mehr davon kann man in [1], [10] und [3] nachlesen.

In dem ersten Abschnitt dieses Kapitels werden die notwendigen Definitionen gegeben, die für die Terminierungsanalyse von grundlegender Bedeutung sind. Von Abschnitt 2.2 bis Abschnitt 2.6 wird dann die eigentliche Terminierungsanalyse beschrieben.

2.1 Vorbereitende Definitionen und Begriffe

Definition 1 Sei B ein SLD-Ableitungsbaum. Auf den Knoten von B wird eine Relation $\succ$ wie folgt definiert: $g_1,\ldots,g_n \succ r_1,\ldots,r_m$, wenn r_1 aus g_1 durch einen Resolutionsschritt gewonnen wird. Der transitive Abschluss von $\succ$ wird als $\succ^+$ bezeichnet. Im Falle $g_1,\ldots,g_n \succ^+ r_1,\ldots,r_m$ bezeichnet man das Paar (g_1, r_1) als *Ableitungspaar*.

Anschaulich besteht ein Ableitungspaar aus zwei Zielen G_1 und G_2, wobei G_2 durch Anwendung von PROLOG-Ableitungen aus G_1 entsteht. Ableitungspaare stellen also PROLOG-Ableitungen dar. Eine notwendige Bedingung für die Nichtterminierung eines PROLOG-Programms bezüglich einer Anfrage ist, dass der dazugehörige SLD-Baum einen unendlichen Ast besitzt. Man kann nun einen Terminierungstest so definieren, dass der SLD-Ableitungsbaum zu einem gegebenen Programm und einer Anfrage genau dann endlich ist, wenn jedes erzeugbare Ableitungspaar diesen Terminierungstest erfüllt. Das Problem, das sich dabei ergibt, ist, dass unendlich viele Ableitungspaare vorkommen können. Die Terminierungsanalyse erzeugt nun abstrahierte Ableitungspaare, so dass sich eine endliche Menge ergibt. Diese endliche Menge kann dann auf Terminierung überprüft werden. Wichtig ist hierbei, dass die Information, die für die Terminierung notwendig ist, durch die Abstraktion nicht verloren geht, und es somit ausreicht die endliche Menge der abstrahierten Ableitungspaare auf Terminierung hin zu testen.

Als nächstes wird der Begriff *ausreichend instantiiert* definiert, was für die gesamte Terminierungsanalyse von Bedeutung ist, da der Nachweis der Terminierung nur an den Argumenten erfolgt, die bezüglich einer symbolischen Term-Norm ausreichend instantiiert sind. Deswegen müssen wir zuerst die symbolischen Term-Norm definieren.

Definition 2 Sei $T = f(t_1,\ldots,t_n)$ ein Term, dann wird die symbolische Term-Norm von T definiert durch $\| f(t_1,\ldots,t_n) \| := c + \sum_{i=1}^{n} a_i \| t_i \|$ mit c, a_i positive Konstanten.
Für Variablen definiert man $\| X \| := X$

Definition 3 Ein Term T ist bezüglich einer symbolischen Term-Norm *ausreichend instantiiert* (**ai**), wenn $\| T \|$ einen Zahlenwert ergibt, sonst ist T nicht ausreichend instantiiert (**nai**).

Example 1 Wir erhalten die Term-Norm, die für einen Grundterm als die Summe der Stelligkeit seiner Funktoren definiert werden kann, indem wir für jedes f/n $c = n$, $a_1 = \ldots a_n = 1$*wählen*. So ist zum Beispiel ist die Term-Norm von T_1 =`f(g(X,X,Y),X)` $5+3X+Y$, von T_2=`p(X,Y)` $2+X+Y$ und von T_3=`p(X,Y)` 1. Für die Terme T_1, T_2 und T_3 gilt: T_1 und T_2 sind **nai** und T_3 ist **ai**.

2.2 Instantiierungsanalyse

Mit Hilfe der Instantiierungsanalyse gewinnt man Informationen bezüglich der Instantiierung der Argumente eines Prädikats. Diese Information kann in Form eines Instantiierungsmusters dargestellt werden, das für jedes Argument angibt, ob es ausreichend instantiiert ist oder nicht. Eine Klausel mit dem dazugehörigen Instantiierungsmuster wird dann als Instanzklausel, deren Atome als Instanzen bezeichnet. Die Instantiierungsanalyse generiert dann für jedes Prädikat die entsprechenden Instanzen.

Definition 4 Unter einer *Instanz* versteht man ein Atom $atom(arg_1, \ldots, arg_n)$ mit $arg_i \in \{\mathbf{ai}, \mathbf{nai}\}$.

Definition 5 Unter einer *Instanzklausel* versteht man eine Klausel der Form
$atom_1(arg_{1,1}, \ldots, arg_{1,n_1}) : -atom_2(arg_{2,1}, \ldots, arg_{2,n_2}), \ldots, atom_m(arg_{m,1}, \ldots, arg_{m,n_k})$.
mit $arg_{i,j} \in \{\mathbf{ai}, \mathbf{nai}\}$.

Als nächstes wird der Konsequenz-Operator definiert. Dazu benötigt man die Menge der Instanzklauseln zu einem gegebenen PROLOG-Programm. Das Verfahren besteht aus den folgenden zwei Schritten:

- Für jede Klausel C mit n verschiedenen Variablen generiert man 2^n Klauseln durch Substitution der Variablen durch **ai** und **nai**.
- Für jede der erzeugten Klauseln wird die symbolische Term-Norm für jedes Argument berechnet, wobei man für **ai** bzw. **nai** folgendes setzt: $\| \mathbf{ai} \| = 1$ $\| \mathbf{nai} \| = \mathrm{X}$. Jedes Argument wird dann durch **ai** ersetzt, wenn dessen symbolische Term-Norm einen Zahlenwert ergibt, sonst durch **nai**.

