

19th Workshop on (Constraint) Logic Programming

Armin Wolf, Thom Frühwirth, Marc Meister (eds.)
University of Ulm



Preface

This volume contains the refereed and accepted papers and system descriptions presented at the 19th Workshop on (Constraint) Logic Programming W(C)LP 2005 held in Ulm, Germany, from February 21 to 23, 2005.

This year marks the 40th anniversary of Alan Robinson's seminal paper on "A Machine-Oriented Logic Based on the Resolution Principle" that forms a basis for automated deduction and in particular logic programming, that started some three decades ago with the work of Robert Kowalski and Alain Colmerauer.

The workshop on (constraint) logic programming is the annual meeting of the Society of Logic Programming (GLP e.V.) and brings together researchers interested in logic programming, constraint programming, and related areas like databases and artificial intelligence. Previous workshops have been held in Germany, Austria, and Switzerland. The technical program of the workshop included an invited talk, presentations of refereed and accepted papers and system descriptions, as well as non-refereed system demonstrations and poster presentations. The contributions in this volume were reviewed by at least two referees.

We would like to thank all authors who submitted papers and all workshop participants for their fruitful discussions. We are grateful to the members of the program committee and of the local organisation as well as to the referees. We would like to express our thanks to the Society of Logic Programming (GLP e.V.) for handling finances and the University of Ulm for hosting the workshop.

February 2005

**Armin Wolf, Thom Frühwirth
Marc Meister**

Program Chair

Armin Wolf Fraunhofer FIRST, Berlin, Germany

Local Chair

Thom Frühwirth University of Ulm, Germany

Local Organisation

Marc Meister University of Ulm, Germany

Program Committee

Slim Abdennadher	German University Cairo, Egypt
Christoph Beierle	FernUniversität Hagen, Germany
Francois Bry	Ludwig Maximilians University, Munich, Germany
Henning Christiansen	Roskilde University, Denmark
Thom Frühwirth	University of Ulm, Germany
Ulrich Geske	Fraunhofer FIRST, Berlin, Germany
Michael Hanus	University of Kiel, Germany
Petra Hofstedt	Technical University Berlin, Germany
Steffen Hölldobler	Technical University Dresden, Germany
Christian Holzbaur	Vienna, Austria
Ulrich John	DaimlerChrysler Research, Berlin, Germany
Ulrich Neumerkel	Technical University Vienna, Austria
Alessandra Raffaeta	Università Ca' Foscari, Venezia, Italy
Georg Ringwelski	4C, Cork, Ireland
Hans Schlenker	Fraunhofer FIRST, Berlin, Germany
Tom Schrijvers	Catholic University Leuven, Belgium
Dietmar Seipel	University of Würzburg, Germany
Michael Thielscher	Technical University Dresden, Germany
Herbert Wicklicky	Imperial College, London, Great Britain
Armin Wolf	Fraunhofer FIRST, Berlin, Germany

Table of Contents

Full Papers

XML Transformations based on Logic Programming	5
<i>Dietmar Seipel and Klaus Praetor</i>	
A PROLOG Tool for Slicing Source Code	17
<i>Marbod Hopfner, Dietmar Seipel, and Joachim Baumeister</i>	
Combining Domain Splitting with Network Decomposition for Application in Model-Based Engineering	29
<i>Rüdiger Lunde</i>	
Towards an Object-Oriented Modeling of Constraint Problems	41
<i>Armin Wolf, Henry Müller, and Matthias Hoche</i>	
Expressing Interaction in Combinatorial Auction through Social Integrity Constraints	53
<i>Marco Alberti, Federico Chesani, Alessio Guerri, Marco Gavanelli, Evelina Lamma, Paola Mello, Michela Milano, Paolo Torroni</i>	
Level Mapping Characterizations of Selector Generated Models for Logic Programs	65
<i>Pascal Hitzler and Sibylle Schwarz</i>	
Truth and knowledge fixpoint semantics for many-valued logic programming	76
<i>Zoran Majkic</i>	
Impact- and Cost-Oriented Propagator Scheduling for Faster Constraint Propagation	88
<i>Georg Ringwelski and Matthias Hoche</i>	
Static and dynamic variable sorting strategies for backtracking-based search algorithms	99
<i>Henry Müller</i>	
The CHR-based Implementation of a System for Generation and Confirmation of Hypotheses	111
<i>M.Alberti, F.Chesani, M.Gavanelli, E.Lamma, P.Mello, P.Torroni</i>	
Guard Simplification in CHR programs	123
<i>Jon Sneyers, Tom Schrijvers, and Bart Demoen</i>	
Analysing the CHR Implementation of Union-Find	135
<i>Tom Schrijvers and Thom Frühwirth</i>	

DB_CSPA Framework and Algorithms for Applying Constraint Solving within Relational Databases	147
<i>Chuang Liu, Ian Foster</i>	
System Descriptions	
Meta-S – Combining Solver Cooperation and Programming Languages . . .	159
<i>Stephan Frank, Petra Hofstedt, and Dirk Reckman</i>	
Cmodels for Tight Disjunctive Logic Programs	163
<i>Yuliya Lierler</i>	

XML Transformations Based on Logic Programming

Dietmar Seipel¹ and Klaus Prator²

¹ University of Wurzburg, Institute for Computer Science
Am Hubland, D – 97074 Wurzburg, Germany
seipel@informatik.uni-wuerzburg.de

² Berlin–Brandenburg Academy of Sciences and the Humanities
Jagerstr. 22–23, D – 10117 Berlin, Germany
praetor@bbaw.de

Abstract. Critical or scientific editions are a promising field for the application of declarative programming, which can facilitate the parsing and the markup of texts, and the transformation of XML documents.

We have used a logic programming–based approach for the production of critical editions: In particular, we propose a transformation of XML documents based on a compact and intuitive substitution formalism. Moreover, we have developed a new XML update language FNUPDATE for adding navigation facilities.

We have implemented a transformation tool in SWI–PROLOG, which integrates and interleaves PROLOG’s well-known definite clause grammars and the new substitution formalism.

Keywords. PROLOG, grammars, critical editions, XSLT

1 Introduction

The mainstream approach to XML processing is based on concepts such as the *Extensible Style Sheet Language for Transformations* (XSLT), the XML query language XQuery, and the underlying path language XPATH. Originally, XSLT was meant as an advanced sort of Cascading Style Sheets (CSS), but then it developed into a tool for the transformation of markup – the real formatting tool was delivered afterwards as XSL Formatting Objects (XSL–FO).

The way XSLT works resembles remarkably the way of PROLOG. The transformation is done by traversal of a tree, and by testing the matching of nodes against patterns in the style sheet. Comparing PROLOG and XSLT, of course the mainstream character and the extent of support are arguments in favour of XSLT, but there are also some drawbacks [9]. Firstly, XSLT is not a universal programming language, which means that not all problems are solvable within this framework. Secondly, XSLT is not designed from ground up as a declarative language. In simple examples this is not easily visible, but as applications get more complex you see the barely disguised imperative background shining through in control structures (xsl:if, xsl:for-each etc.) and pattern matching. A detailed comparison of XSLT and PROLOG can be found on the WWW pages of SWI–PROLOG [13].

Logic programming is very well-suited for natural language processing (NLP) [8] and for text processing in general. In particular, *definite clause grammars* (DCGs) are a powerful declarative approach to writing parsers [11]. According to Richard O’Keefe the great advantage of DCGs is that it is important *not* to think about the details of the translation: Any time you have stereotyped code, using a translator to automatically supply the fixed connection patterns means that the code is dramatically *shorter* and *clearer* than it would otherwise have been, because you are hiding the uninformative parts and revealing the informative parts. Thus, the code is *easier* to write and to read, and it has *fewer* mistakes, because the pattern is established once in the translator and thereafter tirelessly applied with machine consistency. DCGs are just one example for a more general idea (cf. [10, 14]): define a *little language* embedded in PROLOG syntax, write an interpreter for that language, and devise a translator which turns constructs of that language into the (useful, non-book-keeping) code that the interpreter *would* have executed for those constructs.

Prätor [15] has shown that techniques of logic programming are especially apt for the structures and the specific problems of critical editions. DCGs and a graph traversal had been used for transforming between the different layers of XML documents in the pilot project Jean Paul. In the present paper we extend this approach by proposing a more compact and intuitive substitution formalism, which can be interleaved with PROLOG’s built-in DCG formalism, and we have developed a new XML update language FNUPDATE for adding navigation facilities.

The rest of this paper is organized as follows: In Section 2 we review some characteristics of critical editions in general, and we give an introduction to the different types of XML structures used within the pilot project Jean Paul. In Section 3 we show how XML documents can be represented in PROLOG, and we present transformations based on sequential scans and on tree traversals, respectively. Finally, in Section 4 we use an XML update language FNUPDATE, that we have implemented in PROLOG, for adding topology and navigation facilities to XML documents.

2 Critical Editions

Critical editions are notoriously difficult sorts of text, as they form not one linear text but rather a complex of different *variants* and readings of a text, which are to be handled within a critical apparatus (think of a set of foot- or endnotes). They are enriched by *commentaries* with historical and philological information and made accessible by indices and directories. Other peculiarities are the mostly large extent of the editions and the fact that the period of production as well as of usage is very long reaching from decades to centuries. Especially in electronic editions it is necessary to care for sustainable availability and usability.

Some problems are due to the print form. Lack of space leads to elaborate systems of abbreviations and to steady considerations, which material and information can still be incorporated and which one has to be left out. The print can hardly handle the inherent nonlinearity of the documents, and of course there is no thought of adaptation to

different situations of usage. Compared to the print form, electronic editions show some advantages, which result from the advanced ways of navigation and retrieval and from the possibility to provide different types of output. The larger storage capacities provide room for additional information regarding, e.g., involved persons or historical circumstances. Hypertext capacities facilitate the constitution of temporal, spatial or thematic relations. Different editions may be nested and entirely new information spaces may be created in this way.

2.1 The Pilot Project Jean Paul

The Academy of Sciences and Humanities of Berlin–Brandenburg is the home of many edition projects of all ages – editions in a broader sense including aside from work editions also source editions and dictionaries. Most of these editions are still focussed on the print form, but we are working on the migration to genuine electronic editions.

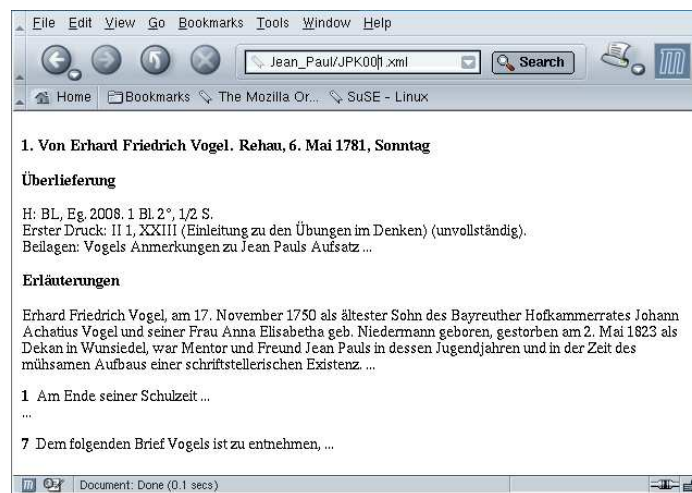


Fig. 1. Comment to a Letter of the Jean Paul Edition

Jean Paul has been a frequently read and appreciated author of novels in the times of Goethe. The Jean Paul Edition is organized in 6 volumes, each of which consists of about 150 letters to Jean Paul. The two aims of the pilot project were to produce an electronic equivalent of the just finished first volume, and to demonstrate some additional features of an electronic edition in an enriched selection. Aside from the usual contents it provides a simple full text search, and – most importantly – three paths to the letters via correspondent, year or place of the respective letter.

In this paper we focus on the *comments* to the letters. For each letter there exists one comment; the comment to the first letter of the first volume is shown in Figure 1.

The source texts of the edition were originally produced with Microsoft Word, which is not a desirable choice as an editor's tool at all, but just legacy. They were subsequently translated to a *migration layer* in HTML by a commercial tool. In the Sections 3 and 4 we will show transformations from this migration layer layer to the *archive layer* of the edition. The timeframe, in which a critical edition should be usable, is very long. Different users or situations may demand different presentations, which can then be produced from the archived documents.

2.2 Transformation Techniques

Two orthogonal concepts are used for transforming documents, DCGs and substitution rules, which can also be mixed by calling them from each other:

- We use *DCGs* for *grouping* elements that are on the same level in the source document to form complex, nested structures. They are well-suited for parsing a sequence of items by a sequential scan.
- We use *substitution rules* for transforming a complex, nested document. They *traverse* a tree-shaped XML document recursively: first the subelements of an element are transformed, then the resulting element is also transformed.

We propose a new, compact and intuitive substitution formalism, which can be interleaved with PROLOG's well-known built-in DCG formalism,

3 Transformation of XML Documents

The transformation of XML documents is a perfect task for declarative programming. The first success is to see how seamlessly an XML document converts into a genuine PROLOG structure. Subsequently, we show how to strip off unnecessary graphical markup and to transform layout markup which transports meaning into explicit XML tagging. This transforms from the migration to the archive layer.

3.1 XML Documents in PROLOG

Essentially, an XML or HTML element can be represented as a nested term structure containing the tag name, the attribute list, and the content (possibly also a list or empty). There are some libraries available to support this sort of transformation in both directions. As far as we know the first was the HTML tool *PiLLoW* for CIAO PROLOG [4], which also contains some CGI- and HTTP-support. A similar library for SGML exists for SWI-PROLOG.

The tool *FNQUERY* [16], which was implemented in SWI-PROLOG, uses a slightly abbreviated – but similar – term structure. E.g., an XML element


```

<p style="margin-top:0;margin-bottom:0;">
  <font face="Times New Roman" size="3">
    <em>1,</em> 18-19
    <strong>Gehen </strong> bis
    <strong>sind] </strong>
    Dem folgenden Brief Vogels ...
  </font>
</p>

```

is represented as

```

p:[style:'margin-top:0;margin-bottom:0;']: [
  font:[face:'Times New Roman', size:3]: [
    em:['1,'], '18-19',
    strong:['Gehen '], bis, strong:['sind] '],
    'Dem folgenden Brief Vogels ...' ] ]

```

called FN triple, where the attribute list can be omitted, if it is empty. Here, a triple is represented by means of operator definition in the form $T:As:C$, where As is an associative list of attribute/value pairs in the form $a:v$. Text elements (such as '18-19') are simply represented as PROLOG atoms. The PROLOG library FNQUERY goes beyond the pure transformation by providing additional means to select and handle substructures, comparable to XQuery or F-logic. E.g., if X_{ml} is the FN triple above, then the call $X := X_{ml}^{font@face}$ selects the value 'Times New Roman' of the face-attribute of the nested font-element and assigns it to X . A detailed description of the selection features of FNQUERY can be found in [16].

3.2 Complex Transformations by Traversal of Documents

The predicate `transform_fn_item/2` transforms an FN triple `Item_1` (e.g., from the migration layer) into another FN triple `Item_2` (e.g., from the archive layer) based on facts for the predicate `---->/2`:

```

transform_fn_item(T:As:Es, Item) :-
  maplist( transform_fn_item,
    Es, Es_2 ),
  ( T:As:Es_2 ----> Item
  ; As = [],
    T:Es_2 ----> Item ).
transform_fn_item(declare(_X), '').
transform_fn_item(Item, Item).

```

This new *substitution formalism* has been implemented within the subsystem FN-TRANSFORM of FNQUERY. It generalizes XSLT style sheets, which can be seen as a collection of transformation rules. These template rules in XSLT have the form of

```
<xsl:template match=Pattern>
  Template</xsl:template>
```

Pattern is an XPATH expression, which matches with some nodes of the document tree, and Template is the content which is to be inserted at this location. This content may contain the possibly recursive application of the same or other template rules. For doing this in PROLOG, we use substitution rules of the form

```
Item_1 ---> Item_2 :- Condition.
```

If a term matches the pattern Item_1 (and of course all capacities of the PROLOG unification can be used to do this matching) and the call of Condition (which can be a standard PROLOG goal) succeeds, then the pattern will be replaced by the template Item_2.

3.3 From the Migration Layer to the Archive Layer

On the migration layer the text is represented as a *flat* sequence of paragraph elements, as shown in the previous subsection. The following *substitution rules* transform this sequence to a *nested* structure.

```
html:_:Es ---> Es.
head:_: _ ---> ' ' :- assert(commentHead).
body:_:Es_1 ---> comment:Es_2 :-
    jean_paul_comment(comment:Es_2, Es_1, []).

p:As:Es ---> signedp:Es :-
    right := As^align.
p:_:Es ---> notep:Es.

font:As:Es ---> lat:Es :-
    'small-caps' := As^'font-variant'.
font:_: [X] ---> X.
em:_:Es ---> commentHead:Es :- retract(commentHead).
em:_:Es ---> page:Es.

strong:_:Es ---> lemma:Es.
u:_:Es ---> spaced:Es.

T:As:Es ---> T:As:Es.
```

The body of the HTML document is transformed into a `comment`-element, and the first `em`-element in the body becomes the header of this comment; this is achieved by the assertion of `commentHead`. Subsequent `em`-elements are transformed to `page`-elements. The following text is the result of the transformation. It is archived, and it

serves as the basis for the production of different presentation layers. In the archive layer the comment consists of a header, which is a `commentHead`-element, and two blocks, which are given by `ednote`-elements with a `type`-attribute having the value `Überlieferung` and `Erläuterungen`, respectively.

```
<comment>
  <commentHead>1. Von Erhard Friedrich Vogel.
    Rehau, 6. Mai 1781, Sonntag
  </commentHead>
  <ednote type="Überlieferung">
    <notep>H: BL, Eg. 2008. 1 Bl. 2ř, 1/2 S.</notep>
    ...
  </ednote>
  <ednote type="Erläuterungen">
    ...
    <notep>
      <page>1,</page> 18-19
      <lemma>Gehen</lemma> bis <lemma>sind</lemma>
      Dem folgenden Brief Vogels ist zu entnehmen, ...
    </notep>
  </ednote>
</comment>
```

The grouping of a comment into a header and two blocks – in the third substitution rule above – has been achieved using the predicate `jean_paul_comment/3`, which is defined by the following DCG rules:

```
jean_paul_comment (comment:Es) -->
  header(H),
  block('Überlieferung', X),
  block('Erläuterungen', Y),
  { append(H, [X, Y], Es) }.

header([notep:[]:[commentHead:Es_1]|Es_2]) -->
  sequence_of_notep([notep:[]:[page:_:Es_1]|Es_2]).

block(Type, ednote:[type:Type]:Seq) -->
  sequence_of_notep([notep:_: [Type]|Seq]).

sequence_of_notep([notep:As:Es]) -->
  [notep:As:Es].
sequence_of_notep([notep:As:Es|Seq]) -->
  [notep:As:Es],
  sequence_of_notep(Seq).
```

According to the DCG rules, the header can consist of several `notep`-elements; the first of these elements is derived from an FN triple `notep:[]:[page:_:Es_1]` that

is transformed to another FN triple of the form `notep:[]:[commentHead:Es_1]`. Using further DCG rules the `notep`-elements of the archive layer can be transformed to `note`-elements, such as the following:

```
<note id="7">
  <remark>
    <position page="1" line="18-19"/>
    <lemma>Gehen</lemma> <lemma>sind</lemma>
  </remark>
  Dem folgenden Brief Vogels ist zu entnehmen, ...
</note>
```

The archive layer preserves all information, which may be helpful in some case, and omits all presentation specific features.

4 Update of XML Documents

In this section we investigate transformations that are applied to the archive layer for producing a presentation layer or a presentation layer with *navigation utilities*, respectively. The transformations are based on the XML update language FNUPDATE, which we have implemented in PROLOG within FNQUERY.

4.1 Pruning of XML Documents

Using FNUPDATE we delete the `remark`-elements within the `ednote`-elements for Erläuterungen, since they shall not be displayed at the presentation layer:

```
xml_extract_notes(Xml_1, Xml_2) :-
  Xml_2 := Xml_1 <-> [
    ^ednote::[@type='Erläuterungen']
    ^note^remark ].
```

In FNUPDATE an XML element given by an FN triple `Xml_1` can be pruned by deleting certain subelements. `<->` is a binary operator, which indicates that within the following pair of brackets it will be specified by a path expression which subelements should be deleted. In the example, certain `remark`-elements are deleted; the condition `@type='Erläuterungen'` assures that only `ednote`-elements with the value `Erläuterungen` in their `type`-attribute are affected.

The PROLOG predicate `xml_extract_notes/2` is applied to a `comment`-element `Xml_1`, and thus we don't have `comment` in the selection path. The result `Xml_2`, which forms the presentation layer, is also a `comment`-element; compared to `Xml_1` the `note`-elements are simplified; in our example we obtain

```
<note id="7">
  Dem folgenden Brief Vogels ist zu entnehmen, ...
</note>
```

as a simplified subelement of the presentation layer structure.

4.2 Topology and Navigation

The presentation layer is not meant as the graphical interface, which is primarily produced by the style sheet and the browser, but as an organisation of the content with regard to different user interests.

A very important task in this context is navigation. We suggest to distinguish between topology and navigation with reference to a document. *Topology* refers to the potential connections of document elements, whereas *navigation* means the realised and used connection paths. In accordance with this convention, providing the topology would be part of the archive layer, whereas the presentation has to care of the navigation. While topology is a matter of conceptual and structural relations, navigation has also to take into account technical and aesthetic aspects of the realisation. Other user interests demand different ways of access and navigation, but these should be generated from the one archive layer, or better: the archive layer should be designed in a way that multiple navigations can be produced with little effort.

The information from the document, even in its tagged archive format, may not be enough to produce the navigation. In such case we need meta information, incorporated into the document (e.g., as RDF) or kept in a separate file or database. Here again PROLOG is a good choice to handle and inference these structures. The presentation layer may be seen as a virtual layer on top of documents and meta information. Derived from such meta information are both the navigation headlines of the letters and the different directories of persons, years and places, which provide access to the content, cf. Figure 2.

Another navigation task is the creation of indices, e.g. for persons mentioned in the document. For this purpose we would insert a durable tag that marks the person names, like `<person>Name</person>`, on the archive layer. This provides the basis for the insertion of the appropriate linking structures on the presentation layer.

Creating an Index for an XML Document

The following predicate adds anchors to a given word within an FN term using FNUPDATE: If a given word *Name* occurs in the content list of a *note*-element, then the word is replaced by an element `<person>Name</person>`:

```
xml_enrich_with_anchors(Name, Xml_1, Xml_2) :-
  Xml_2 := Xml_1 <-+> [
```



Fig. 2. Navigation Utilities

```

^ednote^note^child::'*'::[
  ^self::'*'=N, name_contains(Name, N) ]
^person:[N] ].

```

This is initiated by the binary replacement operator $\langle -+ \rangle / 2$ in the path expression. E.g., if we want to enrich an XML document containing the element

```

<note id="7">
  Dem folgenden Brief Vogels ...
</note>

```

with anchors to words containing *Vogel*, then we assume that the content of this element is split into tokens, i.e., that we have the following FN triple:

```

note:[id:7]:[
  'Dem', folgenden, 'Brief', 'Vogels', ... ].

```

Using the FNUPDATE statement above we obtain the following:

```

<note id="7">
  Dem folgenden Brief <person>Vogels</person> ...
</note>

```

Finally, we create the index structure from the enriched document. For each inserted anchor, the following predicate extracts a corresponding reference of the form `Name` from the updated document, where the ID *N* is generated using the call `get_num(anchor_id, N)`:

```

xml_extract_references(Xml, index:Anchors) :-
  findall( a:[href:R]:[Name],
    ( [Name] := Xml^^person^content::*,
      get_num(anchor_id, N),
      File := Xml@file,
      concat([File, '#', N], R) ),
    Anchors ).

```

Observe, that the selection `Xml^^person` yields a complete person-element, such as `person:[]:['Vogels']`. The subsequent expression `^content::*` selects its content list. Thus, in the rule above `Name` would be assigned to `'Vogels'`.

Given a presentation layer file `letter_17` containing the enriched note-element above, we obtain an index with a reference to `letter_17`:

```

<index>
  <a href="letter_17#1">'Vogels'</a>
  ...
</index>

```

This index will be stored in a separate XML file.

5 Final Remarks

The introduction of the substitution predicate `--->/2` extends the ideas of the DCG operator `-->/2`, which is usually used for transformations on flat sequences of tokens, to transformations on complex structures based on tree traversals. Using `--->/2`, we have simplified the transformation formalism of [15].

We have used the sublanguage `FNUPDATE` of `FNQUERY` for a compact specification of updates to the archive layer of the critical Jean Paul edition. This was used for the construction of further navigation structures in addition to the hierarchical table of contents, cf. Figure 2, which we had before. Now we have a word register as well. In the future we want to base the creation of the whole navigation structure including the hierarchical table of contents on `FNUPDATE` and `FNTRANSFORM`.

At the moment the handling of meta data is a rather modest matter in our project. The next step will be to incorporate meta information in `RDF` and to get nearer to the *Semantic Web*. Without doubt this is a very interesting and important development for the future of critical editions. Examples of logic programming tools and libraries for `RDF` are to be found for example within `SWI-PROLOG` and within the Mozilla project [3]. As `RDF` is a format which must be supplemented by a separate means for inference, `PROLOG` (and companions) will be our favourite also in this field.

References

1. *S. Abiteboul, P. Bunemann, D. Suciu*: Data on the Web – From Relations to Semi-Structured Data and XML, Morgan Kaufmann, 2000.
2. *G. Antoniou, F. van Harmelen*: A Semantic Web Primer, MIT Press, 2004.
3. *D. Brickley*: Enabling Inference,
<http://www.mozilla.org/rdf/doc/inference.html>.
4. *D. Cabeza, M. Hermenegildo*: WWW Programming using Computational Logic Systems (and the PiLLoW / CIAO Library), Proc. Workshop on Logic Programming and the WWW, at WWW6, 1997.
5. *S. Ceri, G. Gottlob, L. Tanca*: Logic Programming and Databases, Springer, 1990.
6. *M.A. Covington*: Natural Language Processing for Prolog Programmers, Prentice Hall, 1993.
7. *G. Gazdar, C. Mellish*: Natural Language Processing in PROLOG: An Introduction to Computational Linguistics, Addison-Wesley, 1989.
8. *G. Gupta, E. Pontelli, D. Ranjan, et al.*: Semantic Filtering: Logic Programming Killer Application, Proc. Symposium on Practical Aspects of Declarative Languages PADL 2002, Springer LNCS 2257.
9. *M. Leventhal*: XSL Considered Harmful,
www.xml.com/pub/a/1999/05/xsl/xslconsidered_1.html.
10. *L.S. Levy*: Taming the Tiger – Software Engineering and Software Economics, Springer, 1987.
11. *R.A. O’Keefe*: The Craft of Prolog, MIT Press, 1990.
12. *C. Lehner*: PROLOG und Linguistik, Oldenbourg Verlag, 1990.
13. *B. Parsia*: Long story about using SWI-PROLOG, RDF and HTML Infrastructure, especially Chapter 6: DCGs Compared to XSLT,
<http://www.xml.com/pub/a/2001/07/25/prologrdf.html>.
14. *F. Pereira, S. Sheiber*: PROLOG and Natural-Language Analysis, Center for the Study of Language and Information, 1987.
15. *K. Prator*: Logic for Critical Editions, Proc. Intl. Conference on Applications of Declarative Programming and Knowledge Management INAP 2004.
16. *D. Seipel*: Processing XML Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.
17. *M. Smith, C. Welty, D. McGuinness*: OWL Web Ontology Language Guide, February 2004,
<http://www.w3.org/TR/2004/REC-owl-guide-20040210/>.
18. *J. Wielemaker, A. Anjewierden*: Programming in XPCE/PROLOG
<http://www.swi-prolog.org/>

A PROLOG Tool for Slicing Source Code

Marbod Hopfner, Dietmar Seipel, and Joachim Baumeister

University of Würzburg, Institute for Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{hopfner, seipel, baumeister}@informatik.uni-wuerzburg.de

Abstract. We describe a PROLOG tool for slicing source code. We assume that there exists an XML representation of the *parse tree* of the code. Then, we can perform an analysis of the extended *call graph* based on methods from the tool VISUR/RAR to determine the relevant predicates for the slice.

User-defined *policies* reflecting the different styles of programming of different users can be plugged into the system; they are formulated in a declarative way using the XML query language FNQUERY.

We have implemented VISUR/RAR and FNQUERY as part of the DISLOG developers toolkit DDK. So far we have applied slicing to extract several subsystems of the DDK.

Keywords. program extraction, comprehension, refactoring, visualization

1 Introduction

In a large software system it often becomes difficult to keep an overview of the entire system, even if it consists of classes, modules, and methods. For debugging a system, or for porting a subsystem to another language or dialect, it is important – but sometimes very difficult – to extract the relevant entities. A *slice* for a certain target method consists of all methods of the considered software system which are necessary to correctly run the subsystem in focus [9, 10].

We have implemented a PROLOG slicing tool for extracting source code for a certain functionality from a software system. Our approach is based on the extended *predicate dependency graph* (call graph for predicates) of the source code, which can be handled using the system VISUR/RAR [6], and on some further *policies* for handling specialities of the source language. We use an XML representation of the parse tree of the considered source code.

So far we have developed an extensible collection of extraction policies for the source language SWI PROLOG, cf. [11], which are specified declaratively as PROLOG predicates using the library FNQUERY for XML [5], but we could also extract methods from source code of other languages, provided that we have an XML representation of the parse tree and corresponding slicing policies for it.

We have in mind several reasons for slicing a software system:

- We can use slicing for *debugging* and for *extracting* certain functionality of a large software system to use it in a foreign application. One can obtain an overview of the relevant methods much more easily, and one does not need to read and understand unnecessary source code.
- Although in general it is very difficult to port PROLOG systems to other PROLOG dialects, porting becomes much easier if we can focus on slices of the systems, which could have only a small fraction of the size of the entire system.
- The integration of the slice into a foreign application becomes easier, since the probability that some methods and global variables of the application conflict with methods from the slice is reduced.

The rest of this paper is organized as follows: In Section 2 we give an overview of the slicing tool. In Section 3 we present a collection of basic policies for slicing PROLOG source code. User-defined slicing policies reflecting the programming styles of further users can be added using the language FNQUERY, which we also describe shortly. If slicing is done as a first step for porting a subsystem to another language or dialect of the same language, then it often is accompanied by *refactoring*; in Section 4 we mention some types of refactorings that could be useful in this context. Finally, in Section 5 we present two case studies applying the proposed slicing approach to the toolkit DDK,

2 An Overview of the Slicing Tool

In this section we give an overview of the slicing tool, and we list some problems that arise when we slice PROLOG source code. Moreover, we describe the XML representation of the source code and the XML query language called FNQUERY, on which the slicing tool is based.

2.1 Slicing PROLOG Source Code

Usually, we slice at the *granularity* of predicates, but we can also slice at the granularity of files. The result of the slicing process is the following:

- an archive, i.e., a directory with subdirectories containing the files with the selected predicates in the same structure as in the file hierarchy of the source system, and
- an index file containing some settings and consult statements for the files in the archive.

Consulting the index file from a foreign PROLOG application causes all necessary files from the archive to be properly consulted.

The following problems arise during the slicing of PROLOG source code: Firstly, it can be very complicated to determine the relevant predicates and files from the predicate dependency graph, if there are meta-call predicates calling other predicates. Secondly, we have to handle external libraries and dynamic loadings, and the consulting order

of the files in the slice needs to be the same as in the original source code. Thirdly, some generic predicates might be defined in many files, and not all of their clauses are relevant for the slice. Finally, the handling of predicate properties (such as dynamic or multifile) and of global variables is difficult. In Section 3 we will discuss how we have solved these problems based on the XML representation of the source code.

2.2 Source Code in Field Notation / XML

In the following we will briefly describe the used XML representation of PROLOG source code, the corresponding PROLOG structure, which we call field notation, and the XML query language FNQUERY; a more detailed introduction to the DDK library FNQUERY can be found in [5]. E.g., we can parse a PROLOG file `inc.pl` consisting of the single clause

```
increment(X, Y) :-
    Y is 1 + X.
```

into the following XML representation:

```
<file path="inc.pl" module="user">
  <rule operator=":-">
    <head>
      <atom module="user" predicate="increment" arity="2">
        <var name="X"/> <var name="Y"/>
      </atom>
    </head>
    <body>
      <atom module="user" predicate="is" arity="2">
        <var name="Y"/>
        <term functor="+">
          <term functor="1"/> <var name="X"/>
        </term>
      </atom>
    </body>
  </rule>
</file>
```

We represent an XML element as a PROLOG term structure $T:As:C$, which we call FN triple: it consists of the tag T , an association list As for the attribute/value pairs, and a list C containing the subelements. E.g., the PROLOG statement `Y is 1 + X` becomes the term structure `atom:As:C` shown below. Our parser from XML to field notation is based on the XML parser of SWI PROLOG, and we derive the field notation as a slightly more compact variant of the XML data structure of SWI PROLOG.

For obtaining the representation `Code` of a PROLOG program in field notation we use the predicate `program_file_to_xml/2` – without converting to an XML file first. From the FN triple `Code` we can select a body atom with the predicate symbol `is` using the infix predicate `:=/2` of FNQUERY:

```
?- program_file_to_xml('inc.pl', Code),
   Atom := Code^rule^body^atom::[@predicate=is].

Code = ...,
Atom =
  atom:[module:user, predicate:is, arity:2]:[
    var:[name:'Y']:[],
    term:[functor:+]::[
      term:[functor:1]:[], var:[name:'X']:[]]]

Yes
```

Every component T of the used path expression selects a corresponding subelement with the tag T , and the condition `[@predicate=is]` checks that the value of the attribute `predicate` is equal to `is`.

3 Basic Policies for Slicing PROLOG Source Code

For slicing a PROLOG system w.r.t. to a target predicate we have to take into account the special aspects of PROLOG. The set of *potentially relevant files* includes files that are loaded using statements such as `consult`, `use_module`, or `ensure_loaded`. Within this set of potentially relevant files we search for the *transitive definition* of the target predicate. Here, we can use the well-known concept of the *predicate dependency graph*, cf. Figure 1, but we have to pay special attention to *meta-call predicates* [1, 2].

Assume, e.g., that we would like to extract the predicate `increment/2`, which is defined in the file `my_arithmetic.pl`:

```
:- use_module(arithmetic).
:- consult([portray_predicates, test_predicates]).

increment(X, Y) :-
  call(add(1, X, Y)),
  portray(increment(X, Y)).

test(my_arithmetic, increment) :-
  increment(3, 4),
  message(sucessful_test, increment(3, 4)).
```

Then, the following file `arithmetic.pl`, which exports the predicate `add/3`, is potentially relevant, and it can be found due to the `use_module`-statement:

```
:- module(arithmetic, [add/3]).

add(U, V, W) :-
  W is U + V.
```

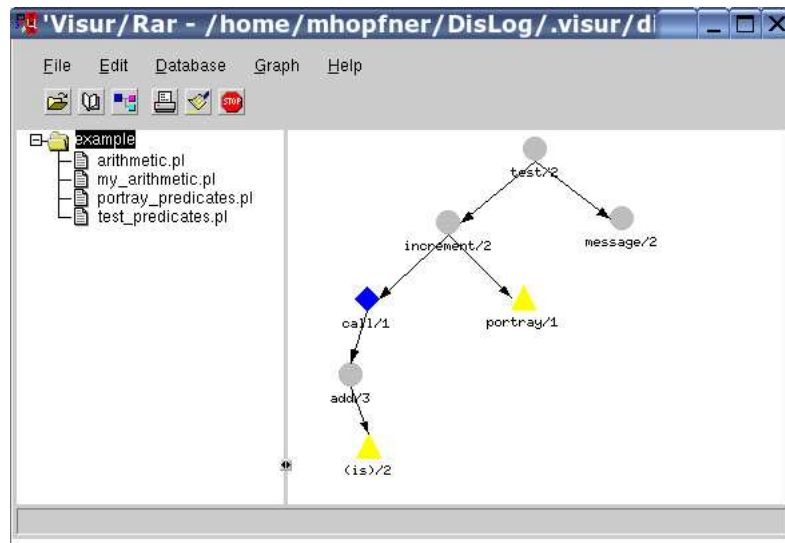


Fig. 1. Extended Predicate Dependency Graph in VISUR/RAR

But, at first sight it is not clear that `increment/2` transitively depends on `add`, which can only be inferred since we know that `call` is a meta-call predicate. Moreover, due to the `consult`-statements in `my_arithmetic.pl`, two further files are potentially relevant. Since `portray_predicates.pl` defines the pretty-printing predicate `portray/1` on which `increment/2` depends, this file needs to be in the slice. For ensuring the correctness of the produced slice, the test clause needs to be in the slice, i.e., we include the file `test_predicates.pl` containing the test suite, too.

3.1 The Potentially Relevant Files

The collection of potentially relevant files cannot be determined from the directory structure. All files that are loaded using `consult/1` from other potentially relevant files are also potentially relevant. But, moreover, the files must be consulted in the correct order. Sometimes, consulting a file calls a predicate that depends on other predicates. Then, it must be ensured that these predicates are available, before the depending predicates are called. We determine the order of the files using a call

```
code_to_sequence(Code, Sequence).
```

The predicate `code_to_sequence/2` has to be defined for each style of programming. For the DDK, it can be based on the file `dislog_units`.

All files that are consulted from other potentially relevant files are potentially relevant for the slice. E.g., external libraries are frequently loaded using `use_module/1`, and the further modules that they require are loaded using `ensure_loaded/1`.

3.2 Dependency Graphs with Meta-Call Predicates

The *transitive definition* of a predicate p consists of all predicates q , which are a member of the definition of the predicate p together with the transitive definition of these predicates q . In the previous example, the transitive definition of `increment/2` consists of the predicates `call/1`, `portray/1`, `add/3`, and `is/2`. The latter two predicates are reached, since the meta-call predicate `call/1` calls `add/3`.

Meta-Call Predicates We need to know which predicates are meta-call predicates and are able to call other predicates in one or more of their arguments. This is important in order to obtain the entire transitive definition of a predicate.

- In general it is impossible to automatically determine the predicates (and their arities) that are called from meta-call predicates. Then, such information has to be provided by the user by specifying the meta-call predicates in a configuration file. Otherwise, the slice will not be complete/correct. However, we can analyze many calls to meta-call predicates using some heuristics.
- For user-defined meta-call predicates, we additionally have to determine the arguments containing potential goals. In order to support the user in collecting the user-defined meta-call predicates, the system is able to automatically generate a list of possible meta-call predicates by searching for rules that call other already known meta-call predicates, such that one of the arguments in the call is connected to an argument of the head of the rule by a sequence of body atoms.
- For built-in meta-call predicates, we already know the arguments containing the goals that will be called, and in many cases we can infer their predicate symbol and arity. If the goal arguments are variables, then we track their bindings. E.g., if we can determine the predicate p and arity N of `Goal`, then we know that `checklist(Goal, Xs)` calls p with the arity $N + 1$; similarly, `maplist/3` adds 2 to the arity of the called goal.

Extended Dependency Graphs The call `calls_pp(Code, P1/A1, P2/A2)` determines the predicates $P2/A2$ that are called by $P1/A1$ in the PROLOG code represented by the FN triple `Code`. If we implement our policies correctly as clauses for `calls_pp/3`, then the transitive definition of the target predicate can be computed as the transitive closure of `calls_pp/3`.

E.g., for an atom `call(Goal)`, we try to determine the predicate and the arity of `Goal`. If the goal is created at runtime, then this is not always possible. But often, `Goal` is bound by statements such as `Goal = add(1, X, Y)` or `Goal = . . [P, 1, X, Y]`, where P was assigned to `add` before. The following rule for the predicate `calls_pp/3` determines a rule `Rule` with the head predicate $P1/A1$, such that there exists a body atom of the form `call(Goal)`. In XML this atom is encoded as

```
<atom module="user" predicate="call" arity="1">
  <var name="Goal"/>
</atom>
```

Rule and Goal are selected from the FN triple Code using suitable path expressions in FNQUERY. Within Rule we search for the predicate P2/A2 of Goal using the predicate goal_to_predicate/4:

```
calls_pp(Code, P1/A1, P2/A2) :-
    Rule := Code^rule::[
        ^head@predicate=P1, ^head@arity=A1],
    Goal := Rule
        ^body^atom::[
            @module=user, @predicate=call, @arity=1]
        ^var@name,
    goal_to_predicate(Rule, Goal, [], P2/A2).

goal_to_predicate(Rule, Goal, Goals, P/A) :-
    Atom := Rule^body^_ ^atom::[^var@name=Goal],
    [user, =, 2] := Atom@[module, predicate, arity],
    ([P, A] := Atom^term@[functor, arity]
    ; G := Atom^var@name,
      not(member(G, Goals)),
      goal_to_predicate(Rule, G, [Goal|Goals], P/A) ).
```

It could happen that P2/A2 becomes the predicate of Goal by a sequence of assignments. E.g., for the following rule for the predicate P1/A1=increment/2, the predicate goal_to_predicate/4 determines P2/A2=add/3 by subsequently looking at Goal and G, respectively:

```
increment(X, Y) :-
    G = add(1, X, Y),
    Goal = G,
    call(Goal).
```

The second call to goal_to_predicate/2 determines the atom Atom representing the equality G = add(1, X, Y), and then it selects the functor add and the arity 3 of the term argument of Atom.

3.3 Slicing of Individual Clauses

For some predicates the slice should only include individual clauses rather than all of the defining clauses. E.g., the defining clauses of generic predicates might be spread over many different files; examples from the DDK are dislog_help_index/0, the pretty-printer portray/1, and the test predicate test/2. In a slice we only need some of the clauses, and including all of them would be highly redundant.

We do not consider these generic predicates when we compute the set of potentially relevant files. But, when we include individual clauses into the slice, then we have to include the transitive definitions of all predicates that are called from them as well.

E.g., in the DDK many files contain some tests – which are usually located at the end of the files. For ensuring that the slice works correctly, the slice should include all clauses for `test/2` calling sliced predicates; in practice, most of these clauses will be contained in the potentially relevant files. In our example, for testing `increment/2`, we obviously need to include the definition of `message/2` in the slice.

3.4 Declarations of Predicate Properties

We have to detect the properties of all predicates in the slice, i.e., the corresponding declarations of the types `dynamic`, `multifile`, or `discontiguous`. Some of these declarations will be located in centralized files of the system, others are contained in separate files for each unit. Since these files might not be reached by the file dependency graph, the properties for these predicates have to be extracted using a special policy, such that they can be included into the slice.

```
predicate_to_property(Code, Pred/Arity, Property) :-
    Atom := Code^rule^body
           ^atom::[@predicate=Property],
    ( Term := Atom^term::[@functor='//']
      ; Term := Atom^_term::[@functor='//'] ),
    Pred := Term-nth(1)^term@functor,
    Arity := Term-nth(2)^term@functor.
```

E.g., in the DDK the declaration `:- multifile test/2.` for the test predicate can be found in the file `test_predicates.pl`.

3.5 Global Variables

In the DDK we have a generic concept of global variables, which are implemented using `assert` and `retract`. The global variables are accessed using the calls

```
dislog_variable_get(+Variable, ?Value),
dislog_variable_set(+Variable, +Value).
```

It might happen that the value of a relevant global variable is set in a file – either at consultation time or at runtime – that is not contained in the slice. Thus, the following predicate determines – on backtracking – the assignments of all variables `Variable` that are set in a file `File` using a call `dislog_variable_set(Variable, Value)`:

```
file_to_assignment(Code, File, Variable:Value) :-
    Variable := Code^rule::[@file=File]^body
              ^atom::[@predicate=dislog_variable_get]
              -nth(1)^term@functor,
    Atom := Code^rule^body
           ^atom::[@predicate=dislog_variable_set],
    Variable := Atom-nth(1)^term@functor,
    Value := Atom-nth(2)^term@functor.
```


In this rule, the path expression $\text{-nth}(N)^{\text{term}}$ selects the N -th subelement (for $N=1, 2$) with the tag term of an FN triple.

In the DDK, there also exist further similar concepts for global variables of subsystems, and we use the well-known predicate `gensym/2`, cf. [2].

4 Refactoring and Slicing

Refactoring methods [3] provide an effective tool in the context of slicing source code: Such methods modify source code without changing its external behavior. Refactoring methods have demonstrated their practical impact in numerous object-oriented software projects. They are strongly connected with appropriate test methods that are applied before and after performing the refactoring; thus, the preservation of the external behavior can be monitored.

Slicing is frequently accompanied or followed by refactoring. In order to adapt the slice to the software system in which it will be integrated, sometimes it is necessary to move or rename predicates, or to remove `consult`-statements. In the following, we briefly discuss some refactoring methods that are suitable in the context of slicing PROLOG code.

4.1 Predicate-Based Refactorings

- *Move Predicate*: A predicate is moved into a module. This refactoring may be useful for test predicates which are located in distributed modules. It is reasonable to apply this refactoring before performing the slicing procedure.
- *Rename Predicate*: This refactoring is usually performed after the slicing operation in order to adapt the slice to the target language, which will usually be a different PROLOG dialect. It is especially useful when slicing source code in order to allow for a simpler porting to another language dialect.
- *Extract Predicate*: Similar to the previous refactoring, this method is also useful, when we want to prepare the slice for the adaptation to another language dialect. Then, we extract *hostile* predicate blocks from the slice. In a subsequent step, the extracted blocks are replaced by suitable predicate calls from the target language.
- *Remove Unused Predicates*: If we slice at the granularity of files, then we may think of removing unused predicates from the files in the slice.

4.2 Module-Based Refactorings

Schrijvers et al. [8] also discuss useful refactorings based on the granularity of *modules*: merge modules, remove dead code intra-module, rename module, split module. In the context of slicing source code at the granularity of files these refactorings can be useful, if the sliced system does not need to allow for file-based updates.

5 Case Studies

We present two case studies applying the proposed slicing approach to the DISLOG Development Kit DDK, which we are developing using XPCE /SWI PROLOG [4]. The functionality ranges from (non-monotonic) reasoning in disjunctive deductive databases to various PROLOG applications, such as a PROLOG software engineering tool, and a tool for the management and the visualization of stock information.

Currently, the DDK consists of about 14.000 clauses; they are located in 440 files containing a total of about 100.000 LoC (lines of code). We have sliced out two subsystems, Minesweeper and FNQUERY. Slicing at the granularity of files has reduced the size of the system to subsystems of 16% (Minesweeper) and 4% (FNQUERY), respectively, of the original size, and slicing at the granularity of predicates has further reduced the size of the system to subsystems of 5% and 0.8%, respectively, of the original size.

The computation time is divided in a preprocessing time and the slicing time. The preprocessing takes about 0.6 msec per LoC on an Intel Pentium 1.1 GHz, 512 MB RAM. For the DDK the preprocessing took about 10 minutes. The preprocessing has to be done only once; its result can be used for each slicing operation, and for further operations like computing dependency graphs. The slicing operations took about 28.4 sec for Minesweeper and about 6.6 sec for FNQUERY, respectively.

Tests were also extracted and did successfully pass after slicing.

5.1 The Game Minesweeper

The slice for the game Minesweeper, which is a part of the DDK, should contain all files that are needed for a stand-alone version including the graphical user interface. Thus, we have extracted the transitive definition of the target predicate `create_board/0`, which creates the GUI of the game, cf. Figure 2.

In addition, we have included the transitive definition of all predicates called by the buttons `New Game`, `Show`, `Help`, `Help*`, and `Quit`, of the GUI. Clicking on these buttons calls a meta-call predicate, which calls another predicate, namely the predicates `new_game/0`, `show_board/0`, `clear_board/0`, and twice the help predicate `mine_sweeper_decision_support/1`, respectively. Since the syntax for meta-calls is often used in different ways, every user can write his own policies for obtaining the transitive definition of these calls, but one can also manually add predicates to the list of relevant predicates.

The consult file `index.pl` for the slice – a fragment is shown below – declares the predicate properties, sets all necessary global variables and consults the relevant files:

```
:- dynamic ...
:- multifile test/2, ...

:- dislog_variable_set(
    smodels_in, 'results/smodels_in').
```

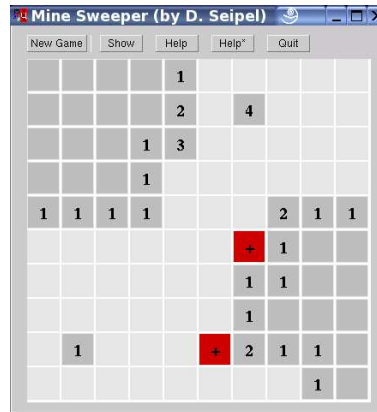


Fig. 2. Minesweeper

```
:- consult([
    'library/loops.pl',
    'library/ordsets.pl', ...
    'sources/basic_algebra/basics/operators',
    'sources/basic_algebra/basics/meta_predicates',
    ...]).

test(minesweeper, create_board) :-
    create_board.
```

5.2 The Field Notation and FNQUERY

In the second case study we have created a slice for the target predicate `:=/2` of `FNQUERY`, which is defined in the DDK module `xml/field_notation`. This slice consists only of about 800 LoC, which are extracted from 21 files of the DDK. 19 of these files are from regular DDK modules, which are located in the directory `sources`, and two of the files are library files from the directory `library`. The slice is derived from

- 8, i.e. 50%, of the files of the module `xml/field_notation`,
- 7, i.e. 25%, of the files of the module `basic_algebra/basics`,
- 3 files from the unit development, e.g., the file `development/refactoring/extract_method`,
- one file from the module `basic_algebra/utilities`, and
- the library files `lists_sicstus.pl` and `loops.pl` from the separate directory `library`.

6 Final Remarks

The slicing tool has been built based on two modules of the toolkit DDK: the system VISUR/RAR for visualizing and reasoning about source code, and the XML query and transformation language FNQUERY. It can be used for extracting slices from large software packages, provided that we have an XML representation for the considered language. So far we have applied it for extracting several subsystems of the DDK.

The tool can be extended by plugging in further user-defined slicing policies reflecting different styles of programming. Since the source code is represented in XML, slicing policies can be specified declaratively using the XML query language FNQUERY, which is fully interleaved with PROLOG.

We have used XML representations of PROLOG or JAVA source code for other purposes as well: for comprehending and visualising software with VISUR/RAR [6], for refactoring PROLOG programs and knowledge bases [5, 7], and for clone detection in PROLOG and JAVA source code.