Im folgenden bezeichne C_I die Menge aller nach obigen Verfahren erzeugten Instanzklauseln. Der Konsequenz-Operator T_p wird folgendermaßen definiert:

Definition 6 Sei I eine Menge von Instanzen, C_I eine Menge von Instanzklauseln. Der *Konsequenz-Operator* T_p wird definiert durch $T_p(I) = \{H \mid (H : -A_1, \ldots, A_n) \in C_I \wedge \forall i\ 1 \leq i \leq n : A_i \in I\}$.

Algorithm 1 Ermittlung aller Instanzen aus einer Menge von Instanzklauseln

Eingabe: die Menge der Instanzklauseln C_I.
Ausgabe: die Menge der Instanzen M.
M := $\{I \in C_I \mid I$ ist Fakt$\}$; M' := $\emptyset$; $C_I := C_I$ - Instanzen-Fakten
WHILE M $\neq$ M'
  M ' := M
  FOR $I \in C_I$ **DO** /*** I = H : - $A_1, \ldots, A_n$ ***/
    IF $\forall i : 1 \leq i \leq n$: $A_i \in M$
    THEN M := M $\cup$ H
  END
END

Algorithmus 1 ermittelt alle Instanzen, die aus der Menge der Instanzklauseln herleitbar sind. Die innere Schleife des Algorithmus realisiert den Konsequenz-Operator. Eine Instanz H wird zur Menge M hinzugefügt, wenn sie Kopfatom einer Instanzklausel ist, deren Rumpfatome in M enthalten sind. M enthält zunächst nur die Instanzen, die Fakten darstellen. Der Algorithmus terminiert, wenn keine weiteren Instanzen durch den Konsequenz-Operator gefolgert werden können. Die Terminierung des Algorithmus ist gesichert, da die Menge aller Instanzen endlich ist, und ein Fixpunkt existiert. Dieser entspricht nach Ablauf des Algorithmus der Menge M.

2.3 Regelgraphen

Regelgraphen stellen die wesentlichen Strukturen zur Terminierungsanalyse dar. Jeder PROLOG-Regel kann ein solcher Regelgraph zugeordnet werden. In diesem Abschnitt folgen zunächst die notwendigen Definitionen. Im nächsten Abschnitt wird dann gezeigt, wie aus den Regelgraphen gewisse Untergraphen erzeugt werden, die gewisse Kriterien erfüllen müssen, damit die Terminierung garantiert werden kann. Im folgenden bezeichne R eine PROLOG-Regel.

Definition 7 Sei R eine Regel eines PROLOG-Programms. Ein *Regelgraph* ist ein gerichteter, gewichteter Graph G = (V, E) mit endlicher Punktemenge V und Kantenmenge $E \subseteq V \times V$.

Die Punkte von G entsprechen den Argumentpositionen aller Atome von R. N_k bezeichne die symbolische Term-Norm eines Arguments, dessen Argumentposition dem Punkt V_k in G entspricht, dann gilt für die Kantenmenge E:

$$(V_i, V_j) \in E, \text{ wenn } N_j \leq N_i$$

Kanten verbinden also Punkte, deren entsprechende Argumente in R bezüglich der symbolischen Term-Norm vergleichbar sind. Aus dieser Definition ergeben sich auch ungerichtete Kanten zwischen Punkten, deren entsprechende Argumente bezüglich der symbolischen Term-Norm gleich sind, das heißt wenn $N_j = N_i$ gilt.

Ungerichtete Kanten besitzen kein Gewicht, für gerichtete Kanten ergibt sich das Gewicht durch die Normdifferenz der entsprechenden Argumente in R.

Die Punkte eines Regelgraphen G sind entweder schwarz oder weiß gefärbt. Diese Färbung der Punkte spiegelt die Instantiierung der in der Regel R entsprechenden Argumente wider. Ein Punkt V eines Regelgraphen ist genau dann schwarz gefärbt, wenn das dem Punkt V entsprechende Argument in R ausreichend instantiiert ist, sonst ist V weiß gefärbt. Zwei weitere Behauptungen kann man einfach nachweisen:

- Sind zwei Punkte eines Regelgraphen durch eine ungerichtete Kante verbunden, so besitzen sie die gleiche Färbung.
- Ist ein Punkt V_i eines Regelgraphen schwarz gefärbt und existiert eine gerichtete Kante von V_i zu einem Punkt V_j, so muß auch V_j schwarz gefärbt sein.

Weitere Kanten gewinnt man durch den transitiven Abschluß mit Hilfe zweier Regeln zur Herleitung von Kanten:

- Regel 1: Eine ungerichtete Kante (V_i, V_j) wird zu E hinzugefügt, wenn
- Regel 2: Eine gerichtete Kante (V_i, V_j) wird zu E hinzugefügt, wenn ein Pfad von V_i nach V_j mit Gesamtgewicht $G > 0$ existiert.

Zu den Regeln ist anzumerken, dass eine gerichtete Kante mit dem Gewicht w auch in Gegenrichtung mit Gewicht $-w$ durchlaufen werden kann.

2.4 Ableitungspaare

Ableitungspaare werden aus Untergraphen eines Regelgraphen gewonnen. Sie bilden diejenigen Strukturen, an denen die Terminierung festgestellt werden kann. Es folgt zunächst die Definition:

Definition 8 Sei R eine PROLOG-Regel, und G der dazugehörige Regelgraph, dann ist ein *Ableitungspaar* ein Untergraph von G, dessen Punktemenge auf die Argumentpositionen des Kopfatoms von R und eines Rumpfatoms von R beschränkt ist. Die dem Kopfatom entsprechenden Punkte werden als *Domain* des Ableitungspaares, die restlichen Punkte als *Range* bezeichnet. Die Kanten eines Ableitungspaares ergeben sich aus den Kanten des Regelgraphen mit Ausnahme von gerichteten Kanten die zwei weiße Punkte verbinden.

Aus der Definition 8 geht hervor, dass aus einer Regel R mit n Rumpfatomen n Ableitungspaare gewonnen werden können.

Wie schon in Kapitel 2.1 erwähnt wurde, stellen reale Ableitungspaare PROLOG-Ableitungen dar. Bisher wurde lediglich gezeigt, wie Ableitungspaare aus einer Regel gewonnen werden. Um PROLOG-Ableitungen vollständig darzustellen, müssen diese Ableitungspaare komponiert werden:

Definition 9 Zwei *Ableitungspaare* AP_1 und AP_2 sind komponierbar, wenn der Range von AP_1 gleich dem Domain von AP_2 ist, und deren Prädikatnamen übereinstimmen. Sind zwei Ableitungspaare AP_1 und AP_2 komponierbar, so ergibt sich das komponierte Ableitungspaar $AP_c = AP_1 \circ AP_2$ aus dem Domain von AP_1 und dem Range von AP_2. Die Kanten von AP_c ergeben sich aus dem transitiven Abschluss der entsprechenden Kanten aus AP_1 und AP_2.

Jetzt sind wir so weit, dass wir die Menge aller erzeugbaren Ableitungspaare bilden können: Man erzeugt zunächst alle Ableitungspaare P_R, die aus allen Regelgraphen gebildet werden können. Diese Ableitungspaare werden unter Abschluss komponiert, wobei sich eine weitere Menge P_C von Ableitungspaaren ergibt. $P_R \cup P_C$ ergibt dann die Menge aller erzeugbaren Ableitungspaare.

2.5 Terminierungstest

In diesem Abschnitt wird ein Terminierungstest definiert, der für alle erzeugbaren Ableitungspaare eines Programms **P** erfüllt sein muß, damit die Terminierung von **P** garantiert werden kann. Der Terminierungstest verlangt, dass es für jede rekursive PROLOG-Regel eine ausreichend instantiierte Argumentposition gibt, dessen Argument bezüglich der symbolischen Term-Norm zwischen zwei Aufrufen dieser Regel abnimmt.

Definition 10 Sei $AP = (V, E)$ ein Ableitungspaar, wenn Domain and Range identisch sind, dann kann man ein *variantes Ableitungspaar* $AP_v = (V, E)$ erzeugen mit $E' = E \cup E_v$, wobei $E_v = \{(V_i, V_j) \mid (V_j, V_i) \in E \wedge (V_i, V_j) \notin E \wedge V_i$ und V_j liegen nicht beide zugleich in Domain oder Range, V_i und V_j entsprechen Knoten der selben Argumentposition in Argumentpaar$\}$. Die Menge E_v und die Menge der ungerichteten Kanten aus E bilden zusammen die Menge der *zirkulären Kanten*.

Für den folgenden Terminierungstest gilt:

- zirkuläre Kanten besitzen das Gewicht 0.
- gerichtete Kanten dürfen nicht in Gegenrichtung durchlaufen werden.
- ein positiver Zyklus ist ein Zyklus, bei dem mindestens eine gerichtete, nicht zirkuläre Kante ein Teil dieses Zyklus ist.

Der Terminierungstest lautet: Ein variantes Ableitungspaar AP_v erfüllt den Terminierungstest, wenn AP_v einen positiven Zyklus enthält, und in diesem Zyklus jede zirkuläre Kante von Range nach Domain durchlaufen wurde.

2.6 Das Verfahren

Das Verfahren bekommt als Eingabe ein PROLOG-Programm **P** und eine Anfrage Q in instantiierter Form, das heißt die Argumentpositionen dieser Anfrage sind entweder **ai** oder **nai**. Der erste Schritt des Verfahrens besteht darin, für jede Regel aus **P**, deren Prädikatname mit Q übereinstimmt, einen Regelgraphen zu erstellen. Sei nun R eine solche Regel:

$$atom_0(\ldots) :- atom_1(\ldots), \ldots atom_m(\ldots).$$

Der zu R erstellte Regelgraph RG besitzt zunächst nur weiße Punkte. Färbe im Regelgraph RG die dem Kopfatom $atom_1(\ldots)$ entsprechenden Punkte gemäß der Instantiierung von Q. Mit Hilfe der Regeln aus Abschnitt 2.3 werden dann neue Kanten erzeugt und, wenn möglich, weitere Punkte schwarz gefärbt. Für jedes i mit $2 \leq i \leq m$ und für jedes j mit $2 \leq j < i$ tue man folgendes:

- Aus den Instanzen, die die Instantiierungsanalyse für das Prädikat $atom_j$ erzeugt, wähle man diejenige Instanz I, dessen Argumentpositionen **ai** sind, wenn die diesen Argumentpositionen entsprechenden Punkte schwarz gefärbt sind.
- Färbe dann die Punkte von $atom_j$ gemäß der Instantiierung von I.
- Färbe aufgrund von Abschnitt 2.3 weitere Punkte von $atom_j$.