References

1. *I. Bratko*: PROLOG—Programming for Artificial Intelligence, 3rd Edition, Addison–Wesley, 2001.
2. *W.F. Clocksin, C.S. Mellish*: Programming in PROLOG, 3rd Edition, Springer, 1987.
3. *M. Fowler*: Refactoring – Improving the Design of Existing Code, Addison–Wesley, 1999.
4. *D. Seipel*: DISLOG—A Disjunctive Deductive Database Prototype, Proc. 12th Workshop on Logic Programming WLP 1997.
5. *D. Seipel*: Processing XML–Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.
6. *D. Seipel, M. Hopfner, B. Heumesser*: Analyzing and Visualizing Prolog Programs based on XML Representations. Proc. International Workshop on Logic Programming Environments WLPE 2003.
7. *D. Seipel, J. Baumeister, M. Hopfner*: Declarative Querying and Visualizing Knowledge Bases in XML, Proc. 15th International Conference on Applications of Declarative Programming and Knowledge Management INAP 2004, pp. 140-151.
8. *A. Serebrenik, B. Demoen*: Refactoring Logic Programs, Proc. Intl. Conference on Logic Programming ICLP 2003 (Poster Session).
9. *M. Weiser*: Programmers use Slices when Debugging, Communications of the ACM, vol. 25, 1982, pp. 446–452.
10. *M. Weiser*: Program Slicing, IEEE Transactions on Software Engineering, vol. 10 (4), 1984, pp. 352–357.
11. *J. Wielemaker*: SWI–PROLOG 5.0 Reference Manual, <http://www.swi-prolog.org/>
J. Wielemaker, A. Anjewierden: Programming in XPCE/PROLOG
<http://www.swi-prolog.org/>

Combining Domain Splitting with Network Decomposition for Application in Model-Based Engineering

Rüdiger Lunde

R.O.S.E. Informatik GmbH, Schloßstr. 34, 89518 Heidenheim, Germany email: r.lunde@rose.de

Abstract. This paper discusses a new approach to combine the branch&prune algorithm with constraint network decomposition methods. We present a version of the well-known base algorithm which interleaves pruning, network decomposition and branching and is especially suited for large constraint networks with low connectivity. Special care is taken to support quantitative as well as qualitative and mixed domains. A sufficient criterion for the generation of decompositions called ‘independent solvability’ is defined, which guarantees maximal domain reduction for finite domains. We present production rules to compute decompositions and give an account of the implementation of the algorithm within a commercial model-based engineering tool.

1 Introduction

In current model-based engineering tools, constraint solving is used to predict the behavior of physical systems, based on component-oriented models. Constraints represent physical laws, whereas variables are used to describe behavioral modes of components as well as physical quantities or observable states of the system. Hybrid behavioral models containing quantitative relations (equations, inequalities, interpolating splines) as well as qualitative ones (tables or boolean formulas) give the modeler flexibility to choose the abstraction level which is best suited to the analysis task at hand.

To support the engineering process, the constraint solver is embedded into a reasoning framework, which uses the behavior prediction as a means to perform its high-level tasks. Typical tasks include predicting the expected behavior of a system over a period of time, estimating the risk that a certain top event occurs, explaining an observed situation, or proposing appropriate test points for candidate discrimination in diagnosis. Especially in diagnosis, models are often underdetermined. For complexity reasons, the constraint solver is required to deliver a compact representation of the solution space instead of point solutions of the network. Consequently, solvers used in this context apply inference techniques rather than search techniques (for a classification, see [5]).

In our approach, domain reduction is used to generate compact representations of solution spaces. The basic algorithm is a variant of the well-known branch&prune algorithm [7], which is extended by a network decomposition step. The paper elaborates on a decomposition strategy based on the so-called ‘independent solvability’ of subnetworks, which is appropriate to the structure of the constraint networks in model-based engineering.

The paper is organized as follows: We start with some preliminary definitions and a formalization of the investigated constraint problem. In Section 3, our version of the branch&prune algorithm is presented. Independent solvability of subnetworks as a sufficient criterion for constraint network decompositions is discussed in Section 4. We report briefly on the practical exploitation of the algorithm in Section 5, and conclude with outlining related work and promising directions of further research.

2 Basic Definitions

We define a constraint network in accordance with the literature (see e.g. [5]), but take special care to keep relation representations independent from variable domains and to support various kinds of domains:

Definition 1 (constraint network). *A constraint network $CN = \langle V, D, C \rangle$ consists of a finite set of variables $V = \{v_1, \dots, v_n\}$, a domain composed of the corresponding variable domains $D = D_1 \times \dots \times D_n$ and a set of constraints $C = \{c_1, \dots, c_m\}$. All variable domains D_i are subset of a common universe domain U . Constraints are relations defined on sets of variables which constrain legal combinations of values for those variables. Each constraint c is represented by a pair $\langle V_c, R_c \rangle$. $V_c \subseteq V$ is called the scope of c and is accessed by the function $\text{scope}(c)$. The relation R_c of c is a subset of $U^{|\text{scope}(c)|}$ and is accessed by $\text{rel}(c)$.*

Discrete variable domains are supported as well as continuous variable domains. If indices are not available, we write D_v to access the variable domain corresponding to variable v . In the algorithms discussed below, domains are represented by vectors of variable domains. If clear without ambiguity, those vectors are identified with the denoted cross product.

The projection function $\pi_i : 2^D \rightarrow 2^{D_i}$ is defined as usual. We write π_v instead of π_i where convenient. Obviously, for all domains and variables $\pi_v D = D_v$. The same notation is used to project a subset D' of a domain onto a set of variables instead of a single variable. For example, $\pi_{\{v_3, v_5\}} D = D_3 \times D_5$.

Subnetworks of CN are characterized by the following definition:

Definition 2 (subnetwork of a constraint network). *Let $CN = \langle V, D, C \rangle$ be a constraint network and $\hat{C} \subseteq C$ a set of constraints. The subnetwork of CN defined by \hat{C} is a constraint network $\langle \hat{V}, \hat{D}, \hat{C} \rangle$ with the following properties: $\hat{V} = \bigcup_{c \in \hat{C}} \text{scope}(c)$, and $\hat{D} = \pi_{\hat{V}} D$. It is denoted by $\text{sn}(\hat{C}, CN)$.*

An *instantiation* e of D is a set of value assignments, noted $\{\langle v_1, a_1 \rangle, \dots, \langle v_n, a_n \rangle\}$ with $v_i \in V$ and $a_i \in D_i$. It is called *complete*, if the set contains assignments for all $v \in V$, and *partial* otherwise. The *scope* of an instantiation e is defined by the set of those variables which occur in an assignment of e . Complete instantiations correspond to elements of D , whereas partial instantiations correspond to elements of projections obtained from D . For example, the instantiation $e = \{\langle v_1, a_1 \rangle, \langle v_3, a_3 \rangle\}$ corresponds to the value vector $\bar{e} = \langle a_1, a_3 \rangle$, which is an element of $\pi_{\{v_1, v_3\}} D$. Value vectors corresponding to instantiations are indicated by overlining.

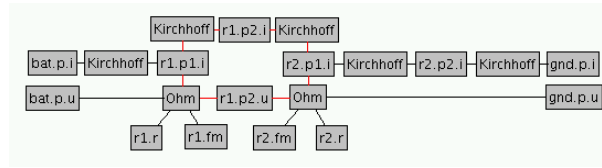


Fig. 1. Constraint net of two serial resistors. Nodes represent variables (dot-notation) and constraints

An instantiation e is said to satisfy a constraint c if $\text{scope}(c) \subseteq \text{scope}(e)$ and $\pi_{\text{scope}(c)}\{e\} \subseteq \text{rel}(c)$. Each complete instantiation e of D , which satisfies all constraints $c \in C$ is called a *solution* of CN . The set of all value vectors which correspond to solutions of CN form a subset of D . It is called solution set of CN and noted $\text{sol}(CN)$.

The space of all solutions of a given constraint network can be described by maximally reduced domains based on the following definition:

Definition 3 (maximally reduced domains). Let $CN = \langle V, D, C \rangle$ be a constraint network and $\hat{D} \subseteq D$ a domain. \hat{D} is called *maximally reduced with respect to CN* iff $\forall v \in V : \hat{D}_v = \pi_v \text{sol}(CN)$.

Our aim is to present efficient algorithms which compute maximally reduced domains \hat{D} for given constraint networks. The discussed algorithms try to reduce the range of each variable as far as possible without losing any solution. If the given constraint network has only one solution, a single value remains for each variable. In underdetermined networks (which is common in model-based diagnosis, due to incomplete or fuzzy information) a set of possible values is computed for each variable. For constraint networks with discrete variable domains the domain \hat{D} can be computed exactly. However, when reasoning about continuous domains it is infeasible to compute more than approximations of \hat{D} .

3 The Branch and Prune Algorithm

Constraint networks which describe real systems on a physical level are in general hard to solve because of cyclic dependencies, which occur in even the simplest aggregations of components (see Fig. 1 for illustration). The branch&prune algorithm [7] solves cycles by combining incomplete local constraint propagation [8] with recursive domain splitting. Our version of that algorithm, which is shown in Fig. 2, follows the idea of the original but adds a network decomposition step.

The algorithm starts with enforcing local consistency by means of the function `prune`, which implements a local constraint propagation method. If local propagation stops because one of the variable domains turns out to be empty, a domain composed of empty variable domains is returned. Otherwise, the constraint network is checked with regard to subnetworks with insufficiently reduced domains. The constraint sets which represent those parts are computed by the function `decomposeNetwork`. It returns a list of sets of constraints.

```

Domain branchAndPrune( $\langle V, D, C \rangle$ ) {
   $D' = \text{prune}(\langle V, D, C \rangle)$ 
  if ( $\neg \text{empty}(D')$ ) {
     $L = \text{decomposeNetwork}(\langle V, D', C \rangle)$ 
    for  $\hat{C}$  across  $L$  {
       $\langle \hat{V}, \hat{D}, \hat{C} \rangle = \text{sn}(\hat{C}, \langle V, D', C \rangle)$ 
       $\langle \hat{D}^1, \hat{D}^2 \rangle = \text{branch}(\langle \hat{V}, \hat{D}, \hat{C} \rangle)$ 
      if ( $\hat{D}^1 \neq \hat{D}^2$ ) {
         $\hat{D}^{1'} = \text{branchAndPrune}(\langle \hat{V}, \hat{D}^1, \hat{C} \rangle)$ 
         $\hat{D}^{2'} = \text{branchAndPrune}(\langle \hat{V}, \hat{D}^2, \hat{C} \rangle)$ 
        for ( $v \in \hat{V}$ )
           $D'_v = \hat{D}_v^{1'} \cup \hat{D}_v^{2'}$ 
        if ( $\text{empty}(D')$ )
          break
      }
    }
  }
  return( $\neg \text{empty}(D')$ ) ?  $D'$  :  $\langle \emptyset, \dots, \emptyset \rangle$ 
}

```

Fig. 2. The branch&prune Algorithm

For each set of constraints, the corresponding subnetwork is determined and its domain split into two parts by means of the function `branch`. If this operation was successful, the algorithm `branch&prune` is applied recursively to both partitions, and the set union of the returned variable domains is used to reduce the resulting domain. The loop is aborted if the resulting domain is empty. In this case, a domain composed of empty variable domains is returned.

The method `branch` can be implemented by selecting a so-called split variable and splitting its domain. Important for the correctness of the algorithm is that it returns either a pair of subdomains which partition the original domain \hat{D} or a pair of two equal domains, and that the former is the case if \hat{D} contains a discrete variable domain with at least two elements.

We now discuss some basic features of the branch&prune algorithm. In the following, we assume that the set of different variable domain representations is finite (but not necessarily the domains themselves) and that the implementation of `decomposeNetwork` terminates for all constraint networks.

It can easily be seen that the algorithm `branch&prune` terminates: Let $<_{\text{dom}}$ be a well-founded ordering on domains which agrees with narrowing, partitioning and subnetwork selection. One obvious realization for $<_{\text{dom}}$ compares the corresponding variable domains with respect to set inclusion. Given such an ordering, in each recursive call, domains become smaller with respect to $<_{\text{dom}}$.

Let $\text{bp}(\langle V, D, C \rangle) = \text{branchAndPrune}(\langle V, D, C \rangle)$. Obviously `branch&prune` is a narrowing algorithm, which does not add new elements to the given domain at any time:

Lemma 1. *Let $\langle V, D, C \rangle$ be a constraint network. $\forall v \in V : \text{bp}(\langle V, D, C \rangle)_v \subseteq D_v$.*

As an important feature, the branch&prune algorithm does not remove any element from any variable domain which occurs in a solution.

Lemma 2. *Let CN be a constraint network. $\forall v \in V : \text{bp}(CN)_v \supseteq \pi_v \text{sol}(CN)$.*

The quality of the domain reduction, which is obtained by our version of the branch&-prune algorithm, strongly depends on the network decomposition strategy. In the next section, we will introduce a sufficient decomposition criterion called ‘independent solvability’, which guarantees maximally reduced domains for constraint networks over finite domains. For decomposition algorithms based on this criterion, the computed domains are identical to the domains which are computed by the original branch&prune algorithm without network decomposition. The approximation accuracy for constraint networks over continuous or mixed domains depends on the chosen interval bound representation precision, the implementation of the method `branch`, and the used narrowing operators. Let us assume that the method `branch` partitions a given domain whenever possible with respect to the chosen interval bound representation, that all constraints describe continuous functions, and that the type of local consistency obtained by the narrowing operators converges with decreasing variable domain diameters to arc-consistency or a strong approximation like hull-consistency [1]. Then, the resulting domain converges with increasing bound representation precision to the maximally reduced domain.

In practice, the required accuracy depends on the analysis task to solve. Often weak approximations of maximally reduced domains suffice. If that is the case, the key to scale the precision of the result with regard to necessary accuracy and available resources lies in the splitting strategy used by the method `branch`. Our implementation selects continuous variables as split-variables only if the absolute and the relative diameters of their current domain bounds exceed specified thresholds.

Definition 4 (absolute and relative diameter). *Let $I = [a \ b]$ be an interval. The absolute diameter of I is defined by $\text{diam}(I) = b - a$ and the relative diameter is defined by*

$$\text{diamRel}(I) = \begin{cases} 0 & : a = 0 = b \\ \infty & : a < 0 \leq b \vee a \leq 0 < b \\ \frac{\text{diam}(I)}{\min(|a|, |b|)} & : \text{otherwise.} \end{cases} \quad (1)$$

Especially in underdetermined networks a careful selection of appropriate thresholds is crucial for the success of the analysis.

4 Constraint Network Decomposition

When applying the branch&prune algorithm to possibly underdetermined constraint networks of several thousand constraints, the main problem is efficiency. We address this issue with a new structure-based approach.

In the field of continuous CSP solving, strong graph decomposition algorithms are known [2] which are based on the Dulmage and Mendelsohn decomposition. The result of this decomposition is a directed acyclic graph of blocks¹. Unfortunately, those algorithms cannot be applied here, due to the hybrid structure of our constraint networks.

¹ Blocks correspond to subnetworks in our approach and generally include cycles. They are connected by directed arcs which represent causal dependencies.

While conditions might be handled by a two step simulation approach, which separates mode identification from continuous behavior prediction, inequalities are definitely not covered by the Dulmage and Mendelsohn decomposition technique. Solvers based on this decomposition techniques compute series of point-solutions, but fail to determine complete ranges for variables in underdetermined networks.²

The decomposition criteria proposed here are much weaker, but do not impose restrictions on the type of relations used. Their aim is to identify parts of the constraint network which can be solved sequentially, one after the other. Besides the graph information obtained from the constraint network structure, we integrate another source of information, namely, the current value ranges of the variables. Since the ranges are subject to change during the recursive constraint solving process, graph decomposition is not applied as a preprocessing step, but is performed anew after each pruning step.³

4.1 Graph Notions

The static dependency structure of a constraint network $CN = \langle V, D, C \rangle$ is represented by an undirected bipartite graph G called *constraint graph of CN* . The set of vertices is defined by the union of all variables and all constraints in CN . Undirected edges connect each constraint with the variables of its scope. We represent undirected edges by sets. So we get

$$G = \langle V \cup C, \{ \{c, v\} \mid c \in C \wedge v \in V \wedge v \in \text{scope}(c) \} \rangle.$$

For the computation of maximally reduced domains, dependencies between variables whose domains are not uniquely restricted are of special interest. We call a variable v *relevant* iff $|D_v| > 1$. Replacing V by the set of all relevant variables $V_r \subseteq V$ in the definition above, a subgraph of G is obtained which we call *relevant subgraph of CN* .

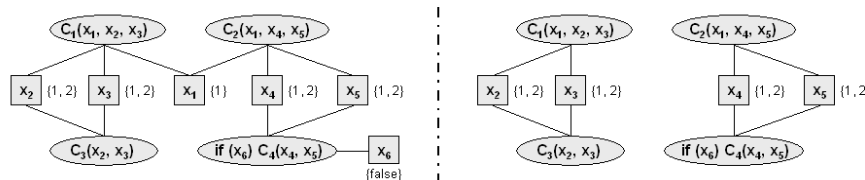


Fig. 3. A relevant subgraph

The complexity of the task to compute maximally reduced domains for a given constraint system strongly depends on the structure of its relevant subgraph. If the graph is acyclic pure local propagation is sufficient, as the following theorem states.

² One reason for this lies in hidden cycles between different blocks. While causality between different blocks is cycle-free, dependencies are not. In general, the corresponding undirected version of the block graph is cyclic.

³ Implementations can profit from the monotonic refinement of ranges by reusing graph analysis results gained one recursion level up.

Theorem 1. *Let $CN = \langle V, D, C \rangle$ be a constraint network with a non-empty domain. If CN is arc-consistent and its relevant subgraph acyclic, then D is maximally reduced with respect to CN .*

Proof. Let CN be arc-consistent and its relevant subgraph G be acyclic. Obviously $\forall v \in V : D_v \supseteq \pi_v \text{sol}(CN)$. We show that $\forall v \in V : D_v \subseteq \pi_v \text{sol}(CN)$.

Let $C_r \subseteq C$ be the set of all constraints, whose scope contains at least one relevant variable. Since CN is arc-consistent, each complete instantiation of D which satisfies all $c \in C_r$ also satisfies all $c \in C \setminus C_r$. Hence $\text{sol}(CN) = \text{sol}(\langle V, D, C_r \rangle)$ and it suffices to show that $\forall v \in V : D_v \subseteq \pi_v \text{sol}(\langle V, D, C_r \rangle)$.

By precondition, each variable domain contains at least one value. If C_r is empty, all complete instantiations of D are solutions of $\langle V, D, C_r \rangle$ and there exists at least one. Hence $\forall v \in V : D_v \subseteq \pi_v \text{sol}(\langle V, D, C_r \rangle)$.

Otherwise let $v \in V$ and $a \in D_v$ be chosen arbitrarily. We have to prove that there exists a solution of $\langle V, D, C_r \rangle$ which contains the assignment $\langle v, a \rangle$. Let G_r be the relevant subgraph of $\text{sn}(C_r, CN)$. G_r contains at least one constraint and a relevant variable which is connected to it.

Case 1: v is a vertex of G_r . Since G is acyclic, G_r is acyclic too. Hence, we can arrange its nodes as a forest, in which one of the trees is rooted by v . Nodes with even depth are variables and with odd depth constraints. Let $<_{C_r}$ be a total ordering on the constraint vertices which agrees with depth, and for each instantiation e let $\text{mc}(e)$ denote the minimal constraint c with respect to $<_{C_r}$ which is not satisfied.

Let us now assume that there is no solution of $\langle V, D, C_r \rangle$ which contains the assignment $\langle v, a \rangle$. Among the complete instantiations of D which contain $\langle v, a \rangle$, there must be an instantiation e with the property that $c = \text{mc}(e)$ is maximal with respect to $<_{C_r}$. Since CN is arc-consistent, regardless of the value assignment which is used in e for the parent variable v_p of c in G_r , there must exist value assignments e_s for all other variables in $\text{scope}(c)$ such that c is satisfied. By replacing in e the assignments for all variables in $\text{scope}(c) \setminus \{v_p\}$ by e_s , we obtain a new instantiation e' which differs from e in some assignments for the child variables of c in G_r . Since it satisfies c and (like e) all smaller constraints with respect to $<_{C_r}$, we get $\text{mc}(e') >_{C_r} \text{mc}(e)$, which is a contradiction to our assumption. So there must be a solution of $\langle V, D, C_r \rangle$ which contains the assignment $\langle v, a \rangle$.

Case 2: v is not a vertex of G_r . Let v' be a variable vertex of G_r and $a' \in D_{v'}$ chosen arbitrarily. As proved in case 1 there exists a solution e of $\langle V, D, C_r \rangle$ which contains $\langle v', a' \rangle$. Since v is irrelevant, $D_v = \{a\}$. We conclude that e must contain the assignment $\langle v, a \rangle$. \square

4.2 Independent Solvability

When searching for realizations of `decomposeNetwork`, the major motivation is the reduction of computational costs to compute maximally reduced domains. The smaller the resulting subnetworks, the better. But we have to be careful not to loose too much narrowing quality, compared to the branch&prune variant without network decomposition. We definitely want to guarantee maximally reduced domains for constraint networks over finite domains.

We start by defining a class of decompositions called independently solvable decompositions. Based on the definition, a sufficient but not necessary criterion for the realization of `decomposeNetwork` is given. The problem of computing independently solvable decompositions is addressed by providing a set of production rules.

Definition 5 (decomposition of a constraint network). Let $CN = \langle V, D, C \rangle$ be a constraint network, L a set of mutually disjoint subsets of C . The set of subnetworks $\Lambda = \bigcup_{\hat{C} \in L} \{\text{sn}(\hat{C}, CN)\}$ is called a decomposition of CN . $\Lambda_v \subseteq \Lambda$ denotes the set of all constraint networks in Λ which contain the variable v .

Definition 6 (independent solvability). Let $CN = \langle V, D, C \rangle$ be a constraint network, Λ a decomposition of CN . Let $D^{\text{is}} \subseteq D$ be a domain whose variable domains are computed from the maximally reduced domains for the corresponding constraint networks in Λ by intersection: $\forall v \in V : D_v^{\text{is}} = \bigcap_{cn \in \Lambda_v} \pi_v \text{sol}(cn) \cap D_v$

Λ is called independently solvable, iff D^{is} is empty or maximally reduced with respect to CN .

An implementation of `decomposeNetwork` can be based on independent solvability. In this case, for any constraint network CN over finite domains, `decomposeNetwork` computes a vector $\langle C_1, \dots, C_n \rangle$ with the property that $\bigcup_{1 \leq i \leq n} \{\text{sn}(C_i, CN)\}$ is an independently solvable decomposition of CN . In the following $\text{bp}^{\text{is}}(CN)$ denotes the result of a variant of the branch&prune algorithm which uses an independent solvability based `decomposeNetwork` implementation.

Theorem 2 (correctness of branchAndPrune). The branch&prune algorithm returns maximally reduced domains for all constraint networks over finite domains if the implementation of `decomposeNetwork` is based on independent solvability.

Proof. (Sketch) Thanks to Lemma 2, it suffices to show, that for all domains D which are composed of finite variable domains, the following proposition holds:

$$\forall C \forall V \forall v \in V : \text{bp}^{\text{is}}(\langle V, D, C \rangle)_v \subseteq \pi_v \text{sol}(\langle V, D, C \rangle). \quad (2)$$

The proof is a strong induction over the domain size. The central loop invariant is

$$\text{empty}(D') \vee \forall v \in V : D'_v \subseteq \bigcap_{cn \in \Lambda_v^i} \pi_v \text{sol}(cn) \cap D_v^{\text{pr}},$$

where i is the number of previously performed loop body evaluations, D^{pr} the contents of the algorithm variable D' after pruning, $\Lambda^i = \bigcup_{j \leq i} \{\text{sn}(C_j, \langle V, D^{\text{pr}}, C \rangle)\}$, and C_j the components of the vector which was returned by `decomposeNetwork`. \square

Consequently, independent solvability is a sufficient criterion to guarantee maximally reduced domains for constraint networks over finite domains. We now face the task how to compute such decompositions. The Lemmas 3 – 6 are called production rules for independently solvable decompositions. They suggest to start with a trivial decomposition, and to refine it subsequently by removing or splitting contained subnetworks.

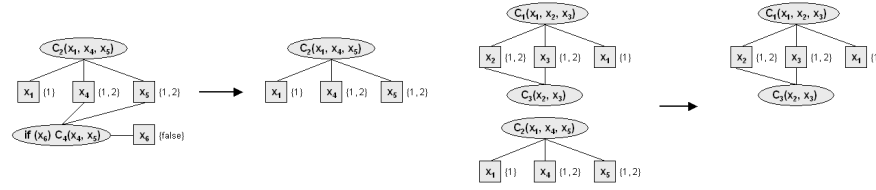


Fig. 4. Constraint removal and subnetwork removal

Lemma 3 (trivial decomposition). Let $CN = \langle V, D, C \rangle$ be a constraint network. The set $\{sn(C, CN)\}$ is an independently solvable decomposition of CN .

Proof. (Sketch) If D is empty, $\{sn(C, CN)\}$ is an independently solvable decomposition of CN . Otherwise let $\langle \hat{V}, \hat{D}, C \rangle = sn(C, CN)$. Since D_v is not empty for all $v \in V \setminus \hat{V}$, every solution of $\langle \hat{V}, \hat{D}, C \rangle$ which contains the assignment $\langle v, a \rangle$ can be converted into a solution of $\langle V, D, C \rangle$ with the same assignment and vice versa. So we get for all $v \in \hat{V}$, $\pi_v sol(\langle \hat{V}, \hat{D}, C \rangle) = \pi_v sol(\langle V, D, C \rangle)$, which implies the lemma. \square

Many interactions between different components depend on the current operational or fault state of the system. An open switch disconnects two parts of a circuit, the same is true for a closed valve in a hydraulic system or a broken belt in an engine. Using conditional constraints to express state-dependent relations allows to disconnect parts of the networks, which are not independent in general, but with respect to the currently analyzed state.

Lemma 4 (constraint removal). Let $CN = \langle V, D, C \rangle$ be a constraint network, $\Lambda \uplus \{sn(\hat{C}, CN)\}$ an independently solvable decomposition of CN and $c \in \hat{C}$ a constraint, whose condition cannot be fulfilled by the instantiations of \hat{C} . Then $\Lambda \cup \{sn(\hat{C} \setminus \{c\}, CN)\}$ is an independently solvable decomposition of CN .

Proof. Since c is satisfied by any instantiation e with $scope(e) \supseteq scope(c)$, we get $sol(sn(\hat{C} \setminus \{c\}, CN)) = sol(sn(\hat{C}, CN))$ which implies the lemma. \square

As stated by Theorem 1, domains of arc-consistent acyclic networks are maximally reduced. Therefore, such subnetworks can be removed from an independently solvable decomposition. Since the branch&prune algorithm performs local propagation before splitting the network, it can provide arc-consistency for certain subnetworks (e.g. subnetworks over finite domains).

Lemma 5 (subnetwork removal). Let CN be a constraint network and $\Lambda \uplus \{\hat{CN}\}$ an independently solvable decomposition of CN . If \hat{CN} is arc-consistent and its relevant subgraph acyclic, then Λ is an independently solvable decomposition of CN .

Finally, subnetworks whose relevant subgraphs do not overlap can be split, as stated in the last production rule.

Lemma 6 (subnetwork splitting). Let CN be a constraint network, $\hat{CN} = sn(\hat{C}, CN)$ a subnetwork of CN and $\Lambda \uplus \{\hat{CN}\}$ an independently solvable decomposition of CN . Let further $C^1 \uplus C^2 = \hat{C}$ be a partitioning of \hat{C} and $CN^i = sn(C^i, CN)$ ($1 \leq i \leq 2$) the

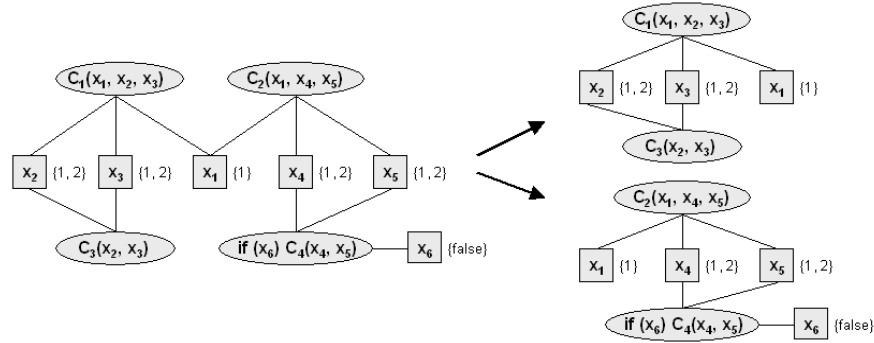


Fig. 5. Subnetwork splitting

corresponding subnetworks. If the relevant subgraphs of CN^1 and CN^2 do not overlap, then $\Lambda \cup \{CN^1, CN^2\}$ is an independently solvable decomposition of CN .

Proof. (Sketch) Let $\langle V, D, C \rangle = CN$ and $\langle V^i, D^i, C^i \rangle = CN^i$ for $1 \leq i \leq 2$. Since $D_v^i = D_v$ for all variables $v \in V^i$, and $|D_v| = 1$ for all common variables $v \in V^1 \cap V^2$, the union of all value assignments contained in any pair of solutions $e^i \in \text{sol}(CN^i)$ ($1 \leq i \leq 2$) forms a solution e of \hat{CN} .

The rest of the proof instantiates the definition of independent solvability for $\Lambda \cup \{CN^1, CN^2\}$ and $\Lambda \uplus \{\hat{CN}\}$ and makes a case differentiation about whether $\text{sol}(\hat{CN})$ is empty or not. \square

5 Experimental Results

We have integrated a Java implementation of the presented concepts into the commercial model-based engineering tool RODON.

To demonstrate the usage of the branch and prune approach in our application focus, we have chosen a diagnostic problem. Given a model of a real system and a symptom describing some abnormal state, the diagnostic module of RODON performs a conflict directed search for diagnostic candidates. The results presented in Table 1 are based on a quantitative model of an automotive exterior lighting system (see Fig. 6). It consists of three electrical control units, 10 drivers providing 20 diagnostic trouble codes, 4 switches, 12 actuators and several connector blocks, wires, fuses, diodes, resistors and relays. Altogether 86 physical components are included, defining 143 single faults including unknown fault modes for all connector boxes. The behavior model is built out of 1816 variables (most of them quantitative and unbounded) and 1565 constraints.

The first diagnosed symptom (\triangle) describes the observation that two rear lights are dimmed which can be explained for example by a disconnected ground node. RODON checks 15 diagnostic candidates and returns five of them which are consistent with the given symptom. The second symptom (\diamond) represents a partial system state in which two side marker fault codes indicate a short to ground failure. Here 18 diagnostic candidates are checked and two returned, including a multiple fault. For both symptoms behavior

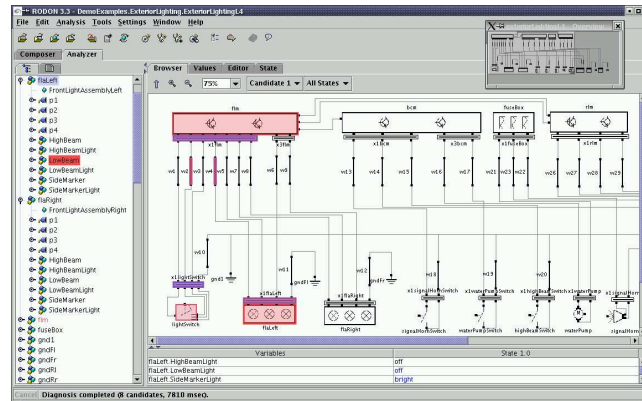


Fig. 6. The model-based engineering tool RODON

prediction based on pure local propagation fails due to a large number of algebraic cycles.

Table 1 shows that while the number of successful network decompositions (#splits) is limited in the presented example runs, the impact on the number of recursive solver calls (#propagations) as well as the overall time to compute the diagnosis (t , measured on a Pentium 4 with 1.1GHz) is significant. We have tested the algorithm in different analysis tasks with various models including much larger ones. As one would expect, the performance gains differ from case to case, but the presented results seem to be typical for the average case. Network decomposition has shown to reduce execution time significantly when underconstraint states are investigated. The smaller the diameters used to select split variables are, the more computation time can be saved.

Task Characterization			Without Decomposition		With Decomposition		
Symptom	diam	diamRel	#propagations	t [sec]	#splits	#propagations	t [sec]
\triangle	1	5	100	5.4	7	80	1.9
\triangle	0.01	1	954	11	7	204	3
\triangle	0.01	0.1	99056	1209	7	4768	34
\diamond	1	5	8453	66	5	261	3.5
\diamond	0.01	1	> 55000	> 1000	5	21821	190

Table 1. Performance gained by network decomposition during diagnostic runs

6 Conclusion and Related Work

Reducing domains in large cyclic constraint structures is generally a hard task, especially if the system includes continuous variables. The currently known inference methods can be divided into algebraic and numeric methods.

Several contributions have addressed the problem of solving cyclic constraint structures by algebraic transformation algorithms (see e. g. [6] for a survey). By linearization,

a linear system of equations is obtained which can be solved by the Gauss-Seidel iterative method. Polynomial systems can be transformed by Gröbner basis computation. A common framework for transformations based on variable elimination called “Bucket Elimination” is presented in [4].

Numeric methods reduce variable domains by combining local consistency criteria with some kind of recursive trial-and-error strategy. The branch&prune algorithm discussed in this paper splits domains and tests local consistency of the resulting boxes. Other numeric methods shift the bounds of variable domains until inconsistency is detected (see e.g. [3]).

While algebraic methods can only be applied to more or less restricted types of constraints, numeric methods suffer from their worst-case complexity, especially when applied to large underconstraint networks. In this paper, we have presented a structure-based approach to improve the performance of the branch&prune algorithm by defining a class of state dependent network decompositions called ‘independently solvable decompositions’ and providing production rules to compute instances of it. The decompositions allow to solve independent parts sequentially instead of recursively, and thereby reduce recursion depth significantly. Parts which are already maximally reduced are identified by cycle analysis and removed from the network. These optimizations increase significantly the size of models whose solution is feasible in practice.

The class of independently solvable decompositions is very general and can improve other numeric domain reduction methods as well. Nevertheless, one obvious limitation of all decompositions which can be derived by the provided production rules is, that the contained subnetworks do not share any common relevant variable. That is partly because the production rules do not span the complete class of independently solvable decompositions. But the required notion of independence also limits the available decompositions more than necessary. We are currently working on relaxations of this criterion.

References

1. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Proceedings Int. Logic Programming Symposium*, 1994.
2. Christian Bliek, Bertrand Neveu, and Gilles Trombettoni. Using graph decomposition for solving continuous CSPs. *Lecture Notes in Computer Science*, 1520:102+, 1998.
3. Lucas Bordeaux, Eric Monfroy, and Frederic Benhamou. Improved bounds on the complexity of kB-consistency. In *IJCAI*, pages 303–308, 2001.
4. Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.
5. Rina Dechter. *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, May 2003.
6. L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-interval cooperation in constraint programming. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pages 150–166, 2001.
7. P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, April 1997.
8. A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

Towards an Object-Oriented Modeling of Constraint Problems^{*}

Armin Wolf, Henry Müller, and Matthias Hoche

Fraunhofer FIRST, Kekuléstraße 7, D-12489 Berlin, Germany
{Armin.Wolf|Henry.Mueller|Matthias.Hoche}@first.fraunhofer.de

Abstract. In this paper we present a first version of a "model and run" paradigm for Constraint Programming (CP) following J.F. Puget's ideas of the next challenge in CP: simplicity of use. To obtain an easy to use modelling of problems we suggest a combination of the Unified Modelling Language UML and the Object Constraint Language OCL as a standard file format. We mainly concentrate in this paper on a way to achieve automatic data driven generation of complex constraint networks by the use of constraint network schemata. Therefore we define what a constraint network schema formally means, how it is modelled correctly and when some information about a concrete network is compatible. Due to the aspect of data driven constraint network generation, we further give an instruction on how *constraint network schemata* (CNS) can be used to fill possible gaps in the problem's data and automatically complete constraint networks.

1 Introduction

The interpretation of real problems and formulation as a constraint satisfaction problem requires experience and expertise of a constraint programmer. We know from our own experience with our constraint solver `firstcs` [2] that business partners are not in the position to model their problems on their own. The only chance to include such non-constraint-experts in the modelling is to simplify the whole process (see figure 1). Making sufficient abstract instruments without loosing the expressiveness is a challenge.

In [4] J.-F. Puget argues for a modelling language for CP like MPS for MP to allow a simple use of CP - a "model-and-run" paradigm. Following him, for a further success of CP in industrial applications these items should be developed:

- A standard file format for expressing CP models,
- a library of CP algorithms that can get models as input and produces solutions as output without requiring additional inputs,
- books that focus on modelling,
- and software improvement that do not require model rewriting.

^{*} The work presented in this paper is funded by the European Union (EFRE) and the state Berlin within the research project "inubit MRP", grant no. 10023515.

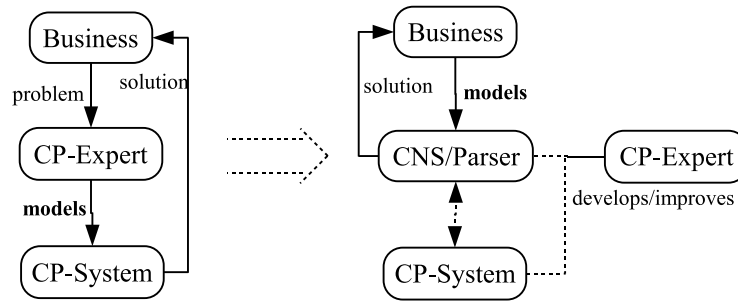


Fig. 1. Towards simplicity

Following this proposal, we will suggest a standard file format to express CP models in the first part. The second and main part will show how to transform the developed file format into adequate constraint networks. The aim of this paper is to provide a programming system independent, object-oriented modelling technique for constraint problems. Model transformation to improve efficiency like symmetry breaking, implied constraints, auxiliary variables, dual models etc. (cf. [6]) are not in the scope of this paper.

Our approach works out the common structures of constraint problems, the so called constraint network schemata. In contrast to constraint patterns [6] these are formal specifications of some constraint problem classes in a object-oriented sense.

Example 1. All job-shop-scheduling problems have a common structure: They consists of a set of n machines and m jobs. These jobs consist in turn of n tasks. The tasks of any job have to be processed one after the other using a different machine.

In the following, we propose a process for a data-driven generation of constraint networks by the use of such schemata. Further, we define some necessary properties for this generation process.

2 Choosing the standard file format

We argue that the standard file format of our choice must provide the following characteristics:

- widely used and accepted in software engineering,
- object-oriented modelling for well-structuring and information re-use,
- well-known and used in industry and science,
- plenty of available tools to support the modelling process.

With respect to these demands, we decided to use a combination of the Unified Modelling Language (UML) and the Object Constraint Language (OCL).

UML not only standardizes object-oriented modelling but also the file format XMI, which enables platform independent model processing and exchange.

It should be mentioned, that our focus on object-oriented modelling will not restrict the used CP systems to this programming paradigm. A lot of work was already done in the community to make modelling transparent to the used system, see [5] for example.

3 Generating Constraint Networks using UML and OCL

Beyond checking given constraint network objects against formally specified structural and semantical aspects the main focus of our work lies on an automatic, data-driven generation of a “complete” constraint network from partially available data, i.e. given some values or variables the interrelating constraints are generated automatically by the use of a given formal schema. We propose the *Unified Modelling Language* UML [3] and its extension the *Object Constraint Language* OCL [7] for the formal specification of constraint network schemata. In our approach we are using some UML class diagrams to describe different types of constraints and the values as well as the variables constrained by them. The idea is to allow the user to build up constraint networks schemata iteratively:

- At the bottom level the “atomic” constraint network schemata are the available/supported constraints types, value and variable types.
- At a higher level the “complex” constraint network schemata consist of constraint networks, values and variables already defined on a lower level.

Example 2. In a resource allocation environment the atomic constraint network schemata are “single resource”, “task”, “weighted sum”, “before”, and other types of constraints. Complex constraint network schemata like “jobs” consists of “task” and “before” constraints.

However, in “pure” UML there are only limited possibilities to describe the interrelations between the classes’ instances. Hence, we have chosen OCL to fill this gap. In detail, the construction of complex constraint network schemata works as follows:

1. Existing atomic or complex constraint networks are either aggregated as members into a new complex constraint network schema (i.e. in a new UML class) or are associated with them.
2. The aggregated or associated constraint network schemata are connected via invariant object equations formulated in OCL.

Here, the introduction of OCL enriches the expressiveness of the UML model. What is needed in the first place to generate a constraint network is:

- the formulation of equations within constraint network schemata. E.g. if a new complex schema is created, we most likely want to express variable identities to connect already defined constraint network schemata.

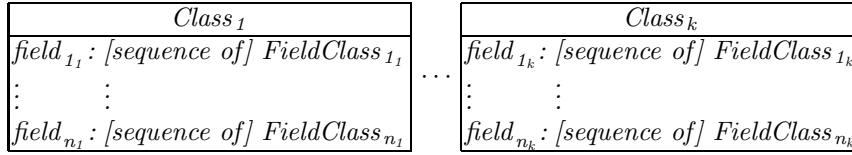
- the formulation of sequences of instances of constraint network schemata which avoids ambiguity and enables “generic” constraint schemata like the sum of a variable number of addends.¹

Later, the formulation of restrictions on basic data types, e.g. $\min \geq 0$ where \min is a number, have to be added, too.

3.1 Constraint Network Schemata (CNS)

Definition 1 (constraint network schema). A constraint network schema (CNS) consists of

- a non-empty finite set UML class diagrams having the structure



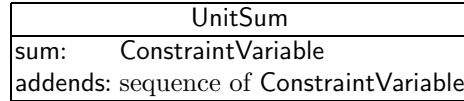
- and of finitely many OCL equations

$$f_{1_i} \dots f_{u_i} = f'_{1_i} \dots f'_{v_i} \quad (1 \leq i \leq m_k)$$

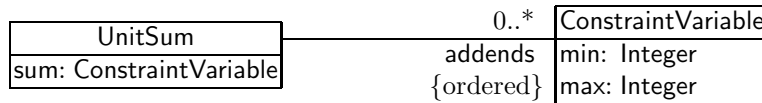
which are the invariants of the class diagrams Class_j (1 ≤ j ≤ k)

where Class₁, ..., Class_k are pairwise different class identifiers which are different from OCL basic types, i.e. Integer, Real, etc. and for each j = 1, ..., k the identifiers field_{1_j}, ..., field_{n_j} are pairwise different, too. Furthermore, for each i_j = 1₁, ..., n₁, ..., 1_k, ..., n_k the field class identifiers FieldClass_{i_j} are either in the set {Class₁, ..., Class_k} or OCL basic types and the identifiers f_{1_i}, ..., f_{u_i} as well as f'_{1_i}, ..., f'_{v_i} are field identifiers, i.e. in the set {field_{1₁}, ..., field_{n₁}, ..., field_{1_k}, ..., field_{n_k}}. □

In the previous definition, the non-UML, optional construct “sequence of” represents an “ordered” association with default multiplicity “0..*” in UML, e.g.

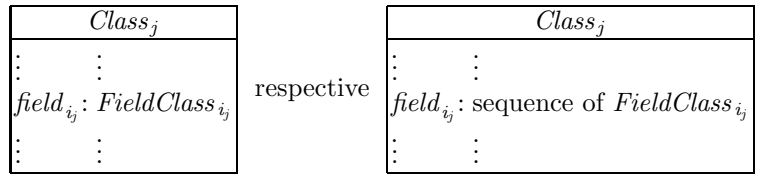


stands for



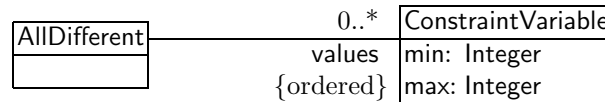
In the following and for simplicity we use the compact textual representation Class_j :: field_{i_j} : FieldClass_{i_j} and Class_j :: field_{i_j} : sequence of FieldClass_{i_j} instead of the class diagrams

¹ This will be realized via the UML annotation *ordered*.

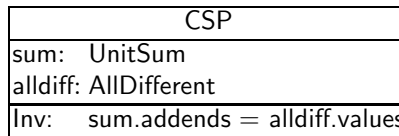


The following example shows an application of these modelling techniques for a constraint satisfaction problem (CSP):

Example 3 (UML using OCL equations). Let be given a simple CSP consisting of the UnitSum constraint presented before and the well-known AllDifferent constraint. The AllDifferent constraint is represented as a UML class diagram as follows:



To model the CSP as UML we can now simply aggregate constraints to another class representing the CSP.



This small example already demonstrates the need for OCL expressions as well as showing how easily these expressions can be used. The problem is that the UML aggregation says nothing about the relations between both constraint. Therefore the OCL equation `sum.addends = alldiff.values` is added, expressing that `sum` and `alldiff` share the same constraint variables.

3.2 Model Restriction — Well-Defined CNS

To reflect the iterative process of the definition of complex constraint network schemata and to avoid non-termination in the definition process of their semantics, recursion is not allowed. Therefore we define an order relation on CNS:

Definition 2. For a CNS with UML class diagrams named $Class_1, \dots, Class_k$ (cf. Definition 1), let ‘ $>$ ’ be the binary relation on these diagrams such that

$A > B$ holds if and only if $A = Class_j$ for an $j \in 1, \dots, k$ and $B = FieldClass_{i_j}$ for an $i_j \in \{1_j \dots, n_j\}$.

Obviously, a CNS is iteratively defined, if the transitive closure of this relation is irreflexive and the smallest elements with respect to this closure are the basic types. Both is important for the well-definition of a CNS:

Definition 3 (well-defined CNS). A CNS is well-defined if exactly one of its UML class diagrams is the schema of a constraint variable:

ConstraintVariable
min: Integer
max: Integer

and the transitive closure ' $>^+$ ' of the relation ' $>$ ' (cf. Definition 2)

- is irreflexive, i.e. $C \not>^+ C$ holds for any class diagram C in the CNS.
- the OCL basic types are the smallest elements; i.e. for any class C in the CNS it holds $C >^+ \text{Integer}$ or $C >^+ \text{Real}$, etc.

Furthermore, for the finitely many OCL equations

$$f_{1_i} \dots f_{u_i} = f'_{1_i} \dots f'_{v_i} \quad (1_k \leq i \leq m_k)$$

which are the invariants of the class diagrams Class_j ($1 \leq j \leq k$) it holds that

- the field identifiers f_{1_i} and f'_{1_i} are in the class diagram Class_j , or in detail $f_{1_i} = \text{field}_{p_j}$ and $f'_{1_i} = \text{field}_{q_j}$ for some $p_j, q_j \in \{1_j, \dots, n_j\}$,
- for any pair $f_l.f_{l+1}$ ($l = 1_i, \dots, u - 1_i$) where $f_l = \text{field}_{p_j}$ in the class diagram Class_j , the identifier f_{l+1} is a field identifier in the class diagram FieldClass_{p_j} ; the same holds for any pair $f'_l.f'_{l+1}$ ($l = 1_i, \dots, v - 1_i$),
- the field identifiers f_{u_i} and f'_{v_i} refer to the same data-type: if $f_{u_i} = \text{field}_{p_j}$ and $f'_{v_i} = \text{field}_{q_l}$ holds then their field classes in the class diagrams Class_j respective Class_l will be the same, i.e. $\text{FieldClass}_{p_j} = \text{FieldClass}_{q_l}$ and it either holds

$$\text{Class}_j :: \text{field}_{p_j} : \text{sequence of FieldClass}_{p_j}$$

and

$$\text{Class}_l :: \text{field}_{q_l} : \text{sequence of FieldClass}_{q_l}$$

or

$$\text{Class}_j :: \text{field}_{p_j} : \text{FieldClass}_{p_j} \quad \text{and} \quad \text{Class}_l :: \text{field}_{q_l} : \text{FieldClass}_{q_l}$$

Example 4. The CNS presented in Example 3 is already a well defined CNS:

- there are no recursions in the model,
- the binary relation $>^+$ is irreflexive, the class diagram CSP is the greatest element and the OCL basic type **Integer** is the smallest element,
- the OCL field identifiers are in the same class diagram, and
- `sum.addends` and `alldiff.values` refer to a sequence of **ConstraintVariables**.

3.3 Compatibility with a CNS

For the generation of a concrete constraint network out of some partial information, some given CNS requires a common semantics. Therefore, we have chosen the Herbrand algebra and a constraint system consisting of syntactical equations.

We assume that the partial information is a term which is compatible with a given well-defined CNS.

Definition 4 (Compatibility with a CNS).

Let a well-defined CNS be given where $Class_k$ is the greatest element with respect to the order relation ' $>^+$ '. Then, every class identifier $Class_j$ represents an n_j -ary function symbol, every field identifier $field_{i_j}$ as well as every OCL basic type Integer, Real, etc. an unary function symbol and there are the usual integer, real, etc. constants.

We define the compatibility condition \mathcal{C} for any given term t with respect to any class diagram named $Class_j$ ($1 \leq j \leq k$) in the following way:

$\mathcal{C}(t, Class_j)$ holds for $1 \leq j \leq k$ if and only if t is a variable or a structure

$$t := 'Class_j(field_{i_j}(t_1), \dots, field_{n_j}(t_n))$$

and for each term t_i with $1 \leq i \leq n$

- either the class diagram $Class_j$ has the structure $Class_j :: field_{i_j} : FieldClass_{i_j}$ and $\mathcal{C}(t_i, FieldClass_{i_j})$ holds,
- or the class diagram $Class_j$ has the structure $Class_j :: field_{i_j} : \text{sequence of } FieldClass_{i_j}$ and the term t_i is
 - either the empty list, i.e. $t_i := []$
 - or a non-empty list of terms, i.e. $t_i := [s_1, \dots, s_m]$ ($m > 0$) and the compatibility condition holds for its sub-terms and their field class diagrams: $\mathcal{C}(s_1, FieldClass_{i_j}), \dots, \mathcal{C}(s_m, FieldClass_{i_j})$.

A term t is compatible with the given well-defined CNS if and only if the compatibility condition $\mathcal{C}(t, Class_k)$ holds.

Example 5. Let the well-defined CNS presented in Example 3 be given. Then, the term $t := 'CSP(\text{sum}('UnitSum(\text{sum}(X), \text{addends}([U, V, W]))), \text{alldiff}(Z))$ with the variables X, U, V, W, Z is compatible to this CNS: Following Definition 4, this term is compatible because the compatibility condition holds for all its sub-terms:

- $\mathcal{C}('UnitSum(\text{sum}(X), \text{addends}([U, V, W]), UnitSum)$ holds because X is a variable and $[U, V, W]$ is a list of variables and the class diagram $UnitSum$ has the structure $UnitSum :: \text{addends} : \text{sequence of } ConstraintVariable$.
- $\mathcal{C}(Z, AllDifferent)$ holds because Z is a variable.

Proposition 1. For any well-defined CNS and any term t the compatibility condition \mathcal{C} is well-defined.