Aus dem so entstandenen Regelgraph können dann die $m-1$ Ableitungspaare gewonnen werden. Auf alle erzeugten Ableitungspaare wird dann der Terminierungstest angewandt. Erfüllt mindestens ein Ableitungspaar diesen Test nicht, so bricht das Verfahren ab und meldet, dass **P** möglicherweise nicht terminiert. Erfüllen alle Ableitungspaare den Terminierungstest, so werden durch Komposition neue Ableitungspaare erzeugt, auf die ebenfalls der Terminierungstest angewandt wird.

Algorithm 2 Terminierungsverfahren

Eingabe: ein PROLOG-Programm **P** und eine Startanfrage Q.
Ausgabe: Die Antwort P terminiert bezüglich der Startanfrage Q, falls die Terminierung von **P** nachgewiesen werden konnte, sonst terminiert **P** möglicherweise nicht.

Alte_Paare = $\emptyset$; Alle_Anfragen = {Q}; Test_Anfragen = {Q}
WHILE Test_Anfragen $\neq \emptyset$ **DO**
  Anfragen = Alle aus der Menge Test_Anfragen erzeugbaren Anfragen
  Paare = Alle aus der Menge Test_Anfragen erzeugbaren Paare
  Führe Terminierungstest für alle Paare durch
  Test_Anfragen = Anfragen - Alle_Anfragen
  Alle_Anfragen = Alle_Anfragen $\cup$ Anfragen
  Neue_Paare = Paare - Alte_Paare
  WHILE Neue_Paare $\neq \emptyset$ **DO**
    Komponierte_Paare = $\{P1 \circ P_2 \mid P_1 \in$ Alte_Paare, $P_2 \in$ Neue_Paare$\}$
    $\cup\{P_1 \circ P_2 \mid P_1 \in$ Neue_Paare, $P_2 \in$ Alte_Paare$\} \cup\{P_1 \circ P_2 \mid P_1 \in$ Neue_Paare, $P_2 \in$ Neue_Paare$\}$
    Führe Terminierungstest für alle komponierten Paare durch
    Alte_Paare = Alte_Paare $\cup$ Neue_Paare
    Neue_Paare = Komponierte_Paare - Alte_Paare
  END
END

Der Range eines Ableitungspaares zusammen mit dem dazugehörigen Prädikatnamen bildet eine neue Anfrage in instantiierter Form. Für jede neue Anfrage, die aus den erzeugten Ableitungspaaren gewonnen werden, wird das Verfahren erneut aufgerufen, woraus sich mögliche neue Regelgraphen, Ableitungspaare und Anfragen ergeben. Das Verfahren terminiert, wenn keine weiteren Anfragen mehr erzeugt werden können. Haben alle Ableitungspaare den Terminierungstest erfüllt, so terminiert **P**.

Das Terminierungsverfahren wird durch den Algorithmus 2 beschrieben. Im Algorithmus 2 werden die Ableitungspaare als Paare bezeichnet, $\circ$ stellt die Komposition dar. Die äußere Schleife wird beendet, wenn keine neuen Anfragen mehr erzeugt werden. In der inneren Schleife werden unter Abschluss alle komponierbaren Ableitungspaare erzeugt. Der Abbruch im Falle, wenn ein Ableitungspaar den Terminierungstest nicht erfüllt, ist im Algorithmus aus Gründen der Übersichtlichkeit nicht angegeben.

3 Optimierung der Terminierungsanalyse

In diesem Kapitel werden zwei Möglichkeiten zur Optimierung der Terminierungsanalyse behandelt.

3.1 Ergänzung der Instantiierungsanalyse um ein Top-down-Verfahren

Wie in Abschnitt 2.2 beschrieben, erzeugt die Instantiierungsanalyse für jede Klausel mit n verschiedenen Variablen 2^n Instanzklauseln. Diese Anzahl kann durch Ausnutzung der Information, die durch die Startanfrage gegeben ist, durch ein Top-down-Verfahren reduziert werden. Dieses Verfahren wird anhand des folgenden Programms 2 zur Berechnung von Potenzen beschrieben.

Example 2 Programm zur Berechnung von Potenzen und das teilinstantiertes Programm.

```
nat(0).                                  nat(ai).
nat(s(X)) :-nat(X).                      nat(ai) :- nat(ai).

plus(0,X,X) :- nat(X).                   plus(ai,ai,ai) :- nat(ai).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).        plus(s(X),ai,s(Z)) :- plus(X,ai,Z).

mal(0,X,0).                              mal(ai,ai,ai).
mal(s(X),Y,Z) :- mal(X,Y,W),plus(W,Y,Z). mal(s(X),ai,Z) :- mal(X,ai,W),plus(W,ai,Z).

exp(s(X),0,0).                           exp(ai,ai,ai).
exp(0,s(X),s(0)).                        exp(ai,ai,ai).
exp(s(N),X,Y) :- exp(N,X,Z),mal(Z,X,Y).  exp(ai,ai,Y) :- exp(ai,ai,Z), mal(Z,ai,Y).
```