Proof. By induction over the term's depth. □

3.4 Term Extension

Now, given term t which is compatible with a well-defined CNS, we are able to extend this term with respect to this CNS such that all variables representing field objects are substituted by a corresponding term. Therefore we define the function **extend**:

Definition 5 (Term Extension). Let a well-defined CNS be given. Then for any term t which is compatible to this CNS and any two class diagrams C and D in this CNS we define $\text{extend}(t, C, D)$ as follows:

$$\begin{aligned} & \text{extend}('Class_j(\text{field}_{1_j}(t_1), \dots, \text{field}_{n_j}(t_{n_j}), Class_j, D) \\ & := 'Class_j(\text{field}_{1_j}(\text{extend}(t_1, FieldClass_{1_j}, Class_j), \dots, \\ & \quad \text{field}_{n_j}(\text{extend}(t_{n_j}, FieldClass_{n_j}, Class_j))) \end{aligned}$$

for any sub-terms t_1, \dots, t_{n_j} .

$$\begin{aligned} & \text{extend}(X, FieldClass_{i_j}, Class_j) \\ & := 'FieldClass_{i_j}(\text{field}_{1_{i_j}}(\text{extend}(Y_1, 'FieldClass_{1_{i_j}}, 'FieldClass_{i_j}), \dots, \\ & \quad \text{field}_{n_{i_j}}(\text{extend}(Y_{n_{i_j}}, 'FieldClass_{n_{i_j}}, 'FieldClass_{i_j}))) \end{aligned}$$

for any variable X if the class diagram $Class_j$ has the structure $Class_j :: \text{field}_{i_j} : FieldClass_{i_j}$. Here $Y_1, \dots, Y_{n_{i_j}}$ are new pairwise different variables.

$$\text{extend}(X, FieldClass_{i_j}, Class_j) := X$$

for any variable X if the class diagram $Class_j$ has the structure $Class_j :: \text{field}_{i_j} : \text{sequence of } FieldClass_{i_j}$.

In all other cases the value of $\text{extend}(t, C, D)$ is undefined.

Proposition 2. Let a well-defined CNS be given where $Class_k$ is the greatest element with respect to the order relation ' $>^+$ ' and a compatible term t . Then, the extended term $\text{extend}(t, Class_k, Class_k)$ is well-defined and compatible to the given CNS. Further, all variables in this term are in sub-terms representing lists or OCL basic types, i.e. any variable X is in a sub-term ' $Class_j(\dots, \text{field}_{i_j}(X), \dots)$ ' and the class diagram $Class_j$ has the structure $Class_j :: \text{field}_{i_j} : \text{sequence of } FieldClass_{i_j}$ or it is in a sub-term ' $Integer(X)$ ' or ' $Real(X)$ ', etc.

Proof. by induction over the term's depth using the properties of a well-defined CNS and the compatibility condition. \square

Example 6. Considering the well-defined CNS presented in Example 3 and the term t presented in Example 5 which compatible to this CNS. Then, the extended term

$$\begin{aligned} & \text{extend}(t, CSP, CSP) \\ & := 'CSP(\text{sum}('UnitSum(\text{sum}('ConstraintVariable(\min(X_1), \max(X_2))), \\ & \quad \text{addends}(['ConstraintVariable(\min(U_1), \max(U_2)), \\ & \quad \quad 'ConstraintVariable(\min(V_1), \max(V_2)), \\ & \quad \quad 'ConstraintVariable(\min(W_1), \max(W_2))])), \\ & \quad \text{alldiff}('AllDifferent(\text{values}(Z_1)))) \end{aligned}$$

is well-defined and compatible to the CNS, too. Further, the new variables $X_1, X_2, U_1, U_2, V_1, V_2, W_1, W_2$ represent values of the OCL basic type `Integer` and the new variable Z_1 represents a list.

3.5 Dissolving OCL equations

Considering the extension of a term which is compatible to a well-defined CNS, we are not yet able to derive the intended constraint network because the interrelations between the constraints and the constraint variables, esp. the sharing of variables, are not reflected. The reason is that the OCL equations in the CNS are not dissolved. Therefore, we define the semantics of a CNS's OCL equations, i.e. we transform them into a set of syntactical equations. Their satisfiability and solutions are effectively computable via unification and the determination of a possible most general unifier (mgu). The necessary transformation is performed in two steps:

1. the two sub-terms in the term t which are referred to by both sides an OCL equation are determined by the use of a function `dissolve`,
2. then, the determined sub-terms are syntactically equated.

In the following we define the function `dissolve`:

Definition 6 (Dissolving). *Let a well-defined CNS be given. Then for any term t which is compatible to this CNS, any sequence of field identifiers F in this CNS we define $\text{dissolve}(t, F)$ as follows:*

$$\text{dissolve}(X, f) := X$$

for any variable X and any field identifier f .

$$\text{dissolve}('Class_j(\text{field}_{i_j}(t_1), \dots, \text{field}_{n_j}(t_{n_j})), \text{field}_{i_j}) := t_{i_j}$$

for any term $'Class_j(\text{field}_{i_j}(t_1), \dots, \text{field}_{n_j}(t_{n_j}))$ and any field identifier f .

$$\text{dissolve}([], f) := []$$

$$\text{dissolve}([s_1, \dots, s_m], f) := \text{flatten}([\text{dissolve}(s_1, f), \dots, \text{dissolve}(s_m, f)])$$

for any terms s_1, \dots, s_m and any field identifier f . Here, `flatten` flattens possibly nested lists such that the lists' elements are the elements of one common list keeping their order, e.g. $\text{flatten}([a, [b]], [[c, d], e]) = [a, b, c, d, e]$.

$$\text{dissolve}(t, \text{field}_{i_j}.f_2 \dots f_n) := \text{dissolve}(\text{dissolve}(t, \text{field}_{i_j}), f_2 \dots f_n)$$

for any term t and any non-empty sequence of field identifiers $f_2 \dots f_n$.

In all other cases the value of $\text{dissolve}(t, F)$ is undefined.

Proposition 3. *Let a well-defined CNS and a term t be given that satisfies the compatibility condition $\mathcal{C}(t, Class_j)$ for a class diagram $Class_j$ in this CNS. Then for any OCL invariant $f_1 \dots f_u = f'_1 \dots f'_v$ in this class diagram*

$$\text{dissolve}(t, f_1 \dots f_u) \quad \text{and} \quad \text{dissolve}(t, f'_1 \dots f'_v)$$

are well-defined.

Proof. by induction over the term's depth using the properties of a well-defined CNS and the compatibility condition. \square

Example 7. Considering the well-defined CNS presented in Example 3 and the extended term $t' := \text{extend}(t, \text{CSP}, \text{CSP})$ determined in Example 6. If we further consider the invariant of the class diagram CSP, i.e. the OCL equation $\text{sum.addends} = \text{alldiff.values}$ then it will hold

$$\begin{aligned} \text{dissolve}(t', \text{sum.addends}) &= ['\text{ConstraintVariable}(\min(U_1), \max(U_2)), \\ &\quad '\text{ConstraintVariable}(\min(V_1), \max(V_2)), \\ &\quad '\text{ConstraintVariable}(\min(W_1), \max(W_2))] \\ \text{dissolve}(t', \text{alldiff.values}) &= Z_1 \end{aligned}$$

Based on this important property, we are now able to consider the OCL equations adequately. Therefore we “extract” all syntactical equations from the constraint network's term representation. Then, after the determination of a possible, most general solution it is only a small step to a concrete constraint network.

Definition 7 (OCL Extensions). *Let a well-defined CNS be given where Class_k is the greatest element with respect to the order relation ' $>^+$ '. Further, let a term t be given which is compatible to this CNS. Then, the extension $t' := \text{extend}(t, \text{Class}_k, \text{Class}_k)$ has the structure*

$$t' := '\text{Class}_k(\text{field}_{1_k}(s_1), \dots, \text{field}_{n_k}(s_{n_k}))$$

where s_1, \dots, s_{n_k} are sub-terms.

Now, for all OCL equations $f_{1_i} \dots f_{u_i} = f'_{1_i} \dots f'_{v_i}$ ($1_k \leq i \leq m_k$) which are the invariants of the class diagram Class_k we compute the pairs of terms

$$\begin{aligned} p_i &:= \text{dissolve}(t', f_{1_i} \dots f_{u_i}) \\ q_i &:= \text{dissolve}(t', f'_{1_i} \dots f'_{v_i}) \end{aligned}$$

Then, for the set of syntactical equations

$$\bigcup_{1_k \leq i \leq m_k} \{p_i \doteq q_i\}$$

we compute a mgu λ , if it exists; otherwise there is no OCL extension of the term t – respective of its extension t' – which is compatible to the given CNS. If λ exists, we will apply this process recursively to all the sub-terms

$$\lambda(s_1), \dots, \lambda(s_{n_k})$$

and for all OCL equations which are the invariants of the field class diagrams $\text{FieldClass}_{1_k}, \dots, \text{FieldClass}_{n_k}$.

If this process terminates without returning the information that no OCL extension exists, there will be a sequence of substitutions; let their combination be the substitution θ . In this case, the extension $\theta(t')$ of the term t' and of the term t will be called an OCL extension of t with respect to the given CNS.

The generation process in Definition 7 terminates, because the functors of the generated sub-terms are smaller than the functor of the aggregating term with respect to the order relation $>^+$. Furthermore, the extension $\theta(t)$ of t is by construction compatible to the given well-defined CNS.

Example 8. Considering the well-defined CNS presented in Example 3 and the extended term $t' := \text{extend}(t, \text{CSP}, \text{CSP})$ determined in Example 6. Then, from Example 7 we know that

$$Z_1 \doteq [\text{'ConstraintVariable}(\min(U_1), \max(U_2)), \\ \text{'ConstraintVariable}(\min(V_1), \max(V_2)), \\ \text{'ConstraintVariable}(\min(W_1), \max(W_2))]$$

is the only syntactical equation to be solved which results from the OCL equation $\text{sum.addends} = \text{alldiff.values}$. Its solution is quite simple, the mgu is

$$\lambda := \{Z_1 \mapsto [\text{'ConstraintVariable}(\min(U_1), \max(U_2)), \\ \text{'ConstraintVariable}(\min(V_1), \max(V_2)), \\ \text{'ConstraintVariable}(\min(W_1), \max(W_2))]\}$$

Thus, the term

$$\lambda(t') := \text{'CSP}(\text{sum}(\text{'UnitSum}(\text{sum}(\text{'ConstraintVariable}(\min(X_1), \max(X_2))), \\ \text{addends}([\text{'ConstraintVariable}(\min(U_1), \max(U_2)), \\ \text{'ConstraintVariable}(\min(V_1), \max(V_2)), \\ \text{'ConstraintVariable}(\min(W_1), \max(W_2))])), \\ \text{alldiff}(\text{'AllDifferent}(\text{values}([\text{'ConstraintVariable}(\min(U_1), \max(U_2)), \\ \text{'ConstraintVariable}(\min(V_1), \max(V_2)), \\ \text{'ConstraintVariable}(\min(W_1), \max(W_2))]))))$$

is an OCL extension of the term t with respect to the considered CNS.

Concluding, the presented examples show the extension of some partial information about a constraint network using a formally specified schema such that in a further step a concrete constraint network is constructible. Therefore, only the variables in the OCL extension have to be replaced by default values, e.g. the min's and max's in the `ConstraintVariables`. Then, the `ConstraintVariables` as well as the basic constraints `AllDifferent` and `UnitSum` have to be generated, respecting the sharing of common objects defined by the most general unifier(s).

4 Future work

At this point we have described how to represent constraint networks using UML and OCL. We showed how missing parts of data-driven models can be added

by the use of a given schema and how data-driven constraint network creation can be achieved. Future work focuses on the solution of these networks. We plan to present a first implementation of a constraint network parser that is able to interpret CNS, to automatically generate a CSP and to solve it. Then, after performing the possibility of generating and solving a constraint network automatically, we will focus our work on “model optimizations”. We believe that the symmetry breaking and other optimization can be either performed explicitly in the formal model or later automatically via model transformation. Another challenge is the integration of adequate search and propagation strategies into the formal model. Or even better, to define analysis processes which choose a good strategy automatically.

On the long term we are willing to make CNS available in information systems. The development of a web service interface, cf. [1], will be the natural extension of constraint network schemata. In doing so, we will most likely stick again to standard techniques like WSDL and SOAP. High level modelling accommodates not only the end-user but makes it also an appropriate foundation for automatic interaction between information systems.

Last but not least, we have to add some well-founded extensions to the formal definitions presented here, e.g. domain restrictions like $\min > 0$.

References

1. D. Seipel B. Heumesser, A. Ludwig. Web services based on prolog and xml. In *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management – INAP 2004*, pages 369–378, March 4 - March 6 2004.
2. Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs - A Pure Java Constraint Programming Engine. In Michael Hanus, Petra Hofstedt, and Armin Wolf, editors, *2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL’03*, 29th September 2003.
3. Object Management Group, Inc. *OMG Unified Modelling Language Specification*, March 2003.
4. Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In Marc Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004, 10th International Conference, Proceedings*, number 3258 in Lecture Notes in Computer Science, pages 5–8, Toronto, Canada, September/October 2004. Springer-Verlag.
5. Hans Schlenker and Georg Ringwelski. Pooc - a platform for object-oriented constraint programming. In B. O’Sullivan, editor, *Recent Advances in Constraints – Selected Papers of the Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2002*, number 2627 in Lecture Notes in Artificial Intelligence, pages 159–170. Springer Verlag, 2003.
6. Toby Walsh. Constraint patterns. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, Proceedings*, number 2833 in Lecture Notes in Computer Science, pages 53–64, Kinsale, Ireland, September/October 2003. Springer-Verlag.
7. Jos B. Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition*. Object Technology Series. Addison-Wesley, 2003.

Expressing Interaction in Combinatorial Auction through Social Integrity Constraints [★]

Marco Alberti¹, Federico Chesani², Marco Gavanelli¹, Alessio Guerri², Evelina Lamma¹, Michela Milano², and Paolo Torroni²

¹ ENDIF - Università di Ferrara - Via Saragat, 1 - 44100 Ferrara, Italy.
{malberti|m gavanelli|elamma}@ing.unife.it

² DEIS - Università di Bologna - Viale Risorgimento, 2 - 40136 Bologna, Italy.
{fchesani|aguerri|mmilano|ptorroni}@deis.unibo.it

Abstract. Combinatorial auctions are an interesting application of intelligent agents. They let the user express *complementarity* relations among the items for sale, and let the seller obtain higher revenues. On the other hand, the solving process, the so-called Winner Determination Problem (WDP) is NP-hard. This restricted the practical use of the framework, because of the fear to be, in some WDP instances, unable to meet reasonable deadlines. Recently, however, efficient solvers have been proposed, so the framework starts to be viable.

A second issue, common to many agent systems, is *trust*: in order for an agent system to be used, the users must *trust* both their representative and the other agents inhabiting the society. The SOCS project addresses such issues, and provided a language, the *social integrity constraints*, for defining the allowed interaction moves, and a proof-procedure able to detect violations.

In this paper we show how to write a protocol for combinatorial auctions by using social integrity constraints. In the devised protocol, the auctioneer interacts with an external solver for the winner determination problem. We also suggest extensions of the scenario, with more auctions in a same society, and suggest to verify whether two auctions interact. We also apply the NetBill protocol for the payment and delivery scheme.

1 Introduction

Auctions have been practically used for centuries in human commerce, and their properties have been studied in detail from economic, social and computer science viewpoints. The raising of electronic commerce has pushed auctions as one

[★] A preliminary version of this paper appeared in [1]. This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32530 project in the context of the Global Computing initiative of the FET (Future Emerging Technology) initiative and by the MIUR COFIN 2003 projects *La Gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici* and *Sviluppo e verifica di sistemi multiagente basati sulla logica*.

of the favourite dealing protocols in the Internet. Now, the software agent technology seems an attractive paradigm to support auctions [2]: agents acting on behalf of end-users could reduce the effort required to complete auction activities. Agents are intrinsically autonomous and can be easily personalised to embody end-user preferences. As the rise of the Internet and electronic commerce continues, dynamic automated markets will be an increasingly important domain for agents.

Combinatorial auctions are types of auctions that give more expressiveness to the bidders: in fact, bidders can place bids on sets of items, expressing complementarity and, in some cases, also substitutability among the items [3,4]. The main drawback is that determining the winners is an NP-hard problem. This delayed the practical applications of combinatorial auctions, mainly for fear not to meet the given deadlines. Recently, however, solvers able to solve the winner determination problem in reasonable time have been developed.

Of course, another issue common to many e-commerce applications is *trust* [5]. Amongst the various aspects of trust in MASs (often related to credibility levels between agents), we find utterly important that human users trust their representatives: in order for the system to be used at all, each user must trust its representative agent in the auction. The agent must be well specified, and a formal proof of a correspondence between specification and implementation is, at least, desirable. Also, even if the agents are compliant to their specifications, the compliance to the social rules and protocols must be provable, in order to avoid, or, at least, detect malicious behaviours.

A typical answer to such issues is to model-check the agents with respect to both their specifications and requirements coming from the society. However, this is not always possible in open environments: agents could join the society at all times and their specifications could be unavailable to the society. Thus, the correct behaviour of agents can be checked only from the external in an open environment: by monitoring the communicative actions of the agents.

The *SOCS* project addresses these issues by providing formal definitions both for the agents, that are based on Computational Logics, and are thus called *Computees*, and for the society in an open environment.

In this paper, we focus on the societal aspects, and on the compliance of the computees (or, in general, agents) to protocols and social rules. These can be easily expressed in a logic language, the *Social Integrity Constraints* (ic_S) that are an extension of the integrity constraints widely used in Abductive Logic Programming, and, in particular, extend those of the IFF proof-procedure [6].

We implemented an abductive proof-procedure, called *SCIFF* (extending the IFF), that is able to check the compliance to protocols and social rules given a history of communicative actions. Besides a posteriori check of compliance, *SCIFF* also accepts dynamically incoming events, so it can check compliance during the evolution of the societal interaction, and raise violations as soon as possible. *SCIFF* extends the IFF in a number of directions: it provides a richer syntax, it caters for interactive event assimilation, it supports fulfillment check and violation detection, and it embodies CLP-like constraints [7] in the

ic_S . **SCI**FF is sound [8] with respect to the declarative semantics of the society model, in its abductive interpretation. The **SCI**FF has been implemented and integrated into a Java-Prolog-CHR based tool [9].

In this paper, we show a definition of the combinatorial auction protocol in Social Integrity Constraints. Since the solving process is NP-hard, we exploit an efficient solver for the Winner Determination Problem. Finally, we propose to extend the framework to check the compliance of two interacting auctions, in a double auction scheme.

The paper is the extended version of a previous informal publication [1]. In Section 2 we recall the *SOCS* social model. We describe the combinatorial auction scenario in Section 3). We present new work on extensions of the given scenario (Section 4.1), and on experimentation (Section 5). We cite some related work and, finally, we conclude.

2 *SOCS* social model

We sketch, for the sake of readability, the *SOCS* social model; the reader is referred to previous publications for more details on the syntax and semantics of the language [10,11]. More details can also be found in another paper in this same volume [12].

The society knowledge is physically memorised in a device, called the *Society Infrastructure*, that has reasoning capabilities and can use the society knowledge to infer new information. We assume that the society infrastructure is time by time aware of social events that dynamically happen in the environment (*happened* events). They are represented as ground atoms $\mathbf{H}(\textit{Description}[, \textit{Time}])$.

The knowledge in a society is given by:

- a *Social Organisation Knowledge Base (SOKB)*: an abductive logic program with constraints;
- a set \mathcal{IC}_S of *Social Integrity Constraints (ic_S)*: implications that can relate dynamic elements, CLP constraints and predicates defined in the SOKB.

The “normative elements” are encoded in the ic_S . Based on the available history of events, and on the ic_S -based specification, the society can define what the “expected social events” are, i.e., what events are expected (*not*) to happen. The expected events, called *social expectations*, reflect the “ideal” behaviour of the agents. Social expectations are represented as atoms $\mathbf{E}(\textit{Description}[, \textit{Time}])$ for events that are expected to happen and as $\mathbf{EN}(\textit{Description}[, \textit{Time}])$ for events expected not to happen. Explicit negation can be applied to expectations, letting the user express concepts like *possibility* ($\neg\mathbf{EN}$) and *optionality* ($\neg\mathbf{E}$).

While \mathbf{H} atoms are always ground, the arguments of expectations can contain variables. Intuitively, an $\mathbf{E}(X)$ atom indicates a wish about an event $\mathbf{H}(Y)$ which unifies with it: X/Y . CLP constraints [7] can be imposed on the variables occurring in expectations, in order to refine the meaning of the expectation, and to improve the propagation. For instance, in an auction context the atom:

$$\mathbf{E}(\textit{tell}(\textit{Bidder}, \textit{Auctioneer}, \textit{bid}(\textit{ItemList}, \textit{Price}), D), T_{\textit{bid}}, T_{\textit{bid}} < 10$$

stands for an expectation about a communicative act *tell* made by a computee *Bidder*, addressed to another computee *Auctioneer*, at a time T_{bid} , with subject $bid(ItemList, Price)$. The expectation is fulfilled if a matching event actually happens, satisfying the imposed constraints (i.e., the deadline $T_{bid} < 10$). D is a dialogue identifier, that can be useful to distinguish different instances of the same interaction scheme.

Negative expectations, instead, suggest actions that should not happen, in order for the protocol to be fulfilled. For this reason, their variables are quantified universally (if they do not occur in positive expectations as well). Constraints can be imposed also on universally quantified variables, through a solver we implemented. For example, $\mathbf{EN}(Bidder, Auctioneer, bid(ItemList, Price), D), T_{bid}, T_{bid} > 10$ means that no bidder is supposed to send any bid to any auctioneer after time 10. Another paper in this book [12] describes the implementation of the proof-procedure that gets happened events and raises such expectations.

3 The Combinatorial Auctions scenario

There exist different kinds of combinatorial auctions. In this paper, we consider *single unit reverse auctions*. In a *single unit auction*, the auctioneer wants to sell a set M of goods/tasks maximising the profit. Goods are distinguishable. Each bidder j posts a bid B_j where a set S_j of goods/tasks $S_j \subseteq M$ is proposed to be bought at the price p_j , i.e., $B_j = (S_j, p_j)$. In a *reverse auction*, the auctioneer tries to buy a set of goods minimising the total cost.

3.1 The Auction Solver

Besides the usual constraints of a combinatorial auction (i.e., two winning bids cannot have elements in common), some real-life auction scenarios also have the so called *side constraints*: other constraints that should be satisfied by the winning bids. One typical example is when the auctioneer needs to allocate tasks, that have precedence relations. Bidders propose to execute (bunches of) tasks, each with an associated time window, at a given price. The auctioneer will try to find a set of tasks that will cover the whole manufacturing process satisfying the time precedence constraints and minimising the cost.

The Winner Determination Problem in combinatorial auctions is NP-hard [13], so it cannot be addressed naively, but we need to exploit smart solving techniques. While the pure WDP is best solved with an Integer Programming (IP) solver [14], adding side constraints makes a Constraint Programming (CP) solver more appealing.

We address the problem by exploiting a module called *Auction solver* [15] that embeds two different algorithms both able to solve efficiently the WDP: one is a pure IP-based approach, and the other is an Hybrid approach based on a CP model with a variable selection heuristic based on the variables reduced costs deriving from the Linear Relaxation of the IP model. For a complete description of the IP and CP models, see [16].

The results obtained using the two algorithms strongly depend on the instance structure. The module embeds an automatic Machine Learning based portfolio selection algorithm, able to select the best algorithm on the basis of few structural features of the problem instance. Guerri and Milano [16] show that the method is able to select the best algorithm in the 90% of the cases, and that the time spent to decide which of the available algorithms fits best with the current instance is always orders of magnitude lower with respect to the search time difference between the two algorithms. This is a fundamental assumption; in fact, if the sum of the times used to extract the features and to solve the problem with the best algorithm was greater than the time used by the worst algorithm to solve the same problem, the selection tool would be useless.

We now give the general auction protocol in terms of ic_S in Section 3.2.

3.2 Auction Protocol

We first describe the auction protocol, then we give its implementation with social integrity constraints. The auction is declared open by

$$\mathbf{H}(\text{tell}(Auc, Bidders, \text{openauction}(ItemList, T_{end}, T_{deadline}), D), T_{open}),$$

containing, as parameters, the deadlines for sending valid bids (T_{end}), and for the winners declaration ($T_{deadline}$). In an open society, bidders can come without registration, so the addressee of the *openauction* is not fundamental (bidders could join even if not explicitly invited with an *openauction*).

After the *openauction*, each bidder can place bids, i.e., declare the subset of the items it is interested in (*ItemList*), and the price (P) it is willing to pay:

$$\mathbf{H}(\text{tell}(Bidder, Auc, \text{bid}(ItemList, P), D), T_{bid}).$$

Finally, the auctioneer replies to each of the bidders either *win* or *lose*:

$$\mathbf{H}(\text{tell}(Auc, Bidder, \text{answer}(\text{win/lose}, Bidder, ItemList, P), D), T_{answer}).$$

The auction protocol in Social Integrity Constraints. Each time a bidding event happens, the auctioneer should have sent an *openauction* event:

$$\begin{aligned} &\mathbf{H}(\text{tell}(Bidder, Auc, \text{bid}(-, -), D), T_{bid}) \rightarrow \\ &\mathbf{E}(\text{tell}(Auc, -, \text{openauction}(-, T_{end}, -), D), T_{open}) \wedge T_{open} < T_{bid} \leq T_{end} \end{aligned} \quad (1)$$

Incorrect bids always lose; e.g., a bid for items not for sale must lose.

$$\begin{aligned} &\mathbf{H}(\text{tell}(Auc, -, \text{openauction}(Items, -, -), D), -) \wedge \\ &\mathbf{H}(\text{tell}(Bidder, Auc, \text{bid}(ItemBids, P), D), -) \wedge \\ &\text{not included}(ItemBids, Items) \\ &\rightarrow \mathbf{E}(\text{tell}(Auc, Bidder, \text{answer}(\text{lose}, Bidder, ItemBids, P), D), -) \end{aligned} \quad (2)$$

$$\text{included}([], -).$$

$$\text{included}([H|T], L) : \neg \text{member}(H, L), \text{included}(T, L).$$

The auctioneer should answer to each bid within the deadline $T_{deadline}$.

$$\begin{aligned}
& \mathbf{H}(\text{tell}(\text{Bidder}, \text{Auc}, \text{bid}(\text{ItemList}, P), D), -) \wedge \\
& \mathbf{H}(\text{tell}(\text{Auc}, -, \text{openauction}(-, T_{end}, T_{deadline}), D), -) \\
& \rightarrow \mathbf{E}(\text{tell}(\text{Auc}, \text{Bidder}, \text{answer}(\text{Ans}, \text{Bidder}, \text{ItemList}, P), D), T_{answer}) \quad (3) \\
& \quad \wedge T_{answer} > T_{end} \wedge T_{answer} < T_{deadline} \wedge \text{Ans} :: [\text{win}, \text{lose}]
\end{aligned}$$

A bidder should not receive for the same bid two conflicting answers:

$$\begin{aligned}
& \mathbf{H}(\text{tell}(\text{Auc}, \text{Bidder}, \text{answer}(\text{Ans}_1, \text{Bidder}, \text{ItemList}, P), D), -) \\
& \rightarrow \mathbf{EN}(\text{tell}(\text{Auc}, \text{Bidder}, \text{answer}(\text{Ans}_2, \text{Bidder}, \text{ItemList}, P), D), -) \quad (4) \\
& \quad \wedge \text{Ans}_1 \neq \text{Ans}_2
\end{aligned}$$

Two different winning bids cannot contain the same item:

$$\begin{aligned}
& \mathbf{H}(\text{tell}(\text{Auc}, \text{Bidder}_1, \text{answer}(\text{win}, \text{Bidder}_1, \text{ItemList}_1, -), D), -) \\
& \wedge \mathbf{H}(\text{tell}(\text{Bidder}_2, \text{Auc}, \text{bid}(\text{ItemList}_2, P_2), D), -) \\
& \wedge \text{Bidder}_1 \neq \text{Bidder}_2 \wedge \text{intersect}(\text{ItemList}_1, \text{ItemList}_2) \\
& \rightarrow \mathbf{EN}(\text{tell}(\text{Auc}, \text{Bidder}_2, \text{answer}(\text{win}, \text{Bidder}_2, \text{ItemList}_2, P_2), D), -) \quad (5)
\end{aligned}$$

$$\begin{aligned}
& \text{intersect}([X|_], L) : -\text{member}(X, L). \\
& \text{intersect}([_|Tx], L) : -\text{intersect}(Tx, L).
\end{aligned}$$

4 Extensions

4.1 Double combinatorial auction

We extended the scenario to allow for multiple auctions in a same society. We assume that all the items are labelled with a unique name. Clearly, in such a situation, we must ensure that a computee will not bid (i.e., try to sell) the same item in two different auctions, as he will be able to provide only one. We add to the previous *ic_S* (1) to (5), the following:

$$\begin{aligned}
& \mathbf{H}(\text{tell}(B, A1, \text{bid}(\text{ItemList1}, P1), D1), -) \wedge \\
& \mathbf{H}(\text{tell}(B, A2, \text{bid}(\text{ItemList2}, P2), D2), -) \wedge \\
& D1 \neq D2 \wedge \text{intersect}(\text{ItemList1}, \text{ItemList2}) \rightarrow \\
& \text{false}. \quad (6)
\end{aligned}$$

This allows for interesting bidding strategies and behaviours of the computees, such as the *double combinatorial auction*. Suppose, for example, that a computee opens an auction for a set of items. Another computee, that owns most of the requested goods but not all of them, could try to make its bid more appealing, by trying to get all of the requested items, and sell them all together. Think, for example, to a collection of items, like comics: the price that a bidder can be able to obtain is much higher if he sells a complete collection, so he will try to buy the missing issues. Then, the bidder may become auctioneer of a second auction. Possible bidders know that the second auction is more appealing,

because the bidder that misses only a few issues will probably be prepared to pay more for the few missing issues than the first auctioneer.

In such a scenario, the society must ensure that a bidder will not try to sell an item he still does not own, so the end of the second auction should not be later than the time allowed in the first auction for placing bids; i.e.:

$$\begin{aligned} & \mathbf{H}(\text{tell}(A, B, \text{openauction}(\text{Items1}, T_{\text{end1}}, T_{\text{deadline1}}), -), -) \wedge \\ & \mathbf{H}(\text{tell}(B, C, \text{openauction}(\text{Items2}, T_{\text{end2}}, T_{\text{deadline2}}), -), -) \wedge \\ & \text{intersect}(\text{Items1}, \text{Items2}) \wedge T_{\text{deadline2}} \geq T_{\text{end1}} \rightarrow \\ & \text{false.} \end{aligned} \tag{7}$$

4.2 Combinatorial auction with Netbill

This protocol extends the tradition auction protocol with the payment at the end. NetBill [17] is a security and transaction protocol for the sale and delivery of low-priced information goods, such as software or journal articles. The protocol rules transactions between two actors: *merchant* and *customer*. Accounts for merchants and customers, linked to traditional financial accounts (like credit cards), are maintained by a NetBill server. The implementation of the NetBill protocol in Social Integrity Constraints can be found in [18].

In our case, it was possible to simply add the *ic_S* defining the NetBill protocol to the auction integrity constraints (1) to (5).

5 Experiments

We performed different types of experiments in the combinatorial auction scenario. First, we have tested the Auction Solver implemented in ILOG to test its efficiency for increasing number of bidders. Then, we experimented the proof-procedure with the protocols defined in Sections 3 and 4.

All the experiments have been performed on a Pentium 4, 2.4 GHz, 512 MB.

5.1 Experiments on the Auction Solver

In the Combinatorial Auction scenario we have to cope with a complex combinatorial optimisation problem: the Winner Determination Problem. Having an efficient, scalable and flexible tool that solves this problem is crucial for the efficiency of the overall system.

In this experiment, we exploit an Auction Solver implemented in ILOG solver [19] suitably wrapped in to Java. The auction solver is able to solve both winner evaluation problems with and without temporal side constraints.

Although the focus of this paper is more on showing the feasibility of such an implementation than comparing with existing platforms, we have some comparisons (shown in Figure 5.1) of the Auction Solver with Magnet [20], both with respect to the Simulated Annealing (SA) and with the Integer Programming (IP) models of Magnet. In [15] more extensive experimentation is reported,

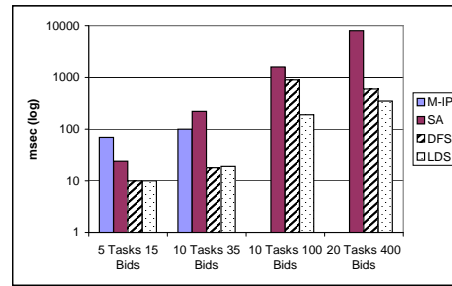


Fig. 1. Comparison with Magnet

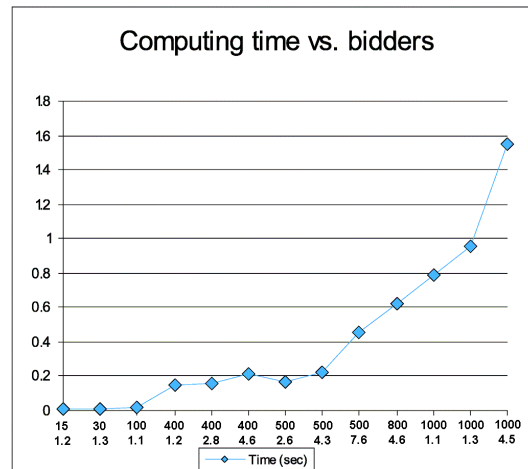


Fig. 2. Test of the Auction Solver performance

showing that the Auction Solver module outperforms, both in search time and in anytime solution quality, any other commercial solver in a WDP with temporal precedence constraints.

The Auction Solver is so efficient that it gives the possibility of scaling the auction size, and test the performances of the SOCS social infrastructure. Results are reported in Figure 2. We can see that auctions with 1000 bidders can be solved in less than 2 seconds. In the graph, we report the number of bids and the average number of tasks per bid. The results refer to the time for finding the optimal solution plus the proof of optimality.

5.2 Experiments on the SCIFF proof-procedure

We have arranged a set of experiments where the proof-procedure performances are tested for an increasing number of bidders. Bidders in fact send messages

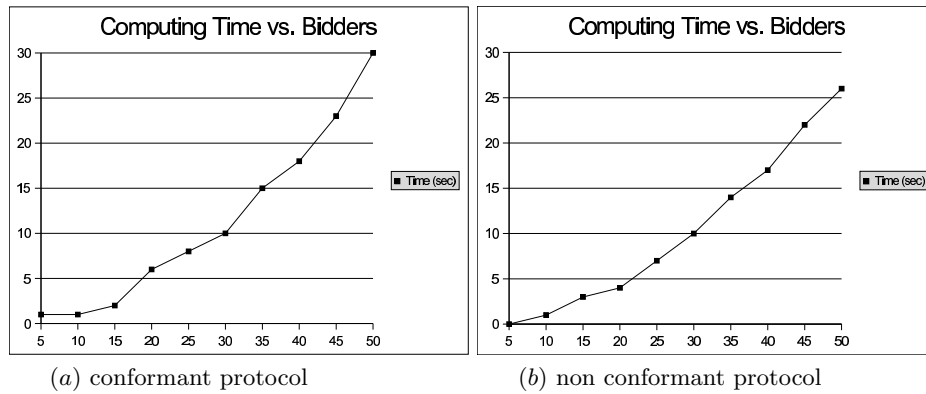


Fig. 3. SCIFF performance on a combinatorial auction

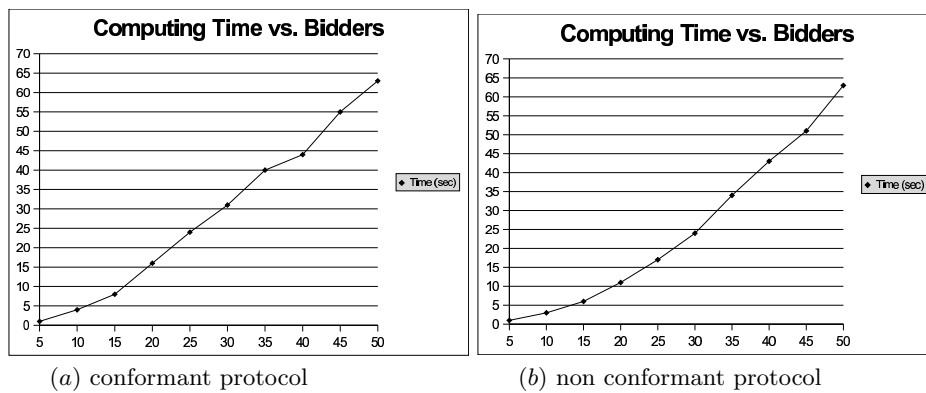


Fig. 4. SCIFF performance on a double auction

which should be checked for conformance by the society and the more the number of bidders the more the number of messages to control. We have tested the two protocols described previously, plus an implementation that also considers the actual selling of the goods, through the NetBill protocol.

Concerning the single auction, we can see in Figure 3 that the tests provide an idea on how the proof scales for an increasing number of bidders and consequently of messages. We can see that results are good, since the prototype we implemented works well up to 50 bidders answering in half a minute.

As far as the double auction is concerned, it is intuitive that the number of exchanged messages is almost doubled with respect to a traditional combinatorial auction. We can see from the Figure 4 that the proof scales very well, being the time for testing conformance almost doubled with respect to a single auction.

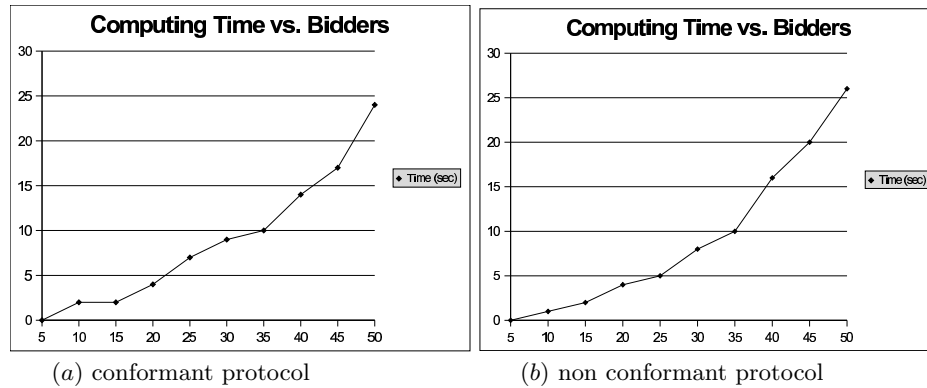


Fig. 5. SCIFF performance on an auction plus NetBill

As far as the auction plus the NetBill protocol, results are very good (Figure 5). In fact, the time for checking conformance is similar to that of the combinatorial auction.

As we can see from Figure 2 the Auction Solver is far more efficient since it scales up to 1000 bids within 2 seconds. Slower performances are achieved by the conformance checking of the society protocol (around 30 seconds for checking messages exchanged in auctions with 50 bidders). However, even if the components have different performances, from the testing, we can conclude that both components can be used in a real combinatorial auction scenario since time limits required for answering are much larger than sum of the answer times of the components.

Note that the implementation of the protocol currently considers only the combinatorial auction without temporal constraints. The full implementation with temporal constraints is left for future work.

6 Related Work

Different systems and methods have been proposed to solve a combinatorial auction in an efficient way, including dynamic programming techniques [13], approximate methods that look for a reasonably good allocation of bids [21,22], integer programming techniques [20,3], search algorithms [4].

Various works are related to the SCIFF proof-procedure and the checking of compliance to protocols; a full discussion can be found in previous publications [10,11]. We will cite here ISLANDER [23], a tool to specify protocols in a system ruled by electronic institutions that has been applied to a Dutch auction (and other scenarios). Their formalism is multi-levelled: agents have *roles*, agents playing a role are constrained to follow *protocols* when they belong to a *scene*; agents can move from a scene to another by means of *transitions*. Protocols are defined by means of transition graphs, in a finite state machine. Our definition

of protocols is wider than finite state machines, and leaves more freedom degrees to the agents. In our model, an event could be *expected to happen*, *expected not to happen* or have no expectations upon, thus there can be three possible values, while in finite state machines there are only two. Also, we focused on *combinatorial* auctions; this provides nice features, widely documented in the literature, but also makes the solving problem NP-hard. For this reason, a general purpose proof-procedure that checks the compliance to the protocol could be inefficient. We proposed a specialised solver and integrated it in our system.

7 Conclusions

Combinatorial auctions are recently starting to withdraw from the set of *practically unusable* applications as more efficient solvers are being produced for the winner determination problem. One of their natural applications involve intelligent agents as both bidders and auctioneers, but this raises the problem of humans trusting their representatives, and the other agents in the society.

Through the tools provided by the *SOCS* project, we give means for the user to specify the fair and trusty behaviour, and a proof-procedure for detecting the unworthy and fallacious one. We defined the combinatorial auctions protocol through social integrity constraints, also exploiting an efficient solver for the winner determination problem.

In future work, we will try other interaction schemes between the auction solver and the auctioneer agent; e.g., by having a centralised auction solver that serves more auctioneers. We are also interested in performing an extensive experimentation, to find how many auctioneers an auction solver can serve.

References

1. Alberti, M., Chesani, F., Gavanelli, M., Guerri, A., Lamma, E., Mello, P., Torroni, P.: Expressing interaction in combinatorial auction through social integrity constraints. In: Atti del Nono Convegno dell'Associazione Italiana per l'Intelligenza Artificiale, Perugia, Italia (2004)
2. Chavez, A., Maes, P.: Kasbah: An agent marketplace for buying and selling goods. In: Proc. of the 1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96), London (1996) 75–90
3. Nisan, N.: Bidding and allocation in combinatorial auctions. [24] 1–12
4. Sandholm, T.: Algorithm for optimal winner determination in combinatorial auction. *Artificial Intelligence* **135** (2002) 1–54
5. Marsh, S.: Trust in distributed artificial intelligence. In Castelfranchi, C., Werner, E., eds.: *Artificial Social Societies*. Number 830 in LNAI, Springer-Verlag (1994)
6. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. *Journal of Logic Programming* **33** (1997) 151–165
7. Jaffar, J., Maher, M.: Constraint logic programming: a survey. *Journal of Logic Programming* **19-20** (1994) 503–582

8. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of interaction protocols: a computational logic approach based on abduction. Technical Report CS-2003-03, Dipartimento di Ingegneria di Ferrara, Ferrara, Italy (2003) Available at <http://www.ing.unife.it/informatica/tr/>.
9. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based tool. In Trappl, R., ed.: Proc. of the 17th European Meeting on Cybernetics and Systems Research, Austrian Society for Cybernetic Studies (2004) 570–575
10. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An Abductive Interpretation for Open Societies. In Cappelli, A., Turini, F., eds.: AI*IA 2003: Advances in Artificial Intelligence. Volume 2829 of LNAI, Springer-Verlag (2003) 287–299
11. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Modeling interactions using *Social Integrity Constraints*: A resource sharing case study. In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: Declarative Agent Languages and Technologies. Volume 2990 of LNAI. Springer-Verlag (2004) 243–262
12. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E.: The CHR-based implementation of a system for generation and confirmation of hypotheses. In Wolf, A., ed.: 19th Workshop on (Constraint) Logic Programming. (2005) This volume.
13. Rothkopf, M., Pekec, A., R.M.Harstad: Computationally manageable combinatorial auctions. *Management Science* **44** (1998) 1131–1147
14. Sandholm, T., Suri, S., Gilpin, A., Levine, D.: Cabob: A fast optimal algorithm for combinatorial auctions. In Nebel, B., ed.: Proc. of IJCAI 01, (Morgan Kaufmann)
15. Guerri, A., Milano, M.: Exploring CP-IP based techniques for the bid evaluation in combinatorial auctions. In Rossi, F., ed.: Principles and Practice of Constraint Programming. Volume 2833 of LNCS, Springer Verlag (2003) 863–867
16. Guerri, A., Milano, M.: Learning techniques for automatic algorithm portfolio selection. In Lopez de Mantaras, R., Saitta, L., eds.: Proc. of ECAI. (2004)
17. Cox, B., Tygar, J., Sirbu, M.: NetBill security and transaction protocol. In: Proc. of the 1st USENIX Workshop on Electronic Commerce, New York (1995)
18. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interactions using social integrity constraints. *Electronic Notes in Theoretical Computer Science* **85** (2003)
19. ILOG S.A. France: ILOG Solver. 5.0 edn. (2003)
20. Collins, J., Gini, M.: An integer programming formulation of the bid evaluation problem for coordinated tasks. In Dietrich, B., Vohra, R.V., eds.: Mathematics of the Internet: E-Auction and Markets. Volume 127 of IMA Volumes in Mathematics and its Applications. Springer-Verlag, New York (2001) 59–74
21. Fujishima, Y., Leyton-Brown, K., Shoham, Y.: Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In Dean, T., ed.: Proc. of IJCAI 99, (Morgan Kaufmann) 548–553
22. Sakurai, Y., Yokoo, M., Kamei, K.: An efficient approximate algorithm for winner determination in combinatorial auctions. [24] 30–37
23. Sierra, C., Noriega, P.: Agent-mediated interaction. From auctions to negotiation and argumentation. In d’Inverno, M., Luck, M., Fisher, M., Preist, C., eds.: Foundations and Applications of Multi-Agent Systems, UKMAS Workshop 1996-2000. Volume 2403 of LNCS, Springer Verlag (2002) 27–48
24. Jhingran, A., ed.: Proc. of the 2nd ACM Conference on Electronic Commerce (EC-00), October 17-20, 2000, Minneapolis, MN, USA. ACM Press (2000)

Level Mapping Characterizations of Selector Generated Models for Logic Programs

Pascal Hitzler^{1*} and Sibylle Schwarz²

¹ AIFB, Universität Karlsruhe (TH)

email: hitzler@aifb.uni-karlsruhe.de

² Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg

email: schwarzs@informatik.uni-halle.de

Abstract. Assigning semantics to logic programs via selector generated models (Schwarz 2002/2003) extends several semantics, like the stable, the inflationary, and the stable generated semantics, to programs with arbitrary formulae in rule heads and bodies. We study this approach by means of a unifying framework for characterizing different logic programming semantics using level mappings (Hitzler and Wendt 200x, Hitzler 2003), thereby supporting the claim that this framework is very flexible and applicable to very diversely defined semantics.

1 Introduction

Hitzler and Wendt [8, 10, 11] have recently proposed a unifying framework for different logic programming semantics. This approach is very flexible and allows to cast semantics of very different origin and style into uniform characterizations using level mappings, i.e. mappings from atoms to ordinals, in the spirit of the definition of acceptable programs [2], the use of stratification [1, 14] and a characterization of stable models by Fages [3]. These characterizations display syntactic and semantic dependencies between language elements by means of the preorders on ground atoms induced by the level mappings, and thus allow inspection of and comparison between different semantics, as exhibited in [8, 10, 11].

For the syntactically restricted class of normal logic programs, the most important semantics — and some others — have already been characterized and compared, and this was spelled out in [8, 10, 11]. Due to the inherent flexibility of the framework, it is clear that studies of extended syntax are also possible, but have so far not been carried out. In this paper, we will present a non-trivial technical result which provides a first step towards a comprehensive comparative study of different semantics for logic programs under extended syntax.

* The first named author acknowledges support by the German Federal Ministry of Education and Research under the SmartWeb project, and by the European Union under the KnowledgeWeb Network of Excellence. He also acknowledges the hospitality of the Graduiertenkolleg *Wissensrepräsentation* at the University of Leipzig, Germany, while working on a first draft of this paper.

Table 1. Notions of specific types of rules.

<i>rule is called</i>	<i>set</i>	<i>condition</i>
<i>definite</i>	LP	$\text{body}(r) \in \text{Lg}(\{\wedge, \mathbf{t}\}, A)$ and $\text{head}(r) \in A$
<i>normal</i>	NLP	$\text{body}(r) \in \text{Lg}(\{\wedge, \mathbf{t}\}, \text{Lit}(A))$ and $\text{head}(r) \in A$
<i>head-atomic</i>	HALP	$\text{body}(r) \in \text{Lg}(\Sigma^{cl}, A)$ and $\text{head}(r) \in A$
<i>pos. head disj.</i>	DLP ⁺	$\text{body}(r) \in \text{Lg}(\{\wedge, \mathbf{t}\}, \text{Lit}(A))$ and $\text{head}(r) \in \text{Lg}(\{\vee\}, A)$
<i>disjunctive</i>	DLP	$\text{body}(r) \in \text{Lg}(\{\wedge, \mathbf{t}\}, \text{Lit}(A))$, $\text{head}(r) \in \text{Lg}(\{\vee, \mathbf{f}\}, \text{Lit}(A))$
<i>head-disjunctive</i>	HDLP	$\text{body}(r) \in \text{Lg}(\Sigma^{cl}, A)$, $\text{head}(r) \in \text{Lg}(\{\vee, \mathbf{f}\}, \text{Lit}(A))$
<i>generalized</i>	GLP	no condition

More precisely, among the many proposals for semantics for logic programs under extended syntax we will study a very general approach due to Schwarz [15, 16]. In this framework, arbitrary formulae are allowed in rule heads and bodies, and it encompasses the inflationary semantics [12], the stable semantics for normal and disjunctive programs [5, 13], and the stable generated semantics [7]. It can itself be understood as a unifying framework for different semantics.

In this paper, we will provide a single theorem — and some corollaries thereof — which gives a characterization of general selector generated models by means of level mappings. It thus provides a link between these two frameworks, and implicitly yields level mapping characterizations of the semantics encompassed by the selector generated approach.

The plan of the paper is as follows. In Section 2 we will fix preliminaries and notation. In Section 3 we will review selector generated models as introduced in [15, 16]. In Section 4, we present our main result, Theorem 4, which gives a level-mapping characterization of general selector generated models in the style of [8, 10, 11]. In Section 5 we study corollaries from Theorem 4 concerning specific cases of interest encompassed by the result. We eventually conclude and discuss further work in Section 6.

2 Preliminaries

Throughout the paper, we will consider a language \mathcal{L} of propositional logic over some set of propositional variables, or *atoms*, A , and connectives $\Sigma^{cl} = \{\neg, \vee, \wedge, \mathbf{t}, \mathbf{f}\}$, as usual. A *rule* r is a pair of formulae from \mathcal{L} denoted by $\varphi \Rightarrow \psi$. φ is called the *body* of the rule, denoted by $\text{body}(r)$, and ψ is called the *head* of the rule, denoted by $\text{head}(r)$. A *program* is a set of rules. A *literal* is an atom or a negated atom, and $\text{Lit}(A)$ denotes the set of all literals in \mathcal{L} . For a set of connectives $C \subseteq \Sigma^{cl}$ we denote by $\text{Lg}(C, A)$ the set of all formulae over \mathcal{L} in which only connectives from C occur.

Further terminology is introduced in Table 1. The abbreviations in the second column denote the sets of all rules with the corresponding property. A program containing only definite (normal, etc.) rules is called *definite* (*normal*, etc.). Programs not containing the negation symbol \neg are called *positive*. *Facts* are rules r where $\text{body}(r) = \mathbf{t}$, denoted by $\Rightarrow \text{head}(r)$.

The *base* \mathbf{B}_P is the set of all atoms occurring in a program P . A two-valued *interpretation* of a program P is represented by a subset of \mathbf{B}_P , as usual. By \mathbf{I}_P we denote the set of all interpretations of P . It is a complete lattice with respect to the subset ordering \subseteq . For an interpretation $I \in \mathbf{I}_P$, we define $\uparrow I = \{J \in \mathbf{I}_P \mid I \subseteq J\}$ and $\downarrow I = \{J \in \mathbf{I}_P \mid J \subseteq I\}$. $[I, J] = \uparrow I \cap \downarrow J$ is called an *interval* of interpretations.

The model relation $M \models \varphi$ for an interpretation M and a propositional formula φ is defined as usual in propositional logic, and $\text{Mod}(\varphi)$ denotes the set of all models of φ . Two formulae φ and ψ are *logically equivalent*, written $\varphi \equiv \psi$, iff $\text{Mod}(\varphi) = \text{Mod}(\psi)$.

A formula φ is *satisfied* by a set $\mathbf{J} \subseteq \mathbf{I}_P$ of interpretations if each interpretation $J \in \mathbf{J}$ is a model of φ . For a program P , a set $\mathbf{J} \subseteq \mathbf{I}_P$ of interpretations determines the set of all rules which *fire* under \mathbf{J} , formally $\text{fire}(P, \mathbf{J}) = \{r \in P \mid \forall J \in \mathbf{J} : J \models \text{body}(r)\}$. An interpretation M is called a *model* of a rule r (or *satisfies* r) if M is a model of the formula $\neg \text{body}(r) \vee \text{head}(r)$. An interpretation M is a *model* of a program P if it satisfies each rule in P .

For conjunctions or disjunctions φ of literals, φ^+ denotes the set of all atoms occurring positively in φ , and φ^- contains all atoms that occur negated in φ . For instance, for the formula $\varphi = (a \wedge \neg b \wedge \neg a)$ we have $\varphi^+ = \{a\}$ and $\varphi^- = \{a, b\}$. In heads φ consisting only of disjunctions of literals, we always assume without loss of generality that $\varphi^+ \cap \varphi^- = \emptyset$.

If φ is a *conjunction* of literals, we abbreviate $M \models \bigwedge_{a \in \varphi^+} a$ (i.e. $\varphi^+ \subseteq M$) by $M \models \varphi^+$ and $M \models \bigwedge_{a \in \varphi^-} \neg a$ (i.e. $\varphi^- \cap M = \emptyset$) by $M \models \varphi^-$, abusing notation. If φ is a *disjunction* of literals, we write $M \models \varphi^+$ for $M \models \bigvee_{a \in \varphi^+} a$ (i.e. $M \cap \varphi^+ \neq \emptyset$) and $M \models \varphi^-$ for $M \models \bigvee_{a \in \varphi^-} \neg a$ (i.e. $\varphi^- \not\subseteq M$).

By iterative application of rules from a program $P \subseteq \text{GLP}$ starting in the least interpretation $\emptyset \in \mathbf{I}_P$, we can create monotonically increasing (transfinite) sequences of interpretations of the program P , as follows.

Definition 1. A (transfinite) sequence C of length α of interpretations of a program $P \subseteq \text{GLP}$ is called a *P-chain* iff

- (C0) $C_0 = \emptyset$,
- (C β) $C_{\beta+1} \in \text{Min}(\uparrow C_\beta \cap \text{Mod}(\text{head}(Q_\beta)))$ for some set of rules $Q_\beta \subseteq P$ and for all β with $\beta + 1 < \alpha$, and
- (C λ) $C_\lambda = \bigcup \{C_\beta \mid \beta < \lambda\}$ for all limit ordinals $\lambda < \alpha$.

\mathbf{C}_P denotes the collection of all *P-chains*.

Note that all *P-chains* increase monotonically with respect to \subseteq .

3 Selector generated models

In [15, 16], a framework for defining declarative semantics of generalized logic programs was introduced, which encompasses several other semantics, as already mentioned in the introduction. Parametrization within this framework is done via so-called *selector functions*, defined as follows.

Definition 2. A selector is a function $\text{Sel} : \mathbf{C}_P \times \mathbf{I}_P \rightarrow 2^{\mathbf{I}_P}$, satisfying $\emptyset \neq \text{Sel}(C, I) \subseteq [I, \sup(C)]$ for all P -chains C and each interpretation $I \in \downarrow \sup(C)$.

We use selectors Sel to define nondeterministic successor functions Ω_P on \mathbf{I}_P , as follows.

Definition 3. Given a selector $\text{Sel} : \mathbf{C}_P \times \mathbf{I}_P \rightarrow 2^{\mathbf{I}_P}$ and a program P , the function $\Omega_P : (\mathbf{C}_P \times \mathbf{I}_P \rightarrow 2^{\mathbf{I}_P}) \times \mathbf{C}_P \times \mathbf{I}_P \rightarrow 2^{\mathbf{I}_P}$ is defined by

$$\Omega_P(\text{Sel}, C, I) = \text{Min}([I, \sup(C)] \cap \text{Mod}(\text{head}(\text{fire}(P, \text{Sel}(C, I))))) .$$

Example 1. In this paper, we will have a closer look at the following selectors.

lower bound selector	$\text{Sel}_l(C, I) = \{I\}$
lower and upper bound selector	$\text{Sel}_u(C, I) = \{I, \sup(C)\}$
interval selector	$\text{Sel}_i(C, I) = [I, \sup(C)]$
chain selector	$\text{Sel}_c(C, I) = [I, \sup(C)] \cap C$

With the first two arguments (the selector Sel and the chain C) fixed, the function $\Omega_P(\text{Sel}, C, I)$ can be understood as a nondeterministic consequence operator. Iteration of the function $\Omega_P(\text{Sel}, C, \cdot)$ from the least interpretation \emptyset creates sequences of interpretations. This leads to the following definition of (P, M, Sel) -chains.

Definition 4. A (P, M, Sel) -chain is a P -chain satisfying

- ($\mathbf{C} \sup$) $M = \sup(C)$ and
 ($\mathbf{C} \beta_{\text{Sel}}$) $C_{\beta+1} \in \Omega_P(\text{Sel}, C, C_\beta)$ for all β , where $\beta+1 < \kappa$ and κ is the length of the transfinite sequence C .

Thus, (P, M, Sel) -chains are monotonically increasing sequences C of interpretations of P , that reproduce themselves by iterating Ω_P . Note that this definition is non-constructive.

The main concept of the selector semantics is fixed in the following definition.

Definition 5. A model M of a program $P \subseteq \text{GLP}$ is Sel -generated if and only if there exists a (P, M, Sel) -chain C . The Sel -semantics of the program P is the set $\text{Mod}_{\text{Sel}}(P)$ of all Sel -generated models of P .

Example 2. The program $P = \{\Rightarrow a, a \Rightarrow b, (a \vee \neg c) \wedge (c \vee \neg a) \Rightarrow c\}$ has the only Sel_l -generated model $\{a, b, c\}$, namely via the chain $C_1 = (\emptyset \xrightarrow{1,3} \{a, c\} \xrightarrow{2} \{a, b, c\})$, where the rules applied in each step are denoted above the arrows. $\{a, b\}$ and $\{a, b, c\}$ are Sel_u -generated (and Sel_c -generated) models, namely via the chains $C_2 = (\emptyset \xrightarrow{1} \{a\} \xrightarrow{2} \{a, b\})$ and C_1 . $\{a, b\}$ is the only Sel_i -generated model of P , namely via C_2 .

Some properties of semantics generated by the selectors in Example 1 were studied in [15]. In Section 5, we will make use of the following results from [15].

Theorem 1 ([16]).

1. For definite programs $P \subseteq \text{DLP}$, the unique element contained in $\text{Mod}_i(P) = \text{Mod}_u(P) = \text{Mod}_c(P) = \text{Mod}_i(P)$ is the least model of P .
2. For normal programs $P \subseteq \text{NLP}$, the unique element of $\text{Mod}_i(P)$ is the inflationary model of P (as introduced in [12]).
3. For normal programs $P \subseteq \text{NLP}$, the set $\text{Mod}_u(P) = \text{Mod}_c(P) = \text{Mod}_i(P)$ contains exactly all stable models of P (as defined in [5]).
4. For disjunctive programs $P \subseteq \text{DLP}^+$, the minimal elements in $\text{Mod}_u(P) = \text{Mod}_c(P) = \text{Mod}_i(P)$ are exactly all stable models of P (as defined in [13]), but for generalized programs $P \subseteq \text{GLP}$, the sets $\text{Mod}_u(P)$, $\text{Mod}_c(P)$, and $\text{Mod}_i(P)$ may differ.
5. For generalized programs $P \subseteq \text{GLP}$, $\text{Mod}_i(P)$ is the set of stable generated models of P (as defined in [7]). \square

This shows that the framework of selector semantics covers some of the most important declarative semantics for normal logic programs. Selector generated models provide a natural extension of these semantics to generalized logic programs and allow systematic comparisons of many new and well-known semantics.

4 Selector generated models via level mappings

In [8, 10, 11], a uniform approach to different semantics for logic programs was given, using the notion of *level mapping*, as follows.

Definition 6. A level mapping for a logic program $P \subseteq \text{GLP}$ is a function $l : \mathcal{B}_P \rightarrow \alpha$, where α is an ordinal.

In order to display the style of level-mapping characterizations for semantics, we cite two examples which we will further discuss later on.

Theorem 2 ([11]). Every definite program $P \subseteq \text{LP}$ has exactly one model M , such that there exists a level mapping $l : \mathcal{B}_P \rightarrow \alpha$ satisfying

(Fd) for every atom $a \in M$ there exists a rule $\bigwedge_{b \in B} b \Rightarrow a \in P$ such that $B \subseteq M$ and $\max \{l(b) \mid b \in B\} < l(a)$.

Furthermore, M coincides with the least model of P . \square

Theorem 3 ([4]). Let P be a normal program and M be an interpretation for P . Then M is a stable model of P iff there exists a level mapping $l : \mathcal{B}_P \rightarrow \alpha$ satisfying

(Fs) for each atom $a \in M$ there exists a rule $r \in P$ with $\text{head}(r) = a$, $\text{body}(r)^+ \subseteq M$, $\text{body}(r)^- \cap M = \emptyset$, and $\max \{l(b) \mid b \in \text{body}(r)^+\} < l(a)$. \square

It is evident, that among the level mappings satisfying the respective conditions in Theorems 2 and 3, there exist pointwise minimal ones.

We set out to prove a general theorem which characterizes selector generated models by means of level mappings, in the style of the results displayed above. The following notion will ease notation considerably.

Definition 7. For a level mapping $l : B_P \rightarrow \alpha$ for a program $P \subseteq \text{GLP}$ and an interpretation $M \subseteq B_P$, the elements of the (transfinite) sequence $C^{l,M}$ consisting of interpretations of P are for all $\beta < \alpha$ defined by

$$C_\beta^{l,M} = \{a \in M \mid l(a) < \beta\} = M \cap \bigcup_{\gamma < \beta} l^{-1}(\gamma).$$

Remark 1. Definition 7 implies that

1. the (transfinite) sequence $C^{l,M}$ is monotonically increasing,
2. $C_0^{l,M} = \emptyset$, and
3. $M = \bigcup_{\beta < \alpha} C_\beta^{l,M} = \sup C^{l,M}$.

The following Theorem provides a mutual translation between the definition of selector semantics and a level mapping characterization.

Theorem 4. Let $P \subseteq \text{HDLP}$ be a head disjunctive program and $M \in \mathbf{I}_P$. Then M is a Sel-generated model of P iff there exists a level mapping $l : B_P \rightarrow \alpha$ satisfying the following properties.

- (L1) $M = \sup (C^{l,M}) \in \text{Mod}(P)$.
 (L2) For all β with $\beta + 1 < \alpha$ we have

$$C_{\beta+1}^{l,M} \setminus C_\beta^{l,M} \in \text{Min} \left\{ J \in \mathbf{I}_P \mid J \models \text{head} \left(R \left(C_\beta^{l,M}, J \right) \right)^+ \right\}, \quad \text{where}$$

$$R \left(C_\beta^{l,M}, J \right) = \left\{ r \in \text{fire} \left(P, \text{Sel} \left(C^{l,M}, C_\beta^{l,M} \right) \right) \mid \begin{array}{l} C_\beta^{l,M} \not\models \text{head}(r)^+ \text{ and} \\ J \cup C_\beta^{l,M} \not\models \text{head}(r)^- \end{array} \right\}.$$

- (L3) For all limit ordinals $\lambda < \alpha$ we have $C_\lambda^{l,M} = \bigcup_{\beta < \lambda} C_\beta^{l,M}$.

The proof of Theorem 4 is omitted for space limitations. It is rather involved and technical, and can be found in detail in [9]

Remark 2. As P is a head disjunctive program, we have $C_\beta^{l,M} \not\models \text{head}(r)^+$ iff $\text{head}(r)^+ \cap C_\beta^{l,M} = \emptyset$, and $J \cup C_\beta^{l,M} \not\models \text{head}(r)^-$ iff $\text{head}(r)^- \subseteq J \cup C_\beta^{l,M}$, thus

$$R \left(C_\beta^{l,M}, J \right) = \left\{ r \in \text{fire} \left(P, \text{Sel} \left(C^{l,M}, C_\beta^{l,M} \right) \right) \mid \begin{array}{l} \text{head}(r)^+ \cap C_\beta^{l,M} = \emptyset \text{ and} \\ \text{head}(r)^- \subseteq J \cup C_\beta^{l,M} \end{array} \right\}.$$

Also note that for every rule $r \in \text{fire} \left(P, \text{Sel} \left(C^{l,M}, C_\beta^{l,M} \right) \right) \setminus R \left(C_\beta^{l,M}, J \right)$, we have $\downarrow (C_\beta^{l,M} \cup J) \subseteq \text{Mod}(\text{head}(r)^-)$ or $\uparrow C_\beta^{l,M} \subseteq \text{Mod}(\text{head}(r)^+)$. Thus all of these rules are satisfied in the interval $[C_\beta^{l,M}, C_\beta^{l,M} \cup J]$.

For all selectors Sel, it was shown in [15] that the Sel-semantics of programs in GLP is invariant with respect to the following transformations: the replacement (\rightarrow_{eq}) of the body and the head of a rule by logically equivalent formulae and the

splitting (\rightarrow_{hs}) of conjunctive heads, more precisely the replacement $P \cup \{\varphi \Rightarrow \psi \wedge \psi'\} \rightarrow_{\text{hs}} P \cup \{\varphi \Rightarrow \psi, \varphi \Rightarrow \psi'\}$.

Since every formula $\text{head}(r)$ is logically equivalent to a formula in conjunctive normal form, each selector semantics Mod_{Sel} of a generalized program P is equivalent to the selector semantics Mod_{Sel} of all head disjunctive programs Q where $P \rightarrow_{\text{eq,hs}}^* Q$. Note that in the transformation $\rightarrow_{\text{eq,hs}}^*$, no shifting of subformulas between the body and the head of a rule is involved. Therefore, Theorem 4 immediately generalizes to our main result.

Corollary 1. *Let P be a generalized program and M an interpretation of P . Then M is a Sel-generated model of P iff for any head disjunctive program Q with $P \rightarrow_{\text{eq,hs}}^* Q$ there exists a level mapping $l : \mathbf{B}_Q \rightarrow \alpha$ satisfying (L1), (L2) and (L3) of Theorem 4.* \square

5 Corollaries

We can now apply Theorem 4 in order to obtain level mapping characterizations for every semantics generated by a selector, in particular for those semantics generated by the selectors defined in Example 1 and listed in Theorem 1. For syntactically restricted programs, we can furthermore simplify the properties (L1), (L2) and (L3) in Theorem 4. Alternative level mapping characterizations for some of these semantics were already obtained directly in [11].

Programs with positive disjunctions in all heads

For rules $r \in \text{HDLP}$, where $\text{head}(r)$ is a disjunction of atoms, we have $\text{head}(r)^- = \emptyset$. Hence we have $\text{head}(r)^- \subseteq I$, i.e. $I \not\models \text{head}(r)^-$, for all interpretations $I \in \mathbf{I}_P$. Thus the set $R(\mathbf{C}_\beta^{l,M}, J)$ from (L2) in Theorem 4 can be specified by

$$R(\mathbf{C}_\beta^{l,M}, J) = \left\{ r \in \text{fire} \left(P, \text{Sel} \left(\mathbf{C}_\beta^{l,M}, \mathbf{C}_\beta^{l,M} \right) \right) \mid \mathbf{C}_\beta^{l,M} \not\models \text{head}(r)^+ \right\}.$$

We furthermore observe that the set $R(\mathbf{C}_\beta^{l,M}, J)$ does not depend on the interpretation J , so we obtain

$$R'(\mathbf{C}_\beta^{l,M}) = \left\{ r \in \text{fire} \left(P, \text{Sel} \left(\mathbf{C}_\beta^{l,M}, \mathbf{C}_\beta^{l,M} \right) \right) \mid \mathbf{C}_\beta^{l,M} \cap \text{head}(r)^+ = \emptyset \right\}$$

and hence

$$\text{Min} \left\{ J \in \mathbf{I}_P \mid J \models \text{head} \left(R(\mathbf{C}_\beta^{l,M}, J) \right)^+ \right\} = \text{Min} \left(\text{Mod} \left(\text{head} \left(R'(\mathbf{C}_\beta^{l,M}) \right) \right) \right).$$

Thus for programs containing only rules whose heads are disjunctions of atoms we can rewrite condition (L2) in Theorem 4, as follows:

(L2d) for every β with $\beta + 1 < \alpha$:

$$\begin{aligned} & \mathbf{C}_{\beta+1}^{l,M} \setminus \mathbf{C}_\beta^{l,M} \in \text{Min} \left(\text{Mod} \left(\text{head} \left(R'(\mathbf{C}_\beta^{l,M}) \right) \right) \right), \text{ where} \\ & R'(\mathbf{C}_\beta^{l,M}) = \left\{ r \in \text{fire} \left(P, \text{Sel} \left(\mathbf{C}_\beta^{l,M}, \mathbf{C}_\beta^{l,M} \right) \right) \mid \mathbf{C}_\beta^{l,M} \cap \text{head}(r)^+ = \emptyset \right\}. \end{aligned}$$

Programs with atomic heads

Single atoms are a specific kind of disjunctions of atoms. Hence for programs with atomic heads we can replace condition **(L2)** in Theorem 4 by **(L2d)**, and further simplify it as follows.

For rules with atomic heads we have $\text{head}(\{r \in P \mid \text{head}(r) \notin I\}) = \text{head}(P) \setminus I$ and therefore

$$\begin{aligned} & \text{head}\left(R'\left(C_{\beta}^{l,M}\right)\right) \\ &= \text{head}\left(\left\{r \in \text{fire}\left(P, \text{Sel}\left(C_{\beta}^{l,M}, C_{\beta}^{l,M}\right)\right) \mid \text{head}(r) \cap C_{\beta}^{l,M} = \emptyset\right\}\right) \\ &= \text{head}\left(\left\{r \in \text{fire}\left(P, \text{Sel}\left(C_{\beta}^{l,M}, C_{\beta}^{l,M}\right)\right) \mid \text{head}(r) \notin C_{\beta}^{l,M}\right\}\right) \\ &= \text{head}\left(\text{fire}\left(P, \text{Sel}\left(C_{\beta}^{l,M}, C_{\beta}^{l,M}\right)\right)\right) \setminus C_{\beta}^{l,M}. \end{aligned}$$

Because all formulae in $\text{head}(P)$ are atoms we obtain

$$\begin{aligned} \text{Min}\left(\text{Mod}\left(\text{head}\left(R'\left(C_{\beta}^{l,M}\right)\right)\right)\right) &= \text{Min}\left(\uparrow\left(\text{head}\left(R'\left(C_{\beta}^{l,M}\right)\right)\right)\right) \\ &= \left\{\text{head}\left(R'\left(C_{\beta}^{l,M}\right)\right)\right\} \end{aligned}$$

and this allows us to simplify **(L2)** in Theorem 4 to the following:

(L2a) for each β with $\beta + 1 < \alpha$:

$$C_{\beta+1}^{l,M} \setminus C_{\beta}^{l,M} = \text{head}\left(\text{fire}\left(P, \text{Sel}\left(C_{\beta}^{l,M}, C_{\beta}^{l,M}\right)\right)\right) \setminus C_{\beta}^{l,M}.$$

Inflationary models From Section 3 we know that for normal programs P the selector Sel_l generates exactly the inflationary model of P as defined in [12]. The generalizations of the definition of inflationary models and this result to head atomic programs are immediate. From [16] we also know that every Sel_l -generated model is generated by a (P, M, Sel_l) -chain of length ω . Thus level mappings $l : B_P \rightarrow \omega$ are sufficient to characterize inflationary models of head atomic programs. In this case, condition **(L3)** applies only to the limit ordinal $0 < \omega$. But by remark 1, all level mappings satisfy this property. Therefore we do not need condition **(L3)** in the characterization of inflationary models.

Using Theorem 4 and the considerations above, we obtain the following characterization of inflationary models.

Corollary 2. *Let $P \subseteq \text{HALP}$ be a head atomic program and M be an interpretation for P . Then M is the inflationary model of P iff there exists a level mapping $l : B_P \rightarrow \omega$ with the following properties.*

(L1) $M = \sup(C_{\beta}^{l,M}) \in \text{Mod}(P)$.

(L2i) for all $n < \omega$: $C_{n+1}^{l,M} \setminus C_n^{l,M} = \text{head}\left(\text{fire}\left(P, C_n^{l,M}\right)\right) \setminus C_n^{l,M}$. □

Normal programs

For normal programs, the heads of all rules are single atoms. Hence the simplification **(L2a)** of condition **(L2)** in Theorem 4 applies for all selector generated semantics for normal programs.

The special structure of the bodies of all rules in normal programs allows an alternative formulation of **(L2a)**. In every normal rule, the body is a conjunction of literals. Thus for any set of interpretations \mathbf{J} we have $\mathbf{J} \models \text{body}(r)$ iff $\text{body}(r)^+ \subseteq J$ and $\text{body}(r)^- \cap J = \emptyset$ for all interpretations $J \in \mathbf{J}$.

Stable models We develop next a characterization for stable models of normal programs, as introduced in [5]. The selector Sel_{lu} generates exactly all stable models for normal programs. In [16], it was also shown that all Sel_{lu} -generated models M of a program P are generated by a (P, M, Sel) -chain of length $\leq \omega$. So for the same reasons as discussed for inflationary models, level mappings with range ω are sufficient to characterize stable models and condition **(L3)** can be neglected.

For a normal rule r and two interpretations $I, M \in \mathbf{I}_P$ with $I \subseteq M$ we have $\{I, M\} \models \text{body}(r)$, i.e. $I \models \text{body}(r)$ and $M \models \text{body}(r)$, iff $\text{body}(r)^+ \subseteq I$ and $\text{body}(r)^- \cap M = \emptyset$. Combining this with **(L2a)** we obtain the following characterization of stable models for normal programs.

Corollary 3. *Let $P \subseteq \text{NLP}$ be a normal program and M an interpretation for P . Then M is a stable model of P iff there exists a level mapping $l : \mathbf{B}_P \rightarrow \omega$ satisfying the following properties:*

(L1) $M = \sup(C^{l,M}) \in \text{Mod}(P)$.

(L2s) for all $n < \omega$:

$$C_{n+1}^{l,M} \setminus C_n^{l,M} = \text{head}(\{r \in P \mid \text{body}(r)^+ \subseteq C_n^{l,M}, \text{body}(r)^- \cap M = \emptyset\}) \setminus C_n^{l,M}. \quad \square$$

Comparing this with Theorem 3, we note that both theorems characterize the same set of models. Thus for a model M of P there exists a level mapping $l : \mathbf{B}_P \rightarrow \omega$ satisfying **(L1)** and **(L2s)** iff there exists a level mapping $l : \mathbf{B}_P \rightarrow \alpha$ satisfying **(Fs)**. The condition imposed on the level mapping in Theorem 3, however, is weaker than the condition in Corollary 3, because level mappings defined by (P, M, Sel) -chains are always pointwise minimal.

Definite programs

In order to characterize the least model of definite programs, we can further simplify condition **(L2)** in Theorem 4. Definite programs are a particular kind of head atomic programs. For definite programs, the inflationary and the least model coincide. We can replace condition **(L2)** in Theorem 4 by **(L2i)** in Corollary 2. Since the body of every definite rule is a conjunction of atoms we obtain

$$\text{fire}(P, I) = \{r \in P \mid \text{body}(r)^+ \subseteq I\}$$

for every interpretation $I \in \mathbf{I}_P$. Thus we get the following result.

Corollary 4. *Let $P \subseteq \text{LP}$ be a definite program and let M be an interpretation for P . Then M is the least model of P iff there exists a level mapping $l : \text{B}_P \rightarrow \omega$ satisfying the following conditions.*

(L1) $M = \sup (C^{l,M}) \in \text{Mod}(P)$.

(L2l) for all $n < \omega$: $C_{n+1}^{l,M} \setminus C_n^{l,M} = \text{head}(\{r \in P \mid \text{body}(r)^+ \subseteq C_n^{l,M}\}) \setminus C_n^{l,M}$ \square

Comparing this to Theorem 2, we note that the relation between the conditions (L2l) and (Fd) are similar to those of the conditions (Fs) und (L2s).

6 Conclusions and Further Work

Our main result, Corollary 1 respectively Theorem 4 in Section 4, provides a characterization of selector generated models — in general form — by means of level mappings in accordance with the uniform approach proposed in [8, 10, 11]. As corollaries from this theorem, we have also achieved level mapping characterizations of several semantics encompassed by the selector generated approach due to [15, 16].

Our contribution is technical, and provides a first step towards a comprehensive comparative study of different semantics of logic programs under extended syntax by means of level mapping characterizations. Indeed, a very large number of syntactic extensions for logic programs are currently being investigated in the community, and even for some of the less fancy proposals there is often no agreement on the preferable way of assigning semantics to these constructs.

A particularly interesting case in point is provided by disjunctive and extended disjunctive programs, as studied in [6]. While there is more or less general agreement on an appropriate notion of stable model, as given by the notion of *answer set* in [6], there exist various different proposals for a corresponding well-founded semantics, see e.g. [17]. We expect that recasting them by means of level-mappings will provide a clearer picture on the specific ways of modelling knowledge underlying these semantics.

Eventually, we expect that the study of level mapping characterizations of different semantics will lead to methods for extracting other, e.g. procedural, semantic properties from the characterizations, like complexity or decidability results.

References

1. Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of deductive databases and logic programs*. Morgan Kaufmann, Los Altos, US, 1988.
2. Krzysztof R. Apt and Dino Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1), September 1993.
3. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

4. François Fages. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. In Peter Szeredi and David H.D. Warren, editors, *Proceedings of the 7th International Conference on Logic Programming (ICLP '90)*, Jerusalem, June 1990. MIT Press.
5. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, Cambridge, Massachusetts, 1988. The MIT Press.
6. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4), 1991.
7. Heinrich Herre and Gerd Wagner. Stable models are generated by a stable chain. *Journal of Logic Programming*, 30(2), February 1997.
8. Pascal Hitzler. Towards a systematic account of different logic programming semantics. In Andreas Günter, Rudolf Kruse, and Bernd Neumann, editors, *KI2003: Advances in Artificial Intelligence. Proceedings of the 26th Annual German Conference on Artificial Intelligence, KI2003, Hamburg, Germany, September 2003*, volume 2821 of *Lecture Notes in Artificial Intelligence*, pages 355–369. Springer, Berlin, 2003.
9. Pascal Hitzler and Sibylle Schwarz. Level mapping characterizations of selector generated models for logic programs. Technical Report WV-04-04, Technische Universität Dresden, 2004. Available from www.aifb.uni-karlsruhe.de/WBS/phi/pub/wv-04-04.ps.gz.
10. Pascal Hitzler and Matthias Wendt. The well-founded semantics is a stratified Fitting semantics. In Matthias Jarke, Jana Koehler, and Gerhard Lakemeyer, editors, *Proceedings of the 25th Annual German Conference on Artificial Intelligence, KI2002, Aachen, Germany, September 2002*, volume 2479 of *Lecture Notes in Artificial Intelligence*, pages 205–221. Springer, Berlin, 2002.
11. Pascal Hitzler and Matthias Wendt. A uniform approach to logic programming semantics. *Theory and Practice of Logic Programming*, 5(1-2):123–159, 2005. To appear.
12. Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? In *PODS '88. Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems: March 21–23, 1988, Austin, Texas*, New York, NY 10036, USA, 1988. ACM Press.
13. Teodor Przymusiński. Stable Semantics for Disjunctive Programs. *New Generation Computing Journal*, 9, 1991.
14. Teodor C. Przymusiński. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.
15. Sibylle Schwarz. Answer sets generated by selector functions. In *Proceedings of the Workshop on Nonmonotonic Reasoning'2002*, pages 247–253, Toulouse, 2002. <http://www.tcs.hut.fi/~ini/nmr2002/schwarz.ps>.
16. Sibylle Schwarz. *Selektor-erzeugte Modelle verallgemeinerter logischer Programme*. PhD thesis, Universität Leipzig, 2004. <http://www.informatik.uni-leipzig.de/~schwarz/ps/thes.ps.gz>.
17. Kewen Wang. A comparative study of well-founded semantics for disjunctive logic programs. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17–19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Artificial Intelligence*, pages 133–146. Springer, 2001.

Truth and knowledge fixpoint semantics for many-valued logic programming

Zoran Majkić

Dept. of Computer Science, UMIACS, University of Maryland, College Park, MD 20742
zoran@cs.umd.edu

Abstract. The variety of semantical approaches that have been invented for logic programs for reasoning about inconsistent databases is quite broad. Especially, we are interested in an ontological encapsulation of a many-valued logics with negation, based on bilattices, into a 2-valued logic: this new approach is a minimalist one based on the concept of the semantic reflection. We define a Model theoretic approach for Herbrand interpretations of an encapsulated logic program and the new semantics for implication of their clauses. We introduce a trilattice of Herbrand interpretations, with the third truth dimension for this 'meta' logic (over a bilattice of the many-valued logic), where is possible to apply the Knaster-Tarski theorem for a truth-monotonic 'immediate consequence operator' instead of knowledge-monotonic Fitting's operator for a many-valued logic programs. We extend the Fitting's fixpoint semantics for 3-valued logic programs and show the strong connection with the fixpoint semantics of 'meta' logic programs obtained by encapsulation of many-valued programs.

1 Introduction

Generally, three-valued, or partial model semantics has had an extensive development for logic programs. So far, research in many-valued logic programming has proceeded along different directions; one of them is *Bilattice-based* logics, [1,2]. One of the key insights behind bilattices was the interplay between the truth values assigned to sentences and the (non classic) notion of *implication*. The problem was to study how truth values should be propagated "across" implications.

In [3], Belnap introduced a logic intended to deal in a useful way with inconsistent or incomplete information. It is the simplest example of a non-trivial bilattice and it illustrates many of the basic ideas concerning them. We denote the four values as $\{t, f, \top, \perp\}$, where t is *true*, f is *false*, \top is inconsistent (both true and false) or *possible*, and \perp is *unknown*. As Belnap observed, these values can be given two natural orders: *truth* order, \leq_t , and *knowledge* order, \leq_k , such that $f \leq_t \top \leq_t t$, $f \leq_t \perp \leq_t t$, and $\perp \leq_k f \leq_k \top$, $\perp \leq_k t \leq_k \top$. These two orderings define corresponding equivalences $=_t$ and $=_k$. Thus any two members α, β in a bilattice are equal, $\alpha = \beta$, if and only if (shortly 'iff') $\alpha =_t \beta$ and $\alpha =_k \beta$.

Meet and join operators under \leq_t are denoted \wedge and \vee ; they are natural generalizations of the usual conjunction and disjunction notions. Meet and join under \leq_k are denoted \otimes (*consensus*, because it produces the most information that two truth values can agree on) and \oplus (*gullibility*, it accepts anything it is told), such that hold: $f \otimes t = \perp$, $f \oplus t = \top$,

$\top \wedge \perp = f$ and $\top \vee \perp = t$.

There is a natural notion of truth negation, denoted \sim , (reverses the \leq_t ordering, while preserving the \leq_k ordering): switching f and t , leaving \perp and \top , and corresponding knowledge negation, denoted $-$, (reverses the \leq_k ordering, while preserving the \leq_t ordering), switching \perp and \top , leaving f and t . These two kinds of negation commute: $- \sim x = \sim -x$ for every member x of a bilattice.

It turns out that the operations \wedge, \vee and \sim , restricted to $\{f, t, \perp\}$ are exactly those of Kleene's strong 3-valued logic. Any bilattice $\langle \mathcal{B}, \leq_t, \leq_k \rangle$ is:

1. *Interlaced*, if each of the operations \wedge, \vee, \otimes and \oplus is monotone with respect to both orderings (for instance, $x \leq_t y$ implies $x \otimes z \leq_t y \otimes z$, $x \leq_k y$ implies $x \wedge z \leq_k y \wedge z$).
2. *Infinitarily interlaced*, if it is complete and four infinitary meet and join operations are monotone with respect to both orderings.
3. *Distributive*, if all 12 distributive laws connecting \wedge, \vee, \otimes and \oplus are valid.
4. *Infinitarily distributive*, if it is complete and infinitary, as well as finitary, distributive laws are valid. (Note that a bilattice is *complete* if all meets and joins exist, w.r.t. both orderings).

We denote infinitary meet and join w.r.t. \leq_t by \bigwedge and \bigvee , and by \prod and \sum for the \leq_k ordering. A more general information about bilattice may be found in [2]. The Belnap's 4-valued bilattice is infinitarily distributive. In the rest of this paper we denote by \mathcal{B}_4 (or simply \mathcal{B}) the Belnap's bilattice.

In this paper we will consider only consistent (without constraints) logic programming. We introduce the *ontological encapsulation* (program transformation) of the many-valued logic programs into the 2-valued 'meta' logic programs, and use it in order to define the notion of model for the many-valued programming.

The resulting ontological 2-valued logic [4], which encapsulates the semantics of many-valued logics is not an issue "per se": it is considered as a minimal logic tool useful to define many-valued entailment and inference. But, recently, it is used also in order to define the coalgebraic semantics for logic programs [5], and in the future, considering the fact that for any predicate at 'object' many-valued level, by encapsulation we obtain a new predicate, with one logic-attribute extension (for the epistemic logic value), we will consider the possibility to use the *ordinary relational database* schemas with such extended predicates in global schemas of data integration systems [6,7], for query answering from their unique minimal many-valued Herbrand model.

This paper is the direct continuation of the work [4], with the aim to provide the fixpoint semantics for such ontologically encapsulated 4-valued logic programs, analog to the knowledge ordered fixpoint semantics, given by Fitting [8], for a 3-valued strong Kleene's logic.

It is well known that the semantics of many-valued logic programs with negations can be defined as a least fixpoint w.r.t. the *knowledge* ordering in the bilattice of Herbrand interpretations (for positive logic programs it can be defined as least fixpoint w.r.t. the *truth* ordering also). What we have now to investigate is which kind of ordering, in the functional space of Herbrand interpretations of encapsulated logic programs, we need to define in order to obtain the least fixpoint as the semantics for encapsulated logic programs. As we will see, the answer comes directly from the fact that the encapsulated logic programs, with negation also, are a *positive* logic programs (with only modified

semantics for logic implication).

The obtained syntax for transformed original many-valued logic programs, that is, encapsulated programs, is *equal* to the standard syntax of logic programs: thus, predicates in such logic programs can *directly* represent the relational database models, differently from other approaches as in *Signed* [9,10,11] or *Annotated* logic programming [12,13,14].

Let's call *meta-truth* ordering, the truth ordering for encapsulated programs (to distinguish it from the truth ordering at many-valued logic programming level): from the fact that the encapsulated programs are syntactically positive logic programs we conclude that their semantics can be defined as a least fixpoint w.r.t. the meta-truth ordering. Because of that we need to enrich the original many-valued level bilattice also with this 3-th meta-truth ordering in order to obtain a *trilattice*, and to define the fixpoint semantics of encapsulated programs w.r.t. this meta-truth ordering. Moreover, we need to prove that such least fixpoint, for a minimal Herbrand model of the encapsulated logic program, corresponds to the least (Fitting's-like) fixpoint of the original 4-valued logic program.

We argue that such logic will be a good framework for supporting also the data integration systems with key and foreign key integrity constraints with incomplete and inconsistent source databases, with more simple query-answering algorithms [15,7].

The plan of this paper is the following: Section 2 is a short overview of the work in [4] which introduces the syntax and the *model theoretic* Herbrand semantics for the encapsulated many-valued (EMV) logic programs. Section 3 presents the theory of a trilattice, \mathcal{B}^A , of interpretations for an encapsulated 'meta' logic program P^A , which generalizes the bilattice structure of a functional space of Herbrand interpretations for multi-valued logic programs. Finally in Section 4 is developed the fixpoint semantics for encapsulated logic programs and it is shown that it corresponds to the minimal stable many-valued models of Fitting's fixpoint semantics.

2 Encapsulation of many-valued logic

We assume that the Herbrand universe is $\Gamma_U = \Gamma \cup \Omega$, where Γ is ordinary domain of database constants, and Ω is an infinite enumerable set of marked null values, $\Omega = \{\omega_0, \omega_1, \dots\}$, and for a given logic program P composed by a set of predicate and function symbols, P_S, F_S respectively, we define a set of all terms, \mathcal{T}_S , and its subset of ground terms \mathcal{T}_0 . Then the atoms are defined as:

$$\mathcal{A}_S = \{p(c_1, \dots, c_n) \mid p \in P_S, n = \text{arity}(p) \text{ and } c_i \in \mathcal{T}_S\}.$$

The Herbrand base, H_P , is the set of all ground (i.e., variable free) atoms. A (ordinary) Herbrand interpretation is a many-valued mapping $I : H_P \rightarrow \mathcal{B}$. If P is a many-valued logic program with the Herbrand base H_P , then the ordering relations and operations in a bilattice \mathcal{B}_4 are propagated to the function space $\mathcal{B}_4^{H_P}$, that is the set of all Herbrand interpretations (functions), $I = v_B : H_P \rightarrow \mathcal{B}_4$. It is straightforward [16] that this makes a function space $\mathcal{B}_4^{H_P}$ itself a complete infinitary distributive bilattice.

One of the key insights behind bilattices [1,2] was the interplay between the truth values assigned to sentences and the (non classic) notion of *implication*. The problem was to study how truth values should be propagated "across" implications. We proposed

[17] the implication based approach, which extends the definition in [18] based on the following intuitionistic many valued implication (a relative pseudo complement w.r.t. the truth ordering):

\rightarrow	t	\perp	f	\top
t	t	\perp	f	\top
\perp	t	t	\top	\top
f	t	t	t	t
\top	t	\perp	\perp	t

We introduce [4] the program encapsulation transformation \mathcal{E} :

Definition 1. Let P be a many-valued logic program with the set of predicate symbols P_S , and for any predicate p in P_S we introduce a mapping $\kappa_p : T_0^{\text{arity}(p)} \rightarrow \mathcal{B}$. The translation \mathcal{E} in the encapsulated syntax version P^A is as follows:

1. Each positive literal in P , we introduce a new predicate p^A as follows
 $\mathcal{E}(p(x_1, \dots, x_n)) = p^A(x_1, \dots, x_n, \kappa_p(x_1, \dots, x_n))$;
2. Each negative literal in P , we introduce a new predicate p^A as follows
 $\mathcal{E}(\sim p(x_1, \dots, x_n)) = p^A(x_1, \dots, x_n, \sim \kappa_p(x_1, \dots, x_n))$;
3. $\mathcal{E}(\phi \wedge \varphi) = \mathcal{E}(\phi) \wedge \mathcal{E}(\varphi)$; $\mathcal{E}(\phi \vee \varphi) = \mathcal{E}(\phi) \vee \mathcal{E}(\varphi)$;
4. $\mathcal{E}(\phi \leftarrow \varphi) = \mathcal{E}(\phi) \leftarrow^A \mathcal{E}(\varphi)$, where \leftarrow^A is a symbol for the implication at the encapsulated 2-valued 'meta' level. Thus, the obtained 'meta' program is equal to $P^A = \{\mathcal{E}(\phi) \mid \phi \text{ is a clause in } P\}$, with the 2-valued Herbrand base $H_P^A = \{p^A(c_1, \dots, c_n, \alpha) \mid p(c_1, \dots, c_n) \in H_P \text{ and } \alpha \in \mathcal{B}\}$.

This embedding of the many-valued logic program P into a 2-valued 'meta' logic program P^A is an *ontological* embedding: views formulae of P as beliefs and interprets negation $\sim p(x_1, \dots, x_n)$ in rather restricted sense - as belief in the falsehood of $p(x_1, \dots, x_n)$, rather as not believing that $p(x_1, \dots, x_n)$ is true (like in an ontological embedding for classical negation). Like for Moore's autoepistemic operator, for the encapsulation operator \mathcal{E} (restricted to atoms), $\mathcal{E}\phi$ is intended to capture the notion of, "I know that ϕ has a value $v_B(\phi)$ ", for a given valuation v_B of the many-valued logic program.

Notice, that with the transformation of the original many-valued logic program P into its encapsulated 'meta' version program P^A we obtain a *positive* logic program.

We denote by \mathcal{F}_r the restriction function from the set of all encapsulated logical programs into the set of (usual) logical programs, such that for any 'meta' program P^A the $P = \mathcal{F}_r(P^A)$ is its reduction where all encapsulations of κ_p in clauses of P^A are eliminated.

An encapsulated Herbrand interpretation (en-interpretation) of P^A is a 2-valued mapping $I^A : H_P^A \rightarrow \mathbf{2}$, where $\mathbf{2} = \{t, f\}$. We denote by $\mathbf{2}^{H_P^A}$ the set of all en-interpretations from H_P^A into $\mathbf{2}$, and by \mathcal{B}^{H_P} the set of all *consistent* Herbrand many-valued interpretations, from H_P to the bilattice \mathcal{B} .

Let \mathcal{L} be the set of all ground well-formed formulae defined by this Herbrand base H_P and bilattice operations (included many-valued implication \leftarrow also), with $\mathcal{B} \subseteq \mathcal{L}$, then the set of all well-formed encapsulated formulae is:

$$\mathcal{L}^A =_{\text{def}} \{\mathcal{E}(\psi) \mid \psi \in \mathcal{L}\}, \text{ so that } H_P^A \subseteq \mathcal{L}^A.$$

The meaning of the *encapsulation* of a many-valued logic program P into this 'meta'

logic program P^A is fixed into the kind of interpretation to give to such new introduced functional symbols κ_p : in fact we want that they represent (encapsulate) the semantics of the many-valued logic program P (shortly, "ontological semantic-reflection \equiv epistemic semantics", [4]).

Definition 2. (Satisfaction) The **encapsulation** of an epistemic many-valued logic program P into a 'meta' program P^A means that, for any consistent many-valued Herbrand interpretation $I \in \mathcal{B}^{H_P}$ and its extension $v_B : \mathcal{L} \rightarrow \mathcal{B}$, the function symbols κ_p , $p \in P_S$ reflect this semantics, i.e. for any tuple $\mathbf{c} \in T_0^{arity(p)}$, $\kappa_p(\mathbf{c}) = I(p(\mathbf{c}))$. So, we obtain a mapping, $\Theta : \mathcal{B}^{H_P} \rightarrow 2^{H_P^A}$, such that $I^A = \Theta(I) \in 2^{H_P^A}$, with: for any ground atom $p(\mathbf{c})$, $I^A(\mathcal{E}(p(\mathbf{c}))) = t$, if $\kappa_p(\mathbf{c}) = I(p(\mathbf{c}))$; f otherwise.

The subset, $Im\Theta \subseteq \{t, f\}^{H_P^A}$, is denominated encapsulated (en-)interpretations). Let g be a variable assignment which assigns values from T_0 to object variables. We extent it to atoms with variables, so that $g(\mathcal{E}(p(x_1, \dots, x_n))) = \mathcal{E}(p(g(x_1), \dots, g(x_n)))$, and to all formulas in the usual way: ψ/g denotes a ground formula obtained from ψ by assignment g , then

1. $I^A \models_g \mathcal{E}(p(x_1, \dots, x_n))$ iff $\kappa_p((g(x_1), \dots, g(x_n))) = I(p(g(x_1), \dots, g(x_n)))$.
- $I^A \models_g \mathcal{E}(\sim p(x_1, \dots, x_n))$ iff $\sim \kappa_p((g(x_1), \dots, g(x_n))) = I(p(g(x_1), \dots, g(x_n)))$.
2. $I^A \models_g \mathcal{E}(\phi \wedge \psi)$ iff $I^A \models_g \mathcal{E}(\phi)$ and $I^A \models_g \mathcal{E}(\psi)$.
3. $I^A \models_g \mathcal{E}(\phi \vee \psi)$ iff $I^A \models_g \mathcal{E}(\phi)$ or $I^A \models_g \mathcal{E}(\psi)$.
4. $I^A \models_g \mathcal{E}(\phi \leftarrow \psi)$ iff $v_B(\phi/g \leftarrow \psi/g) = t$, i.e., $v_B(\phi/g) \geq_t v_B(\psi/g)$.

Notice that in this semantics the 'meta' implication \leftarrow^A (the point 4 above), in $\mathcal{E}(\phi) \leftarrow^A \mathcal{E}(\psi) = \mathcal{E}(\phi \leftarrow \psi)$, is based on the 'object' epistemic many-valued implication \leftarrow (which is not classical, i.e., $\phi \leftarrow \psi \neq \phi \vee \sim \psi$) and determines how the logical value of a body of clause "propagates" to its head.

Definition 3. The **Herbrand instantiation** P^* , of P^A , is constructed as follows: first, put in P^* all ground instances, substituting terms in the Herbrand universe for variables in every possible way, of clauses of P^A ; second (let $\alpha = \kappa_p(\mathbf{c})$), if a clause $p^A(\mathbf{c}, \alpha) \leftarrow^A$, with empty body, occurs in P^* , replace it with $p^A(\mathbf{c}, \alpha) \leftarrow^A t$. Next, if there are several clauses in the resulting set having the same head (only for non built-in predicates), $p^A(\mathbf{c}, \alpha) \leftarrow^A C_1$, $p^A(\mathbf{c}, \alpha) \leftarrow^A C_2$, ..., replace them with $p^A(\mathbf{c}, \alpha) \leftarrow^A C_1 \vee C_2 \vee \dots$; Finally, if the ground atom $p^A(\mathbf{c}, \alpha)$ is not the head of any member of P^* , add $p^A(\mathbf{c}, \alpha) \leftarrow f$. P^* will generally be infinite but in P^* a ground atom, $p^A(\mathbf{c}, \alpha)$, of a non built-in predicate, turns up as the head of exactly one member.

Proposition 1 If P^A is an encapsulated logic program, then its Herbrand instantiation, P^* , is positive. That is, all literals in its instantiated clauses are (positive) encapsulated ground atoms. Thus we can consider P^* as positive logic program w.r.t. replaced predicates (only the semantics for 'meta' implication of these logic programs is modified).

For multi-valued logic programs we apply the standard satisfaction rule for implication, where the logic value of the body of some clause is directly assigned (propagated) to the head of a clause by the 'immediate consequence operator': thus, for encapsulated logic programs, which are always positive, we obtain the unique minimal Herbrand model.

3 The trilattice of interpretations

What is the reason for introducing a trilattice? Well, it is well known that the semantics of many-valued logic programs with negations can be defined as a least fixpoint w.r.t. the *knowledge* ordering in the bilattice of Herbrand interpretations (for positive logic programs it can be defined as least fixpoint w.r.t. the *truth* ordering also). What we have now to consider is in which kind of ordering, in the functional space of Herbrand interpretations of encapsulated logic programs, we have to define the least fixpoint as its semantics. The answer comes directly from the fact that the encapsulated logic programs, obtained from a many-valued programs with negation also, are *positive* programs: thus, their semantics can be defined as a least fixpoint w.r.t. the meta-truth ordering, which is a new third trilattice's dimension.

Any interpretation $I^A : H_P^A \rightarrow \mathbf{2}$ of an encapsulated logical program P^A may be equivalently represented by the set of true, ontologically encapsulated, ground atoms, $m^A =_{\text{def}} \{ p^A(\mathbf{c}, \alpha) \mid I^A(p^A(\mathbf{c}, \alpha)) = t \} \in \mathcal{P}(H_P^A)$, where $\mathcal{P}(H_P^A)$ is the set of all subsets of H_P^A . In order to give an intuitive understanding of the definition of a trilattice let us consider any m^A as a quadruple of sets $[T, P, U, F]$, with true, possible, unknown and false atoms in H_P respectively. Then, given two en-interpretations m^A, m_1^A , represented by $[T, P, U, F]$ and $[T_1, P_1, U_1, F_1]$ respectively, we can introduce the following three preorders:

1. Set inclusion lattice preorder (truth preorder at 'meta' level), $m^A \subseteq m_1^A$.
2. Truth lattice preorder, $m^A \leq_t m_1^A$ iff $T \subseteq T_1$ and $F \supseteq F_1$.
3. Knowledge lattice preorder, $m^A \leq_k m_1^A$ iff $T \subseteq T_1, P \subseteq P_1$ and $F \subseteq F_1$.

Now we can introduce the following lattice operators (meet, join and negation):

1. Set intersection, set union and set complement, for the inclusion lattice preorder.

The strong equivalence $' = '$, is the set equivalence, empty set, m_\perp^A , is its bottom element, and $m_\top^A = H_P^A$ is its top element.

2. For the truth preorder (many-valued level) lattice:

$$[T, P, U, F] \wedge [T_1, P_1, U_1, F_1] =_{\text{def}} [T \cap T_1, P \cap P_1, \{\}, F \cup F_1]$$

$$[T, P, U, F] \vee [T_1, P_1, U_1, F_1] =_{\text{def}} [T \cup T_1, P \cap P_1, \{\}, F \cap F_1]$$

$$\sim [T, P, U, F] =_{\text{def}} [F, P, \{\}, T], \text{ false atoms become true and viceversa.}$$

Each $m^A = [T, P, U, F]$ such that $F = H_P$ and $T = \{\}$ is its bottom element; each m^A such that $T = H_P$ and $F = \{\}$ is its top element.

3. For the knowledge preorder (many-valued level) lattice:

$$[T, P, U, F] \otimes [T_1, P_1, U_1, F_1] =_{\text{def}} [T \cap T_1, P \cap P_1, \{\}, F \cap F_1]$$

$$[T, P, U, F] \oplus [T_1, P_1, U_1, F_1] =_{\text{def}} [T \cup T_1, P \cup P_1, \{\}, F \cup F_1]$$

$$-[T, P, U, F] =_{\text{def}} [\overline{F}, \overline{P}, \{\}, \overline{T}], \text{ where } \overline{S} \text{ denotes set complement, } \overline{S} = H_P - S.$$

Empty set, m_\perp^A , is its bottom element, and each m^A , such that $\sim \sim H_P^A \subseteq m^A \subseteq H_P^A$ is its top element.

The unary operations, n_t (change the truth in possibility), and p_t (change the falsehood in possibility), $n_t, p_t : \mathcal{P}(H_P^A) \rightarrow \mathcal{P}(H_P^A)$, are defined as follows:

$$n_t([T, P, U, F]) =_{\text{def}} [\{\}, T, \{\}, F], \quad p_t([T, P, U, F]) =_{\text{def}} [T, F, \{\}, \{\}],$$

with their bilattice corresponding, $n_t, p_t : \mathcal{B} \rightarrow \mathcal{B}$

$$n_t(t) = n_t(\perp) = \perp, \quad n_t(f) = n_t(\top) = f, \quad p_t(t) = p_t(\top) = t, \quad p_t(f) = p_t(\perp) = \perp.$$

There is also another weak equivalence in a trilattice \mathcal{B}^A , generated by two introduced many-valued orderings.