Sei exp(ai, ai, nai) als Startanfrage gegeben, so hat man die Information, dass die ersten beiden Argumente des `exp`-Prädikats ausreichend instantiiert sind. Betrachtet man die Regel des `exp`-Prädikats, so ergibt sich daraus, dass die Variablen (N, X) an den ersten beiden Argumentpositionen des Regelkopfes ausreichend instantiiert sind. Dies gilt natürlich für die gesamte Regel, und somit können die Variablen N und X in dieser Regel durch **ai** ersetzt werden. Argumente, die schon ausreichend instantiiert sind, werden ebenfalls durch **ai** substituiert. Die resultierende `exp`-Regel sieht dann folgendermaßen aus:

exp(ai, ai, Y) :- exp(ai, ai, Z), mal(Z, ai, Y)

Für diese Regel müssen jetzt statt 16 nur noch vier Instanzklauseln erzeugt werden. Diese Variablensubstitution wird nun mit den Rumpfatomen als Anfragen fortgesetzt, dabei werden die Argumente an den Argumentpositionen, an denen nicht substituiert wurde, als **nai** betrachtet. Beispielsweise gewinnt man aus obiger `exp`-Regel die neue Anfrage `mal`(**nai**, **ai**, **nai**). Substitution innerhalb der mal-Regel ergibt:

mal(s(X), ai, Z) :- mal(X, ai, W), plus(W, ai, Z)

Die Substitution wird für jede Klausel eines Prädikats entsprechend der gegebenen Anfrage durchgeführt. Das Verfahren endet, wenn keine neuen Anfragen mehr erzeugt werden. Als Ergebnis dieses Verfahrens erhält man ein teilinstantiiertes Programm, bei dem einige Argumente durch **ai** substituiert wurden. Beispiel 2 zeigt das teilinstantiierte Programm zur Berechnung von Potenzen nach

Durchführung des Verfahrens. Für dieses teilinstantiierte Programm müssen nur noch 22 Instanzklauseln erzeugt werden, wohingegen es im ursprünglichen Programm 51 Instanzklauseln wären.

3.2 Einschränkung der Anzahl der zu erzeugenden Ableitungspaare

In diesem Abschnitt wird eine Verbesserung der Terminierungsanalyse beschrieben, die durch Einschränkung der Anzahl der Ableitungspaare, die erzeugt werden müssen, erreicht wird.

Definition 11 Ein *Prädikat-Abhängigkeitsgraph* ist ein gerichteter Graph dessen Knoten Prädikate eines Programms darstellen und der für jede Regel $H : -B_1, \ldots, B_n$ und jedes i mit $1 \leq i \leq n$ eine Kante besitzt, die von A zu B_i führt.

Der Graph aus Abbildung 1 besitzt zwei starke Zusammenhangskomponenten. Die erste besteht aus dem Teilgraph, der aus den Knoten a, b, d, e und den dazugehörigen Kanten gebildet wird, die zweite starke Zusammenhangskomponente besteht nur aus dem Knoten c. Befinden sich zwei Knoten in unterschiedlichen starken Zusammenhangskomponenten, zum Beispiel die Knoten b und c aus Abbildung 1, so gibt es keinen Pfad, der vom Knoten b nach c und wieder zurück zu Knoten b führt. Da die Knoten Prädikate darstellen, bedeutet dies, dass, sobald Prädikat c aufgerufen wird, kein Prädikat aus der ersten starken Zusammenhangskomponente mehr erreichbar ist. Somit wird klar, dass es für den Terminierungsnachweis ausreicht, diejenigen Ableitungspaare zu testen, deren Domain und Range innerhalb der selben starken Zusammenhangskomponente liegen.

Definition 12 Ein *Ableitungspaar* heißt *rekursiv*, wenn die Prädikate des Domain und des Range zur selben starken Zusammenhangskomponente des Prädikat-Abhängigkeitsgraphen gehören.

Example 3 Prädikat-Abhängigkeitsgraph von einem einfachen Programm

```
a :- b, c.
b :- d.
c.
d :- b, e.
e :- a.
```

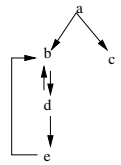


Abbildung 1: Prädikat-Abhängigkeitsgraph

Für den Terminierungsnachweis genügt es, rekursive Ableitungspaare zu erzeugen, was die Anzahl der zu untersuchenden Ableitungspaare reduziert und das Verfahren effizienter macht.

4 Testresultate

Dieser Abschnitt gibt einen Überblick über die getesteten Programm sowie die Effizienz des Terminierungsverfahrens.

In Tabelle 2 sind die Ergebnisse längerer und komplexerer Programme angegeben. `bid` ist Teil eines Bridge-Programms, `plan` und `blocks` sind Programme zum Lösen des Blockwelt-Problems, bei denen durch Backtracking Nichtterminierung auftreten kann. `browse` und `vangelder` sind reine Test beziehungsweise Benchmark-Programme. `mmatrix` ist ein Programm das Matrizenberechnungen durchführt. `color_map` löst das Karten-Färbungsproblem und `credit` ist ein kleines Expertensystem zur Bewertung von Kreditanträgen. Die Zeit ist in Sekunden angegeben. Das Verfahren kommt in allen zum korrekten Ergebnis.

Programm	Anzahl der Fakten	Anzahl der Regeln	Ref.	Startanfrage	Zeit	Ausgabe
bid	24	26	[3]	bid(ai, nai, nai, nai)	5	T
blocks	12	5	[9]	tower(ai, ai, ai, nai)	<1	N
browse	4	25	[3]	main	25	N
color_map	7	6	[1]	test_color(ai, nai)	3	T
credit	33	24	[13]	credit(ai, nai)	3,5	T
mmatrix	7	8	[3]	mmultiply(ai, ai, nai)	<1	T
				trans_m(ai, nai)	<1	N
plan	12	17	[13]	transform(ai, ai, nai)	1,5	N
vangelder	1	10	-	q(ai, ai)	1,5	T

Abbildung 2: Testergebnisse komplexerer Programme.