Definition 4. Let $\mathcal{B}^A =_{\text{def}} \langle \mathcal{P}(H_P^A), \subseteq, \leq_t^A, \leq_k^A \rangle$ be a trilattice of en-interpretations for an encapsulated program P^A . Then the weak equivalence relation " \approx " of a trilattice is defined by: $m_1^A \approx m_2^A$ iff $m_1^A =_t m_2^A$ and $m_1^A =_k m_2^A$. This equivalence determines equivalence classes of members in \mathcal{B}^A . In each equivalence class of members $\{m_1^A, \dots, m_n^A\}$, with $m_i^A \approx m_j^A$ for any $1 \leq i, j \leq n$, the subset of ground atoms encapsulated by \perp of each member m_i^A is left to be free.

1. We define a representative member, of any equivalence class, a member $m^A = [T, P, U, F]$ with $U = H_P - (T \cup P \cup F)$. We also introduce the function f_R which for any $m^A = [T, P, U, F]$ returns with its representative:

$$m^A \approx f_R(m^A) =_{\text{def}} [T, P, H_P - (T \cup P \cup F), F].$$

2. We define minimal members $m^A = [T, P, U, F] \in \mathcal{B}^A$, such that $U = \{\}$; it is easy to see that each trilattice operator, different from \cap and \cup , returns with a minimal member.

3. We define minimal consistent members $m^A \in \mathcal{B}^A$, such that they are minimal and $\bigcup_{\alpha \neq \beta, \alpha, \beta \in [T, P, U, F]} (\alpha \cap \beta) = \{\}$.

We denote by m_{bin}^A the minimal consistent member which represents the prefixed set of ground atoms for built-in predicates; it remains equal for all consistent interpretations of a given program P^A ,

$$m_{\text{bin}}^A = \{p^A(\mathbf{c}, t) \mid p(\mathbf{c}) \in H_P \text{ is a true ground atom of the built-in predicate } p\} \cup \{p^A(\mathbf{c}, f) \mid p(\mathbf{c}) \in H_P \text{ is a false ground atom of built-in predicate } p\}.$$

If a program P^A does not contain built-in predicates then $m_{\text{bin}}^A = m_{\perp}^A$.

Proposition 2 For any $m^A, m_1^A, m_2^A \in \mathcal{B}^A$ hold:

1. if $m_1^A = m_2^A$ then $m_1^A \approx m_2^A$, but not viceversa: only if m_1^A, m_2^A are minimal then $m_1^A \approx m_2^A$ implies $m_1^A = m_2^A$
2. $m^A \supseteq \sim \sim m^A = - - m^A \approx m^A$, $\sim - m^A = - \sim m^A$.
3. $-(m_1^A \cap m_2^A) = -m_1^A \cup -m_2^A$, $-(m_1^A \cup m_2^A) = -m_1^A \cap -m_2^A$.
4. $\sim (m_1^A \wedge m_2^A) = \sim m_1^A \vee \sim m_2^A$, $\sim (m_1^A \vee m_2^A) = \sim m_1^A \wedge \sim m_2^A$.
5. $-(m_1^A \otimes m_2^A) = -m_1^A \oplus -m_2^A$, $-(m_1^A \oplus m_2^A) = -m_1^A \otimes -m_2^A$.
6. $\sim (m_1^A \cap m_2^A) = \sim m_1^A \cap \sim m_2^A$, $\sim (m_1^A \cup m_2^A) = \sim m_1^A \cup \sim m_2^A$.
7. $\sim (m_1^A \otimes m_2^A) = \sim m_1^A \otimes \sim m_2^A$, $\sim (m_1^A \oplus m_2^A) = \sim m_1^A \oplus \sim m_2^A$.
8. $-(m_1^A \wedge m_2^A) =_t -m_1^A \wedge -m_2^A$, while $-(m_1^A \wedge m_2^A) \geq_k^A -m_1^A \wedge -m_2^A$.
9. $-(m_1^A \vee m_2^A) =_t -m_1^A \vee -m_2^A$, while $-(m_1^A \vee m_2^A) \geq_k^A -m_1^A \vee -m_2^A$.

It is easy to verify that a representative $f_R(m^A)$, of any minimal consistent m^A , is an en-interpretation such that $m^A \subseteq f_R(m^A)$.

Proposition 3 $\langle \mathcal{P}(H_P^A), \leq_t^A, \leq_k^A \rangle$ is a complete infinitarily distributive bilattice: its knowledge negation, $-$, coincides with the truth negation of the truth ordering \subseteq . Thus, \sim reverses the \leq_t ordering, while preserving the \leq_k and \subseteq orderings; $-$ reverses the \leq_k and \subseteq , while preserving the \leq_t ordering.

There is a bijection, $\delta : 2^{H_P^A} \simeq \mathcal{B}^A$, such that for any $m^A \in \mathcal{B}^A$, $I^A = \delta^{-1}(m^A) : H_P^A \rightarrow 2$ is an en-interpretation, where, for any ground atom $p^A(\mathbf{c}, \alpha)$, $I^A(p^A(\mathbf{c}, \alpha)) = t$, if $p^A(\mathbf{c}, \alpha) \in m^A$; f , otherwise; and for any $I^A \in 2^{H_P^A}$, $m^A = \delta(I^A) =_{\text{def}} \{p^A(\mathbf{c}, \alpha) \mid I^A(p^A(\mathbf{c}, \alpha)) = t\}$.

Proposition 4 (Consistency) If P^A is an encapsulated logic program based on a 4-valued bilattice \mathcal{B}_4 , and $\mathcal{B}^A =_{\text{def}} \langle \mathcal{P}(H_P^A), \subseteq, \leq_t^A, \leq_k^A \rangle$ is the trilattice of all its en-

interpretations, with $H_P = \{p(c) \mid p(c, \alpha) \in H_P^A\}$, than for any $m^A \in \mathcal{B}^A$

1. if $m^A =_t -m^A$ then it is representative member and a 2-valued consistent (or "exact") Herbrand interpretation $I : H_P \rightarrow \{t, f\}$, (t, f are exact). For any m^A the member $m^A \vee -m^A$ is "exact".

2. if $m^A \leq_k^A -m^A$ then its representative is a 3-valued consistent Herbrand interpretation $I : H_P \rightarrow \{t, f, \perp\}$, (t, f, \perp are 3-valued consistent).

3. if $n_t(m^A) \oplus p_t(m^A) \leq_k^A -m^A$ then its representative is a 4-valued consistent Herbrand interpretation $I : H_P \rightarrow \mathcal{B}_4$. All values in \mathcal{B}_4 are 4-valued consistent.

Proof. Easy to verify. From the points 5 and 6 of the Proposition 2 holds that each 2-valued consistent member is also 3- and 4-valued consistent, and that each 3-valued consistent member is also 4-valued consistent.

Definition 5. A complete semilattice is a partially ordered set, that is closed under arbitrary meets and under joins of directed subsets. A subset D is directed if for all $x, y \in D$ there is $z \in D$ such that $x \leq z$ and $y \leq z$.

Now we will define a sublattice of \mathcal{B}^A which corresponds to consistent many-valued interpretations $I : H_P \rightarrow \mathcal{B}_4$ of a program $P = \mathcal{F}_r(P^A)$. Recall that each representative member m^A is not partial w.r.t. the Herbrand base, and that such completeness is obtained by assigning the default value unknown, \perp , to all remaining ground atoms.

Proposition 5 The 4-valued consistent members in \mathcal{B}^A , i.e., c-interpretations, are closed under \sim , \otimes , and under \wedge , \vee and their infinitary versions. The 4-valued consistent members, or c-interpretations, constitute a complete semilattice under \leq_k^A .

$\mathcal{B}_C^A =_{\text{def}} \{f_R(m^A) \mid n_t(m^A) \oplus p_t(m^A) \leq_k^A -m^A\}$, being closed under Π and under directed Σ (applied to directed sets). The $f_R : (\mathcal{B}^A, \subseteq) \rightarrow (\mathcal{B}^A, \leq_k^A)$ and $\sim : (\mathcal{B}^A, \leq_k^A) \rightarrow (\mathcal{B}^A, \subseteq)$ are two homomorphisms which preserve orderings.

There is an isomorphism, $(\delta\Theta)^{-1} = \Theta^{-1}\delta^{-1} : \mathcal{B}_C^A \rightarrow \mathcal{B}_4^{H_P}$, where $\Theta^{-1} : 2^{H_P} \rightarrow \mathcal{B}_4^{H_P}$ is inverse of Θ , such that for any $I^A \in 2^{H_P}$, $v_B = \Theta^{-1}(I^A)$ and for every annotated ground atom, if $I^A(p^A(c, \alpha)) = t$ then $v_B(p(c)) = \alpha$.

Proof. Let prove the closure for \sim : if m^A is a c-interpretation, then $n_t(m^A) \oplus p_t(m^A) \leq_k^A -m^A$, and $\sim (n_t(m^A) \oplus p_t(m^A)) \leq_k^A \sim -m^A$, and $\sim n_t(m^A) \oplus \sim p_t(m^A) \leq_k^A -\sim m^A$; so we obtain $p_t(\sim m^A) \oplus n_t(\sim m^A) \leq_k^A -\sim m^A$, thus $\sim m^A$ is a c-interpretation. For a 4-valued consistent members in \mathcal{B}^A hold:

(a) $n_t^2(m_1^A) \otimes n_t^2(m_1^A) \leq_k^A -(m_1^A \vee m_2^A)$, and $p_t^2(m_1^A) \otimes p_t^2(m_1^A) \leq_k^A -(m_1^A \wedge m_2^A)$. From $n_t(m^A) \oplus p_t(m^A) \leq_k^A -m^A$ we have (p_t is monotonic) $p_t(n_t(m^A) \oplus p_t(m^A)) \leq_k^A p_t(-m^A) \leq_k^A -m^A$, i.e., $p_t n_t(m^A) \oplus p_t^2(m^A) \leq_k^A -m^A$, thus $p_t^2(m^A) \leq_k^A -m^A$.

Analogously, (n_t is monotonic), $n_t^2(m^A) \leq_k^A -m^A$. Thus, $n_t^2(m_1^A) \otimes n_t^2(m_2^A) = n_t^2(m_1^A) \vee n_t^2(m_2^A) \leq_k^A -m_1^A \vee -m_2^A = -(m_1^A \vee m_2^A)$, and $p_t^2(m_1^A) \otimes p_t^2(m_2^A) = p_t^2(m_1^A) \wedge p_t^2(m_2^A) \leq_k^A -m_1^A \wedge -m_2^A = -(m_1^A \wedge m_2^A)$.

(b) $n_t p_t^2(m_1^A) \oplus n_t p_t^2(m_2^A) \oplus p_t n_t^2(m_1^A) \oplus p_t n_t^2(m_2^A) \leq_k^A -(m_1^A \odot m_2^A)$, where $\odot \in \{\wedge, \vee\}$.

(c) $n_t(m_1^A \wedge m_2^A) \leq_k^A -(m_1^A \wedge m_2^A)$. (d) $p_t(m_1^A \wedge m_2^A) \leq_k^A -(m_1^A \wedge m_2^A)$.
(e) $n_t(m_1^A \vee m_2^A) \leq_k^A -(m_1^A \vee m_2^A)$. (f) $p_t(m_1^A \vee m_2^A) \leq_k^A -(m_1^A \vee m_2^A)$.

Now, from (c) and (d), we conclude that $n_t(m_1^A \wedge m_2^A) \oplus p_t(m_1^A \wedge m_2^A) \leq_k^A -(m_1^A \wedge m_2^A)$, and, from (e) and (f), we conclude $n_t(m_1^A \vee m_2^A) \oplus p_t(m_1^A \vee m_2^A) \leq_k^A -(m_1^A \vee m_2^A)$, i.e., the closure of r-interpretations under \wedge and \vee . For \otimes , instead, we have that

$n_t(m_1^A \otimes m_2^A) \oplus p_t(m_1^A \otimes m_2^A) = (n_t(m_1^A) \oplus p_t(m_1^A)) \otimes (n_t(m_2^A) \oplus p_t(m_2^A)) \otimes \dots$,
 by distributivity, $\leq_k^A (n_t(m_1^A) \oplus p_t(m_1^A)) \otimes (n_t(m_2^A) \oplus p_t(m_2^A))$
 $\leq_k^A -(m_1^A) \oplus -(m_2^A) = -(m_1^A \otimes m_2^A)$, by De Morgan.

Suppose S is a set of r -interpretations that also is *directed* by \leq_k^A . Let prove that $\sum S$ is a r -interpretation. Since S is directed, for any two $m_1^A, m_2^A \in S$ there is some $m^A \in S$ with $m_1^A \leq_k^A m^A$ and $m_2^A \leq_k^A m^A$. Thus,

$n_t(m_1^A) \oplus p_t(m_1^A) \leq_k^A n_t(m^A) \oplus p_t(m^A) \leq_k^A -m^A$, and
 $n_t(m_2^A) \oplus p_t(m_2^A) \leq_k^A n_t(m^A) \oplus p_t(m^A) \leq_k^A -m^A$, so
 $(n_t(m_1^A) \oplus p_t(m_1^A)) \oplus (n_t(m_2^A) \oplus p_t(m_2^A)) \leq_k^A -m^A$.

Consequently, (for brevity we denote $\sum_{m \in S}$ by \sum),

$n_t(\sum(m^A)) \oplus p_t(\sum(m^A)) = \sum(n_t(m^A) \oplus p_t(m^A)) \leq_k^A -\sum m^A$.

It is easy to verify that for any two $m_1^A \subseteq m_2^A$ in \mathcal{B}^A holds $f_R(m_1^A) \leq_k^A f_R(m_2^A)$, and for any two $m_1^A \leq_k^A m_2^A$ in \mathcal{B}^A holds $\sim\sim(m_1^A) \subseteq \sim\sim(m_2^A)$, thus f_R and $\sim\sim$ are two order preserving homomorphisms.

From definitions we have that $\Theta\Theta^{-1}$ and $\Theta^{-1}\Theta$ are identities, thus Θ is an isomorphism, and having that also δ is an isomorphism, then also their composition is an isomorphism.

The Knaster-Tarski like theorem for monotonic functions in a complete sublattice is partially valid (the greatest fixpoints may not exist as in complete lattices): the *least fixpoint exists* for each monotonic function ('truth revision') w.r.t. \leq_k^A ordering. As consequence of the isomorphism $\delta\Theta$, and two order-preserving homomorphisms, f_R , $\sim\sim$, all of Fitting's fixpoint semantics definitions for a 4-valued stable models for a logic programming [8], based on a monotonic 'truth revision' operators w.r.t. \leq_k^A ordering over a Function space bilattice \mathcal{B}_4^{HP} , are directly translated into the complete semilattice w.r.t. \leq_k^A of c -interpretations \mathcal{B}_C^A , and, successively, into the complete semilattice w.r.t. \subseteq of the minimal consistent interpretations.

4 Fixpoint semantics of encapsulated logic programs

All encapsulated logic programs have a nice property: their Herbrand instantiations are *positive* logic programs w.r.t. c -interpretations; such c -interpretations define a complete semilattice, \mathcal{B}^A under \subseteq .

Such property, analogously as for positive logic programs, induces that the monotonic 'immediate consequence operator', T_P , of a program P^A has the least fixpoint (the least fixpoint corresponds to the least Herbrand model of P^A). In [5] is given also the coalgebraic semantics for such logic programs.

Now we can introduce the suitable transcription, taking the particular rule of built-in predicates, of the Fitting's definition (Def 18 in [8]) of a 4-valued single-step truth revision operator over a Function space \mathcal{B}_4^{HP} (of 4-valued Herbrand interpretations) for many-valued logic programming, $\Phi_P : \mathcal{B}_4^{HP} \rightarrow \mathcal{B}_4^{HP}$.

We have to remark that the *incompleteness* of database defined by ordinary 3-valued logic programs is expressed by existentially quantified heads of clauses; that is, some of the attributes of the head-predicate are not defined by a program (consider for example the foreign key integrity constraints in a global schema of data integration systems for incomplete data sources [15,7]) and for such attributes are introduced Skolem func-

tions. Consequently, to any ground atom which contains the Skolem terms, the *unknown* logic value \perp is associated.

Definition 6. (Fitting's like 4-valued operator [8]) Let P^* be a Herbrand instantiation of an encapsulated logic program P^A . Then $\mathcal{F}_r(P^*)$ is a Herbrand instantiation of the (standard) logic program $P =_{def} \mathcal{F}_r(P^A)$. We define the Fitting's 4-valued operator $\Phi_P : \mathcal{B}_4^{H_P} \rightarrow \mathcal{B}_4^{H_P}$, such that for any 4-valued interpretation $v \in \mathcal{B}_4^{H_P}$, $\Phi_P(v) = w$, where $w : H_P \rightarrow \mathcal{B}_4$ is the unique interpretation determined as follows:

1. if the pure ground atom $A \in H_P$ is not the head of any member of $\mathcal{F}_r(P^*)$, then $w(A) = f$.
2. If $A \leftarrow B$ occurs in $\mathcal{F}_r(P^*)$, then $w(A) = v_B(B)$, where v_B is the valuation extension to all ground formulae.

Fitting showed that Φ_P is monotonic w.r.t. \leq_k ordering and that the usual Knaster-Tarski theorem gives smallest fixed points, which correspond to the 4-valued stable models of a logic program P .

Definition 7. Let P^A be an encapsulated logic program, obtained from the many-valued program P . An associated 'encapsulated immediate consequence' operator, $T_P^A : \mathcal{B}^A \rightarrow \mathcal{B}^A$, is defined as follows: let $m_0^A = m_{bin}^A$, and $m_k^A \in \mathcal{B}^A$ be determined in the k -th step ($k \geq 0$). Then

$T_{P,\sigma}^A(m_k^A) = m_{k+1}^A$, where m_{k+1}^A is the unique valuation determined by the following: for a ground atom $p^A(\mathbf{c}, \alpha)$,

1. $p^A(\mathbf{c}, \alpha) \in m_{k+1}^A$, if there is a ground clause $p^A(\mathbf{c}, \alpha) \leftarrow p_1^A(\mathbf{c}_1, \alpha_1), \dots, p_n^A(\mathbf{c}_n, \alpha_n)$ such that $p_i^A(\mathbf{c}_i, \alpha_i) \in m_k^A$ for all $1 \leq i \leq n$, and $\alpha = \alpha_1 \wedge \dots \wedge \alpha_n$.
2. $p^A(\mathbf{c}, \alpha) \notin m_{k+1}^A$, otherwise.

Proposition 6 (Well-founded semantics) If P^A is an encapsulated logic program then the 'encapsulated immediate consequence', $T_P^A : \mathcal{B}^A \rightarrow \mathcal{B}^A$, is monotonic, i.e., $m^A \subseteq n^A$ implies $T_P^A(m^A) \subseteq T_P^A(n^A)$, and holds $m^A \subseteq T_P^A(m^A)$.

The set of all minimal consistent members in \mathcal{B}^A is closed under the ontological operator T_P , that is, if m^A is a minimal consistent then also $T_P^A(m^A)$ is a minimal consistent. Let the smallest fixed point of T_P^A be $m^A \in \mathcal{B}^A$, and the well-founded model of P be the interpretation $I : H_P \rightarrow \mathcal{B}$, then for any ground atom $p^A(\mathbf{c}, \alpha) \in H_P^A$ holds $\delta^{-1}(m^A)(p^A(\mathbf{c}, \alpha)) = t$, if $I(p^A(\mathbf{c})) = \alpha$; f otherwise.

Thus, the well-founded semantics for general logic programs, obtained by the Fitting's fixpoint operator Φ_P , monotonic with respect to the knowledge bilattice ordering, corresponds to the minimal model of the encapsulated logic programs, obtained from the truth-monotonic operator T_P^A , as consequence of the semantics of the encapsulation.

In fact the following homomorphism between many-valued functional space of interpretations and 2-valued ontological interpretations holds:

Theorem 1 (Truth-knowledge correspondence)

Let $T_P^A : \mathcal{B}^A \rightarrow \mathcal{B}^A$ be an immediate consequence operator of an encapsulated program P^A , then

$\Phi_P = (\delta\Theta)^{-1} f_R T_P^A \sim \delta\Theta$ is the correspondent Fitting's 4-valued operator of a program $P = \mathcal{F}_r(P^A)$, where $\sim \delta\Theta : (\mathcal{B}_4^{H_P}, \leq_k) \rightarrow (\mathcal{B}^A, \subseteq)$ and $(\delta\Theta)^{-1} : (\mathcal{B}_C^A, \subseteq) \rightarrow (\mathcal{B}_4^{H_P}, \leq_k)$ are two homomorphisms which preserve orderings.

And viceversa, if $\Phi_P : \mathcal{B}_4^{H_P} \rightarrow \mathcal{B}_4^{H_P}$ is an immediate Fitting's consequence operator, then $T_P^A = \sim\sim \delta\Theta \Phi_P(\delta\Theta)^{-1} f_R$ is the correspondent encapsulated immediate consequence operator.

Consequently, if m^A is the smallest fixed point of the encapsulated logic program operator T_P^A , then $(\delta\Theta)^{-1} f_R(m^A)$ is the smallest fixed point of the Fitting's operator Φ_P , i.e., it is a 4-valued stable model of a program $P =_{def} \mathcal{F}_r(P^A)$.

Proof. Let verify the order preserving property of these two homomorphisms. Let $w, v \in \mathcal{B}_4^{H_P}$, such that $w \geq_k v$, i.e., that for every $\alpha \in \{t, f, \top\}$, $\{w(p(\mathbf{c})) \mid w(p(\mathbf{c})) = \alpha\} \supseteq \{v(p(\mathbf{c})) \mid v(p(\mathbf{c})) = \alpha\}$. We obtain that $\sim\sim \delta\Theta(w) = \{p(\mathbf{c}, \alpha) \mid w(p(\mathbf{c})) = \alpha \text{ and } \alpha \neq \perp\}$, thus $\sim\sim \delta\Theta(w) \supseteq \sim\sim \delta\Theta(v)$.

Let a minimal consistent m^A is a smallest fixpoint for T_P^A , i.e., $m^A = T_P^A(m^A)$, then $v = (\delta\Theta)^{-1} f_R(m^A)$ is the smallest fixpoint of Φ_P . In fact,

$$\begin{aligned} \Phi_P(v) &= \Phi_P(\delta\Theta)^{-1} f_R(m^A) \\ &= (\delta\Theta)^{-1} f_R T_P^A \sim\sim \delta\Theta(\delta\Theta)^{-1} f_R(m^A) \\ &= (\delta\Theta)^{-1} f_R T_P^A \sim\sim f_R(m^A), \text{ from the isomorphism } \delta\Theta \\ &= (\delta\Theta)^{-1} f_R T_P^A(m^A), \text{ from } \sim\sim f_R(m^A) = m^A \\ &= (\delta\Theta)^{-1} f_R(m^A), \text{ by hypothesis } m^A \text{ is a fixpoint} \\ &= v \end{aligned}$$

Suppose that v is not the smallest fixpoint, i.e., that there is an other fixpoint $v' < v$.

Let prove that $m_1^A = \sim\sim \delta\Theta(v')$ is a fixpoint for T_P^A :

$$\begin{aligned} T_P^A(m_1^A) &= \sim\sim \delta\Theta \Phi_P(\delta\Theta)^{-1} f_R(m_1^A) \\ &= \sim\sim \delta\Theta \Phi_P(\delta\Theta)^{-1} f_R \sim\sim \delta\Theta(v') \\ &= \sim\sim \delta\Theta \Phi_P(\delta\Theta)^{-1} \delta\Theta(v'), \text{ from identity } f_R \sim\sim \\ &= \sim\sim \delta\Theta \Phi_P(v'), \text{ from the isomorphism } \delta\Theta \\ &= \sim\sim \delta\Theta(v'), \text{ by hypothesis } v' \text{ is a fixpoint} \\ &= m_1^A. \end{aligned}$$

From $v' < v$ holds that $\exists p(\mathbf{c}, \alpha)$, $\alpha \in \{t, f, \top\}$ such that $p(\mathbf{c}, \alpha) \in v$ and $p(\mathbf{c}, \alpha) \notin v'$, thus $m_1^A = \sim\sim \delta\Theta(v') \subset \sim\sim \delta\Theta(v) = m^A$, i.e., that m^A is not the smallest fixpoint of T_P^A which is in contradiction with hypothesis: thus we conclude that $v = (\delta\Theta)^{-1} f_R(m^A)$ is really the smallest fixpoint of Φ_P .

Note that two homomorphisms, $(\delta\Theta)^{-1} f_R$, $\sim\sim \delta\Theta$, are key facts for the truth-knowledge ordering correspondence in encapsulated ('meta') and classical many-valued logic programming levels. In fact, an increase in a many-valued knowledge order \leq_k , loosely means more literals acquire truth values from a truth space \mathcal{B}_4 , and that is just an increase in a 'meta' 2-valued truth order \subseteq , in which an increase means 'more encapsulated ground atoms are true'.

5 Conclusion

We have presented a programming logic capable of handling inconsistent beliefs, based on the 4-valued Belnap's bilattice, which has clear model theory and fixed point semantics. In the process of the encapsulation we distinguish two levels: the many-valued level of ordinary logic programs with epistemic negation based on a bilattice operators, and the 'meta' level of encapsulated logic programs. In such an abstraction we obtained

a kind of an ontological Predicate Logic where fixpoint 'immediate consequence' operator is always continuous, and which is *computationally equivalent* to standard Fitting's fixpoint semantics.

A contribution from the theoretical point of view is given in understanding why (when we pass from a 2-valued to many-valued logic programming) we pass from truth to knowledge ordering. The knowledge ordering at many-valued epistemic level is homomorphic to the truth ordering at ontological 'meta' level of logic programming.

References

1. M.Ginsberg, "Multivalued logics: A uniform approach to reasoning in artificial intelligence," *Computational Intelligence*, vol.4, pp. 265–316, 1988.
2. M.C.Fitting, "Billattices and the semantics of logic programming," *Journal of Logic Programming*, 11, pp. 91–116, 1991.
3. N.D.Belnap, "A useful four-valued logic," In J.-M.Dunn and G.Epstein, editors, *Modern Uses of Multiple-Valued Logic*. D.Reidel, 1977.
4. Z. Majkić, "Ontological encapsulation of many-valued logic," *19th Italian Symposium of Computational Logic (CILC04)*, June 16-17, Parma, Italy, 2004.
5. Z. Majkić, "Coalgebraic semantics for logic programming," *18th Workshop on (Constraint) Logic Programming, WLP 2004*, March 04-06, Berlin, Germany, 2004.
6. Maurizio Lenzerini, "Data integration: A theoretical perspective.," in *Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002)*, 2002, pp. 233–246.
7. Z. Majkić, "Fixpoint semantic for query answering in data integration systems," *AGP03 - 8.th Joint Conference on Declarative Programming*, Reggio Calabria, pp. 135–146, 2003.
8. M.C.Fitting, "Fixpoint semantics for logic programming: A survey," *Theoretical Computer Science*, 278, pp. 25–31, 2002.
9. G.Escalada Imaz and F.Manyá, "The satisfiability problem for multiple-valued horn formulae," In *Proc. International Symposium on Multiple-Valued Logics (ISMVL)*, Boston, IEEE Press, Los Alamitos, pp. 250–256, 1994.
10. R.Hahnle, "Automated deduction in multiple-valued logics," *Oxford University Press*, 1994.
11. B.Beckert, R.Hahnle, and F.Manyá, "Transformations between signed and classical clause logic," In *Proc. 29th Int.Symposium on Multiple-Valued Logics*, Freiburg, Germany, pp. 248–255, 1999.
12. H.A.Blair and V.S.Subrahmanian, "Paraconsistent logic programming," *Theoretical Computer Science*, 68, pp. 135–154, 1989.
13. M.Kifer and E.L.Loizinskii, "A logic for reasoning with inconsistency," *Journal of Automated reasoning* 9(2), pp. 179–215, 1992.
14. M.Kifer and V.S.Subrahmanian, "Theory of generalized annotated logic programming and its applications," *Journal of Logic Programming* 12(4), pp. 335–368, 1992.
15. A.Cali, D.Calvanese, G.De Giacomo, and M.Lenzerini, "Data integration under integrity constraints," in *Proc. of the 14th Conf. on Advanced Information Systems Engineering (CAiSE 2002)*, 2002, pp. 262–279.
16. M.C.Fitting, "Billattices are nice things," *Proceedings of Conference on Self-Reference*, Copenhagen, 2002.
17. Z. Majkić, "Many-valued intuitionistic implication and inference closure in a bilattice based logic," *Notes in* <http://www.dis.uniroma1.it/~majkic/>, 2004.
18. T.Przymusiński, "Every logic program has a natural stratification and an iterated fixed point model," In *Eighth ACM Symposium on Principles of Databases Systems*, pp. 11–21, 1989.

Impact- and Cost-Oriented Propagator Scheduling for Faster Constraint Propagation

Georg Ringwelski¹ and Matthias Hoche²

¹ 4C, University College Cork, Ireland g.ringwelski@4c.ucc.ie

² Fraunhofer FIRST, Kekuléstr.7, 12489 Berlin, Germany mathoc@first.fhg.de

Abstract. Constraint Propagation can be speeded up significantly by choosing a good execution order for propagators. A propagator is an implicit representation of a constraint which is widely used in today's powerful constraint solvers. In this paper we evaluate different ways to find good execution orders automatically during runtime. We extend previous work in this area by two new techniques: fair-scheduling and impact-oriented prioritization of propagators.

1 Introduction

Constraint Propagation is one of the key techniques of Constraint Programming. In finite integer domains Propagation is often based on efficient filtering algorithms of constraints in compiled form. These algorithms implement so-called *propagators*. Thus, they represent the semantics of their associated constraint in an implicit way and detect inconsistencies by inferring wipe-outs of variable domains. Propagators are the basis of constraint propagation in the currently fastest finite domain constraint solvers which include SICStus Prolog, ECLⁱPS^e, ILOG Solver, CHIP, Mozart, GNU Prolog and others.

The propagators are used by the constraint solver to compute a fixed point of constraint propagation as a preprocessing step or during search. This fixed point can be a bounds-consistent CSP but there are no general requirements on the properties of that fixed point. However, given a set of propagators the fixed point is uniquely determined. To compute this determined fixed point, the propagators have to be re-executed until no further domain restrictions are inferable. Naturally this can be done in arbitrary ways and there are many different orders of execution which lead to different performance. Very little is known (to the public) about the execution order in the mentioned commercial solvers.

Related Work. It seems that most of the mentioned constraint solvers use a priority queue in which the propagators are buffered for execution. CLP-based languages, such as ECLⁱPS^e or SICStus use (2 – 13 different) static priority values associated to each constraint respectively its propagator. The propagators with higher priority are executed earlier and thus more often. Little was published about which priority-values are chosen for the constraints [7, 8, 11]. Most of the used priority values seem to be set according to the complexity

of the propagator which they are associated to: expensive computations are delayed until all the cheap things are finished. In ECLⁱPS^e user-defined constraints can be associated to priority values between 1 and 12, but we are not aware of any guidelines on which numbers to choose. Similar mechanisms are available in implementations of CHR [6], where the application of certain rules can be prioritized. However, CHR does not provide any automatic mechanisms to find appropriate values at all. Another way to prioritize was found in the context of propagator learning for GNU Prolog [9]: propagators are prioritized regarding the tightness of their respective constraints. Propagators of tight constraints are executed earlier. In some CP systems also other data structures than priority queues, such as stacks for example, can be used to buffer the propagators (e.g. CHOCO). But again, the decision of which data structure to use is left to the user and guidance on how to make this decision is missing. In early work on CSP Wallace and Freuder [12] investigated several heuristics to choose the best arc during Arc-consistency enforcement with AC-3. Many of their ideas can also be found in work on propagator scheduling. They additionally proposed some heuristics which we plan to apply to propagator based constraint propagation such as learning good priority-values during execution.

Contribution. In this paper we investigate the correlation of execution orders for propagators with the result and performance of constraint solving. In order to ensure correct results we prove that the execution order can be safely manipulated and that idempotent propagators can be omitted from being re-inserted when they are already waiting for execution. Knowing this, we can compare the performance of various execution orders which can be automatically computed during runtime because the result is uniquely determined. The order will be specified by various buffers to store propagators for their execution: FIFO, LIFO, a priority queue and a fair scheduling buffer known from CPU scheduling in Operating Systems. Fairness was not considered in propagator scheduling before. These buffers are varied to become sets, i.e. not store any idempotent propagator twice [10, 8, 11]. Furthermore, we consider two classes of prioritization: cost-oriented [7, 11] and impact-oriented. The first prioritizes computationally cheap propagators, the latter prioritizes propagators which may have a large impact, i.e. which can be expected to prune large portions of the search space. The latter is a new class of execution orders which was not considered before. Finally we compare the use of static priority-values to the dynamic adaptation of the values during runtime.

Organization. In the next Section we describe the theoretical background of constraint propagation with propagators and show that the execution order can be safely varied. In Section 3 we specify more precisely the setting of this investigation. We identify a (necessarily supplied) data structure with which we can manipulate the execution order of propagators and propose and verify several implementations of it. In Section 4 we evaluate these implementations with different benchmark problems and conclude in Section 5.

2 Theoretical Background

A Constraint Satisfaction Problem (V, C, D) is given by a set of variables $V = \{v_1, v_2, \dots, v_n\}$, a set of constraints C and a set of variable domains $D = \{D_1, D_2, \dots, D_n\}$ associated to the variables. Each constraint $c \in C$ is specified by its scope which is a vector of variables (v_i, \dots, v_j) and its semantics, which describes the allowed simultaneous assignments of the variables and is thus a subset of $D_i \times \dots \times D_j$. In this paper we assume all domains to be finite sets of integers.

Constraint Propagation is generally formalized by the Chaotic Iteration (CI) of propagators³ [4]. Given a CSP (V, C, D) a set of propagators $prop(c)$ is deduced from the semantics of any constraint $c \in C$. Each constraint can use more than one propagator to implement its semantics (by sequentially executing all of them) [10, 11]. Each propagator p uses input variables $in(p)$ from which it may infer reductions of the domains of its output-variables $out(p)$ of which it can change the domain. Propagators are monotonic functions (i.e. $D \sqsubset D' \Leftrightarrow p(D) \sqsubset p(D')$) in the Cartesian product of the powersets of all variable domains $\mathcal{P}(D_1) \times \dots \times \mathcal{P}(D_n)$. Furthermore, propagators must ensure that for each variable the domain can only be restricted but not extended: given a propagator p with $p((D_1, \dots, D_n)) = (D'_1, \dots, D'_n)$, then $D'_i \subseteq D_i$ must hold. Given this, the Chaotic Iteration over all propagators induced by C (i.e. $\bigcup_{c \in C} prop(c)$) will reach a uniquely determined fixed point [1]. This is a vector of variable domains for which no further restrictions can be inferred with the used propagators. The starting point for this approximation are the initial domains D .

However, Chaotic Iteration does not define how the fixed point is computed. In theory, each propagator is executed infinitely often to reach the fixed point. When we actually want to solve a CSP we naturally have to find ways to reach this fixed point without infinite computations. A round-robin algorithm will have poor performance, as most propagators that are executed will not be able to infer any further domain reductions. Thus, a more sophisticated algorithm is used in most solvers today. It is described in detail for `clp(FD)` (i.e. the predecessor of GNU Prolog) in [3]. A more formal definition of this concept can be found in [10, 11]. The idea is to store for each variable v all propagators p with $v \in in(p)$ and relate them to domain events on v after which they have to be triggered. With this, only those propagators are re-executed that can possibly infer further domain reductions.

Example 1. A propagator of a constraint $v < w$ only needs to be re-executed upon changes on the largest value of D_w and the smallest value of D_v , all other changes of D_v or D_w may not lead to further domain reductions and can thus be omitted.

³ Many different names for these procedures can be found in the literature. When we restrict ourselves to finite integer domains, they are all practically the same: propagators [11], domain reduction functions [1], closure-operators [5], constraint frames [3] etc.

The generally desired result of Constraint Propagation in a CSP (V, C, D) is generally considered the fixed point of the Chaotic Iteration of the propagators which are induced by C . To start, each propagator will have to be executed at least once. After that, new domain reductions may possibly be inferred (only) from the newly reduced domains. Thus some propagators will have to be executed again in order to approximate the desired fixed point and so on. We define Event-based Propagation to use a simplified version of this execution model. Without restricting generality, we consider only one sort of events, namely *any* change of the domain of one variable. For any other kind of event, as they are listed in [3] or [11] for example, we could define a further iteration rule in the following definition.

Definition 1. *In A CSP (V, C, D) , Event-based Propagation (EBP) is defined iteratively by*

1. *For each $c \in C$ the propagators $\text{prop}(c)$ are executed once*
2. *If D_v is reduced, then all propagators $p \in \text{prop}(c)$ with $v \in \text{in}(p)$ and $c \in C$ are executed.*

We can show that EBP leads to the desired result. This is the fixed point of the Chaotic Iteration of the propagators that are implied by the posted constraints.

Theorem 1. *The Event-Based Propagation of a CSP $\mathcal{A}(C, V, D)$ will lead to the same result as the Chaotic Iteration of $\bigcup_{c \in C} \text{prop}(c)$ starting at D .*

Proof. Soundness : Since the same propagators are used, there cannot exist any values that are pruned in EBP, but not in the Chaotic Iteration.

Completeness : If EBP was not complete, then there would have to be a value that is pruned with CI, but not with EBP. This would imply that a propagator which has the potential to prune that value is not executed in EBP. As every propagator is defined to be executed at least once, this propagator would have to be not re-executed after a domain reduction. However, this contradicts the definition of EBP and can thus not have happened.

This theorem implies that the order in which the propagators are executed is irrelevant. The only crucial requirement for the correctness is that each propagator is re-executed whenever a respective event has occurred.

Corollary 1. *EBP is correct with any execution order of the propagators.*

3 Propagator Scheduling

During propagation, the propagators are executed and re-executed when they are triggered due to a domain reduction. Since many constraints can be triggered by one reduction we need a buffer to store all the propagators which remain to

be executed. The basic version of such a buffer is a queue, i.e. a FIFO data structure. However, there seems to be a lot of potential to speed up constraint propagation by using more sophisticated buffers [10]. The desired effect of this is to make the execution of certain propagators obsolete, because their effect has already been achieved.

Example 2. Consider the CSP $A < B, B < C$. A propagator for each constraint must be executed at least once such that the initial buffer will be $\langle A < B, B < C \rangle$. After one execution we might obtain $\langle B < C, B < C \rangle$ which will lead to an obsolete execution of $B < C$.

The question we want to answer in this paper is: how can we implement the buffer in order to speed up EBP by omitting useless executions of propagators? Schulte and Stuckey have already described three classes of optimizations in [11]: EBP as described above; prevent the re-insertion of idempotent propagators; execute preferably the propagators with low computational complexity. In this paper we concentrate on EBP as this is most widely used and seems to be most efficient. We investigate a wider set of possible buffers and consider also impact-oriented prioritization and thus qualitatively extend the state-of-the-art.

The characterizing property of a buffer is its implementation of the methods *push* and *pop* which add respectively delete values from the buffer. We consider four basic data structures for the buffer:

FIFO a standard queue, *pop* will always return the item which was *pushed* first.

LIFO a standard stack, *pop* will always return the item which was *pushed* last.

Order- \prec a static priority queue, *pop* will always return the item which has the highest priority wrt. \prec

Sched- \prec a fair priority queue, *pop* will usually return the item with the largest wrt. \prec , this may differ when low-priority items are waiting too long.

Order- \prec will always return the highest priority item it stores. When two items have the same priority, we schedule them in FIFO manner. To depend the scheduling only on priorities can lead to “starvation” of items with low priority. Starvation can be prevented by a fair scheduling algorithm as used in many areas of computer science (e.g. CPU scheduling) today. For propagator scheduling we can only use non-preemptive algorithms to prevent Reader-Writer-Problems on the variable domains. The efficiency-results known from CPU scheduling for example cannot be expected to hold in propagator scheduling, because the number of future jobs depends on the prioritization⁴. We use a simple form of an “aging” algorithm to implement **Sched- \prec** . Aging prevents starvation by considering a combination of the actual priority and the time a task is waiting for execution. We implement this by setting the priority in Sched- \prec to the quotient of the priority in Order- \prec over the solver lifetime when the propagator is added to the buffer.

⁴ “shortest job first” will not necessarily be the best for our purposes although we know the costs of the jobs, i.e. the propagator’s computational complexity.

In order to improve the basic buffers, we considered methods to prevent the multiple storage of idempotent propagators [10, 8, 11]. This means, that any propagator will only be stored, if it is not already in the buffer. As we thus potentially execute less propagators than EBP, we need to show that Propagation remains correct.

Theorem 2. *EBP will remain correct, if idempotent propagators are not multiply stored in the propagator queue.*

Proof. Because of Corollary 1 we do not restrict generality when we make the following assumption: whenever two identical propagators are stored in the buffer, then they will be executed directly after each other. Since we assume propagators to be idempotent, the execution of the second will not have any effect and can thus be omitted.

Knowing this, we can safely prevent the storage of idempotent propagators which are already in the buffer. For this we propose a variant for all of the above described basic data types:

X-set will not *push* an item, if it is already stored in buffer **X**.

What remains to specify are the possible orders \prec we use for the **Order-** \prec and **Sched-** \prec buffers. We propose the following:

- comp** the complexity of the propagator, as also described in [7, 11]
- bound** a lower bound of the expected pruning achieved by the propagator (e.g. the portion of bounds-consistent values of its variables)
- tight** an estimation of the tightness of the constraint

The latter two orders were to our knowledge not (knowingly⁵) applied to propagator scheduling before. With considering the expected pruning (**bound**) and the estimated tightness (**tight**) we tackle the topic of the predicted impact of a propagator. A tight constraint can be expected to have a larger impact than a loose constraint, i.e. it can be expected to prune more values and thus make the execution of other propagators obsolete. According to the fail-first-principle [2] propagators of constraints with a high expected impact should thus be executed first. They have the highest potential to prune the domains and will usually be able to prune the most.

Finally all these prioritization-strategies can be applied statically and dynamically. In the static version the respective values will be computed once, when the constraint is first inserted. In the dynamic case, the priority is computed whenever the propagator is (re-)inserted in the buffer.

X-Y-dyn will compute the priority **Y** dynamically before every insertion to the buffer **X**.

⁵ Arnaud Lallout denied that this is done by their algorithm after the presentation of [9] at the FLAIRS'04 conference.

4 Empirical Evaluation

All our methods target the reduction of the number of executed propagators during constraint solving. However, they all yield extra computational costs which have in our implementations constant or linear worst case complexity. Thus an empirical evaluation is necessary to show the usefulness of the proposed methods. We think that it does not make much sense to count the executed propagators for evaluation purposes, since the propagators differ a lot in complexity. A more fine-grained metric, which counts separately for propagators of different complexity classes would make more sense but the results would be very hard to compare. Instead we use good old runtime to check whether the extra effort used for smart propagator scheduling pays. All experiments were run with the firstcs solver [7] on a 1.8GHz Linux PC. In the following we use the abbreviations described in the previous section to specify the used algorithm. For example Order-set-comp-dyn will thus stand for experiments with an complexity-ordered dynamic queue buffer where multiple insertions of idempotent propagators are prevented.

It can be seen in [11] that (in contrast to the counted propagation steps) no single method can be expected to be always the best. Thus it is essential to consider various benchmark problems, which use varying combinations of used constraints:

queens The famous 27-queens problem implemented with primitive constraints only

queens-a 27-queens implemented with three AllDifferent constraints

golomb The golomb ruler problem with 10 marks and the optimal length of 55 implemented with primitive constraints only

golomb-a The golomb ruler with global constraints

L10x5,L15x5,L10x10.1,L10x10.3,L10x10.5,FT10x10 Job-Shop-Scheduling

Problems from the OR library which can be found at

<http://www.brunel.ac.uk/depts/ma/research/jeb/info.html>.

These problems are implemented with global scheduling constraints and simple in-equations.

First we compare the basic buffer types. Figure 1 shows the respective results. They can be summarized as follows:

- The effort to use the **set**-variant of **LIFO** almost always pays significantly. Thus we used this variant for all further tests.
- **FIFO** is always better than **LIFO**.
- **Order** is better than non-prioritized buffers whenever some diversity in the priority of the propagators exist. This applies to all investigated problems but **Order** profits only when global constraints are used.
- The extra effort to ensure fairness (**Sched** vs. **Order**) during propagation in priority queues does not pay. This may, however, result from a poor aging algorithm. We plan to find better parameters for this in future work.

Next we evaluate the various priority-computations. The results are shown in Figure 2. It can be seen that:

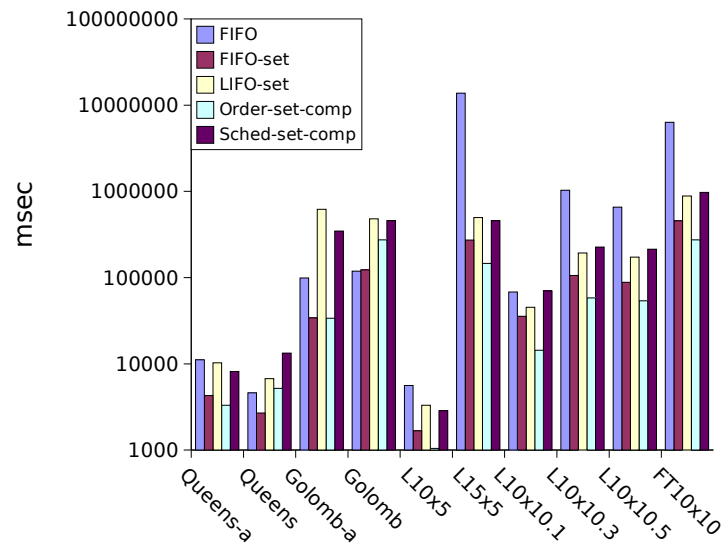


Fig. 1. Comparison basic buffer types.

- The estimation of the tightness seems to be a good measure to express the expected impact of a constraint. The performance of **bound** and **tight** are very similar in almost all tests.
- Whenever global constraints are involved, the complexity based prioritization **compl** is better and otherwise the impact based **tight/bound** are. The poor performance of impact-based methods with global constraints may result from bad estimations of the priority values. The impact of arithmetic constraints can be predicted much more precisely.

Finally we check whether the evaluation of the priority before every insertion of a propagator into the buffer pays. Thus we compare the **dyn** versions of the buffers to the standard case where the priority is only computed once upon the construction of the constraint. The results are presented in Figure 3, they can be summarized as follows:

- It seems that only in problems without global constraints and in combination with the **comp**-priority the dynamic versions perform better than the static versions.
- The relatively complex computation of the expected impact seems to be far too costly to be computed dynamically. The combination **bound-dyn** yields poor results in all experiments.

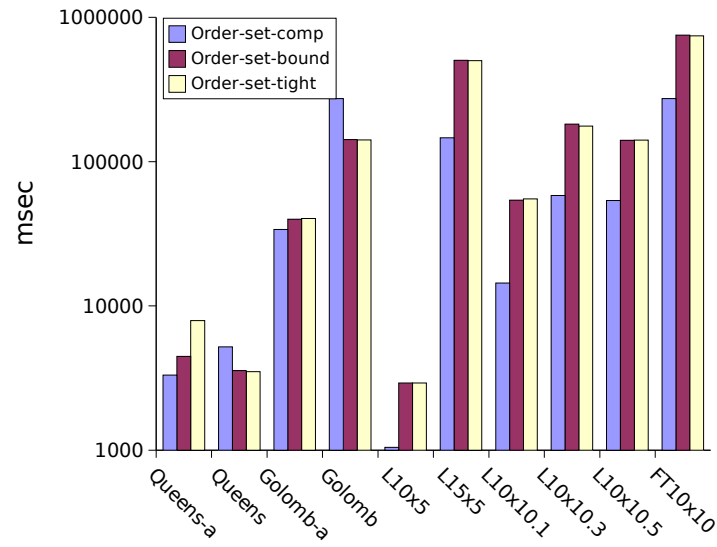


Fig. 2. Comparison of priority orders in **Order** buffer.

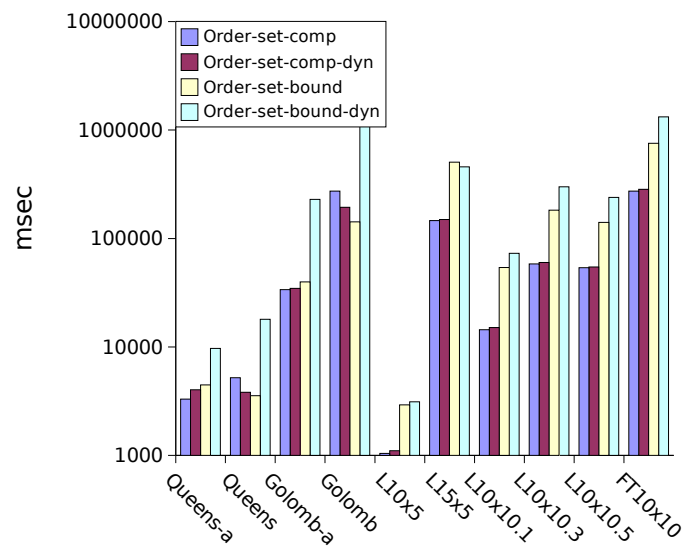


Fig. 3. Comparison of dynamic and static prioritization.

5 Conclusion

We have shown that the execution order of propagators in event-based propagation can be varied while retaining correct propagation results. Furthermore we showed that the re-insertion of idempotent propagators may be safely omitted when they are already buffered for execution. These theoretical results allow to safely implement arbitrary buffers for propagator scheduling.

We implemented such buffers which adapt the execution order of propagators dynamically during runtime in a standard constraint solver. Doing this, we extended the state of the art with two new scheduling techniques: fair-scheduling and impact-oriented prioritization. Ensuring fairness during propagator scheduling does not show any positive effect in our experiments so far. Despite the low computational complexity of our algorithm we could never achieve a runtime improvement. Impact-oriented ordering seems to depend a lot on the used notion of “impact”. Whenever good estimations of the impact are available, the impact-oriented scheduling is better than cost-oriented propagator scheduling. Preventing the multiple storage of propagators seems to be a good way to speed up constraint solving in all our experiments. Computing the priority values dynamically as opposed to just once upon the creation of a constraint does not pay in general. The computation of priority values of propagators should preferably only be executed once.

6 Future Work

In future work we plan to make many more and larger scale experiments and to evaluate more types of prioritization. These include combinations of the heuristics presented here, such as a quotient of cost over impact or the application of different methods in differing contexts (staged propagators [10, 11] or constraint specific metrics such as cost-oriented for global and impact-oriented for primitive constraints). Furthermore we plan to follow up research on the impact of fairness during propagator scheduling. We think results from CPU scheduling can be exploited much more in order to find near to optimal orderings of propagators. Finally we plan to apply learning techniques during scheduling. Propagators that have had a large impact in earlier executions may want to be preferably executed (or not, because their potential is already exhausted). We hope that with all the results gained in these experiments we will be able to find a generally efficient propagator scheduling heuristic which can be chosen as a standard for constraint solvers.

References

1. Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.
2. C.J. Beck, P. Prosser, and R.J. Wallace. Trying again to fail-first. In *Proc. ERCIM/CologNet workshop*, 2004.

3. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.
4. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, 1977. ACM Symposium on Artificial Intelligence and Programming Languages.
5. F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37:1-3:185–212, Oct-Dec 1998.
6. Thom Frühwirth. Constraint handling rules. *Constraint Programming: Basics and Trends, LNCS 910*, 1995.
7. M. Hoche, H. Müller, H. Schlenker, and A. Wolf. firstcs – a pure java constraint programming engine. In *Proc. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL’03*, 2003.
8. Matthias Hoche. Analyse und entwicklung unterschiedlicher scheduling-verfahren zur laufzeit-optimierung der propagation eines java-basierten constraint-lösers. Master’s thesis, TU Berlin, January 2004.
9. A. Legtchenko, A. Lallouet, and A. Ed-Dbali. Intermediate consistencies by delaying expensive propagators. In *Proc. FLAIRS 04*, 2004.
10. Georg Ringwelski. *Asynchrone Constraintlösen*. PhD thesis, Technical University Berlin, 2003.
11. C. Schulte and P. Stuckey. Speeding up constraint propagation. In *Proc. CP04*, September 2004.
12. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proc. 9th Canadian Conf. on AI*, pages 163–169, 1992.

Static and dynamic variable sorting strategies for backtracking-based search algorithms

Henry Müller

Fraunhofer FIRST, Kekuléstr. 7, D-12489 Berlin, Germany
henry.mueller@first.fraunhofer.de

Abstract. This paper presents techniques for the improvement of backtracking based search algorithms in the context of constraint programming. The main idea is to improve the efficiency of the search process by manipulating the variable order within the search algorithm.

Two main strategies of sequence manipulation were developed: Static variable ordering (SVO) analyses the problem structure a priori to set the variables in a convenient sequence. This technique can be applied to any backtracking algorithm. Dynamic variable ordering (DVO) is a backjumping improvement. It reorders not yet labelled variables during search in a way which could be regarded as "forward jumping".

Both static and dynamic variable ordering decrease the number of needed backtracks for the tested problem classes often by several magnitudes. Especially the dynamic reordering is lightweight and robust at the same time. The combination of static and dynamic sorting brought the performance even on par with state of the art SAT solvers, although no SAT-specific adjustments were made.

Keywords. Constraint programming, search, backtracking, backjumping, conflict-directed backjumping, static variable ordering, dynamic variable ordering, AIM, SAT.

1 Introduction

Backtracking search algorithms [5,6,13,14] are prone to thrashing [11] – successive failure because of the same reason – and do initially a lot of needless work. Conflict-directed backjumping (CBJ) [14,15] is an improved backtracking-based search algorithm, which tries to avoid thrashing. CBJ calculates the cause of an inconsistent assignment and jumps over the search tree skipping all the variables between the current inconsistent variable v_i and the calculated cause v_h . Thereby the thrashing occurring on the way from v_i back to v_h is avoided.

In [13] the author improved the performance of conflict-directed backjumping considerably with variable sorting strategies. Investigation led to the creation of two kinds of sorting heuristics: a) Static variable ordering (SVO), which is geared towards the problem structure and applied before the search process is started, and b) dynamic variable ordering (DVO), which is applied during search and optimises backjumping. SVO and DVO try to minimise thrashing even more by reducing the number of visited nodes in the search tree. They are completely orthogonal and can be combined without problems.

The two sorting strategies were tested comprehensively but only briefly described. In this paper both are presented and studied in greater detail. Therefore the discussion will neither go much into the basics of constraint programming, which can be looked up in [3,12,13], nor into the details of the circumstances or the interpretation of the experimental results, which [13] describes in detail.

Section 2 describes the necessary basics and conventions. Section 3 presents two variable sorting strategies. Section 4 shows a roundup of the experimental results concerning the sorting strategies. Section 5 considers related and future work, and in section 6 some conclusions are drawn.

2 Basics and conventions

The methods presented in this paper regard either chronological backtracking (BT) or conflict-directed backjumping (CBJ) [14]. Accurate knowledge of the concrete design, implementation and testing details is not necessary for the understanding, but as a matter of completeness they should be outlined at least:

Among other problems AIM instances [2] and a collection of other SAT problems from the SAT-Ex platform [17] in cnf-form [1] were used for empirical testing of static and dynamic variable ordering. SAT problems are satisfiability problems in propositional logic, they are logic statements in conjunctive normal form (conjunction of disjunctions). AIM instances, which are named after their inventors Asahiro, Iwama and Miyano, are 3-SAT-problems consisting of three variables per disjunction. The problems from the SAT-Ex platform are of different structure and from different domains, they were chosen to enable a sound comparison with other SAT solvers.

The processing of the problems required the implementation of the search algorithms BT and CBJ, a justification system, a cnf-parser [1] and boolean constraints. The test environment was embedded into `firstcs` [9,10,13]. As this solver always delivers full look ahead for constraints of any arity, and the boolean constraints create local consistency, local consistency is always assured for the whole CSP. This consistency level is called Maintaining Local Consistency (MLC), hence we get MLC-BT and MLC-CBJ.

Search situations often focus on the *current* variable, which is the one currently processed by the algorithm. Already assigned variables are called *past* variables, not yet assigned variables are called *future* variables. If the assignment of the current variable is *inconsistent* with an earlier assignment, a *conflict* occurs. Because of MLC, only future variables may report a conflict. The cause of such a conflict, which is an already assigned variable, is acquired via the justification system.

The presented ideas are about variable orderings. When no ordering is especially mentioned, the default ordering is in effect. That is the ordering which is established during the process of problem parsing.

Note 1. All AIM-instances with 50 and 100 variables were tested, problems with 200 variables were too hard for MLC-BT. The group of SAT-problems for the comparison with the top ten SAT-solvers were: bf0432-007, ssa2670-141, ii16e1,

par16-1-c, sw100-49, ais12, pret60_60, hole9. These problems had to be solved in a certain time, otherwise the attempt was considered invalid. For both the AIM instances and the SAT-solver group the algorithms processed each problem until one solution was found or the search tree was traversed.

3 Variable ordering

The idea of variable ordering is to acquire a "good" variable order, which hopefully results in a more efficient search process. What a good ordering is, depends on the circumstances. But in the core it boils down to the character of the problem structure and the behaviour of the applied solution methods. Thus static variable sorting deals with the problem character and dynamic variable sorting refines search behaviour.

3.1 Static variable ordering

The idea of static variable ordering (SVO) is to arrange the variables of the CSP into a convenient sequence, before the search algorithm begins to work. But what is a good initial ordering? A rule of thumb concerning the solution of constraint problems says: "Propagate as much as possible as early as possible." Following this idea, a variable assignment resulting in a lot of propagation would have to be favoured. An assignment causing less propagation would be deferred. Therefore a good measurement for the anticipated amount of propagation for every single variable - the propagation power - is needed.

It is reasonable to use a variable's dependencies on other variables as the propagation measurement. The term "dependency" is quite flexible and can be interpreted differently depending on the problem space and level of abstraction: Looking on a constraint satisfaction problem as a constraint graph, an intuitive candidate for the dependency value is a variable's number of edges. That is the number of constraints the variable participates in. This general view holds for all kinds of CSPs, but one surely also can find more domain-specific measures for concrete problems. If for example a SAT-problem were to be transformed into a CSP, one could increase a variable's dependency value for every other variable it relates to over a clause.

Implementation The intuitive data structure for dependency values is an adjacency matrix [18], which can hold any kind of graph. During the parsing process or construction of a problem the matrix can be built up alongside. The result is a matrix holding the dependencies between the variables of the CSP. Now the propagation power of a certain variable can simply be obtained by adding all its single dependencies up. With the propagation power determined for every variable an ordering over all variables can be established.

Fortunately the choice of an adjacency matrix as the data structure allows the use of plenty of available graph and matrix analysis algorithms. As a

prominent representative the Floyd-Warshall algorithm, which calculates shortest paths between graph nodes, was implemented and tested as an SVO expansion. The algorithm was not especially picked, the choice of the algorithm was made by chance to monitor its influence. Static variable ordering combined with the Floyd-Warshall algorithm will abbreviated SVO+F. Let us summarise the whole process:

1. Interpretation of the variables' dependencies into an adjacency matrix.
2. Application of useful transformations or analyses on the matrix.
3. Summation of single dependencies to a variable's overall dependency value - its propagation power.
4. Ordering of the variables according to their propagation power.

Examples Figure 1 shows the interpretation of a small example CSP. The efficiency of the original sequence $\{v_0, v_1, v_2, v_3, v_4\}$ can't be judged, as the ordering is accidental. With just SVO the sequence $\{v_2, v_3, v_0, v_1, v_4\}$ is calculated. That is a good choice, because the assignment of v_2 or v_3 likely produces much propagation. The appliance of the Floyd-Warshall algorithm before summation seriously changes the outcome: Floyd-Warshall pushes isolated variables ahead, because they have longer paths than well connected variables. Thus the resulting sequence is $\{v_4, v_0, v_1, v_3, v_2\}$. This turnaround shows the impact of the problem analysis, which should be chosen wisely per application.

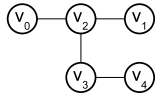
a) CSP Creation	b) Matrix interpretation	c) Extra analysis
	<pre> v0 v1 v2 v3 v4 v0 0 0 1 0 0 = 1 v1 0 0 1 0 0 = 1 v2 1 1 0 1 0 = 3 v3 0 0 1 0 1 = 2 v4 0 0 0 1 0 = 1 </pre>	<pre> v0 v1 v2 v3 v4 v0 2 2 1 2 3 = 10 v1 2 2 1 2 3 = 10 v2 1 1 2 1 2 = 7 v3 2 2 1 2 1 = 8 v4 3 3 2 1 2 = 11 </pre>

Fig. 1. a) Creation of the constraint net, b) interpretation into an adjacency matrix and c) appliance of extra analysis algorithms and summation.

Figure 2 shows the calculated propagation power values for an AIM-instance with 50 variables. The problem's synthetic origin reflects in the display of only few dependency levels. A regular problem structure provokes a regular dependency matrix. That is a problem, because an ordering will only be possible if a distinction between the variables' dependencies can be made.

If the Floyd-Warshall algorithm is applied on the dependency matrix before summation, the propagation power values in figure 3 will result. This algorithm works more deeply than SVO alone, as it operates on paths of dependencies. The dependency values in the figure are high, because direct dependencies between neighbouring variables get added by the algorithm. Each variable's dependency

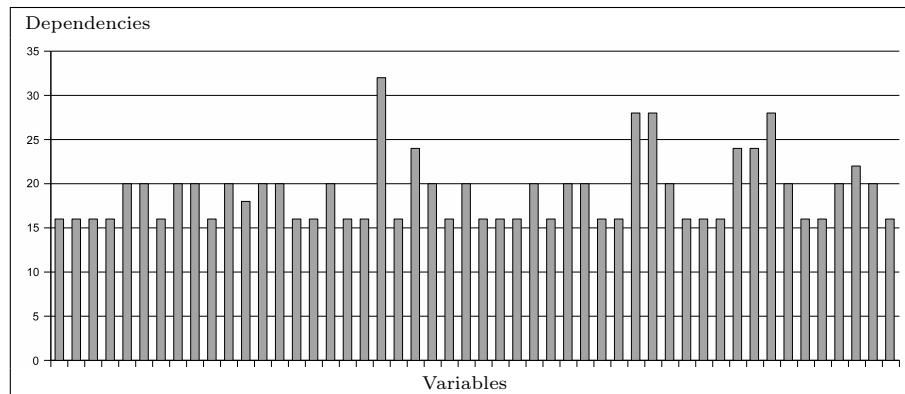


Fig. 2. SVO acquires the dependencies of the AIM-instance aim-50-1_6-no-1.cnf with 50 variables as shown.

value is the sum of direct dependencies on the shortest path to each other variable. A greater distinction is reached, which makes a meaningful ordering possible.

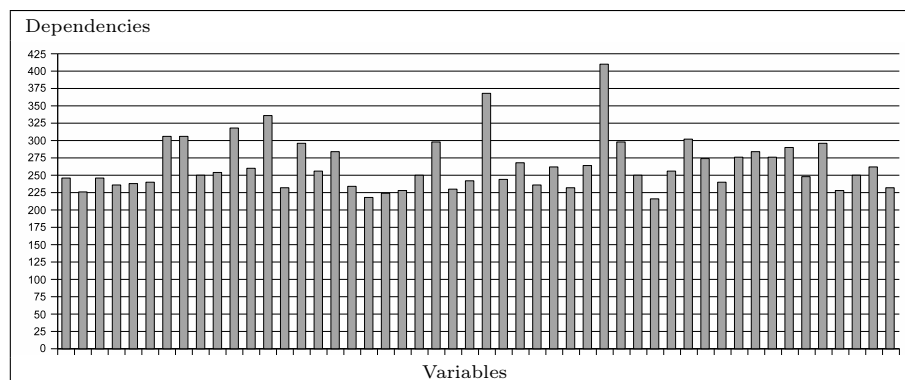


Fig. 3. Appliance of the Floyd-Warshall algorithm breaks the synthetic character of the adjacency matrix.

3.2 Dynamic variable ordering

Dynamic variable ordering (DVO) follows a completely different approach than SVO. The technique enhances the jumping-phase of a backjumping algorithm. If a backjump occurs, DVO will not just reassign and go on but reorder future variables to check the conflict again immediately.

Let us examine the behaviour of the combination of MLC-CBJ and DVO. The reordering hooks exactly in the moment of backjumping: The assignment of the current variable v_i just conflicted with past assignments. The search algorithm detects the past variable v_h as the deepest cause¹ and thus as the backjumping target. v_i now gets sorted right after v_h . After v_h is reassigned, v_i immediately is the next variable in line and assigned in the next step. That means, a conflict is checked right after its occurrence. This saves the propagation work on the way from v_h to v_i and eventual thrashing between both variables. The bigger the jumps are the more work is saved. Of course no reordering will happen if there are no variables between v_h and v_i . The behaviour of DVO is summarised in figure 4.

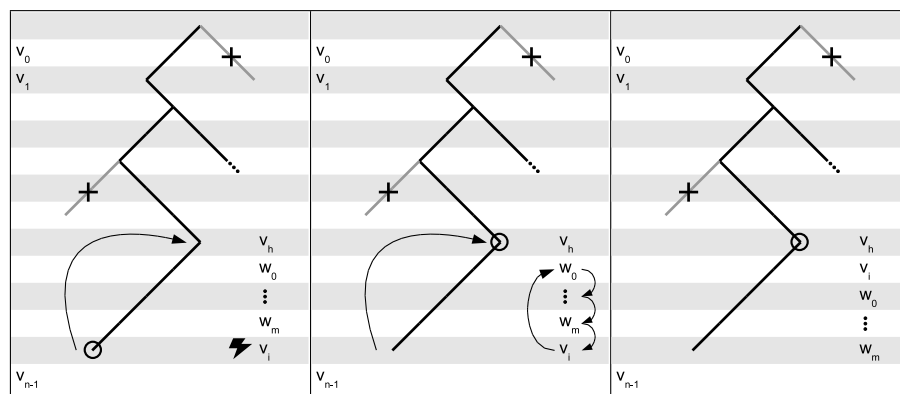


Fig. 4. a) An inconsistency causes a backjump, b) v_i gets sorted right after the cause v_h and c) the search continues with a reassignment of v_h .

The successive reordering as described has interesting effects. For most examined problems DVO causes the relevant variables to move up². As conflicting variables ascend in the search tree, bit by bit an overall "reasonable" sequence is obtained. Thus further backjumping and reordering often will become needless when two variables still stand in succession from a previous reordering. The effect of CBJ diminishes, but that is no misbehaviour but a sign of an efficient variable order. A transition from a phase with numerous and far jumps to a phase with only few and short jumps can be observed. DVO is always correct and terminates, as only unassigned future variables get reordered. There cannot emerge a situation with endless circular sorting, because the target variable of a backjump always gets immediately reassigned another value.

¹ In the context of MLC-CBJ the cause is a mix of assigned variables. Future variables are not of interest in determining a target for backjumping. The deepest variable in the search tree, which lies before the current variable, is the jump target.

² One very synthetic problem caused some terminating circular variable movement.

Implementation There is an easy and efficient way to enhance backjumping algorithms with the ability of dynamic ordering. To enable sorting, a reference data structure can be included into the original algorithm. All access on the variable holding data structure simply has now to be done indirectly over the reference structure, whose entries finally point at the variables. Thus a permutation over the variables is possible without changing the original algorithm much. E.g.: If there is a variable holding array *variable*[] and a reference array *ref*[] the redirection will simply be done with *variable*[*ref*[*i*]].

Of course the sorting can also be realised without redirection, but the author found this approach to give some extra flexibility. Furthermore the reference array can be reused for the implementation of static ordering, which is quite convenient from an object oriented view. For example, the reference array can simply be passed to a constructor of a search class, which does not have to know about static ordering to benefit from it.

The actual sorting process consists of reassignment of references, which is fast and lightweight in Java and should be in most other programming languages. The realisation of the dynamic ordering is also quite easy. After the jump from v_i to v_h and the backtracking step, the sequence $\{\dots, v_h, w_1, w_2, \dots, v_i, \dots\}$ gets arranged to $\{\dots, v_h, v_i, w_1, w_2, \dots\}$ by sorting v_i right after v_h and shifting the variables in between one to the right. The tested implementation uses Java int-arrays and does simple reference shifting. The data structure has not been optimized up to now, because the author did not expect a big gain: The shifting itself consumes only a minor part of the search algorithm's computing time.

Examples Figure 6 shows the behaviour of MLC-CBJ & DVO on an exemplary unsolvable CSP. The problem is depicted in figure 5, it consists of five variables A to E , which are all different and all have the domain $\{0..4\}$. The five variables are associated pairwise with a NotEqual-constraint. Additionally there are two rings of variables $B_{1..n}$ and $C_{1..n}$, which are connected with the CSP but have no influence on the main problem. Step by step the cause-oriented jumping creates a good sequence in respect of the main problem. Thereby work is saved as the variables $B_{1..n}$ and $C_{1..n}$ get deferred. As soon as the variables A to D stand in a row, CBJ degenerates to BT until the tree is traversed.

4 Experimental evaluation

The experimental environment and the collected figures are relatively complex and voluminous – several computers worked several weeks to collect the material. Detailed information on the used methods and the acquired results can be found in [13].

The hardest of the tested AIM-instances were unsolvable for MLC-BT, only MLC-CBJ managed to solve all instances. The hardest set of instances, which also could be solved by MLC-BT, was the group *100-2_0-no*. It consist of four unsolvable problems with 100 variables and 200 clauses (clause-variable-quotient of 2,0). Table 1 shows the needed backsteps and runtime concerning this problem

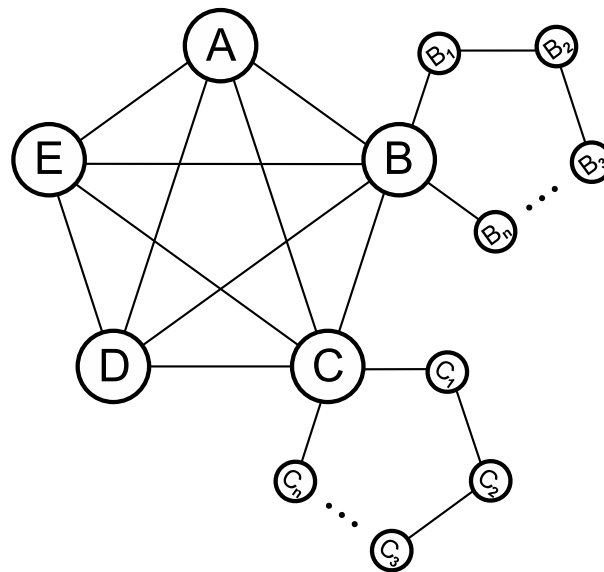


Fig. 5. The NotEqual-example: A to E have to be different, the rings are independent subproblems.

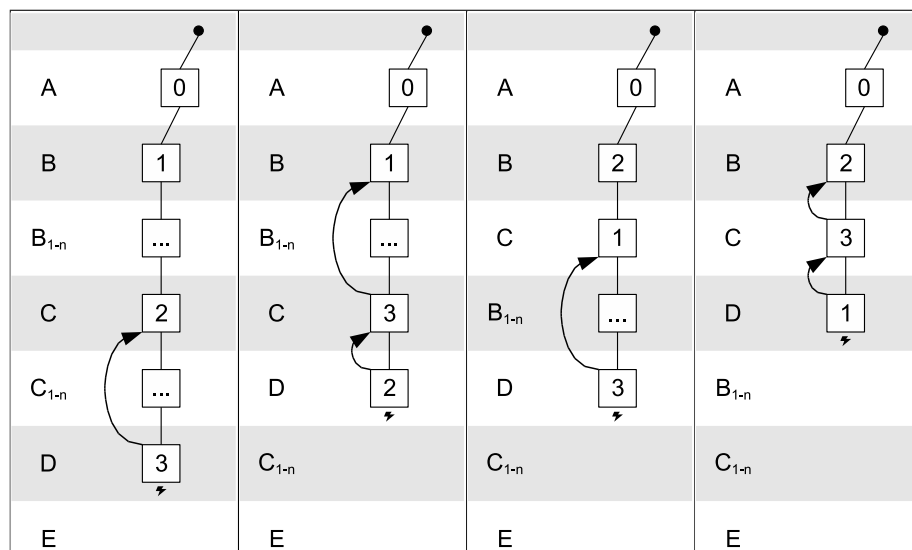


Fig. 6. MLC-CBJ & DVO processing the NotEqual-example up to the point where the "reasonable" sequence is achieved.

group. SVO is only shown in combination with the Floyd-Warshall algorithm, because it performed better for AIM-instances than just SVO. Static ordering has a positive effect on both MLC-BT and MLC-CBJ, that could be expected. DVO is very effective and improves the performance of MLC-CBJ by several magnitudes, and the combination of SVO and DVO is even a bit more effective. Unfortunately not all runtimes were measured, but at least one can see that the needed backsteps and the runtime is roughly proportional for the best variation MLC-CBJ with SVO+F and DVO.

It is interesting to note that the combination of SVO and DVO produces the best results. For most hard problems both complemented each other to a more efficient processing. As a rule of thumb can be stated: If the current SVO heuristic improves efficiency, then the combination of SVO and DVO will most often perform better than DVO or SVO alone.

	MLC-BT	MLC-BT & SVO+F	MLC-CBJ	MLC-CBJ & SVO+F	MLC-CBJ & DVO	MLC-CBJ & SVO+F & DVO
Backsteps	$1,8 \cdot 10^{11}$	$3,19 \cdot 10^7$	$3,2 \cdot 10^7$	1275	3950	1043
[%]	100	$1,77 \cdot 10^{-2}$	$1,78 \cdot 10^{-2}$	$7,08 \cdot 10^{-7}$	$2,19 \cdot 10^{-6}$	$5,8 \cdot 10^{-7}$
Runtime	402,42 h	-	-	-	-	27,61 ms
[%]	100	-	-	-	-	$1,91 \cdot 10^{-6}$

Table 1. Comparison of the needed backsteps and runtime for the hard to solve group of AIM-instances *100-2_0-no*. MLC-BT is the base for the percent-values.

Table 2 shows a performance comparison between MLC-CBJ and the top ten SAT solvers of the SAT-Ex platform [17]. DVO does again a strong performance: Only combinations with DVO were able to solve all seven problems, which are very different from each other. That recommends dynamic ordering as a flexible technique. Another thing to notice is the huge performance difference of SVO and SVO+F compared to the tests with AIM-instances. The Floyd-Warshall algorithm improved performance greatly for AIM-instances compared to normal static ordering, but here SVO and SVO+F change places. Altogether the strongest combination of MLC-CBJ & SVO & DVO brings the performance in range of the top ten SAT-solvers. Based on the SAT-solvers' average performance on the problems, *firstcs* visits only a few more nodes. The runtime is five times slower, but every result in the same magnitude is great. After all no SAT-specific optimisations were done.

5 Related and future Work

There are numerous techniques and heuristics which try to improve search by manipulating the variable order, and it would be quite pointless trying to list them here. In the constraint world especially those heuristics enjoy great popularity which minimise the search tree dynamically [5,16]. Static techniques are

Solver	all values		no outliers	
	time	nodes	time	nodes
top ten avg	588,75 100%	2023262,50 100%	193,64 100%	502853,90 100%
median	214,16 36%	239624 12%	214,16 111%	526170 105%
firstcs MLC-CBJ	-	-	1669,61 862%	4488697 893%
firstcs MLC-CBJ & DVO	10086,35 1713%	3495504 173%	-	-
firstcs MLC-CBJ & SVO	-	-	375,87 194%	4171369 829%
firstcs MLC-CBJ & SVO & DVO	2997,25 509%	2360267 117%	-	-
firstcs MLC-CBJ & SVO+F	-	-	18569,73 9590%	65469248 13019%
firstcs MLC-CBJ & SVO+F & DVO	4600,27 781%	6612419 327%	-	-

time Needed runtime in seconds
nodes Number of visited nodes
methods The percent value relates to *top ten avg*. If all problems can be solved, the result will stand in the column "all values", otherwise in the column "no outliers".
configuration Runtime is normalised to the SAT-Ex norm.

Table 2. MLC-CBJ versus top ten SAT-solver

probably rather neglected, because dynamic methods seem to outperform static ones generally [16]. It is a common effort to analyse and use constrainedness information [15] to reduce the search space as much as possible. Especially the *first fail* [8] heuristic, which chooses for the next assignment always the variable with the smallest domain, and its variations are constraint programmers' favourite. Some heuristics even try to sort the values of a single variable [6,7].

Although dependency analysis for the static variable ordering (SVO) seems to be obvious, the author could not find anything similar in recent publications. Nevertheless the technique works good and opens the door for other graph analysing algorithms. Static ordering should not be disregarded, only because dynamic ordering is a better performer. The best results were always achieved with a combination of static and dynamic ordering. Just as one often propagates a CSP before search, one should also have a look on the variable order. The right static sorting strategy can make a difference.

The approach of dynamic variable ordering (DVO) seems to be new. In [4] the existence of a "*perfect*" *dynamic variable ordering* is shown, which makes CBJ redundant by choosing the right variable in every search step. The behaviour of DVO strongly accords with this fact, as both the generation of a certain variable sequence and the diminishing of backjumps can be observed. The "*perfect*" *dynamic variable ordering* cannot be guessed, but occurring conflicts can immediately be used to repair the sequence as DVO does. The technique has proven to be very reliable and effective. The term "dynamic variable ordering"

is a little stressed in the constraint community: As DVO tries to spare work by pulling a future variable up, it could be called "forward jumping" as well.

At the moment the author is working on an enhanced variant of DVO, which moves not only the last conflicting variable v_i but also all variables which previously conflicted with and initiated a jump to v_i . E.g.: If variables $\{x_o, x_1, \dots, x_n\}$ conflicted with v_h previously, and v_i just conflicted with v_h , then the sequence $\{v_h, v_i, x_o, x_1, \dots, x_n\}$ will be established. The hope is to create a "good" sequence in one step instead of with a series of jumps and reorderings. As this behaviour demands maintenance of an additional data structure for variable references, the approach is more costly.

Another interesting task would be to check the applicability of the sorting techniques on different problem domains. Up to now the results were mainly positive. SAT problems were chosen as the test domain for reasons which have nothing to do with the sorting methods itself.

In the long run the various approaches of static and dynamic will be generalised, revised in respect of object orientation and integrated into `firstcs` as optional modules. This fits in the author's effort of the creation of a more intelligent constraint solving system [13].

6 Conclusion

This paper presented two major strategies which increase the efficiency of backtracking search algorithms drastically, at least for the tested domain. The strategy of static variable ordering (SVO) analyses the variables' dependencies a priori to create an initial sequence which is advantageous for any search algorithm. The strategy of dynamic variable ordering (DVO) enhances backjumping algorithms. It manipulates the variable ordering during search and improves efficiency by immediately checking conflicts again after backjumping.

A meaningful problem analysis and the combination of static and dynamic variable sorting procedures are an effective approach and improve the CSP solution process often by magnitudes regarding the number of visited nodes. The author sees many points for evolution of the developed methods and will pursue them in the future.

References

1. Satisfiability Suggested Format. The document `satformat.ps` describes the cnf-format, <http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/satformat.ps>.
2. Y. Asahiro, K. Iwama, and E. Miyano. Random Generation of Test Instances with Controlled Attributes. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:377–394, 1996. <http://dimacs.rutgers.edu/Volumes/Vol126.html>.
3. Roman Barták. Online Guide To Constraint Programming, 1998. First Edition, <http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html>.

4. X. Chen and P. Beek. Conflict-directed backjumping revisited, 2001. <http://citeseer.ist.psu.edu/chen01conflictirected.html>.
5. Rina Dechter and Daniel Frost. Backjump-based Backtracking for Constraint Satisfaction Problems. 2001. <http://www.ics.uci.edu/~dechter/publications/r56.html>.
6. Daniel Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems*. PhD thesis, University of California, 1997. <http://www.ics.uci.edu/~dechter/publications/r69.html>.
7. Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, Montreal, Canada, 1995. citeseer.ist.psu.edu/frost95lookahead.html.
8. R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. In *Artificial Intelligence 14*, pages 263–313, 1980.
9. Matthias Hoche. Analyse und Entwicklung unterschiedlicher Scheduling-Verfahren zur Laufzeit-Optimierung der Propagation eines Java-basierten Constraint-Lösers. Diplomarbeit, Technische Universität Berlin, 2004.
10. Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs - A Pure Java Constraint Programming Engine. Juli 2003. submitted to the 2nd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'03) at the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003. <http://uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf>.
11. A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.
12. Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.
13. Henry Müller. Analyse und Entwicklung von intelligenten abhängigkeitsgesteuerten Suchverfahren für einen Java-basierten Constraintlöser. Diplomarbeit, Technische Universität Berlin, 2004.
14. Patrick Prosser. Hybrid Algorithms For The Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268, 1993.
15. Patrick Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report/95/177, Dept. of Computer Science, University of Strathclyde, 1995. <http://citeseer.nj.nec.com/prosser95maccbj.html>.
16. Patrick Prosser. The dynamics of dynamic variable ordering heuristics. *Lecture Notes in Computer Science*, 1520:17–??, 1998. <http://citeseer.ist.psu.edu/prosser98dynamics.html>.
17. Laurent Simon and Philippe Chatalic. SAT-Ex: a web-based framework for SAT experimentation. <http://citeseer.nj.nec.com/simon01satex.html>, The web-portal SatEX <http://www.lri.fr/~simon/satex/>, reference-computer und scaling-benchmark dfmax <http://www.lri.fr/~simon/satex/aim/satex-aim.php3#machine>.
18. Stephen Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, 1962. <http://doi.acm.org/10.1145/321105.321107>.

The CHR-based Implementation of a System for Generation and Confirmation of Hypotheses

Marco Alberti¹, Federico Chesani², Marco Gavanelli¹, and
Evelina Lamma¹

¹ ENDIF - Università di Ferrara - Via Saragat, 1 - 44100 Ferrara, Italy.
{malberti|mgavanelli|elamma}@ing.unife.it

² DEIS - Università di Bologna - Viale Risorgimento, 2 - 40136 Bologna, Italy.
{fchesani}@deis.unibo.it

Abstract. Hypothetical reasoning makes it possible to reason with incomplete information in a wide range of knowledge-based applications. It is usually necessary to constrain the generation of hypotheses, so to avoid inconsistent sets or to infer new hypotheses from already made ones. These requirements are met by several abductive frameworks. In order to tackle many practical cases, however, it would also be desirable to support the dynamical acquisition of new facts, which can confirm the hypotheses, or possibly disconfirm them, leading to the generation of alternative sets of hypotheses.

In this paper, we present a system which supports the generation of hypotheses, as well as their confirmation or disconfirmation. We also describe the implementation of an abductive proof procedure, used as a reasoning engine for the generation and (dis)confirmation of hypotheses.

1 Introduction

Human reasoning often has to face incomplete information. In such cases, making hypotheses (i.e., assuming the truth of an assertion that cannot be proved) allows the reasoning process to reach conclusions that would be impossible otherwise. Sometimes, more than one hypothesis can be made; also, the truth of a hypothesis may imply the truth of another; and some sets of hypotheses cannot be made, because they are contradictory. While a person is reasoning, he/she may become aware of new facts which confirm the hypotheses, or possibly disconfirm them: in this case, the person will revise the disconfirmed hypothesis, together with its consequences.

Abduction is a reasoning paradigm that formalizes this kind of human reasoning. In particular, in the field of Abductive Logic Programming [1], many frameworks have been proposed which support many aspects of hypothetical reasoning: in many of them, it is possible to specify the class of predicates that can be hypothesized (*abducibles*), and the relation that have to hold among abducibles so that they can remain consistent (*integrity constraints*).

Most abductive frameworks assume a static knowledge, i.e., than no new facts can be acquired during the reasoning process. Yet, in many cases it would be

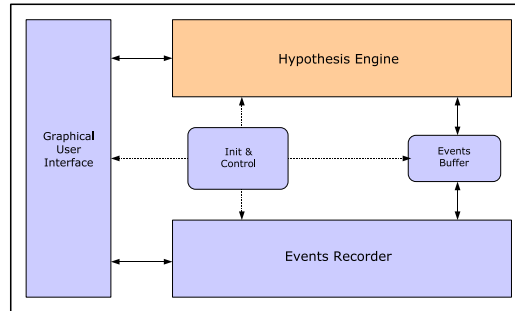


Fig. 1. A functional view of the system

useful to support the *dynamic* acquisition of new facts, so to confirm or disconfirm the generated hypotheses, and possibly lead to a revision of the generated hypotheses, just like humans do.

In this paper, we present the implementation of a system for hypothetical reasoning which, together with the features usually found in abductive systems, also supports the dynamic acquisition of facts which can confirm or disconfirm hypotheses, or cause the generation of new hypotheses.

We first describe the general architecture of the system and the abductive framework used for hypothetical reasoning in Section 2; in Section 3 we present the implementation of the abductive framework. Discussion of related work and directions for future work conclude the paper.

2 A Hypothetical Reasoning System

In this section, we give a brief overview of the architecture of the system that we have implemented (Sect. 2.1) and of the abductive framework used for hypothetical reasoning (Sect. 2.2).

2.1 Architecture

The functional architecture of the system is shown in Figure 1. The system is composed of two main components:

- An internal reasoning component (**Hypothesis Engine** in Fig. 1), which generates the hypotheses and verifies their confirmation or disconfirmation according to the information acquired during the reasoning process, revising its hypotheses if necessary;
- An external component, providing an interface to the external world (observing the events, i.e., the pieces of information acquired during the computation, **Events Recorder** in Fig. 1) and to the user (displaying the results of the computation, **Graphical User Interface** in Fig. 1). The **Init**

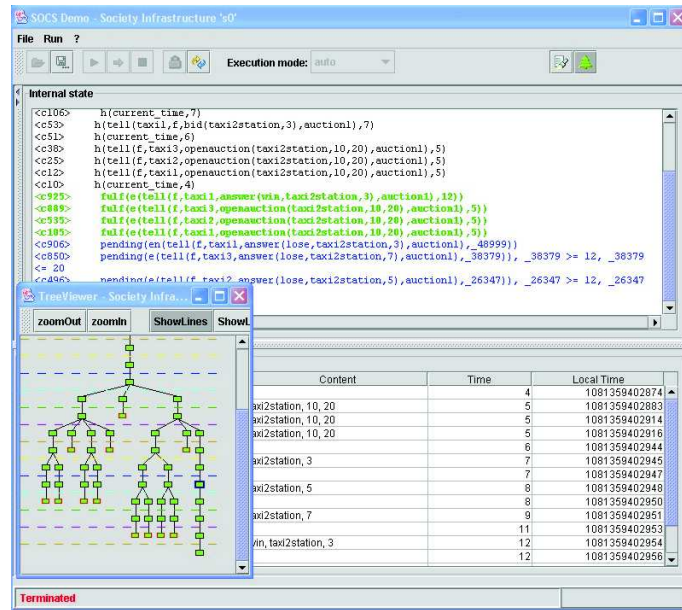


Fig. 2. A screenshot of the system

& Control and Event Buffer modules in Fig. 1 provide interfacing and synchronization between the other modules.

The two components are meant to interleave their operation: each time new events are acquired, they will be passed over to the Hypothesis Engine, which will consequently update its current set of hypotheses (by adding new ones and withdrawing disconfirmed ones) and return them for the external component to display.

This separation makes the system modular, in that the hypothesis engine used in this implementation (SCIFF, see Sects. 2.2 and 3) can be replaced by another, to support a different schema of hypothetical reasoning.

The GUI displays to the user the following information:

- The current set of hypotheses, both confirmed/disconfirmed and pending ones. It is also possible also to apply filters in order to select only subsets of hypotheses.
- The set of the happened events, considering both preloaded static events and dynamic events that have happened.
- The tree showing the state of the computation of the Hypothesis Engine.

The user can select the Hypothesis Engine to be used and the source of events. The graphical user interface also lets the user inspect any node of the computation tree.

A screenshot of the system is shown in Figure 2.

2.2 An abductive framework

In the following, we informally recall the *SCIFF* abductive logic framework with hypotheses confirmation, whose declarative and operational semantics are described in [2]. *SCIFF*, originally developed for the verification of interaction in multiagent systems [3, 4], is an extension of the IFF proof procedure by Fung and Kowalski [5]; the extensions are mainly motivated by the need to support the dynamic acquisition of facts and CLP [6] constraints over variables. This section is aimed at pointing out the features of the *SCIFF* framework that we believe more significant with respect to reasoning with hypotheses confirmation; the interested reader can refer to [2] for a formal presentation of *SCIFF*'s declarative and operational semantics, and for examples of its expressiveness.

In the *SCIFF* framework, the happened events are represented by a set (called *history*) **HAP**. This set is composed of ground atoms of the form $\mathbf{H}(\textit{Description}, \textit{Time})$, which tell what event has happened at what time. For instance, the atom $\mathbf{H}(\textit{temp}(38), 10)$ can represent the fact that, at time 10, the temperature of a patient was detected to be 38°C. This set can grow dynamically, during the computation, so implementing a dynamic acquisition of events.

Hypotheses are mapped to abducibles called *expectations*, gathered in the set **EXP**. Expectations can be *positive* ($\mathbf{E}(\textit{Description}, \textit{Time})$) representing an event that is expected to happen, or *negative* (of the form $\mathbf{EN}(\textit{Description}, \textit{Time})$), representing an event that is expected *not* to happen. Differently from events, expectations will typically contain variables; in particular, the variables can be quantified existentially or universally (the latter case can be used to express that an event is expected not to happen at any time), and CLP [6] constraints can be imposed on them (which allows, for instance, to express that an event is expected to happen in a given time interval). For example, the atom $\mathbf{E}(\textit{temp}(H), T)$, together with the CLP constraints $H < 39$, $T < 10$, can express that a patient's temperature is expected to be lower than 39°C at some time point before 10. The explicit negations of expectations are, also, abducibles, indicated by the functors $\neg\mathbf{E}$ and $\neg\mathbf{EN}$.

Confirmation and disconfirmation of expectations are mapped very simply in the *SCIFF* framework: a positive expectation that unifies with an event is confirmed, a negative expectation that unifies with an event is disconfirmed. For instance, the event $\mathbf{H}(\textit{temp}(38), 10)$ confirms the expectation $\mathbf{E}(\textit{temp}(H), T)$ with the CLP constraints $H < 39$, $T < 15$, but not the same expectation with the CLP constraints $H < 37$, $T < 15$, due to the violated constraint on H .

A positive (resp. negative) expectation may also be disconfirmed (resp. confirmed) by the fact that the CLP constraints on its variables are not satisfiable: typically, this will happen when the deadline for a hypothesis passes without a unifying event.

A *skeptical* reasoning attitude (i.e., all hypotheses that are not explicitly confirmed are rejected) is implemented by *closing* the history, i.e., it is assumed that no more events can happen, and the pending positive (resp. negative) expectations are disconfirmed (resp. confirmed). A *credulous* reasoning attitude

(i.e., a set hypotheses is acceptable even if some hypotheses are not explicitly confirmed, as long as it is consistent) is implemented by not closing the history.

Expectations may be generated either because of a set \mathcal{G} of goals (according to the knowledge expressed by a logic program KB), or because of *integrity constraints*, whose set is called \mathcal{IC}_S . Integrity constraints are forward rules which, basically, express that if in a state of the computation some events have (not) happened and some expectations have (not) been generated, then some other expectations will be generated (a simple example is $\mathbf{H}(p) \wedge \mathbf{E}(q) \rightarrow \mathbf{EN}(r)$, meaning that if p has happened and q has been hypothesized, then r should be hypothesized not to hold). The set \mathcal{IC}_S always contains the *consistency* integrity constraints, which are needed to avoid inconsistent sets of expectations: the \mathbf{E} -consistency constraint $(\mathbf{E}(p) \wedge \mathbf{EN}(p) \rightarrow \perp)$ expresses that the same event cannot be expected both to happen and not to happen, and the \neg -consistency constraints $(\mathbf{E}(p) \wedge \neg\mathbf{E}(p) \rightarrow \perp)$ and $(\mathbf{EN}(p) \wedge \neg\mathbf{EN}(p) \rightarrow \perp)$ prevent an expectation and its negation from being abduced in the same computation.

3 Implementation of the **SCIFF** proof procedure

In this section, we briefly introduce the *CHR* [7] language, which for its declarative and operational semantics appeared as a natural choice to implement the SCIFF rewriting proof procedure, and we describe the implementation.

3.1 A brief introduction to Constraint Handling Rules

In this brief introduction, we focus on the declarative and operational semantics of *CHR*; implementation-specific details (we used the SICStus [8] implementation of *CHR*) will be explained as they appear in Section 3.2.

Constraint Handling Rules [7] (*CHR* for brevity hereafter) are essentially a committed-choice language consisting of guarded rules that rewrite constraints in a store into simpler ones until they are solved. *CHR* define both *simplification* (replacing constraints by simpler constraints while preserving logical equivalence) and *propagation* (adding new, logically redundant but computationally useful, constraints) over user-defined constraints.

The main intended use for *CHR* is to write constraint solvers, or to extend existing ones. However, the computational model of *CHR* presents features that make it a useful tool for the implementation of the SCIFF proof procedure.

There are three types of *CHRs*: *simplification*, *propagation* and *simplification*.

Simplification CHRs. Simplification rules are of the form

$$H_1, \dots, H_i \iff G_1, \dots, G_j | B_1, \dots, B_k \quad (1)$$

with $i > 0$, $j \geq 0$, $k \geq 0$ and where the multi-head H_1, \dots, H_i is a nonempty sequence of *CHR* constraints, the guard G_1, \dots, G_j is a sequence of built-in constraints, and the body B_1, \dots, B_k is a sequence of built-in and *CHR* constraints.

Declaratively, a simplification rule is a logical equivalence, provided that the guard is true. Operationally, when constraints H_1, \dots, H_i in the head are in the store and the guard G_1, \dots, G_j is true, they are replaced by constraints B_1, \dots, B_k in the body.

*Propagation CHR*s. Propagation rules have the form

$$H_1, \dots, H_i \Longrightarrow G_1, \dots, G_j | B_1, \dots, B_k \quad (2)$$

where the symbols have the same meaning and constraints of those in the simplification rules (1).

Declaratively, a propagation rule is an implication, provided that the guard is true. Operationally, when the constraints in the head are in the store, and the guard is true, the constraints in the body are added to the store.

*Simpagation CHR*s. Simpagation rules have the form

$$H_1, \dots, H_l \setminus H_{l+1}, \dots, H_i \Longleftrightarrow G_1, \dots, G_j | B_1, \dots, B_k \quad (3)$$

where $l > 0$ and the other symbols have the same meaning and constraints of those of simplification *CHR*s (1).

Declaratively, the rule of Eq. (3) is equivalent to

$$H_1, \dots, H_l, H_{l+1}, \dots, H_i \Longleftrightarrow G_1, \dots, G_j | B_1, \dots, B_k, H_1, \dots, H_l \quad (4)$$

Operationally, when the constraints in the head are in the store and the guard is true, H_1, \dots, H_l remain in the store, and H_{l+1}, \dots, H_i are replaced by B_1, \dots, B_k .

3.2 SCIFF Implementation

In this section, we describe the implementation of the SCIFF proof procedure, focusing on the features described in Section 2.2.

As the IFF, the SCIFF proof procedure starts from an initial *node*; to each node *transitions* can be applied to generate new nodes, so building the proof tree (such as the one depicted in Fig. 2).

The proof procedure is implemented in SICStus Prolog [8], and makes an extensive use of its *CHR* library.

The proof tree is searched with a depth-first strategy, so to exploit the Prolog stack for backtracking. Most of the data structures are implemented as *CHR* constraints; in this way, it is quite straightforward to express the proof transitions by means of *CHR* rules.

An *ad-hoc CHR* constraint (`reif_unify/3`) implements reified unification between variables: `reif_unify(A,B,V)` means that A and B unify if and only if V=1.

Data Structures Each node of the proof tree is represented by a tuple with the following structure:

$$T \equiv \langle R, CS, PSIC, \mathbf{EXP}, \mathbf{HAP}, \mathbf{CONF}, \mathbf{DISC} \rangle$$

The data structures are implemented by means of Prolog built-in structures and the *CHR* constraint store.

In the following, we describe the implementation of each element of the tuple.

Resolvent R. The resolvent of the proof is represented as the Prolog resolvent. This allows us to exploit the Prolog stack for depth-first exploration of the tree of states.

Constraint Store CS. The constraint store of the proof³ is represented as the union of the CLP constraint stores. For the implementation of the proof, the CLPFD and CLPB libraries of SICStus Prolog, a *CHR*-based solver on finite and infinite domains, and an *ad-hoc* solver for reified unification have been used. However, in principle, it should be possible to integrate with the proof any constraint solver that works on top of SICStus Prolog.

Partially Solved Integrity Constraints PSIC. Each partially solved integrity constraint is represented by means of a `psic/2` *CHR* constraint, which has two arguments, representing the *body* (condition) and the *head* (conclusion) of the integrity constraint. For example, `psic([h(p),e(q)],[[en(r)]])` would represent⁴ the partially solved integrity constraint $\mathbf{H}(p) \wedge \mathbf{E}(q) \rightarrow \mathbf{EN}(r)$, explained in Sect. 2.2.

History HAP. Each event is represented by means of a `h/2` *CHR* constraint, whose (ground) arguments are the content and the time of the event. An example of event is:

`h(temperature(38),10)`

Expectations EXP. Expectations that are neither confirmed nor disconfirmed are represented by means of a `pending/1` *CHR* constraint, whose content is a term (with functor `e` for \mathbf{E} expectations and `en` for \mathbf{EN} expectations) representing the pending expectations. The `pending/1` constraint, obviously, does not apply to $\neg\mathbf{E}$ or $\neg\mathbf{EN}$. An example of pending expectation is:

`pending(e(temperature(A),T))`

³ This constraint store, which contains CLP constraints over variables, should not be confused with the *CHR* constraint store, which is used for the implementation of the other data structures.

⁴ Although the body of a partially solved integrity constraint is internally represented as a list of lists for efficiency, here and in the remainder of the paper, for better readability, we represent it as a flat list.

The reader should note that the representation of CLP constraints on variable T , such as $T\#<39$, are represented in the CLP constraint store, rather than in the expectation itself.

Additionally, *CHR* constraints are used to represent all expectations, either pending, confirmed or disconfirmed: this is needed because transitions apply to pending, confirmed or disconfirmed expectations in the same way. These constraints are $e/2$, $en/2$, $note/2$ or $noten/2$, for E , EN , $\neg E$ or $\neg EN$ expectations, respectively. The two arguments of these *CHR* constraints are the content and the time of the expectation.

Confirmed Expectations CONF. Each confirmed expectation is represented by a $conf/1$ *CHR* constraint, whose argument is a term representing the confirmed expectation.

Disconfirmed Expectations DISC. Each disconfirmed expectation is represented by a $disc/1$ *CHR* constraint, whose argument is a term representing the disconfirmed expectation.

Transitions The implementation of transitions has been designed so to exploit the built-in Prolog mechanisms whenever possible, both for simplicity and for efficiency. This has been made possible by the choice of a depth-first strategy for the exploration of the proof tree. The *CHR* representation of most data structures allows to map the transitions to *CHR* rules.

For lack of space, in the following we only present the implementation of the transitions that we find more significant in the context of this paper.

Propagation. Propagation is applied when two *CHR* constraints A and P that respect the following conditions are in the *CHR* store:

1. A represents an event (h), a hypothesis (e , en) or the negation of a hypothesis ($note$, $noten$): for example, $A = e(q)$;
2. P represents a partially solved integrity constraint ($psic/2$) with, in its body, an atom B that can unify with A : for example, $P = psic([h(p), e(q)], [[en(r)]])$ and $B = e(q)$.

In this case, two new nodes are generated:

- one where unification between A and B is imposed (in the example above, $reify_unify(e(q), e(q), 1)$ is imposed and immediately resolved to *true*, and removed from the constraint store) and a new constraint P' , representing the same as P but with B removed from its body (in our example, $psic([h(p)], [[en(r)]])$), is added to the *CHR* store;
- one where disunification between A and B is imposed: in the example, $reify_unify(e(q), e(q), 0)$ is imposed and immediately resolved to *false*, thus making the node one of failure.

Dynamically Growing History.

1. Happening
Happening of events is achieved by imposing (i.e., calling) a `h/2 CHR` constraint, whose (ground) arguments are the content and the time of the event.
2. Closure
Closure of the history of the society is achieved by imposing a `close_history/0 CHR` constraint. The presence of this constraint in the store will be checked by other transitions such as confirmation of **EN** expectations.

Confirmation, Disconfirmation and Consistency.

1. **E** Confirmation and **EN** Disconfirmation
Confirmation of **E** and disconfirmation of **EN** can be detected while the history is still open. The following *CHR* implements disconfirmation of **EN** expectations:

```
disconfirmation @
    h(HEvent,HTime),
    pending(e(ENEvent,ENTime)) # _pending
==>
    fn_ok(HEvent,ENEvent) |
    ccopy(p(ENEvent,ENTime),p(ENEvent1,ENTime1)),
    case_analysis_disconfirmation(HEvent,HTime,ENEvent,ENTime,
                                  ENEvent1,ENTime1,_pending).
```

The rule is applied when an event and a pending **EN** expectation whose content have the same functor and arity (this is checked by the `fn_ok/2` predicate in the guard of the rule) are in the *CHR* store. In this case, a copy is made of the expectation⁵ and the `case_analysis_disconfirmation/7` predicate is called. The arguments of this predicate represent, respectively, the content of the event, the time of the event, the content of the expectation, the time of the expectation, a copy of the content of the expectation, a copy of the time of the expectation, and the internal index representing⁶ the `pending/1` constraint for the expectation. Two nodes are created by `case_analysis_disconfirmation/7`: one where unification is imposed between the expectation and the event, the `pending/1` constraint for the expectation is removed and the `disc/1 CHR` constraint for the expectation is imposed; and another one where non-unification between the expectation and the event is imposed.

⁵ This allows for the universally quantified variables in the original expectation (such as **X** in `en(p(X))`) to remain unbound after the unification: binding them to a value would restrict their quantification.