5 Komplexität und frühere Arbeiten

Die Laufzeit des Algorithmus ist exponentiell abhängig von der maximalen Stelligkeit des Prädikates in dem Programm, von der maximalen Anzahl der unterschiedlichen Variablen in einer Klauseln und von der maximalen Anzahl von Atomen in einer Klausel. Zum Beispiel ist die Zeit der Instantiierungsanalyse, wie sie hier beschrieben wurde, exponentiell in der maximalen Anzahl von unterschiedlichen Variablen in den Klauseln. Wenn wir uns nur für die Terminierung einer Abfrage interessieren, brauchen wir nicht die globale Instantiierungsanalyse durchzuführen. Jedoch sind wir an der Installation interessiert, in der ein Programm gegeben wird, und dann die Terminierung für sämtliche mögliche Abfragen betrachtet wird. Wir erwarten, wenn die maximale Stelligkeit der Prädikate, die maximale Anzahl der unterschiedlichen Variablen in den Klauseln und die maximale Anzahl der Atome in den Klausel durch eine relativ kleine Zahl begrenzt ist, dass der Algorithmus nahezu linear in der Länge des Programms arbeitet.

Die Terminierungsanalyse von PROLOG Programmen wurde in den letzten Jahren intensiver erforscht. Systeme für eine automatische Terminierungsanalyse werden z.B. in [2], [10] und [14] vorgeschlagen. Methoden und Techniken für die Prüfung der Terminierung werden in [1], [5], [6] und [15] behandelt. In den früheren Arbeiten wurden verschiedene Arten von Normen vorgeschlagen. Dieses System benutzt eine allgemeinere Norm als frühere und aus diesem Grund hoffen wir, dass es Programme analysieren kann, die früher nicht behandelbar waren.

Die Ableitungspaarmethode, die in diesem Papier verwendet wird, ist eine optimierte Erweiterung zu den allgemeinen Logikprogrammen, die von Sagiv ([12]) eingeführt werden. Seine Methode wird für DATALOG-Programme mit endlosem EDB verwendet. Es benutzt jedoch die Annahme, dass jede Variable im Kopf einer Klausel auch im Rumpf erscheint. Die Anwendung auf Logikprogramme dieses Algorithmus basiert auf der Beobachtung in [11], dass Logikprogramme in DATALOG-Programme umgewandelt werden können, so dass, wenn die Abfrage des DATALOG-Programms terminiert, die Abfrage zum PROLOG-Programm es auch tut. Die schwere Einschrnkung kann hinsichtlich des Auftretens von Variablen im Kopf einer Klausel wurde in diesem Artikel durch die Instantiierungsanalyse überwunden, entsprechend den Ideen der abstrakten Interpretation von [4].

6 Zusammenfassung

Das Ziel war die Erstellung eines Systems zur statischen Codeanalyse von PROLOG-Programmen zur Unterstützung des *IDTS*-Systems. Dabei wurden verschiedene einfache Fehlerklassen erkannt.

Den Schwerpunkt bildete die Terminierungsanalyse sowie die Instantiierungsanalyse. Die Instantiierungsanalyse basiert auf einem Bottom-up-Verfahren, bei dem aus einer gegebenen Anfrage in instantiierter Form alle möglichen Instanzen erzeugt werden. Durch ein Top-down-Verfahren kann die Instantiierungsanalyse bezüglich der Effizienz verbessert werden. Anhand eines Terminierungs-

tests, der für alle Ableitungspaare durchgeführt wird, kann entschieden werden, ob ein gegebenes Programm terminiert oder ob möglicherweise Nichtterminierung vorliegt. Durch die Einschränkung der zu erzeugenden Ableitungspaare auf rekursive Ableitungspaare kann ein Effizienzgewinn der Terminierungsanalyse erreicht werden. Um das zu beweisen und außerdem zu zeigen, welche terminierenden Programme nicht erkannt werden und einen Vergleich mit anderen existierenden Ansätzen ziehen zu können, ist noch ein ausführlicherer Testprozess durchzuführen.

Literatur

[1] Apt, K. R., Pedreschi, D. : *Modular Termination Proofs for Logic and Pure Proog Programs,* In Advances in Logic Programming Theory, 183-229, Oxford University Press, 1994.

[2] Bezem, M.: *Strong termination of logic programs* J. Logic Programming, 15:79-97, 1993

[3] Bueno, F. , Garcia de la Banda M. , Hermenegildo M. : *Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization* International Symposium on Logic Programming, 320-336, MIT Press, 1994.

[4] Cousot, P., Cousot, R.: *Abstract Interpretation and Application to Logic Programs* J. Logic Programming, 13:103-179, 1992

[5] De Schreye, D. Decorte, S.:*Termination of Logic Programs: the Never-ending Story* J. Logic Programming, 19/20:199-260, 1994

[6] Falaschi, M., Levi, G., C. Palamidessi:*Declarative modeling of the operational behaviour of logic languages* Theoretical Computer Science, 69:289-318, 1989

[7] Kókai, G.: *Debugging und maschinelles Lernen von PROLOG-Programmen*; Ph.D. Dissertation am IMMD 2, Friedrich Alexander Universität Erlangen-Nürnberg, 1998.