⁶ In the SICStus [8] *CHR* implementation, writing the constraint `pending(e(ENEvent,ENTime)) # _pending` in the head of a rule will bind `_pending` to the internal index that represents the constraint in the *CHR* store when the rule is activated, for future reference; in this case, the index is used to remove the constraint from the store in the confirmation branch.

2. **E** Disconfirmation and **EN** Confirmation (closed history)

When the history of the society is closed (by means of a closure transitions), all pending **E** are marked as disconfirmed and all pending **EN** are declared confirmed. This is achieved by the following two rules:

```
closure_e @
  (close_history) \ (pending(e(Event,Time)))
  <=>
  disc(e(Event,Time)).
```

An analogous rules implements closure-driven confirmation of negative expectations.

These rules implement the *skeptical* reasoning mentioned in Section 2.2. A *credulous* attitude can be achieved by simply omitting these rules.

3. **E**-Consistency

E-consistency is implemented by imposing non-unification on the (*Content, Time*) pairs of **E** and **EN** expectations in the store:

```
e_consistency @
  e(EEvent,ETime), en(ENEvent,ENTime)
  ==>
  reif_unify(p(EEvent,ETime),p(ENEvent,ENTime),0).
```

4. \neg -Consistency

Analogously to **E**-Consistency, \neg -Consistency is implemented by imposing non-unification on the (*Content, Time*) pairs of **E** and \neg **E** (or **EN** and \neg **EN**) expectations in the store.

4 Related work

The *SCIFF* abductive framework is mostly related to the IFF proof procedure [5], which it extends in several directions: dynamic update of the knowledge base by happening events, confirmation and disconfirmation of hypotheses, hypotheses with universally quantified variables, CLP constraints.

In [9], Sergot proposed a general framework, called *query-the-user*, in which some of the predicates are labelled as “askable”; the truth of askable atoms can be asked to the user. Our **E** predicates may in a sense be seen as asking information, while **H** atoms may be considered as new information provided during search. However, **E** atoms may also mean *expected* behavior, and the *SCIFF* can cope with abducibles containing universally quantified variables.

The idea of hypotheses confirmation has been studied also by Kakas and Evans [10], where hypotheses can be corroborated or refuted by matching them with observable atoms: an explanation fails to be corroborated if some of its logical consequences are not observed. The authors suggest that their framework could be extended to take into account dynamic events, possibly queried to the user.

In a sense, the *SCIFF* framework can be considered as an extension of these works: it provides the concept of confirmation of hypotheses, as in corroboration, and an operational semantics for dynamically incoming events. Moreover, we extend the work by imposing integrity constraints to better define the feasible combinations of hypotheses, and we let the program abduce non-ground atoms.

Christiansen and Dahl [11] propose to exploit the *CHR* language to extend SICStus Prolog to support abduction more efficiently than with metainterpretation-based solutions. They represent abducibles as *CHR* constraints as we do, but they represent integrity constraints directly as *CHR* propagation rules, using the built-in *CHR* matching mechanism for propagation: this does not seem possible in our framework, which also needs to handle universally quantified variables and CLP constraints. Other implementations have been given of abductive proof procedures in Constraint Handling Rules [12, 13]; as these works, our implementation exploits the uniform understanding of constraints and abducibles noted by Kowalski et al. [14]. However, the *SCIFF* framework also supports the dynamic confirmation, or disconfirmation, of hypotheses.

5 Conclusions and future work

In this paper, we have presented the implementation of a system for hypothetical reasoning which supports the confirmation and disconfirmation of hypotheses.

There are many possible extensions of this work, which we intend to pursue in the future. For instance, it would be worthwhile to let the user impose the failure of a branch of the reasoning tree, regardless of the confirmation or disconfirmation of the hypotheses made in the branch, in order to explore branches that the user finds more promising. We also intend to support a breadth-first exploration of the computation tree, as an alternative to the depth-first exploration of the current implementation. Besides, we believe that the formal framework would benefit from the introduction of a formalism to express priorities among the possible alternative hypotheses, in a given state of the computation.

Another direction of improvement could be towards better computational performance, possibly exploiting alternative efficient *CHR* implementations, such as the one proposed by Wolf [15].

Acknowledgments

This work has been supported by the European Commission within the SOCS project (IST-2001-32530), funded within the Global Computing Programme and by the MIUR COFIN 2003 projects *La Gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici* and *Sviluppo e verifica di sistemi multiagente basati sulla logica*.

We wish to thank the anonymous reviewers for their detailed and useful comments on a previous version of this paper.

References

1. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. *Journal of Logic and Computation* **2** (1993) 719–770
2. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Abduction with hypothesis confirmation. In Rossi, G., Panegai, E., eds.: *Proceedings of CILC'04 - Italian Conference on Computational Logic*, Parma, 16-17 Giugno 2004. Number 390 in *Quaderno del Dipartimento di Matematica*, Research Report, Università di Parma (2004)
3. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interactions using social integrity constraints. *Electronic Notes in Theoretical Computer Science* **85** (2003)
4. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An Abductive Interpretation for Open Societies. In Cappelli, A., Turini, F., eds.: *AI*IA 2003: Advances in Artificial Intelligence*, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa. Volume 2829 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag (2003) 287–299
5. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. *Journal of Logic Programming* **33** (1997) 151–165
6. Jaffar, J., Maher, M.: Constraint logic programming: a survey. *Journal of Logic Programming* **19-20** (1994) 503–582
7. Frühwirth, T.: Theory and practice of constraint handling rules. *Journal of Logic Programming* **37** (1998) 95–138
8. : SICStus prolog user manual, release 3.11.0 (2003) <http://www.sics.se/is1/sicstus/>.
9. Sergot, M.J.: A query-the-user facility of logic programming. In Degano, P., Sandwell, E., eds.: *Integrated Interactive Computer Systems*, North Holland (1983) 27–41
10. Evans, C., Kakas, A.: Hypotheticodeductive reasoning. In: *Proc. International Conference on Fifth Generation Computer Systems*, Tokyo (1992) 546–554
11. Christiansen, H., Dahl, V.: Assumptions and abduction in prolog. In Muñoz-Hernández, S., Gómez-Perez, J.M., Hofstedt, P., eds.: *Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'04)*, Saint-Malo, France (2004) Workshop notes.
12. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In Larsen, H., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H., eds.: *FQAS, Flexible Query Answering Systems*. LNCS, Warsaw, Poland, Springer-Verlag (2000) 141–152
13. Gavanelli, M., Lamma, E., Mello, P., Milano, M., Torroni, P.: Interpreting abduction in CLP. In Buccafurri, F., ed.: *APPIA-GULP-PRODE Joint Conference on Declarative Programming*, Reggio Calabria, Italy, Università Mediterranea di Reggio Calabria (2003) 25–35
14. Kowalski, R., Toni, F., Wetzel, G.: Executing suspended logic programs. *Fundamenta Informaticae* **34** (1998) 203–224
15. Wolf, A.: Adaptive constraint handling with chr in java. In Walsh, T., ed.: *Principles and Practice of Constraint Programming - CP 2001*, 7th International Conference. Volume 2239 of *Lecture Notes in Computer Science*, Paphos, Cyprus, Springer Verlag (2001) 256–270

Guard Simplification in CHR programs

Jon Sneyers, Tom Schrijvers*, Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
{jon,toms,bmd}@cs.kuleuven.ac.be

Abstract. Constraint Handling Rules (CHR) is a high-level language commonly used to write constraint solvers. Most CHR programs depend on the refined operational semantics, obfuscating their logical reading and causing different (termination) behavior under the theoretical operational semantics. We introduce a source to source transformation called *guard simplification* which allows CHR programmers to write self-documented rules with a clear logical reading. Performance is improved by removing guards entailed by the implicit “no earlier (sub)rule fired” precondition and optional type and mode declarations. A formal description of the transformation is given, its implementation in the K.U.Leuven CHR compiler is presented and experimental results are discussed.

1 Introduction

Constraint Handling Rules (CHR) is a high-level multi-headed rule-based programming language extension commonly used to write constraint solvers. We will assume the reader to be familiar with the syntax and semantics of CHR, referring to [5] for an overview. Examples are given in a Prolog context, although the results are valid in general.

The theoretical operational semantics ω_t of CHRs, as defined in [5], is relatively nondeterministic as the order in which rules are tried is not specified. However, all implementations of CHR we know of use a more specific operational semantics, called the *refined* operational semantics ω_r [4]. In ω_r , the order in which rules are tried is the textual order in which the rules occur in the CHR program. Usually, CHR programmers take this refined operational semantics into account when they write CHR programs. As a result, their CHR programs could be non-terminating or could even produce incorrect results under ω_t semantics.

The dilemma CHR programmers face is the following: either they make sure their programs are valid under ω_t semantics, or they write programs that only work correctly under ω_r semantics. Sticking to ω_t semantics has the advantage that it results in more declarative code with

* Research Assistant of the fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen)

a clear logical reading, but it has the disadvantages that it is harder to implement some programming idioms and that the compiled code is less efficient. Using ω_r semantics results in more efficient compiled code and allows easier implementation of some programming idioms like key lookup, but at a cost: it becomes much less obvious from the CHR program what the preconditions for application of a rule really are. Indeed, under ω_t semantics, rules have to contain in their guards all the preconditions needed, while under ω_r semantics, the CHR programmer can and does omit the preconditions that are implicitly entailed by the rule order. Omitting these redundant preconditions may contribute to more efficient compiled code, but at the same time it makes the program less self-documented.

In this paper, we propose a compiler optimization that is a major step towards allowing CHR programmers to write more readable and declarative programs while getting the same efficiency as programs written with the specifics of the refined operational semantics in mind. This optimization, called *Guard Simplification*, is a source-to-source transformation of CHR programs, removing redundant guard conditions (and head matchings, an implicit part of the guard) based on reasoning about behavior of the program under the refined operational semantics. The transformed program is simpler, possibly allowing more optimization from other analyses. For example, guard simplification can reveal the never-stored property [2], as we will show later. Thanks to guard simplification, the CHR programmer can focus on writing a declarative specification and rely on the compiler to produce efficient code.

The next section presents a short intuitive overview of guard simplification, illustrated with some examples. In section 3, a formal definition of the guard simplification transformation is given. Section 4 briefly deals with the implementation of the guard simplification analysis in the K.U.Leuven CHR compiler [8]. Then, in section 5, the results of several benchmarks are discussed, in order to compare the efficiency of CHR programs before and after guard simplification. Finally, section 6 concludes this paper, summarizing our contributions.

2 Overview

The source to source transformation discussed in this paper transforms a CHR program P into another CHR program $P' = GS(P)$ which is equivalent under the refined operational semantics ω_r . Although the original program might have been valid under any execution strategy covered by

the theoretical operational semantics ω_t , the transformed program will in general only exhibit identical behavior when ω_r semantics is used. This is not an issue, since all recent CHR implementations use ω_r semantics.

Under the refined operational semantics of CHRs, the order in which the rules are tried is the textual order of the rules in the CHR program. We number the rules accordingly, so that for $i < j$, rule R_i appears before rule R_j in the CHR program.

2.1 Guard simplification

When a simpagation rule or a simplification rule fires, some or all of its head constraints are removed. As a result, for every rule R_i , we know that when this rule is tried, any non-propagation rule R_j with $j < i$, where the set of head constraints of rule R_j is a (multiset) subset of that of rule R_i , did not fire for some reason. Either the heads did not match, or the guard failed. Let us illustrate this with some simple examples.

Example 1: an entailed guard

```
pos  @ sign(P,S) <=> P > 0 | S = positive.
zero @ sign(Z,S) <=> Z := 0 | S = zero.
neg  @ sign(N,S) <=> N < 0 | S = negative.
```

If the third rule, **neg**, is tried, we know **pos** and **zero** did not fire, because if they would have fired, the **sign/2** constraint would have been removed. Because the first rule, **pos**, did not fire, its guard must have failed, so we know that $N \leq 0$. The second rule, **zero**, did not fire either, so we derive that $N \neq 0$. Now we can combine these results to get $N < 0$, which is exactly the guard of the third rule. Because we know this guard will always be true, we can safely remove it. This will result in slightly more efficient generated code (because the redundant test is removed), but – more importantly – this might also be useful for other analyses. In this example, after the guard simplification, the *never-stored* analysis [2] is able to detect that the constraint **sign/2** is never-stored because now the third rule is an unguarded single-head simplification rule, removing all **sign/2** constraints immediately.

Example 2: a rule that can never fire

```
neq  @ p(A) \ q(B) <=> A \== B | ...
eq   @ q(C) \ p(C) <=> true | ...
prop @ p(X), q(Y) ==> ...
```

In this case, we can detect that the third rule, **prop**, will never fire. Indeed, because the first rule, **neq**, did not fire, we know that **X** and **Y** are equal and because the second rule, **eq**, did not fire, we know **X** and **Y** are not equal. This is of course a contradiction, so we know the third rule can never fire. Most often such never firing rules are in fact bugs in the CHR program – there is no reason to write rules that cannot fire – so it seems appropriate for the CHR compiler to give a warning message when it encounters such rules.

Generalizing from the previous examples, we can summarize guard simplification as follows: If a (part of a) guard is entailed by knowledge given by the negation of earlier guards, we can replace it by **true**, thus removing it. However, if the *negation* of (part of a) guard is entailed by that knowledge, we know the rule will never fire and we can remove the entire rule.

2.2 Head matching simplification

Matchings in the arguments of head constraints can be seen as an implicit guard condition that can also be simplified. Consider the following example:

```
p(X,Y) <=> X \== Y | ...
p(X,X) <=> ...
```

Never-stored analysis as it is currently implemented in the K.U.Leuven CHR system is not able to detect **p/2** to be a never-stored constraint, because none of these two rules remove all **p/2** constraints. We can rewrite the second rule to **p(X,Y) <=> ...**, because the (implicit) condition **X == Y** is entailed by the negation of the guard of the first rule. In the refined operational semantics, this does not change the behavior of the program. Now we say the head matchings of the second rule are simplified, because the head contains less matching conditions. As a result, never-stored analysis can now detect **p/2** to be never-stored, and more efficient code can be generated.

2.3 Type and mode declarations

Head matching simplification can be much more effective if some knowledge of the argument types of constraints is given. Consider this example:

```
sum([],S) <=> S = 0.
sum([X|Xs],S) <=> sum(Xs,S2), S is X + S2.
```

If we know the first argument of constraint `sum/2` is a (ground) list, these two rules cover all possible cases and thus the constraint is never-stored. In [11], optional mode declarations were introduced to specify the mode – ground (+) or unknown (?) – of constraint arguments. Inspired by the Mercury type system [13], we have added optional type declarations to define types and specify the type of constraint arguments. For the above example, the CHR programmer would add the following lines:

```
option(type_definition,
      type(list(X), [ [], [X | list(X)] ])).
option(type_declaration, sum(list(int),int)).
option(mode, sum(+,?)).
```

The first line is a recursive and generic type definition for lists of some type `X`, a variable that can be instantiated with builtin types like `int`, `float`, the general type `any`, or any user-defined type. The next line says the first argument of constraint `sum/2` is of type ‘list of integers’ and the second is an integer. In the last line, the first argument of `sum/2` is declared to be ground on call while the second argument can be a variable. Using this knowledge, we can rewrite the second rule of the example program to “`sum(A,S) <=> A = [X|Xs], sum(Xs,S2), S is X + S2.`”, keeping its behavior intact while again helping never-stored analysis to detect `sum/2` to be a never-stored constraint.

3 Formal description

We will now formalize the guard simplification transformation intuitively described above. Constraints are either CHR constraints or *builtin* constraints in some constraint domain \mathcal{D} . The former are manipulated by the CHR execution mechanism while the latter are handled by an underlying constraint solver. We will consider all three types of CHR rules to be special cases of simpagation rules:

Definition 1 (CHR program). A CHR program P is a sequence of CHR rules R_i of the form

$$R_i = H_i^k \setminus H_i^r \iff g_i \mid B_i$$

where H_i^k (kept head constraints) and H_i^r (removed head constraints) are sequences of CHR constraints (not both empty), g_i (guard) is a conjunction of builtin constraints, and B_i (body) is a conjunction of constraints.

We assume all arguments of the CHR constraints in the head to be unique variables, making any head matchings explicit in the guard. This head normalization procedure is explained in more detail in [3] and an illustrating example can be found e.g. in section 2.1 of [10].

We will consider rules that must have been tried according to the refined operational semantics before trying some rule R_i , calling them *earlier subrules* of R_i .

Definition 2 (Earlier subrule). *The rule R_j is an earlier subrule of rule R_i (notation: $R_j \prec R_i$) iff $j < i$ and the (renamed) constraints occurring in the head of R_j form a (multiset) subset of the head constraints of R_i .*

Now we can define a logical expression $nesr(R_i)$ stating the implications of the fact that all constraint-removing earlier subrules of rule R_i have been tried unsuccessfully.

Definition 3 (“No earlier subrule fired”). *For every rule R_i , we define:*

$$nesr(R_i) = \bigwedge \{ (\neg(\theta_j \wedge g_j)) \mid R_j \prec R_i \text{ and } H_j^r \text{ is not empty} \}$$

where θ_j is a matching substitution mapping the head constraints of R_j to corresponding head constraints of R_i .

Consider a CHR program P with rules R_i which have guards $g_i = \bigwedge_k g_{i,k}$. If we apply guard simplification to this program, we rewrite some guards to **true** (or **false**) if they (or their negations) are entailed by the “no earlier subrule fired” condition.

Definition 4 (Guard simplification). *Applying guard simplification to a CHR program P results in a new CHR program $P' = GS(P)$ with rules $R'_i = H_i^k \setminus H_i^r \iff \bigwedge_k g'_{i,k} \mid B_i$, where*

$$g'_{i,k} = \begin{cases} \mathbf{true} & \text{if } \mathcal{D} \models nesr(R_i) \rightarrow g_{i,k}; \\ \mathbf{false} & \text{if } \mathcal{D} \models nesr(R_i) \rightarrow \neg g_{i,k}; \\ g_{i,k} & \text{otherwise.} \end{cases}$$

Because of space limitations, we will simply formulate our correctness result without a proof. A detailed, but rather straightforward proof of the following theorem can be found in [12].

Theorem 1 (Guard simplification and applicability of transitions).

Given a CHR program P and its guard-simplified version $P' = GS(P)$. Given an execution state $S_i = \langle A, S, B, T \rangle_n$ occurring in some derivation for the P program under ω_r semantics, exactly the same transitions are possible from S_i for P and for P' .

Corollary 1. *Under the refined operational semantics, any CHR program P and its guard-simplified version P' are operationally equivalent.*

4 Implementation

We have implemented guard simplification as a new compilation phase in the K.U.Leuven CHR compiler [8]. Essentially, the guard simplification phase does the following: For every rule R , head matchings are made explicit, $nesr(R)$ is constructed (the conjunction of the the negations of the guards of the earlier subrules $R_i \prec R$) and type information is added to this. Then any part of the guard entailed by this big conjunction is replaced by `true` – if its negation is entailed, it is replaced by `fail`. Finally, entailed head matchings are moved to the body if possible.

A separate entailment checking module has been written to test whether some condition B (e.g. $X < Z$) is entailed by another condition A (e.g. $X < Y \wedge Y < Z$), i.e. $A \rightarrow B$. Since in general this problem is undecidable, the entailment checker will try to prove that B is entailed by A by propagating the implications of (host language) builtin conditions in A , like `<`, `==`, `functor/3`, `==` and unification, succeeding if B is found and failing otherwise. Hence if the entailment checker succeeds, $A \rightarrow B$ must hold, but if it fails, either $A \not\rightarrow B$ holds or $A \rightarrow B$ holds but was not detected. It does not try to discover implications of user-defined predicates, which would require a complex analysis of the host-language program. The core of this entailment checker is written in CHR. A detailed description of our implementation of both guard simplification itself and the entailment checker can be found in [12].

In order to compare the generated code both with and without guard simplification, we present the Prolog code the CHR compiler generates for some example CHR program. In this fragment from a prime number generating program (taken from the CHR web site [14]):

```
filter([X|In],P,Out) <=> 0 =\= X mod P |
                                Out=[X|Out1], filter(In,P,Out1).
filter([X|In],P,Out) <=> 0 == X mod P | filter(In,P,Out).
filter([],P,Out) <=> Out=[].
```

the CHR compiler (without guard simplification) generates the typical general code (as in [7]) for the `filter/3` constraint:

```
filter(List,P,Out) :- filter(List,P,Out, _ ) .

% first occurrence
filter(List,P,Out,C) :-
    nonvar(List), List = [X|In], 0 =\= X mod P, !,
    ... % remove from constraint store if needed
    Out = [E|Out1], filter(In,P,Out1) .

% second occurrence
filter(List,P,Out,C) :-
    nonvar(List), List = [X|In], 0 := X mod P, !,
    ... % remove from constraint store if needed
    filter(In,P,Out) .

% third occurrence
filter(List, _ ,Out,C) :-
    List == [], !,
    ... % remove from constraint store if needed
    Out = [] .

% insert into store in case none of the rules matched
filter(List,P,Out,C) :-
    ... % insert into constraint store
```

If we enable the guard simplification phase, the guard in the second rule is removed, but this alone does not considerably improve efficiency. However, we can add type and mode information and then use the guard simplification analysis to transform the program to an equivalent and more efficient form. In this example, the programmer intends to use the `filter/3` constraint with the first two arguments ground, while the third one can have any instantiation. The first and the third argument are lists of integers, while the second argument is an integer. So we add the following type and mode declarations to the CHR program:

```
option(type_declaration, filter(list(int),int,list(int))).
option(mode, filter(+,+,?)).
```

Using this type and mode information, guard simplification now detects that all possibilities are covered by the three rules. The guard in

the second rule can be removed, so the `filter/3` constraint with the first argument being a non-empty list is always removed after the second rule. Thus in order to reach the third rule, the first argument has to be the empty list – it cannot be a variable because it is ground and it cannot be anything else because of its type. As a result, we can drop the head matching in the third rule:

```
filter([X|In],P,Out) <=> 0 =\= X mod P |
                        Out=[X|Out1], filter(In,P,Out1).
filter([],P,Out) <=> filter(In,P,Out).
filter(_,P,Out) <=> Out=[].
```

This transformed program is compiled to more efficient Prolog-code, because never-stored analysis can detect `filter/3` to be never-stored after the third rule. Also no variable triggering needs to be considered since the relevant arguments are known to be ground:

```
filter([X|In],P,Out) :- 0 =\= X mod P, !,
                        Out = [X|Out1], filter(In,P,Out1).
filter([],P,Out) :- !, filter(In,P,Out).
filter(_,_,[]).
```

5 Experimental results

In order to get an idea of the efficiency gain obtained by guard simplification, we have measured the performance of several CHR benchmarks, both with and without guard simplification. All benchmarks were performed in hProlog 2.4.5-32 [1], on a Pentium 4 (1.7 GHz) machine running Debian GNU/Linux (kernel version 2.4.25) with a low load. Figure 1 gives an overview of our results. These benchmarks are available at [9]. For every benchmark, the results for a hand-written Prolog version are included, representing the ideal target code.

Overall, for these benchmarks, the net effect of the guard simplification transformation – together with never-stored analysis and usage of mode information to remove redundant variable triggering code – is cleaner generated code which is much closer to what a Prolog programmer would write. As a result, a major performance improvement is observed in these benchmarks, which are CHR programs that basically implement a deterministic algorithm.

Other CHR programs, like typical constraint solvers, where variable triggering occurs and the constraints are typically not never-stored, will

<i>Benchmark</i>	<i>Language</i>	<i>Guard simpl.</i>	<i>Mode decl.</i>	<i>Type decl.</i>	<i>Clauses</i>	<i>Lines</i>	<i>Run time (ms)</i>	<i>Relative run time</i>
sum (10000,500)	CHR	yes/no	no	no	4	46	1,890	100.0%
		yes/no	yes	no	3	10	1,680	88.9%
		yes	yes	yes	2	6	1,260	66.7%
	handwritten Prolog code				2	5	1,250	66.1%
Takeuchi (1000)	CHR	no	no	yes/no	4	50	15,060	100.0%
		no	yes	yes/no	3	17	9,910	65.8%
		yes	yes/no	yes/no	2	12	9,190	61.0%
	handwritten Prolog code				2	12	9,190	61.0%
nrev (30,50000)	CHR	yes/no	no	no	8	92	4,480	100.0%
		yes/no	yes	no	6	20	2,820	62.9%
		yes	yes	yes	4	11	1,030	23.0%
	handwritten Prolog code				4	7	920	20.5%
cprimes (100000)	CHR	no	no	yes/no	14	160	10,730	100.0%
		no	yes	yes/no	11	42	6,230	58.1%
		yes	no	no	12	120	10,670	99.4%
		yes	yes	no	10	35	6,140	57.2%
		yes	yes	yes	8	25	5,990	55.8%
	handwritten Prolog code				8	23	5,990	55.8%
dfsearch (16,500)	CHR	no	no	yes/no	5	67	20,290	100.0%
		no	yes	yes/no	4	16	17,130	84.4%
		yes	no	no	5	66	18,410	90.7%
		yes	yes	no	4	15	16,120	79.4%
		yes	yes	yes	3	11	12,080	59.5%
	handwritten Prolog code				3	8	11,330	55.8%

Fig. 1. Benchmark results.

not benefit this much from guard simplification. Redundant guards will of course be removed, but in most cases this will not result in a drastic improvement in code size or performance since guards are usually relatively cheap. The main advantage of guard simplification is that relying on it, the CHR programmer is able to write programs that have a more declarative reading and that are more self-documenting. All preconditions needed for a rule to fire can be put in the guard – guard simplification will eliminate all redundant conditions so this will not affect efficiency.

The only difference between the original program and the guard-simplified transformed program is that some conditions (namely those that can be proved to be entailed) are not evaluated in the transformed program. This should only improve efficiency. Thus there are no cases in which guard simplification transforms a program to a less efficient version.

In most cases, the additional compile time spent in the guard simplification phase is very reasonable. For relatively small CHR programs like

the benchmarks discussed above, the time cost of applying guard simplification is more or less insignificant, in the order of 50 milliseconds. For larger CHR programs, the time complexity of the guard simplification compilation phase depends heavily on the number of earlier subrules for every rule. In extreme cases where this number is exceptionally large, the guard simplification phase tends to dominate the compilation time.

For an extensive discussion of the experimental results we refer the reader to [12].

6 Conclusion

We have presented a compiler analysis called guard simplification that allows CHR programmers to write more declarative CHR programs that are more self-documented. Indeed, all preconditions for rule application can now be included in the guard, without efficiency loss. Earlier work introduced mode declarations used for hash tabling and other optimizations. In addition, we have provided a way for CHR programmers to add type declarations to their programs. Using both mode and type declarations we have realized further optimization of the generated code.

In order to achieve higher efficiency, CHR programmers often write parts of their program in Prolog if they do not require the additional power of CHR. Now they no longer need to write mixed-language programs for efficiency: they can simply write the entire program in CHR, because thanks to guard simplification and other analyses like storage analysis, the K.U.Leuven CHR compiler is able to generate efficient code with the constraint store related overhead reduced to a minimum. While guard simplification in itself does not reduce this overhead (although it does remove the overhead of checking entailed guard conditions), it enables other analyses to do so.

The guard simplification analysis is somewhat similar to switch detection in Mercury [6]. Switch detection is used in determinism analysis to check unifications involving variables that are bound on entry to a disjunction and occurring in the different branches. In a sense, switch detection is a special case of guard simplification. Similarities and differences are elaborated in [12].

Possibilities for future work include: improving the scalability of our implementation, adding support for declarations of certain properties of guards and using information derived in the guard simplification phase to enhance other analyses and to do program specialization on calls in the rule body.

References

1. Bart Demoen. The hProlog home page, October 2004. <http://www.cs.kuleuven.ac.be/~bmd/hProlog>.
2. Gregory J. Duck, Tom Schrijvers, and Peter J. Stuckey. Abstract Interpretation for Constraint Handling Rules. Technical Report CW 391, K.U.Leuven, Departement of Computer Science, 2004.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, 2003.
4. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *20th International Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, September 2004.
5. T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, volume 37 (1–3), October 1998.
6. Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.
7. Christian Holzbaur and Thom Frühwirth. Compiling Constraint Handling Rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, 1998.
8. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004. ISSN 0939-5091.
9. Tom Schrijvers. CHR benchmarks and programs, October 2004. Available at <http://www.cs.kuleuven.ac.be/~toms/Research/CHR/>.
10. Tom Schrijvers and Bart Demoen. Antimonotony-based Delay Avoidance for CHR. Technical Report CW 385, K.U.Leuven, Department of Computer Science, July 2004.
11. Tom Schrijvers and Thom Frühwirth. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Department of Computer Science, July 2004.
12. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. Technical Report CW 396, K.U.Leuven, Department of Computer Science, November 2004.
13. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, February 1995.
14. Various. 40 CHR Constraint Solvers Online, December 2004. Available at <http://www.pms.informatik.uni-muenchen.de/~webchr/>.

Analysing the CHR Implementation of Union-Find

Tom Schrijvers* and Thom Frühwirth

¹ Department of Computer Science, K.U.Leuven, Belgium
www.cs.kuleuven.ac.be/~toms/

² Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. CHR (Constraint Handling Rules) is a committed-choice rule-based language that was originally intended for writing constraint solvers. Over time, CHR is used more and more as a general-purpose programming language. In companion paper [12] we show that it is possible to write the classic union-find algorithm and variants in CHR with best-known time complexity, which is believed impossible in Prolog. In this paper, using CHR analysis techniques, we study logical correctness and confluence of these programs. We observe the essential destructive update of the algorithm which makes it non-logical.

1 Introduction

When a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. For example, one often hears the argument that in Prolog some graph algorithms cannot be implemented with best known complexity because Prolog lacks destructive assignment that is needed for efficient update of the graph data structures. In particular, it is not clear if the union-find algorithm can be implemented with best-known complexity in pure (i.e. side-effect-free) Prolog [10].

We give a positive answer for the Constraint Handling Rule (CHR) programming language. There is an CHR implementation with the optimal worst case and amortized time complexity known for the classical union-find algorithm with path compression for find and union-by-rank. This is particularly remarkable, since originally, CHR was intended for implementing constraint solvers only.

CHR is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) into simpler ones until they are solved. In CHR, one distinguishes two main kinds of rules: Simplification rules replace constraints by simpler constraints while preserving logical equivalence, e.g. $X \geq Y \wedge Y \geq X \Leftrightarrow X=Y$. Propagation rules add new constraints, which are logically redundant, but may cause

* Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen). Part of this work was performed while visiting the University of Ulm in November 2004.

further simplification, e.g. $X \geq Y \wedge Y \geq Z \Rightarrow X \geq Z$. The combination of propagation and multi-set transformation of logical formulae in a rule-based language that is concurrent, guarded and constraint-based make CHR a rather unique and powerful declarative programming language.

Closest to our work is the presentation of a logical algorithm for the union-find problem in [10]. In a hypothetical bottom-up inference rule programming system with permanent deletions and rule priorities, a set of rules for union-find is given. The direct efficient implementation of these inference rule system seems not feasible. It is also not clear if the rules given in [10] describe the standard union-find algorithm as can be found in text books such as [4]. The authors remark that giving a rule set with optimal amortized complexity is complicated.

In contrast, we give an executable and efficient implementation that directly follows the pseudo-code presentations found in text books and that has also optimal amortized time complexity. Moreover, we do not need to rely on rule priorities. Here we analyse confluence and logical reading as well as logical correctness of our union-find program.

This paper is an revised extract of our technical report [13]. A programming pearl describing the implementation and giving a proof for the optimal time complexity is under submission [12]. This paper is structured as follows. In the next Section, we review the classical union-find algorithms. Constraint Handling Rules (CHR) are briefly introduced in Section 3. Then, in Section 4 we present the first basic implementation of the classical union-find algorithm in CHR. Relying on established analysis techniques for CHR, we investigate the logical meaning of the program. The logical reading shows that there is an inherent destructive update in the union-find algorithm. In Section 6, the detailed confluence analysis helps to understand under which conditions the algorithm works as expected. It also shows in which way the results of the algorithm depend on the order of its operations. An improved version of the implementation, featuring path compression and union-by-rank, is presented and analysed next in Section 7. Finally, Section 8 concludes.

2 The Union-Find Algorithm

The classical union-find (also: disjoint set union) algorithm was introduced by Tarjan in the seventies [14]. A classic survey on the topic is [9]. The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations on the sets:

- **make**(X): create a new set with the single element X.
- **find**(X): return the representative of the set in which X is contained.
- **union**(X,Y): join the two sets that contain X and Y, respectively (possibly destroying the old sets and changing the representative).

In the naive algorithm, these three operations are implemented as follows.

- **make**(X): generate a new tree with the only node X, i.e. X is the root.
- **find**(X): follow the path from the node X to the root of the tree by repeatedly going to the parent node of the current node until the root is reached. Return the root as representative.
- **union**(X,Y): find the representatives of X and Y, respectively. To join the two trees, it suffices to **link** them by making one root point to the other root.

The naive algorithm requires $\mathcal{O}(N)$ time per find (and union) in the worst case, where N is the number of elements (make operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve quasi-constant (i.e. almost constant) *amortized* running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root after a find. After **find**(X) returned the root of the tree, we make every node on the path from X to the root point directly to the root. The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth. If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one.

For each optimization alone and for using both of them together, the worst case time complexity for a single find or union operation is $\mathcal{O}(\log(N))$. For a sequence of M operations on N elements, the worst complexity is $\mathcal{O}(M + N \log(N))$. When both optimizations are used, the amortized complexity is quasi-linear, $\mathcal{O}(M + N\alpha(N))$, where $\alpha(N)$ is an inverse of the Ackermann function and is less than 5 for all practical N (see e.g. [4]).

The union-find algorithm has applications in graph theory (e.g. efficient computation of spanning trees). We can also view the sets as equivalence classes with the union operation as equivalence. When the union-find algorithm is extended to deal with nested terms to perform congruence closure, the algorithm can be used for term unification in theorem provers and in Prolog. The WAM [3], Prolog's traditional abstract machine, uses the basic version of union-find for variable aliasing. While *variable shunting*, a limited form of path compression, is used in some Prolog implementations [11], we do not know of any implementation of the optimized union-find that keeps track of ranks or other weights.

3 Constraint Handling Rules (CHR)

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [6, 8, 5] and about termination and confluence analysis.

3.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given

constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

Simplification rule: $Name @ H \Leftrightarrow C \mid B,$
Propagation rule: $Name @ H \Rightarrow C \mid B,$
Simpagation rule: $Name @ H \setminus H' \Leftrightarrow C \mid B,$

where *Name* is an optional, unique identifier of a rule, the *head* H , H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (query)* is a conjunction of built-in and CHR constraints. A trivial guard expression “true” can be omitted from a rule. Simpagation rules abbreviate a simplification rules of the form $Name @ H, H' \Leftrightarrow C \mid H, B$.

3.2 Operational Semantics of CHR

Given a query, the rules of the program are applied to exhaustion. A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog). When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule, when a propagation rule is applied, the body of the rule is added to the goal without removing any constraints. When a simpagation rule is applied, all constraints to the right of the backslash are replaced by the body of the rule.

This high-level description of the operational semantics of CHR leaves two main sources of non-determinism: the order in which constraints of a query are processed and the order in which rules are applied. As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program ³. This behavior has been formalized in the so-called refined semantics that was also proven to be a concretization of the standard operational semantics [5].

In this refined semantics of actual implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written, and when it matches a head constraint of a rule, it will look for the other, *partner constraints* of the head in the *constraint store* and check the guard until an applicable rule is found. We consider such a constraint to be *active*. If the active constraint has not been removed after trying all rules, it will be put into the constraint store. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied. Obviously, ground constraints need never to be considered for waking.

³ Nondeterminism due to wake-up order of delayed constraints and multiple matches for a rule are not relevant for our union-find programs [12].

3.3 Well-Behavedness: Termination and Confluence

For many existing CHR programs simple well-founded orderings are sufficient to prove termination [7]. Problems arise with non-trivial interactions between simplification and propagation rules.

Confluence of a CHR program guarantees that the result of a terminating computation for a given query is independent from the order in which rules are applied. This also implies that the order of constraints in a goal does not matter. The papers [1, 2] give a decidable, sufficient and necessary condition for confluence for terminating CHR programs. (It is also shown that confluent CHR programs have a consistent logical reading.) The condition can be readily implemented by an algorithm that is described informally in the following.

For checking confluence, one takes copies (with fresh variables) of two rules (not necessarily different) from a terminating CHR program. The heads of the rules are *overlapped* by equating at least one head constraint from each rule. For each overlap, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a non-confluence. In any consistent state that contains the overlap of a non-joinable critical pair, the application of the two rules to the overlap will usually lead to different results.

4 Implementing Union-Find in CHR

The following CHR program in concrete ASCII syntax implements the operations and data structures of the basic union-find algorithm without optimizations.

```

                                ufd_basic
make      @ make(A) <=> root(A).

union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B \ find(A,X) <=> find(B,X).
findRoot  @ root(A) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
link      @ link(A,B), root(A), root(B) <=> B ~> A, root(A).

```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the operations. `link/2` is an auxiliary relation for performing union. The constraints `root/2` and `~>/2` represent the tree data structure.

Remark. The use of the built-in constraint `=` in the rule `findRoot` is restricted to returning the element `A` in the parameter `X`, in particular no full unification is ever performed (that could rely on union-find itself).

Remark. The rule `link` can be interpreted as performing *abduction*. If the nodes `A` and `B` are not equivalent, introduce the minimal assumption $B \sim A$ so that they are equivalent (i.e. performing union afterwards leads to application of rule `linkEq`).

As usual in union-find, we will allow the following queries:

- An *allowed query* consists of `make/1`, `union/2` and `find/2` constraints only. We call these the *external* operations (constraints). The other constraints (including those for the data structure) are generated and used internally by the program only.
- The elements we use are constants. A new constant must be introduced exactly once with `make/1` before being subject to `union/2` and `find/2`.
- The arguments of all constraints are constants, with exception of the second argument of `find/2` that must be a variable that will be bound to a constant, and the second argument of `root/2`, that must be an integer.

5 Logical Properties

The logical reading of our `ufd_basic` union-find CHR program is as follows:

<code>make</code>	$make(A) \Leftrightarrow root(A)$
<code>union</code>	$union(A, B) \Leftrightarrow \exists XY (find(A, X) \wedge find(B, Y) \wedge link(X, Y))$
<code>findNode</code>	$find(A, X) \wedge A \rightarrow B \Leftrightarrow find(B, X) \wedge A \rightarrow B$
<code>findRoot</code>	$root(A) \wedge find(A, X) \Leftrightarrow root(A) \wedge X = A$
<code>linkEq</code>	$link(A, A) \Leftrightarrow true$
<code>link</code>	$link(A, B) \wedge root(A) \wedge root(B) \Leftrightarrow B \rightarrow A \wedge root(A)$

From the logical reading of the rule `link` it follows that $B \rightarrow A \wedge root(A) \Rightarrow root(B)$, i.e. *root* holds for every node in the tree, not only for root nodes. Indeed, we cannot adequately model the update from a root node to a non-root node in first order logic, since first order logic is monotonic, formulas that hold cannot cease to hold. In other words, the `link` step is where the union-find algorithm is *non-logical* since it requires an update which is destructive in order to make the algorithm efficient.

In the union-find algorithm, by definition of set operations, a union operator working on representatives of sets is an equivalence relation observing the usual axioms:

<code>reflexivity</code>	$union(A, A) \Leftrightarrow true$
<code>symmetry</code>	$union(A, B) \Leftrightarrow union(B, A)$
<code>transitivity</code>	$union(A, B) \wedge union(B, C) \Rightarrow union(A, C)$

To show that these axioms hold for the logical reading of the program, we can use the following observations: Since the unary constraints `make` and `root`

must hold for any node in the logical reading, we can drop them. By the rule `findRoot`, the constraint `find` must be an equivalence. Hence its occurrences can be replaced by `=`. Now `union` is defined in terms of `link`, which is reflexive by rule `linkEq` and logically equivalent to `~>` by rule `link`. But `~>` must be syntactic equivalence like `find` because of rule `findNode`. Hence all binary constraints define syntactic equivalence. After renaming the constraints accordingly, we arrive at the following theory:

union	$A=B \Leftrightarrow \exists XY (A=X \wedge B=Y \wedge X=Y)$
findNode	$A=X \wedge A=B \Leftrightarrow B=X \wedge A=B$
findRoot	$A=X \Leftrightarrow X=A$
linkEq	$A=A \Leftrightarrow \text{true}$
link	$A=B \Leftrightarrow B=A$

It is easy to see that these formulas are logically equivalent to the axioms for equality, hence the program is logically correct.

6 Confluence

We have analysed confluence of the union-find implementation with a small confluence checker written in Prolog and CHR. For the union-find implementation `ufd_basic`, we have found 8 non-joinable critical pairs. Two non-joinable critical pairs stem from overlapping the rules for `find`. Four non-joinable critical pairs stem from overlapping the rules for `link`. The remaining two critical pairs are overlaps between `find` and `link`.

We found one non-joinable critical pair that is unavoidable (and inherent in the union-find algorithm), three critical pairs that feature incompatible tree constraints (that cannot occur when computing allowed queries), and four critical pairs that feature pending link constraints (that cannot occur for allowed queries in the standard left-to-right execution order). In the technical report [13] associated with this paper, we also add rules by completion and by hand to make the critical pairs joinable.

The Unavoidable Non-Joinable Critical Pair The non-joinable critical pair between the rule `findRoot` and `link` exhibits that the relative order of `find` and `link` operations matters.

Overlap	<code>find(B,A),root(B),root(C),link(C,B)</code>
<code>findRoot</code>	<code>root(C),B~>C,A=B</code>
<code>link</code>	<code>root(C),B~>C,A=C</code>

It is not surprising that a `find` after a `link` operation has a different outcome if linking updated the root. As remarked in Section 5, this update is unavoidable and inherent in the union-find algorithm.

Incompatible Tree Constraints Cannot Occur The two non-joinable critical pairs for `find` correspond to queries where a `find` operation is confronted with two tree constraints to which it could apply. Also the non-joinable critical pair involving the rule `linkEq` features incompatible tree constraints.

Overlap	$A \sim B, A \sim D, \text{find}(A, C)$
findNode	$A \sim B, A \sim D, \text{find}(B, C)$
findNode	$A \sim B, A \sim D, \text{find}(D, C)$
Overlap	$\text{root}(A), A \sim B, \text{find}(A, C)$
findNode	$\text{root}(A), A \sim B, \text{find}(B, C)$
findRoot	$\text{root}(A), A \sim B, A = C$
Overlap	$\text{root}(A), \text{root}(A), \text{link}(A, A)$
linkEq	$\text{root}(A), \text{root}(A)$
link	$\text{root}(A), A \sim A$

The conjunctions $(A \sim B, A \sim D)$, $(\text{root}(A), A \sim B)$, $(\text{root}(A), A \sim A)$ and $(\text{root}(A), \text{root}(A))$ that can be found in the overlaps (and non-joinable critical pairs) correspond to the cases that violate the definition of a tree: a node with two parents, a root with a parent, a root node that is its own parent, and a tree with two identical roots, respectively. Clearly, these four conjunctions should never occur during a run of the program.

We show now that the four dangerous conjunctions indeed cannot occur as the result of running the program for an allowed query. We observe that the rule `make` is the only one that produces a `root`, and the rule `link` is the only one that produces a \sim . The rule `link` needs `root(A)` and `root(B)` to produce $A \sim B$, and it will absorb `root(A)`.

In order to produce one of the first three dangerous conjunctions, the link operation(s) need duplicate `root` constraints (as in the fourth conjunction) to start from. But only a query containing identical copies of `make` (e.g. `make(A), make(A)`) can produce the fourth dangerous conjunction. Since duplicate make operations are not an allowed query, we cannot produce any of the dangerous conjunctions (and non-joinable critical pairs) for allowed queries.

Pending Links Cannot Occur The remaining four non-joinable critical pairs stem from overlapping the rule for `link` with itself. They correspond to queries where two `link` operations have at least one node in common such that when one link is performed, at least one node in the other link operation is not a root anymore. When we analyse these non-joinable critical pairs we see that the two conjunctions $(A \sim C, \text{link}(A, B))$ and $(A \sim C, \text{link}(B, A))$ are dangerous.

Overlap	$\text{root}(A), \text{root}(B), \text{link}(B, A), \text{link}(A, B)$
link	$\text{root}(B), A \sim B, \text{link}(A, B)$
link	$\text{root}(A), \text{link}(B, A), B \sim A$
Overlap	$\text{root}(A), \text{root}(B), \text{root}(C), \text{link}(B, A), \text{link}(C, B)$
link	$\text{root}(C), A \sim B, B \sim C$
link	$\text{root}(A), \text{root}(C), \text{link}(B, A), B \sim C$
Overlap	$\text{root}(A), \text{root}(B), \text{root}(C), \text{link}(B, A), \text{link}(A, C)$
link	$\text{root}(B), \text{root}(C), A \sim B, \text{link}(A, C)$
link	$\text{root}(B), C \sim A, A \sim B$
Overlap	$\text{root}(A), \text{root}(B), \text{root}(C), \text{link}(B, A), \text{link}(C, A)$
link	$\text{root}(B), \text{root}(C), A \sim B, \text{link}(C, A)$
link	$\text{root}(B), \text{root}(C), \text{link}(B, A), A \sim C$

Once again, we argue now that the critical pairs can never arise in practice in an allowed query. `link` is an internal operation, it can only be the result of a `union`, which is an external operation. In the union, the link constraint gets its arguments from `find`. In the standard left-to-right execution order of most sequential CHR implementations [5], first the two find constraints will be executed and when they have finished, the link constraint will be processed. In addition, no other operations will be performed inbetween these operations. Then the results from the find constraints will still be roots when the link constraint receives them. Note that such an execution order is always possible, provided `make` has been performed for the nodes that are subject to union (as is required for allowed queries).

7 Optimized Union-Find

The following CHR program implements the optimized classical Union-Find Algorithm, derived from the basic version by adding path compression for find and union-by-rank [14].

ufd_rank	
<code>make</code>	<code>@ make(A) <=> root(A,0).</code>
<code>union</code>	<code>@ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).</code>
<code>findNode</code>	<code>@ A ~> B, find(A,X) <=> find(B,X), A ~> X.</code>
<code>findRoot</code>	<code>@ root(A,_) \ find(A,X) <=> X=A.</code>
<code>linkEq</code>	<code>@ link(A,A) <=> true.</code>
<code>linkLeft</code>	<code>@ link(A,B), root(A,N), root(B,M) <=> N>=M B ~> A, N1 is max(N,M+1), root(A,N1).</code>
<code>linkRight</code>	<code>@ link(B,A), root(A,N), root(B,M) <=> N>=M B ~> A, N1 is max(N,M+1), root(A,N1).</code>

When compared to the basic version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. The rule `findNode` has been extended for path compression already during the first pass along the path to the root of the tree. This is achieved by the help of the logical variable `X` that serves as a place holder for the result of the find operation. The `link` rule has been split into two rules `linkLeft` and `linkRight` to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is chosen (both rules are applicable) and the rank is incremented by one.

Remark. Path compression (cf. rule `findNode`) can be interpreted as *memoization* or *tabling* of all the (intermediate) results of the recursive find operation, where the memoized `find(A,X)` is stored as $A \rightsquigarrow X$.

The results for logical reading and logical correctness of the optimized union-find are analogous to the ones for `ufd_basic`.

Confluence Revisited The non-joinable critical pairs (CPs) are in principle analogous to the ones discussed for `ufd_basic` in Section 6, but their numbers significantly increases due to the optimizations of path compression and union-by-rank that complicate the rules for the find and link operations.

Our confluence checker found 73 non-joinable critical pairs. The number of critical pairs is dominated by those 68 of the link rules. Not surprisingly, each critical pair involving `linkLeft` has a corresponding analogous critical pair involving `linkRight`.

The CPs between `findRoot` and a link rule are the unavoidable critical pairs as in `ufd_basic`. These show the expected behavior that the result of `find` will differ if its executed before or after a link operation, for example:

Overlap	<code>find(B,A),root(B,C),link(E,B),root(E,D),D>=C</code>
<code>findRoot</code>	$A=B, D>=C, N \text{ is } \max(D, C+1), \text{root}(E, N), B \rightsquigarrow E$
<code>linkLeft</code>	$A=E, D>=C, N \text{ is } \max(D, C+1), \text{root}(E, N), B \rightsquigarrow E$

Two `findNode` rule applications on the same node will interact, because one will compress, and then the other cannot proceed until the first find operation has finished:

Overlap	<code>find(B,A),B~>C,find(B,D)</code>
<code>findNode</code>	<code>find(A,D),find(C,A),B~>D</code>
<code>findNode</code>	<code>find(D,A),find(C,D),B~>A</code>

We see that `A` and `D` are interchanged in the states of the critical pair. In the first state, since the result of `find(C,A)` is `A`, the `find(A,D)` can eventually only reduce to $A=D$. Analogously for the second state. But under $A=D$ the two states of the critical pair are identical. The other two critical pairs involving a `findNode` rule correspond to impossible queries $B \rightsquigarrow C, B \rightsquigarrow D$ and $\text{root}(B, N), B \rightsquigarrow C$ as discussed for the confluence of `ufd_basic`.

All critical pairs between link rules only, except those for `linkEq`, consist of pairs of states that have the same constraints and variables, but that differ in

the tree that is represented. Just as in the case of `ufd.basic` the problem of pending links occurs without a left-to-right execution order. For more details see [13].

8 Conclusion

We have analysed in this paper basic and optimal implementations of classical union-find algorithms. We have used and adapted established reasoning techniques for CHR to investigate the logical properties and confluence (rule application order independence). The logical reading and the confluence check showed the essential destructive update of the algorithm when trees are linked. Non-confluence can be caused by incompatible tree constraints (that cannot occur when computing with allowed queries), and due to competing link operations (that cannot occur with allowed queries in the standard left-to-right execution order).

Clearly, inspecting dozens of critical pairs is cumbersome and error-prone, so a refined notion of confluence should be developed that takes into account allowed queries and syntactical variations in the resulting answer.

At <http://www.cs.kuleuven.ac.be/~toms/Research/CHR/UnionFind/> all presented programs as well as related material are available for download. The programs can be run with the proper time complexity in the latest release of SWI-Prolog.

In future work we intend to investigate implementations for other variants of the union-find algorithm. For a parallel version of the union-find algorithm parallel operational semantics of CHR have to be investigated (confluence may be helpful here). A dynamic version of the algorithm, e.g. where unions can be undone, would presumably benefit from dynamic CHR constraints as defined in [15].

Acknowledgements. We would like to thank the participants of the first workshop on CHR for raising our interest in the subject. Marc Meister and the students of the constraint programming course at the University of Ulm in 2004 helped by implementing and discussing their versions of the union-find algorithm.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2), 1999.
3. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

5. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, 2004.
6. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.
7. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
10. H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *International Joint Conference on Automated Reasoning*, LNCS 2083, pages 514–528. Springer, 2001.
11. D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.
12. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules. Technical report, November 2004.
13. T. Schrijvers and T. Frühwirth. Union-find in chr. Technical Report CW389, Department of Computer Science, K.U.Leuven, Belgium, July 2004.
14. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.
15. A. Wolf. Adaptive constraint handling with chr in java. In *7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, LNCS 2239. Springer, 2001.