[8] Lindenstrauss, N.; Sagiv, Y.: *Automatic Termination Analysis of Logic Programs*; Tech. Report, Dept. of Computer Science, Hebrew University Jerusalem, Israel, 1997, http://www.cs.huji.ac.il/ naomil/ .

[9] Nilsson, N. J.: *Principles of Artificial Intelligence* Tioga Publishing Company, 1980.

[10] Plümer, L: *Termination Proofs for Logic Programs, Springer Verlag*, LNAI 446, 1990.

[11] Ramakrishnan, R., Bancilhon, F., Silberschatz, A.,: *Safety of Recursive Horn Clauses with Infinite relation* in. the Proc. of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems 1987

[12] Sagiv, Y. : *A Termination Test for Logic Programs*; International Logic Programming Symposium, MIT Press, 1991.

[13] Sterling, L., Shapiro, E.: *Prolog - Fortgeschrittene Programmiertechniken* Bonn [u.a.] : Addison-Wesley, 1988.

[14] Ullman, J. D., Van Gelder, A.:*Efficient test for top-down termination of logic rules* JACM 35:2 (1988), 345-373

[15] Verschaetse, K. and De Schreye, D.: *Deriving Termination Proofs for Logic Programs, using Abstract Procedures* In proc of the 8th ICLP, 301-315, 1991

Slicing zur Fehlersuche in (Constraint-) Logikprogrammen

Stefan Kral, Fred Mesnard, Ulrich Neumerkel
Technische Universität Wien, Université de la Réunion

Zusammenfassung

Wir stellen drei Verfahren zur Fehlersuche in Logikprogrammen vor, die durch Slicing-Techniken den für ein unerwartetes Phänomen verantwortlichen Bereich einschränken. Als Erklärung des Fehlers wird ein Programmfragment (slice) erzeugt, das modifiziert werden *muß*, um den Fehler zu beheben. Im Gegensatz zu den üblichen Techniken zur Fehlersuche (Tracing und deklaratives Debugging [6]) benötigt unser Ansatz keinerlei Interaktion mit dem Benutzer. Er ist daher besonders für Anfänger geeignet und wurde in eine Programmierumgebung integriert. Da unser Ansatz ohne den üblichen Begriff des Beweisbaumes auskommt, eignet er sich auch zur Fehlersuche von Constraint-Programmen. Insbesonders kann auch die Ursache für das Scheitern eines Zieles in Constraint-Programmen lokalisiert werden.

Beim Erlernen einer Programmiersprache nimmt die Fehlersuche einen prominenten Platz ein. Durch sie kann im Idealfall das Verständnis der Eigenschaften einer Programmiersprache vertieft werden. Bei Logik-Programmiersprachen wie Prolog sind die gegenwärtigen Techniken zur Fehlersuche jedoch kaum geeignet, das Sprachverständnis zu fördern. Die üblichen Techniken beruhen darauf, den Programmablauf in einer —gelegentlich etwas gerafften, aber letztlich noch immer prozeduralen Form— darzustellen. Derartige Ansätze eignen sich kaum, ein vertiefendes Verständnis der deklarativen Eigenschaften zu fördern.

Das von Shapiro entwickelte Algorithmic Debugging [6] versucht, die Fehlerursache unter Zuhilfenahme eines Orakels einzugrenzen. Dabei nimmt meist der Benutzer die Rolle des Orakels ein, das entscheidet, ob Ziele im Ableitungsbaum wahr sein sollen oder nicht. Die Richtigkeit der Diagnose hängt also hier von Richtigkeit der Benutzerantworten ab. Gerade bei Anfängern ist es naheliegend, daß nicht alle Antworten für das Orakel richtig sein werden. Zur eigentlichen Fehlersuche gesellt sich so die Fehlersuche im Diagnoseprozess. Zudem sind viele der zu beantwortenden Fragen sehr komplex und schwer zu lesen. Bei Constraint-Systemen, die keine vollständigen Gleichungslöser besitzen, ist eine lesbare Darstellung ausgeschlossen. Im Gegensatz dazu benötigen unsere Verfahren keinerlei zusätzliche Benutzereingabe. Die erzeugten Erklärungen werden direkt aus dem gegebenen Programm und einer Anfrage erzeugt.

Die drei vorgestellten Verfahren basieren auf einigen (informellen) Lesarten [2], die über die allgemein übliche Trennung zwischen deklarativer und prozeduraler Sichtweise hinausgehen. Insbesonders wurde die deklarative Lesart verfeinert, um ihre analytische Erklärungskraft zu steigern. Die Grundidee beruht darauf, nur ein gewisses Fragment eines Programms auf einmal zu betrachten. Gegenwärtig werden solche Programmfragmente durch zwei zueinander komplementäre Transformationen erhalten: durch Generalisierung und Spezialisierung. Bei der Generalisierung werden Klauseln durch Wegstreichen eines Ziels verallgemeinert. Bei der Spezialisierung kommt ein neues Ziel zu einer Klausel hinzu. Ist dieses Ziel das Ziel false/0, so kann bei deklarativen Lesarten die gesamte Klausel, bei prozeduralen Lesarten, alle

Ziele hinter dem neu eingefügten gelöscht werden. Ein besonderer didaktischer Vorteil unserer Lesarten liegt darin, daß sie sich verhältnismäßig einfach erklären lassen. Das Zudecken läßt sich etwa auch von Hand an der Tafel ohne weitere Hilfsmittel veranschaulichen.

Unsere Verfahren eignen sich für pure Logik- und constraintlogische Programme. Bei der Verwendung von Negation sind unsere Verfahren nur mehr eingeschränkt einsetzbar. Prolog-Programme mit Seiteneffekten und anderen außerlogischen Konstrukten können nicht analysiert werden. Analog zu Shapiros drei Diagnosen, erklären auch wir die Fehlerfälle Unerwartetes Scheitern, Unerwartete Lösung und Nichttermination.

Unerwartetes Scheitern

Scheitert eine Anfrage unerwarteterweise, so ist das dazugehörige Programm zu speziell definiert. Scheitert nun auch eine Verallgemeinerung des Programms, muß sich bereits in dieser Verallgemeinerung ein Fehler befinden. Wir betrachten derzeit nur Verallgemeinerungen, die durch das Löschen von Zielen in Regeln entstehen, da bei komplexeren Verallgemeinerungen die textliche Entsprechung nicht mehr offensichtlich ist. Unser Verfahren versucht, minimale scheiternde Fragmente zu finden, bei denen durch jede weitere Verallgemeinerung die Anfrage erfüllt ist.

← bruder_von(B, P).	
	männlich(franz_I).
bruder_von(B, P) ←	männlich(joseph_II).
* ~~dif(B,P)~~,	männlich(leopold_II).
* ~~männlich(B)~~,	
* ~~kind_von(B,V)~~,	weiblich(maria_th).
männlich(V),	
* ~~kind_von(P,V)~~,	~~kind_von(joseph_II, maria_th).~~
* ~~kind_von(B,M)~~,	~~kind_von(joseph_II, franz_I).~~
* ~~kind_von(P,M)~~,	~~kind_von(leopold_II, maria_th).~~
weiblich(V).	~~kind_von(leopold_II, franz_I).~~

Das erzeugte Programmfragment reduziert das Programm auf einen kleinen Bereich, in dem sich zumindest ein Fehler befinden *muß*. Im angeführten Beispiel könnten neben der offenbar fehlerhaften Definition von bruder_von/2 auch die Prädikate männlich/1 und weiblich/1 —bei entsprechender Interpretation der Prädikatsnamen— zu speziell definiert sein. Auch die Anfrage selbst könnte der eigentliche Fehler sein.

In rekursiven Programmen terminieren viele durch * erhaltene Verallgemeinerungen nicht. Um solche Fragmente möglichst zu umgehen, verwenden wir eine constraintbasierte Terminationsinferenz [1], die im Gegensatz zu den üblichen Terminationsbeweisern, ganze Klassen von terminierenden Anfragen auf einmal bestimmt. Zudem versuchen wir mit die Anzahl der tatsächlich probeweise auszuführenden Fragmente zu minimieren.

Unerwartete Lösungen

Ist eine Anfrage, die keine Lösung finden sollte, dennoch erfüllt, muß die Programmdefinition zu allgemein sein. Das Programm wird gegenwärtig nur durch Einfügen des Ziels false/0 spezialisiert. Diese Ziele erlauben das Wegstreichen ganzer Klauseln. Detaillierte Erklärungen sind textlich nur wesentlich komplexer darstellbar.

$\not\leftarrow$ kind_von(K, K).

~~kind_von(joseph_II, maria_theresia) ← **false**~~.
~~kind_von(joseph_II, franz_I) ← **false**~~.
kind_von(leopold_II, leopold_II).
~~kind_von(leopold_II, franz_I) ← **false**~~.

Nichttermination

Auch wenn Termination zu den prozeduralen Eigenschaften eines Logik-Programms zählt, lassen auch hier sich analog Programmfragmente erzeugt, die im Falle der existentiellen Nichttermination des Programms bereits diese Eigenschaft bedingen. Dazu werden an Programmpunkten Ziele false eingesetzt und wiederum minimale Fragmente erzeugt [5].

← vorfahre_von(Vorfahre, Nachfahre), **false**.

~~vorfahre_von(Vorfahre, Nachfahre) ← **false**,~~
~~kind_von(Nachfahre, Vorfahre)~~.
vorfahre_von(Vorfahre, Nachfahre) ←
vorfahre_von(Person, Nachfahre), **false**,
~~kind_von(Person, Vorfahre)~~.

~~kind_von(joseph_II, maria_theresia) ← **false**~~.
~~kind_von(joseph_II, franz_I) ← **false**~~.
~~kind_von(leopold_II, leopold_II) ← **false**~~.
~~kind_von(leopold_II, franz_I) ← **false**~~.

Literatur

[1] F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs: JICSLP 1996 7–21, 1996. http://www.complang.tuwien.ac.at/ulrich/cti/

[2] U. Neumerkel. Mathematische Logik und logikorientierte Programmierung, Skriptum zur Laborübung, 1993-1999.

[3] U. Neumerkel. Teaching Prolog and CLP. Tutorial. PAP Paris, 1995 und ICLP Leuven, 1997.

[4] U. Neumerkel. GUPU: A Prolog course environment and its programming methodology. Proc. of the Poster Session at JICSLP (Fuchs, Geske Eds.), GMD-Studien Nr. 296, 1996 Bonn.

[5] U. Neumerkel, F. Mesnard. Localizing and explaining reasons for non-terminating logic programs with failure-slices. PPDP'99, LNCS 1702, pp. 328-341, 1999.

[6] E. Y. Shapiro. Algorithmic Program Diagnosis. POPL 1982, 299-308.