DB_CSP: A Framework and Algorithms for Applying Constraint Solving within Relational Databases

Chuang Liu¹, Ian Foster^{1,2}

¹University of Chicago, 1100 E. 58th street
Chicago, IL, 60637, USA

chliu@cs.uchicago.edu

²Mathematics and Computer Science Division, Argonne National Laboratory,
Argonne, IL 60439, USA

Abstract. We propose an approach to address combinatorial search problems in relational database system with constraint solving technology. First, we formalize combinatorial queries in databases as a new type of constraint satisfaction problem: what we call Database Constraint Satisfaction Problems (DB_CSPs). Then, we develop algorithms for solving DB_CSPs that integrate database data management technology with conventional CSP solving techniques. Our algorithms enable efficient domain refinement for tuple-valued data and minimize I/O operations when relations are stored on disk. Finally, we present experimental studies that both quantify the performance of our approach and permit comparison with a conventional database system. We show that our approach is superior for solving complex combinatorial queries.

1 Introduction

The relational data model and relational database technology have achieved great success as a means of managing, and implementing queries on, large amounts of data. However, new application domains and decision procedures lead to new classes of query that are not handled efficiently by today's database systems. In the *combinatorial queries* that we consider in this paper, a request for tuples from multiple relations that satisfy a conjunction of constraints on tuple attribute values leads to a combinatorial search problem.

To illustrate how combinatorial search problems can arise, we consider an Internet information system that maintains a relation *C* storing descriptions of compute resources, a relation *N* storing descriptions of network resources, and a relation *S* storing descriptions of storage resources. Relation *C* has attributes *cpuSpeed*, *price*, and *os*; *N* has attributes *bandwidth* and *price*; and *S* has attributes *ioSpeed* and *price*. In the combinatorial query shown in Figure 1, a user requests a compute, network, and storage resource that collectively satisfy deadline (10 minutes) and budget (\$5) constraints for an application that first calculates on a compute resource (time $10/\text{cpuSpeed}$), then transmits results over a network resource ($10/\text{bandwidth}$), and finally stores results on a storage resource ($4/\text{ioSpeed}$). Note that the query condition is a conjunction of two arithmetic constraints on attributes from three relations.

Database systems implement such combinatorial queries by a set of pair-wise join operators. However, optimization techniques for pair-wise joins, such as sorted-join and hash-join, deal only with simple constraints such as equality or inequality

comparisons between two attributes, and cannot, in general, process combinatorial queries efficiently. Thus, evaluation involves a complete traversal of the search space.

C				N			D		
tuple#	cpuSpeed	price	os	tuple#	bandwidth	price	tuple#	ioSpeed	price
1	6	1	linux	1	1	1	1	6	1
2	1	4	unix	2	2	2	2	1	4
3	2	2	windows	3	4	3	3	2	5
4	3	8	unix	4	8	6	4	3	8
5	4	5	linux				5	3	3
6	6	1	unix						
7	3	4	windows						
8	2	3	linux						


```

SELECT * FROM C, N, D WHERE
  C.price + N.price + D.price <= 5          AND
  10/C.cpuSpeed + 10/N.bandwidth + 4/D.ioSpeed < 10

```

Figure 1. An example combinatorial query, showing the three relations and query.

Combinatorial search has of course been thoroughly studied by researchers in constraint programming [8,13,15], who have developed efficient search algorithms that avoid evaluating all possible states by pruning unsatisfiable results from the search space. Thus, the question that we ask in this paper: can we exploit combinatorial search algorithms from constraint programming to optimize the execution of combinatorial queries in relational databases?

An investigation of this question is not straightforward. Algorithms developed for constraint programming apply only to simple data types and assume that all data is stored in memory. In contrast, combinatorial queries involve searches for tuples that may be stored on disk. Furthermore, it is important that new implementation techniques can be incorporated easily into existing database systems. Nevertheless, we show that it is indeed feasible to overcome these challenges and to integrate highly optimized data management mechanisms from databases with efficient combinatorial search algorithms from constraint programming.

Our approach to this problem proceeds in three stages. First, in Section 3, we show how combinatorial queries can be modeled as a new class of constraint satisfaction problem: what we call Database Constraint Satisfaction Problems (DB_CSPs). Constraint satisfaction problems (CSP) are often used to model combinatorial search problems, and numerous search algorithms are available. However, searching large databases introduces new challenges. We define DB_CSP to model these challenges.

In Section 4, we propose an algorithm for solving DB_CSPs. We address the difficulties introduced by the large value domains encountered in DB_CSPs by designing a value domain representation and a constraint-solving algorithm that together enable both compact storage and efficient execution. We use constraint-solving technology to control the search process, and database indexing technology to implement data manipulation operations such as domain refinement.

In Section 5, we report on experiments that allow us to quantify the performance characteristics of our approach, and to compare this performance with that of a conventional database system. We find that our approach is significantly more efficient than conventional database join algorithms when handling queries with complex join conditions and small selectivity.

2 Related Work

Combinatorial search problems arise in many contexts, and we find a wide variety of approaches to their solution. Here we review the most relevant previous work.

In relational database management systems, combinatorial search is implemented by a multi-join operation, which combines information from multiple relations to determine if an aggregation of tuples from these relations satisfies search conditions. However, existing join algorithms, such as nested-loop, sorted-join, and hash-join [6], apply only to simple constraints. Heuristic combinatorial search algorithms [5,11] have been developed, but apply only to particular types of constraints.

Constraint programming [8,13,15] has been used to formulate combinatorial search problems in areas such as scheduling, routing, and timetabling that involve choosing from among a finite number of possibilities. Correspondingly, constraint-solving algorithms have been developed in several research communities, including node and arc consistency techniques [9,16] in artificial intelligence, bounds propagation techniques [13] in constraint programming, and integer programming techniques [17] in operations research. However, although these algorithms solve combinatorial search problems efficiently, implementations have been within modeling languages [1,3,4,10] and have not been used to manage queries on large data sets, as a database system does.

Constraint databases [12,14] extend the expressiveness of conventional relational databases by using constraints to model relations with an infinite number of data. In contrast, we use constraints to model the search process on conventional databases, with a view to improving the performance of combinatorial queries by incorporating constraint-solving algorithms into a database system.

3 Database Constraint Satisfaction Problems

The essence of our approach is to treat combinatorial search on databases as a constraint satisfaction problem and apply constraint-solving technologies to implement search and selection functions.

A combinatorial search on a database is triggered by a *combinatorial query*: a query that specifies selection conditions for multiple tuples from one or several relations. Existing constraint programming languages [2,3] usually formalize a combinatorial query as a constraint satisfaction problem (CSP) by mapping each database relation to a constraint, and associating a variable to each attribute appeared in the query condition. For example, we can express the search problem in Figure 1 as the following CSP, in which the constraint c , n , and d denote the relations.

$$\begin{aligned} &C_price + N_price + D_price \leq 5 \wedge \\ &10/C_cpuSpeed + 10/N_bandwidth + 4/D_ioSpeed < 10 \wedge \\ &C_price \in [1, 2, 3, 4, 5, 8] \wedge C_cpuSpeed \in [1, 2, 3, 4, 6] \dots \wedge D_ioSpeed \in [\dots] \wedge \\ &c(_, C_price, C_cpuSpeed) \wedge n(_, N_bandwidth, N_price) \wedge d(_, D_ioSpeed, D_price). \end{aligned}$$

We argue that this method may have following limitations when dealing with queries on large size database.

- The constraint solver usually needs to read in tuples in database relations and transform them to a constraint in memory before solving a query. It may cause unnecessary I/O operations because a lot of tuples in a database relation won't lead to any solution.

- The value domains of variables are often too large to maintain in memory, and thus domain refinement may cause expensive I/O operations. Database system may process multiples combinatorial queries concurrently that makes the memory efficiency of constraint solving more important.
- Using variables to represent attributes of a tuple may cause extra constraint checks in constraint propagation. For the constraint $c(C_tuple\#, C_Price, C_cpuSpeed)$ in previous example, if several values are removed from the domain of C_Price , we need to check the constraint c in order to remove values from the domain of $C_cpuSpeed$.

Based on these considerations, we propose to formalize a combinatorial query as a constraint satisfaction problem (CSP) by associating a variable with every required tuple. The domain of each variable comprises all tuples in the associated relation. Constraints on variables describe selection conditions. We call such a CSP a DB_CSP . For example, we can express the search problem in Figure 1 as the following CSP, in which the variables c , n , and d denote the required tuples.

$$\begin{aligned} &c.price + n.price + d.price \leq 5 \wedge \\ &10/c.cpuSpeed + 10/n.bandwidth + 4/d.ioSpeed < 10 \wedge \\ &c \in C \wedge n \in N \wedge d \in D \end{aligned}$$

Definition 1: A database constraint satisfaction problem (DB_CSP) is a CSP in which variable domains are database relations. ■

Any valid combinatorial database query can be expressed as a DB_CSP as just shown. DB_CSP s share some common features different from conventional CSPs, as follows.

- The values of variables are tuples with multiple attributes.
- Constraints on variables are expressed as constraints on attributes of those variables.
- The number of variables is typically not large, but the domain of each variable may be extremely large.

These features lead us instead to pursue an approach to the solution of DB_CSP problems that focuses on their unique characteristics.

4 Solving DB_CSP Problems

Having modeling combinatorial queries as DB_CSP s, we now investigate how constraint-solving algorithms can be integrated with database technologies to implement the required combinatorial search. In considering this question, we must recognize that database systems incorporate highly optimized mechanisms for managing large numbers of record-like data. We use a constraint-solving algorithm to control the search process and database index technology to implement data manipulation operations such as domain refinement, which removes values from variable domains.

Considering the large size of variable domains of a DB_CSP , we require a value domain representation and constraint-solving algorithm that together enables both compact storage and efficient execution for multiple concurrent or successive DB_CSP s, so that we can perform domain refinement efficiently without requiring numerous I/O operations and high memory costs.

4.1 Representation of Variable Domains

We represent a variable domain in terms of (a) the relation that stores tuples in the variable domain and (b) a set of simple constraints on variables that we term the *filter*. The variable domain then corresponds to all tuples from the relation that satisfies the constraints in the filter. All DB_CSPs on a database share the same relations, but each has its own filter. The constraint-solving algorithm refines the domains of variables by adding constraints to the filter. Thus, relations do not change during constraint solving, ensuring that different DB_CSPs do not interfere with each other.

c	n	d
price	price	price
cpuSpeed	bandwidth	ioSpeed
$\geq 1, \leq 8$	$\geq 1, \leq 6$	$\geq 1, \leq 8$
$\geq 1, \leq 6$	$\geq 1, \leq 8$	$\geq 1, \leq 6$

Figure 2. Representation of the filter for the example in Figure 1.

We implement the filter as a set of symbol tables, one per variable appearing in the DB_CSP. These symbol tables maintain information about the attributes appearing in the DB_CSP. Every symbol table entry contains an attribute name and known basic constraints on this attribute. We maintain two types of *basic constraints* on attributes in the symbol table: *assignment* constraints such as $attr = N$ and *range* constraints such as $attr > N$. We use these two types of constraint because they can be easily used by database index techniques to locate values belonging to a value domain from a relation. Initially, attributes are described by range constraints representing attribute value bounds derived from relations. For example, the value domain for c as specified in Figure 1 provides the constraints $c.cpuSpeed \geq 1$ and $c.cpuSpeed \leq 6$ (Figure 2).

A filter requires little space and thus can be kept in memory. The basic constraints contained in the filter can be seen as a summary of the variable domains. As shown in Section 4.2, this information is both used to refine the variable domain and is updated by the constraint-solving algorithm as the variable domain is reduced.

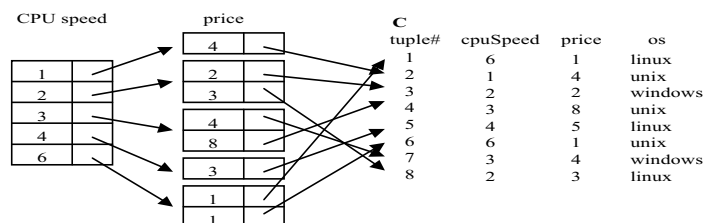


Figure 3. Multiple-key indexes for attributes cpuSpeed and price in relation C of Figure 1.

Values in variable domains are tuples satisfying the constraints in the filter. A naive way to pick a value from variable domain is to read tuples repeatedly from the relation and evaluate the constraints in the filter until a satisfying tuple is found. A more efficient approach is to build indices on relations and use those indexes to locate satisfying tuples. A filter may contain multiple basic constraints on attributes and thus we use multi-dimensional indexing techniques from database systems. Different types of multi-dimensional index are available (e.g., grid file, bitmap) with efficiencies that depend on the data in the relations. In this paper, we use multi-key indexing to illustrate the use of indices for solving DB_CSPs.

For every variable, we build a multi-key index [6] on those of its attributes that appear in the DB_CSP. Figure 3 presents an example that illustrates the approach. The root of the tree is an index for attribute *cpuSpeed*. The index could be any type of conventional index that support range queries, such as a B-tree. The index associated with each of its values is a pointer to another index on attribute *price*. If V is a value of attribute *cpuSpeed*, then by following value V and its pointer we reach an index into the set of tuples that have *cpuSpeed* equal to V and any value for attribute *price*.

The multiple-key index works well for locating tuples with range constraints and assignment constraints, as long as the indexes themselves support range queries on their attributes. To locate tuples based on range constraints, we use the root index and the range of the first attribute to find all sub-indexes that might contain answer points. We then search each sub-index, using the range specified for the second attribute.

4.2 Clustering Value Domains

The search space of a CSP is the Cartesian product of the size of its variable domains. Variable domains may be large and thus even an efficient constraint-solving algorithm may still take a lot of time to find results.

We observe that many tuples have similar attribute values. For a DB_CSP with constraints on a subset of the attributes of a required tuple, tuples with the same value on those attributes can be viewed as identical even if they have different values for other attributes. For convenience, we call these tuples a *cluster*. If one tuple in a cluster will not lead to a solution, no tuple in the cluster will do so. On the other hand, if we find a solution containing one tuple in a cluster, we can get new solutions by selecting any other tuple in the cluster. Because we need to check only one tuple per cluster during the search process, the size of the search space is reduced to the Cartesian product of the cluster sizes.

This approach of clustering value domains may reduce the search space of a DB_CSP remarkably if attributes appearing in the constraint have a value chosen from a finite domain. For example, computers might have an attribute *operating system*, with permitted values of only *Linux* and *Windows*; thus, only two clusters are formed for this attribute.

For every DB_CSP, we preprocess variable domains such that tuples are organized into clusters. Indices on variable domains (relations) make it easy to cluster value domains. For the example in Figure 3, we check the five indexes on attribute *price*. For every index, tuples with the same value on attribute *price* belong to the same cluster. Thus, tuples 1 and 6 in Figure 3 belongs to a cluster.

4.3 Solving Algorithm

To allow for variable domains stored on disk, we need a constraint-solving algorithm with I/O efficiency in addition to computational efficiency. We only consider a complete algorithm in this paper because, in many cases, database queries require that the search algorithm either return exact results or determine that the query is unsatisfiable. However, our approach can be easily applied to incomplete algorithms.

Based on these considerations, we develop a solving algorithm that combines complete backtracking search with the consistency techniques shown in Figure 4.

The function *Search_Algorithm* in Figure 4 uses backtracking to return all solutions for a constraint C . If all variables in C are instantiated, the function returns a

solution if C holds (lines 3-4). Otherwise, the function assigns a value to an undetermined variable with minimum domain size and searches for solutions consistent with this assignment (lines 10-13).

```

// C is a constraint of form  $c_1 \wedge \dots \wedge c_n$ 
// D is variable domains
1  Search_Algorithm(C, D)
2      If all variables in C are bound to a value
3          If constraint C holds
4              Output a solution
5          Else
6              Return //no solution for this constraint
7      If ( !Consistency_Algorithm(C, D) )
8          Return //no solution for this constraint
9      Choose variable v with the minimal value domain
10     For each  $d \in D(v)$  // D(v) is domain of variable v
11         Assign d to v
12         Search_Algorithm(C  $\wedge$  v=d)
13     Return

14 Consistency_Algorithm(C, D)
15 While (C is not node or bounds consistent)
16     For i = 1 to n do
17         a = NodeConsistencyAlgorithm( $c_i$ , D)
18         b = BoundsConsistencyAlgorithm( $c_i$ , D)
19         If (!a || !b) return false
20 Return true

```

Figure 4. The DB_CSP solving algorithm.

We also use the node consistency algorithm and the bounds consistency algorithms to refine variable domains (function *Consistency_Algorithm* in Figure 4). These algorithms can easily be integrated with database index techniques to efficiently refine value domains that are stored in disk.

4.4 Consistency Algorithms

Consistency algorithms are usually developed for basic data types. Here we propose new consistency algorithms for structured data that refine variable domains without accessing tuples in the relation. Thus, if tuples are stored in disk, our algorithms are more I/O efficient than conventional node and bound consistency algorithms.

In a DB_CSP, variables are structured data, and constraints on variables are expressed by a set of primitive constraints on the attributes of variables. We define *attribute node consistent* and *attribute bound consistent* as follows.

Definition 2: A primitive constraint c is *attribute node consistent* [13] if either the constraint involves multiple variable attributes or if all values in the domain of the single attribute are a solution of c . The domain of an attribute is the collection of attribute values appeared in the variable domain. ■

Given a primitive constraint c and a variable domain, the attribute node consistency algorithm removes values from the variable domain until c is attribute node consistent. As discussed in Section 4.1, we represent the variable domain by a filter and a relation. The attribute node algorithm reduces the variable domain by adding the unary constraint on one attribute to the filter. Thus, the refinement does not need to access the relation. To illustrate, consider the example in Figure 2 and a primitive constraint $c.price \leq 5$. The variable domain after the attribute node consistency

algorithm is shown as Figure 5. In the symbol table for variable c , the range constraint on the attribute $price$ has been changed to $1 \leq c.price \leq 5$, i.e., the combination of original constraint $1 \leq c.price \leq 8$ and $c.price \leq 5$.

c		n		d	
price	$\geq 1, \leq 5$	price	$\geq 1, \leq 6$	price	$\geq 1, \leq 8$
cpuspeed	$\geq 1, \leq 6$	bandwidth	$\geq 1, \leq 8$	ioSpeed	$\geq 1, \leq 6$

Figure 5. The variable domains of Figure 1 after the execution of the node consistency algorithm on the constraint $c.price \leq 5$.

Definition 3: An arithmetic primitive constraint c is *attribute bounds consistent* [13] if for each variable attribute $attr$ that appears in this constraint there is:

- An assignment of values d_1, d_2, \dots, d_k to the remaining variable attributes in c , such that $\min(attr_i) \leq d_i \leq \max(attr_i)$ for each d_i , and $\{attr = \max(attr), attr_1 = d_1, attr_2 = d_2, \dots, attr_k = d_k\}$ is a solution of c ; and
- An assignment of values d_1', d_2', \dots, d_k' to the remaining variable attributes in c , such that $\min(attr_i) \leq d_i \leq \max(attr_i)$ for each d_i and $\{attr = \min(attr), attr_1 = d_1', attr_2 = d_2', \dots, attr_k = d_k'\}$ is a solution of c .

Where $\min(attr)$ (or $\max(attr)$) represents the minimal (maximal) value of attribute $attr$ in the domain of the corresponding variable. ■

Given a constraint $c.price + n.price + d.price \leq 5$ and variable domain in Figure 1, we can get the new range of attribute $price$ of variable c by

$$c.price \leq 5 - n.price - d.price \leq 5 - \min(n.price) - \min(d.price) = 3$$

We can get the new range of attribute $price$ of variables d and n in the same way. We refine the variable domain using this new range by adding it to the filter. We show the variable domains that are attribute bounds consistent for constraint $c.price + n.price + d.price \leq 5$ in Figure 6. Note that the range of the attribute $price$ has been changed for each of the three variables.

c		n		d	
price	$\geq 1, \leq 3$	price	$\geq 1, \leq 3$	price	$\geq 1, \leq 3$
cpuspeed	$\geq 1, \leq 6$	bandwidth	$\geq 1, \leq 8$	ioSpeed	$\geq 1, \leq 6$

Figure 6. The variable domains of Figure 1 after the execution of the bounds consistency algorithm on the constraint $c.price + n.price + d.price \leq 5$.

The bounds consistency algorithm needs only range information for attributes that are maintained in the filter. Because we can keep the filter in memory, the algorithm requires no additional I/O operations to read tuples in the relations.

4.5 Assignment Operation

During backtracking, the solver picks a value from the variable domain and assigns this value to the variable (line 11). The variable domain is represented as a filter and relations (Section 4.1). The values of a variable are tuples in relations and thus we use the basic constraints in the filter to pick a value that belongs to the variable domain. Recall that basic constraints are range constraints and assignment constraints on one attribute; thus selection operations can be implemented efficiently by using database index techniques. For example, given the filter shown in Figure 6, when picking a value for variable c we use constraints in the corresponding symbol table ($1 \leq price \leq$

3 and $1 \leq \text{cpuSpeed} \leq 6$) to query the relation C . The multi-key index shown in Figure 3 allows the database to locate tuples that satisfy this selection condition efficiently.

5 Performance Evaluation

We now present an experimental evaluation of the combinatorial query performance of both our DB_CSP solving algorithm and a conventional database system.

5.1 The Benchmark Query Q

We first define a simple combinatorial query Q with parameters that can be used to vary its execution complexity and selectivity. This query operates on three relations A , B , and C , defined according to relations *TENKTUPI* from the Wisconsin Benchmark [7]. In order to simulate complex combinatorial queries, Q comprises two constraints, each involving three relations.

```
Q: SELECT * FROM A, B, C
WHERE
  A.K50 + B.K50 + C.K50 < N1 AND
  A.K200 + B.K200 + C.K200 > N2
```

This query specifies a request for three tuples from relations A , B , and C such that the sum of their attribute $K50$ is less than a constant N_1 and the sum of their attribute $K200$ is more than a constant N_2 . The value of attribute $K50$ is an integer uniformly distributed between 1 and 50, and the value of attribute $K200$ is an integer uniformly distributed between 1 and 200.

Notice that varying the sizes of A , B , and C changes the size of the problem to be solved, while varying the parameter(s) N_1 and N_2 changes the number of results obtained, i.e., the selectivity of Q . We vary both the relation sizes and N_2 in our experiments to obtain queries with various problem sizes and selectivity.

5.2 Experimental Studies

We use three performance metrics to evaluate the search efficiency of both our algorithm and conventional database search algorithms: the total number of I/O operation performed, the number of constraint check operations per result, and the elapsed time required to obtain query results.

- *I/O operations.* It is widely used to measure the performance of a database search. Because the algorithm of Section 4.3 reads a tuple only when it extends an intermediate result and we perform a constraint check for every intermediate result, the number of read operations is equal to the number of constraint check operations. Thus, we use the number of constraint check to measure I/O operations.
- *Constraint check operations per result.* The number of query results returned by a search represents a lower bound on the number of constraint check operations that a search algorithm must perform. We use the ratio of the number of constraint checks performed to the number of results obtained as a normalized measure of how close an algorithm is to this bound.
- *Elapsed time.* The final metric is the time between query submission and the return of results.

In our experiments, we compare the elapsed times of the implementation of our DB_CSP algorithm and the algorithm implemented in the Postgres relational database system (Version 7.3), which we choose for its accessibility. Postgres implement the

search process by pair-wise join operations. We recognize that Postgres may perform less well than other database systems, but argue that because different database systems use similar execution algorithms, our results extend broadly.

We conducted all experiments on an IBM T20 with a single Pentium III 700MHz processor and 128 MB memory, running Suse Linux 7.2.

5.3 Changing Relation Size

In this first set of experiments, we vary the number of tuples in each of the relations A, B, and C from 125 to 64,000 while fixing the join selectivity by setting $N_1=6$ and $N_2=520$. Figure 7 shows the elapsed time measured for both our DB_CSP algorithm and Postgres. Our algorithm is faster than Postgres by from two to four orders of magnitudes with the difference becoming larger as the relation size increases.

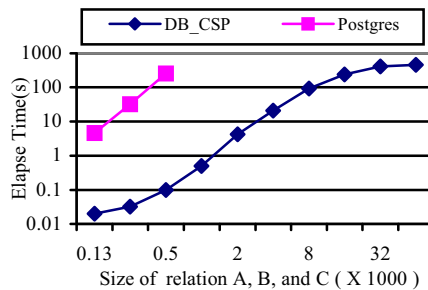


Figure 7. Time taken to evaluate Q , as a function of the size of the relations A, B and C.

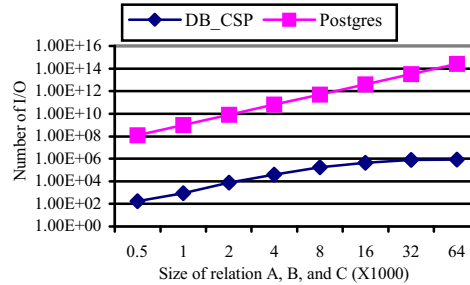


Figure 8. Number of I/O operations performed to evaluate Q , as a function of the size of relations A, B, and C.

The reason for this difference in execution times is made clear by Figure 8, which shows the number of I/O operations performed. We see that Postgres performs six orders of magnitude more constraint checks than our algorithm. This difference arises because Postgres uses an execution plan that creates all combinations of tuples and checks them one by one. For a query with low selectivity, this strategy may lead to many unnecessary computations. In contrast, our DB_CSP algorithm can use constraints in Q to filter intermediate results and thus performs far fewer constraint checks. This improvement in efficiency is achieved by using consistency techniques to prune parts of the search space containing no solutions.

We also see that, for our algorithm, the execution time and the number of constraint checks both increase less rapidly with relation size than in the case of Postgres. For example, the execution time of Q does not change much when the relation size increases from 32,000 to 64,000. The reason is that our algorithm uses clustering (Section 4.2) to organize tuples into clusters and considers only one tuple per cluster during the search process. Query Q specifies constraints on attribute $K50$ and $K200$ that are used to cluster value tuples. $K50$ picks values from 1 to 50 and $K200$ picks values from 1 to 200; thus, the number of possible combinations of $K50$ and $K200$ is 10,000. After clustering, no matter what the relation sizes, the variable domains of the DB_CSP are smaller than 10,000, which put an upper bound on the execution time of query Q .

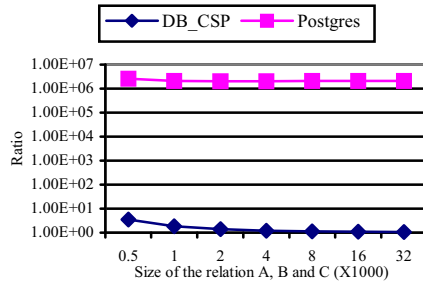


Figure 9. Ratio of constraint checks performed to results obtained when evaluating Q, as a function of size of relations A, B, and C.

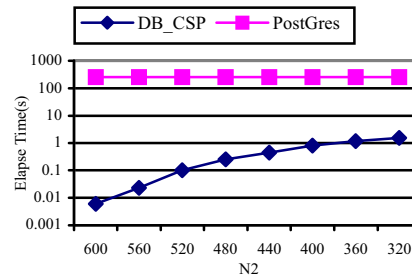


Figure 10. Time taken to evaluate Q when the relations A, B, and C have size 500, as a function of parameter N_2 .

Figure 9 shows the ratio of the number of constraints checked to the number of query results. Postgres performs around 2,100,000 constraint checks per result. In contrast, our DB_CSP algorithm performs only a few constraint checks per result. Indeed, for large relations this number declines to little more than one, which is the lower bound for any search algorithm.

In summary, our algorithm is more efficient at evaluating queries with arithmetic constraints such as Q because it performs many fewer constraint checks. We are confident that similar results would be obtained for other similar queries.

5.4 Changing Selectivity

In this second set of experiments, we fix the number of tuples in relations A, B, and C to 500 and vary the join selectivity by varying the constant N_2 in the selection condition of Q.

Figure 10 compares the elapsed times to evaluate the query. When query selectivity is small, our DB_CSP algorithm has a large performance advantage relative to Postgres. As selectivity increases, the difference between the two systems decreases. The reason is that as the number of results increases, the search space becomes filled with more results, leaving fewer opportunities for a more efficient search algorithm to improve search performance. In summary, our results show that our algorithm performs particularly well when combinatorial queries are complex and query selectivity is small.

6 Summary and Future Work

Combinatorial queries in database systems seek tuples from multiple relations with a particular relationship. Existing relational database systems can express combinatorial queries but cannot execute them efficiently. To improve on this situation, we show how combinatorial queries can be modeled as a new type of CSP problem, DB_CSP, and design and implement a combinatorial search algorithm that integrates constraint-solving techniques with database search techniques. Experimental results show that our algorithm has significant performance advantages relative to traditional database technology when solving combinatorial queries with complex selection conditions, particularly when query selectivity is low. Our algorithm can also be incorporated easily into existing relational database systems.

Acknowledgements

This work was supported by the Grid Application Development Software project of the NSF Next Generation Software program, under Grant No. 9975020. We are grateful to Alvaro Fernandes, Mike Franklin, Svetlozar Nestorov, Anne Rogers, Matei Ripeanu, Alain Roy, and Lingyun Yang for comments on a draft of this paper.

References

- [1] Aggoun, A. and Beldiceanu, N., Overview of the CHIP compiler system. In F. Benhamou and A. Colmerauer (Eds.), *Constraint Logic Programming: Selected Research*, MIT Press, 1993, pp. 421-435.
- [2] Carlsson, M., Ottosson, G. and Carlson, B., An open-ended finite domain constraint solver. *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.
- [3] Cheadle, A., Harvey, W., Sadler, A., Schimpf, J., Shen, K. and Wallace, M., ECLiPSe: An Introduction. Imperial College London, IC-Parc-03-01, 2003.
- [4] Diaz, D. and Codognet, P., A minimal extension of the WAM for clp(FD). In D.S. Warren (Ed.), *Logic Programming: Proceedings of the 10th International Conference*, Budapest, Hungary, 1993, pp. 774-792.
- [5] Fagin, R., Lotem, A. and Naor, M., Optimal aggregation algorithms for middleware, *Journal of Computer and System Sciences*, 66 (2003) 614-656.
- [6] Garcia-Molina, H., Ullman, J.D. and Widom, J., *Database Systems: the Complete Book*, Prentice Hall, Upper Saddle River, NJ, 2002, xxvii, 1119 pp.
- [7] Gray, J., *The Benchmark Handbook for Database and Transaction Systems*, 2nd edn., Morgan Kaufmann, 1993.
- [8] Hentenryck, P.V., *Constraint Satisfaction in Logic Programming*, The MIT Press, 1989.
- [9] Hentenryck, P.V., Deville, Y. and Teng, C.-M., Generic arc-consistency algorithm and its specializations, *Artificial Intelligence*, 57 (1992) 291-321.
- [10] ILOG, ILOG Optimization Suite White Paper. 2001.
- [11] Ilyas, I.F., Aref, W.G. and Elmagarmid, A.K., Supporting Top-K Join Queries in Relational Databases. *VLDB 2003*, Morgan Kaufmann, Berlin, Germany, 2003.
- [12] Kanellakis, P., Kuper, G. and Revesz, P., Constraint query languages. *Proc. 9th ACM PODS*, ACM press, 1990, pp. 299-313.
- [13] Marriott, K. and Stuckey, P.J., *Programming with Constraints: An Introduction*, The MIT Press, Cambridge, Massachusetts, 1998.
- [14] Revesz, P., *Introduction to Constraint Databases*, Springer, New York, 2002.
- [15] Tsang, E., *Foundations of Constraint Satisfaction*, Academic Press, London, 1993.
- [16] Waltz, D.L., Understanding line drawings of scenes with shadows. In P.H. Winston (Ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975, pp. 19-21.
- [17] Wolsey, L.A. and Nemhauser, G.L., *Integer and Combinatorial Optimization*, first edn., Wiley-Interscience, 1999.

System Description: Meta-S – Combining Solver Cooperation and Programming Languages

Stephan Frank, Petra Hofstedt, Dirk Reckmann

Berlin University of Technology, Germany
{sfrank,ph,dyrgh}@cs.tu-berlin.de

Abstract. Meta-S is a constraint solver cooperation system which allows the dynamic integration of arbitrary external (stand-alone) solvers and their combination with declarative languages. We sketch the main aspects of Meta-S including solver and language integration as well as its strategy definition framework for specifying solver cooperation and language evaluation strategies by means of an example.

1 Motivation

Constraint solvers offer problem solving algorithms in an encapsulated way. Many solvers have been designed and implemented, covering several different constraint domains, for example finite domain problems, linear arithmetic or interval arithmetic. However, many real world problems do not fit nicely into one of these categories, but contain constraints of various domains. Writing specialized solvers that can tackle particular multi-domain problems is a time consuming and error prone task. A more promising approach is the integration of existing constraint solvers into a more powerful overall system for solver cooperation. Meta-S – our flexible and extendable constraint solver cooperation system with support for integration of arbitrary declarative languages – implements this idea.

2 Meta-S

Constraint Solver Cooperation Figure 1 shows the general architecture of our constraint solver cooperation framework [3]. The system consists of several constraint *solvers*, each maintaining its own *store*. The collaboration is coordinated by the *meta solver* that establishes an exchange of information between the connected solvers. This meta solver maintains a *pool* of constraints to be solved.

The communication between the meta solver and the individual solvers is done solely through two interface functions (readily supported by many pre-existing solvers). A *propagation* function adds a constraint from the constraint pool into a solver's store, hereby ensuring consistency of the resulting constraint store. The second interface function handles *projection*, i.e. it infers knowledge implied by a constraint store in form of constraints that are put into the pool and passed to other solvers.

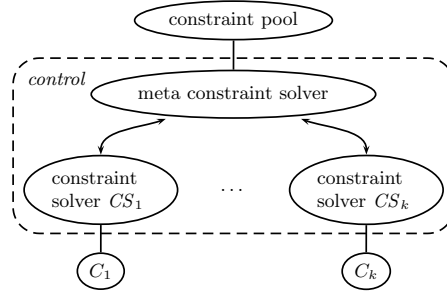


Fig. 1. Architecture of Meta-S

```

1  (define-meta-solver *smm*
2    (meta-solver eager-propagation-reordering)
3    ((my-fd fd-rational)                      ; solver integration
4     (my-lin cllin)
5     (my-cll cll-solver :file "smm.cll")))
6
7    (((in S,E,N,D,M,O,R,Y #0 1 2 3 4 5 6 7 8 9)) ; constraints
8     (alldifferent {S E N D M O R Y})
9     (= (+ SEND MORE) MONEY)
10    (word ( [S,E,N,D] , SEND ))
11    (word ( [M,O,R,E] , MORE ))
12    (word ( [M,O,N,E,Y] , MONEY )))))

```

Fig. 2. The Send-More-Money problem specification

Example 1. To illustrate the usage of Meta-S, we consider the famous Send-More-Money problem.¹ Figure 2 shows the problem specification file. We use a collaboration of three individual constraint solvers: a *finite domain solver* for rational numbers (Line 3), a *linear arithmetic solver* (Line 4) and a constraint solver for *logic goals* (Line 5) based on a logic language (to be discussed later).

The problem to solve is given by a set of constraints (Lines 7-12). The domain constraint in Line 7 and the **alldifferent**-constraint in Line 8 are handled by the finite domain solver. The equation in Line 9 is handled by the linear arithmetic solver, and Lines 10-12 describe three goals for the logic language solver. The **word** predicate combines a sequence of digits to a number.

Language Integration In [2, 4] we show how to integrate declarative languages into Meta-S by treating the language evaluation mechanism as a constraint solver. This easily enables to integrate multi-domain constraints into a language.

Integrating a logic language into Meta-S yields *constraint logic programming*. Logic goals are evaluated by the constraint solver *cll* based on resolution.

¹ As is well known the problem can be solved by a usual finite domain solver with arithmetics. Nevertheless it is suitable here as a short and simple example.

1	(<- (word L X) (word L 0 X))	; word(L,X) :- word(L,0,X).
2	(<- (word [] ACC ACC))	; word([], ACC, ACC).
3	(<- (word [FT RT] ACC SUM)	; word([FT RT], ACC, SUM) :-
4	(word RT (+ (* 10 ACC) FT) SUM))	; word(RT, 10*ACC+FT, SUM).

Fig. 3. Combination of letters to words (file `smm.cll`)

1	(== (word [] ACC) ACC)	; word [] ACC = ACC
2	(== (word [FT RT] ACC)	; word [FT RT] ACC =
3	(word RT (+ (* 10 ACC) FT)))	; word RT 10*ACC+FT

Fig. 4. Functional logic combination of letters to words (file `smm.fcll`)

Consider again our example in Fig.2. Here the logic language solver was configured to read its rule definitions from the file `smm.cll` (cf. Fig.3). E.g. the goal `word([S,E,N,D],SEND)` is solved using the first rule for initializing an accumulator `ACC` with 0, calling thereafter the goal `word([S,E,N,D],0,SEND)`. The predicate `word/3` finally computes the value of the variable `SEND`. To ease the reading, the corresponding PROLOG rules are given as comments.

Of course, **Meta-S** allows the integration of other declarative languages as well. Consider for example the functional logic language solver *fell*. Its integration into **Meta-S** yields *constraint functional logic programming*. Our *fell* solver is based on narrowing using functional logic rules. Again the program must be given by the user in an extra file. E.g. Fig.4 shows the definition of a function `word/2` (equivalent to the predicate `word/3` in Fig.3). For better understanding the rules are given as comments in a HASKELL-like syntax. If we would have used the *fell* solver to describe the Send-More-Money problem, the corresponding constraints in the problem specification in Fig.2 would have been equality constraints like `(= (word [S,E,N,D]) SEND)`.

Problem descriptions may even freely mix logic predicates and functions, i.e. it is possible to integrate both the *cll* and the *fell* solvers.

The Strategy Framework **Meta-S** provides the user with strategy definition mechanisms at two levels (cf. [1]). The definition of generic strategies on the first level mainly regulates the search tree traversal. On top of this, **Meta-S'** strategy specification language allows the user to define strategy attributes which concern the solver behaviour and their cooperation, like the order of propagation and projection, priorities of solvers within a cooperation or the usage of particular heuristics. Within the scope of the description of the solver behaviour, the user is able to specify and refine evaluation strategies for the language solvers, e.g. narrowing strategies. In [2] we compare variations of the classical evaluation strategies for logic languages (including residuation) as well as new advanced strategies in this context. **Meta-S** already comes with a collection of predefined search and cooperation strategies, including typical ones like depth-/breadth-first-search, eager and lazy narrowing and standard solver cooperation schemes.

```

1 (define-strategy eager-propagation-reordering (eager-strategy)
2   (:step (select ((eq-constr (= t t))
3                 (in-constr (in t t))
4                 (rest t))
5             (tell-all in-constr) (tell-all eq-constr) (tell-all rest)
6             (project-one linear-solver)
7             (tell-all) (project-all))))

```

Fig. 5. Reordering constraints for propagation

In the Send-More-Money example in Line 2 we state the usage of the strategy **eager-propagation-reordering**. Its definition in Fig. 5 illustrates the usage of the strategy definition language. This strategy is a refinement of the generic strategy **eager** (Line 1), which has a depth-first-search behaviour. The idea for the refinement is to classify the constraints according to their propagation cost (Lines 2-4) and to redefine the order of propagations and projections accordingly (Lines 5-7). This allows a distinct reduction of computation times (cf. [1]).

3 Conclusion

Besides solver cooperation systems with fixed solvers and fixed cooperation strategies, like [7], there are systems which allow a more flexible strategy handling, e.g. [5], up to the definition of problem specific solver cooperation strategies and the integration of new solvers, like the approach in [6] or **Meta-S**. Our system differs from others by its flexible coordination mechanism that is easily extensible by further solvers and makes it possible for users to adopt the solver cooperation strategy to the needs of the problem at hand. **Meta-S** furthermore allows the integration of declarative languages and multi-domain constraints and the definition and usage of new language evaluation strategies in an easy way.

References

1. St. Frank, P. Hofstedt, and P.R. Mai. **Meta-S: A Strategy-oriented Meta-Solver Framework**. In *Proc. of the 16th FLAIRS Conference*. The AAAI Press, 2003.
2. St. Frank, P. Hofstedt, and D. Reckmann. **Strategies for the Efficient Solution of Hybrid Constraint Logic Programs**. In *MultiCPL Workshop*, Saint-Malo, 2004.
3. P. Hofstedt. **Cooperating Constraint Solvers**. In *Sixth Conference on Principles and Practice of Constraint Programming - CP*, volume 1894 of *LNCS*. Springer, 2000.
4. P. Hofstedt. **A general Approach for Building Constraint Languages**. In *Advances in Artificial Intelligence*, volume 2557 of *LNCS*, pages 431–442. Springer, 2002.
5. E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy. INRIA, 1996.
6. B. Pajot and E. Monfroy. **Separating search and strategy in solver cooperations**. In *Perspectives of Systems Informatics – PSI*, volume 2890 of *LNCS*. Springer, 2003.
7. M. Rueher. **An Architecture for Cooperating Constraint Solvers on Reals**. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer, 1995.

Cmodels for Tight Disjunctive Logic Programs^{*}

Yuliya Lierler

AI, Erlangen-Nürnberg Universität,
yuliya.lierler@informatik.uni-erlangen.de

1 Introduction

Disjunctive logic programming under the stable model semantics [GL91] is a new *answer set programming* (ASP) methodology for solving combinatorial search problems. It is a form of declarative programming related to logic programming languages, such as Prolog, where the solutions to a problem are represented by answer sets, and not by answer substitutions produced in response to a query as in conventional logic programming. Instead of Prolog systems, this programming method uses answer set solvers, such as Smodels¹, Smodelscc², Cmodels³, DLV⁴, and GNT¹. These systems made it possible for ASP to be successfully applied in such areas as planning, bounded model checking, and space shuttle control. DLV and GNT are more general as they work with the class of disjunctive logic programs, while other systems cover nondisjunctive programs. System Cmodels uses SAT solvers as search engines, which allows it to take advantage of rapid progress in the area of SAT. Cmodels proved to be an efficient system in providing the solution to the wire-routing problem [EW04], and to the problem of reconstructing probable phylogenies in the area of historical linguistics [BEMR05]. In this work we extend Cmodels [GLM04] to tight disjunctive programs. Complexity of finding a solution for such programs is NP, as in the case of nondisjunctive programs. Extending the syntax of the input language of Cmodels to tight disjunctive programs permits the knowledge engineer to be more flexible with the encoding of the problems in the NP complexity class. Experimental analyses demonstrate that the approach is computationally promising and may advance applications of disjunctive logic programming.

2 Theory, Implementation, Usage, Experiments

We base our work on the relationship between the completion [Cla78] and answer set semantics for logic programs. For the large class of *tight* programs the answer

^{*} I would like to thank V. Lifschitz for many valuable suggestions for the format of this paper and E. Giunchiglia, G. Görz, J. Lee, and M. Maratea for the comments related to the subject.

¹ <http://www.tcs.hut.fi/Software/> .

² <http://www.ececs.uc.edu/~schlipf/> .

³ <http://www.cs.utexas.edu/users/tag/cmodels> .

⁴ <http://www.dbai.tuwien.ac.at/proj/dlv/> .

```

% Sample graph encoding, i.e. graph contains 3 nodes, and 3 edges:
% edges between nodes 1 and 2, 2 and 3, 3 and 1.
node(1..3). edge(1,2).edge(2,3).edge(3,1).
% Declaration of three colors
col(red). col(green). col(blue).
% Disjunctive rule: stating that node has some color
colored(X,red) | colored(X,green) | colored(X,blue) :- node(X).
% Neighboring nodes should not have the same color
:- edge(X,Y), colored(X,C), colored(Y,C), col(C).

```

Fig. 1. Encoding of tight 3-colorability problem for grounder LPARSE: *3-col.lp*

sets of the program are the same as the models of its completion, and hence SAT solvers can play the role of answer set enumerators. [LL03] introduced the notion of completion and tightness for disjunctive programs. A *disjunctive program* Π is a set of disjunctive rules of the form $A \leftarrow B, F$ where A is the head of the rule, and is either a disjunction of atoms or symbol \perp , B is a conjunction of atoms, and F is a formula of the form $\text{not } a_1, \dots, \text{not } a_m$. We identify the disjunction of atoms A with the set of the atoms occurring in A . The completion of Π [LL03] is the set of propositional formulas that consists of the implication $B \wedge F \supset A$ for every rule in Π , and the implication $a \supset \bigvee_{A \leftarrow B, F \in \Pi; a \in A} (B \wedge F \wedge \bigwedge_{p \in A \setminus \{a\}} \neg p)$ for each atom $a \in \Pi$. The *positive dependency graph* of Π is directed graph G such that the vertices of G are the atoms occurring in Π , and for every rule in Π , G has an edge from each atom in A to each atom in B . Program Π is *tight* if its positive dependency graph is acyclic.

Theorem. [LL03] *For any tight disjunctive program Π and any set X of atoms, X is an answer set for Π iff X satisfies the completion of Π .*

Figure 1 presents the tight disjunctive program *3-col.lp* based on the encoding of 3-colorability problem provided at the DLV web page.

We based our implementation on systems LPARSE *--dlp* and CMODELS. LPARSE *--dlp* takes a disjunctive logic program with variables as an input and grounds the program. In order to use CMODELS for solving disjunctive programs flag *-dlp* should be used. In the process of its operation, CMODELS *-dlp* first verifies that the program is tight, by building the positive dependency graph and using a depth first search algorithm to detect a cycle in it. This step may be omitted from the execution sequence using flag *-t*. Second, CMODELS *-dlp* forms the program's completion, and last it calls a SAT solver to find the models of the completion. Flags *number*, *-si*, *-rs*, *-mc* are available, where *number* is an integer that stands for a number of solutions to find (0 stands for all solutions, 1 is a default), and *-mc* (*default*), *-si*, *-rs* specify that SAT solver CHAFF, SIMO, RELSAT, respectively, is invoked during the search. For example, command line

```
lparse --dlp 3-col.lp | cmodels -dlp
```

produces one answer set for the program in Figure 1:

Answer set: colored(1,red) colored(2,green) colored(3,blue)

It is worth noticing that *3-col.lp* program is syntactically identical to the 3-colorability program with choice rules supported by systems SMOELS, SMOELSc and CMOELS. The disjunctive rule of program *3-col.lp* is interpreted as the choice rule by these systems. Semantically, the rules are nevertheless different. The choice rule encodes the exclusive disjunction in the head of this rule. In case of 3-colorability problem this is acceptable interpretation of a rule and this allows us to find answer sets of the program also by means of nondisjunctive answer set programming.

For experimental analyses we used the encoding of the 3-colorability problem as in Figure 1. We compared the performance of CMOELS *-dlp* with systems DLV, GNT on disjunctive program and also SMOELS, SMOELSc and CMOELS on choice rule encoding of a problem. All experiments were run on Pentium 4, CPU 3.00GHz and presented in Figures 2 and 3.

In Figure 2 we show the results of running CMOELS with SIMO and DLV on the disjunctive programs, and SMOELS on the corresponding program with choice rules. The instances of Ladder graphs presented in the table were taken from the DLV web page. Columns LPARSE *-dlp*, CMOELS, DLV and SMOELS present the running times of the systems in seconds. DLV running time also includes the time spent by the system on grounding the program. We can see that CMOELS outperforms two other systems by more than an order of magnitude.

In Figure 3 we present the experiments with harder instances of the graphs. Letters L, S in the names of the graph instances stand for ladder and simplex, while the number stands for the number of nodes divided by 1000 in the ladder graphs, and the number of levels in the simplex graphs. SMOELS, DLV and GNT were not able to terminate on our test programs within the 30 minutes cutoff time. Columns LPARSE, CMOELS *simo*, CMOELS *chaff*, and SMOELSc present the running times of the systems in seconds. The numbers before and after "\ " stand for invocation of LPARSE *--dlp*, CMOELS *-dlp* on disjunctive programs, and LPARSE, CMOELS on corresponding programs with choice rules, respectively. The second and the third columns demonstrate that the ground disjunctive program is smaller than the corresponding ground program with choice rules: LPARSE encodes disjunctive rules more economically than choice rules. CMOELS *-dlp*, in its turn, takes an advantage of a smaller ground program and produces fewer clauses. For example, the performance of CMOELS *-dlp simo* on the disjunctive program saves 18 to 29% of running time in comparison with CMOELS *simo* performance on the program with choice rules. CMOELS *-dlp simo* also outperforms SMOELSc on ladder graphs by an order of magnitude. In case of simplex graph instances CMOELS *-dlp chaff* also outperforms SMOELSc. Capability of using different search engines may prove to be useful in practical applications.

3 Conclusions and Future Work

The evaluation of CMOELS *-dlp* shows that it is a promising approach that might advance the use of the disjunctive answer set programming paradigm in

# of nodes	LPARSE <i>-dlp</i>	CMODELS <i>simo</i>	SMODELS	DLV
1080	0.06	0.10	2.42	4.00
1320	0.07	0.12	3.74	6.00
1680	0.10	0.17	5.94	9.53
1920	0.12	0.19	7.91	12.46
2400	0.14	0.25	11.97	19.98

Fig. 2. 3-colorability problem on Ladder graphs: CMODELS *-dlp* with SAT solver SIMO on disjunctive program vs. SMODELS on choice rule program and DLV

Pr.	LPARSE disj\ch	# rules *10 ⁻⁴ disj\ch	# clauses *10 ⁻⁴ disj\ch	CMODELS <i>simo</i> disj\ch	CMODELS <i>chaff</i> disjunctive	SMODELScc choice
L32	2\3	25\38	44\51	10\15	27	105
L64	4\6	51\76	89\102	36\51	157	417
L120	8\11	96\144	168\192	122\165	727	1460
L240	16\24	192\288	336\384	496\701	-	-
S480	14\17	161\207	230\253	174\213	20	26
S600	21\27	251\323	359\395	378\490	35	46

Fig. 3. 3-colorability problem on large Ladder and Simplex graphs: CMODELS *-dlp* with SAT solvers SIMO and CHAFF on disjunctive programs vs. CMODELS with SIMO and SMODELScc on choice rule programs

practice. [LZ02] provided the theoretical base for using SAT solvers for computing answer sets for nontight nondisjunctive programs. Systems ASSAT [LZ02] and CMODELS [GLM04] are the implementations that demonstrated promising experimental results. [LL03] extended the theory used by the approach to the case of nontight disjunctive programs. Future work is to add the capability to CMODELS to find answer sets for nontight disjunctive programs.

References

- [BEMR05] D. R. Brooks, E. Erdem, J. W. Minett, and D. Ringe. Character-based cladistics and answer set programming. In *Proc. PADL'05*, pages 37–51, 2005.
- [Cla78] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
- [EW04] E. Erdem and M.D.F. Wong. Rectilinear steiner tree construction using answer set programming. In *Proc. ICLP'04*, pages 386–399, 2004.
- [GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
- [GLM04] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Sat-based answer set programming. In *Proc. AAAI-04*, pages 61–66, 2004.
- [LL03] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. ICLP-03*, pages 451–465, 2003.
- [LZ02] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI-02*, pages 112–117, 2002.