Dietmar Seipel, Michael Hanus,
Armin Wolf, Joachim Baumeister (Eds.)

17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)

Würzburg, Germany, October 4–6, 2007
Proceedings

Technical Report 434, October 2007
Bayerische Julius–Maximilians–Universität Würzburg
Institut für Informatik
Am Hubland, 97074 Würzburg, Germany

Preface

This volume contains the papers presented at the *17th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP 2007) and the *21st Workshop on (Constraint) Logic Programming* (WLP 2007), which were held jointly in Würzburg, Germany, on October 4–6, 2007.

Declarative Programming is an advanced paradigm for the modeling and solving of complex problems. This specification method has got more and more attraction over the last years, e.g., in the domains of databases and the processing of natural language, for the modeling and processing of combinatorial problems, and for establishing systems for the Web.

INAP is a communicative conference for intensive discussion of applications of important technologies around Prolog, Logic Programming, Constraint Problem Solving and closely related advanced software. It comprehensively covers the impact of programmable logic solvers in the Internet Society, its underlying technologies, and leading edge applications in industry, commerce, government, and societal services.

The papers of the conference covered the topics described above, especially, but not excluding, different aspects of Declarative Programming, Constraint Processing and Knowledge Management as well as their use for Distributed Systems and the Web:

- Knowledge Management: Relational/Deductive Databases, Data Mining, Decision Support, Xml Databases
- Distributed Systems and the Web: Agents and Concurrent Engineering, Semantic Web
- Constraints: Constraint Systems, Extensions of Constraint Logic Programming
- Theoretical Foundations: Deductive Databases, Nonmonotonic Reasoning, Extensions of Logic Programming
- Systems and Tools for Academic and Industrial Use

The *WLP workshops* are the annual meeting of the Society for Logic Programming (GLP e.V.). They bring 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.

Contributions were solicited on theoretical, experimental, and application aspects of Constraint Programming and Logic Programming, including, but not limited to (the order does not reflect priorities):

- Foundations of Constraint/Logic Programming
- Constraint Solving and Optimization
- Extensions: Functional Logic Programming, Objects
- Deductive Databases, Data Mining
- Nonmonotonic Reasoning, Answer–Set Programming
- Dynamics, Updates, States, Transactions
- Interaction of Constraint/Logic Programming with other Formalisms like Agents, XML, JAVA
- Program Analysis, Program Transformation, Program Verification, Meta Programming
- Parallelism and Concurrency
- Implementation Techniques
- Software Techniques (e.g., Types, Modularity, Design Patterns)
- Applications (e.g., in Production, Environment, Education, Internet)
- Constraint/Logic Programming for Semantic Web Systems and Applications
- Reasoning on the Semantic Web
- Data Modelling for the Web, Semistructured Data, and Web Query Languages

In the year 2007 the two conferences were organized together by the Institute for Computer Science of the University of Würzburg and the Society for Logic Programming (GLP e.V.). We would like to thank all authors who submitted papers and all workshop participants for the fruitful discussions. We are grateful to the members of the programme committee and the external referees for their timely expertise in carefully reviewing the papers. We would like to express our thanks to Petra Braun for helping with the local organization and to the Department for Biology for hosting the conference.

October 2007

Dietmar Seipel, **Michael Hanus**,
Armin Wolf, **Joachim Baumeister**

Program Chair

Dietmar Seipel	University of Würzburg, Germany

Organization

Dietmar Seipel	University of Würzburg, Germany
Joachim Baumeister	University of Würzburg, Germany

Program Commitee of INAP

Dietmar Seipel	University of Würzburg, Germany (Chair)
Sergio A. Alvarez	Boston College, USA
Oskar Bartenstein	IF Computer Japan, Japan
Joachim Baumeister	University of Würzburg, Germany
Henning Christiansen	Roskilde University, Denmark
Ulrich Geske	University of Potsdam, Germany
Parke Godfrey	York University, Canada
Petra Hofstedt	Technical University of Berlin, Germany
Thomas Kleemann	University of Würzburg, Germany
Ilkka Niemelä	Helsinki University of Technology, Finland
David Pearce	Universidad Rey Juan Carlos, Madrid, Spain
Carolina Ruiz	Worcester Polytechnic Institute, USA
Osamu Takata	Kyushu Institute of Technology, Japan
Hans Tompits	Vienna University of Technology, Austria
Masanobu Umeda	Kyushu Institute of Technology, Japan
Armin Wolf	Fraunhofer FIRST, Germany
Osamu Yoshie	Waseda University, Japan

Program Commitee of WLP

Michael Hanus	Christian–Albrechts–University Kiel, Germany (Chair)
Slim Abdennadher	German University Cairo, Egypt
Christoph Beierle	Fern–University Hagen, Germany
Jürgen Dix	Technical University of Clausthal, Germany
Thomas Eiter	Technical University of Vienna, Austria
Tim Furche	University of München, Germany
Ulrich Geske	University of Potsdam, Germany
Petra Hofstedt	Technical University of Berlin, Germany
Sebastian Schaffert	Salzburg Research, Austria
Torsten Schaub	University of Potsdam, Germany
Sibylle Schwarz	University of Halle, Germany
Dietmar Seipel	University of Würzburg, Germany
Michael Thielscher	Technical University of Dresden, Germany
Hans Tompits	Vienna University of Technology, Austria
Armin Wolf	Fraunhofer FIRST, Berlin, Germany

External Referees for INAP and WLP

Martin Atzmueller	Andreas Böhm
Steve Dworschak	Stephan Frank
Martin Gebser	Martin Grabmüller
Matthias Hintzmann	Matthias Hoche
Marbod Hopfner	Dirk Kleeblatt
Benedikt Linse	André Metzner
Sven Thiele	Johannes Waldmann

Table of Contents

Invited Talk

Constraints

Databases and Data Mining

Logic Languages

System Demonstrations

Invited Talk

A guide for manual construction of difference-list procedures

Ulrich Geske
University of Potsdam
ugeske@uni-potsdam.de

Hans-Joachim Goltz
Fraunhofer FIRST, Berlin
goltz@first.fraunhofer.de

Abstract. Difference-list technique is an effective method for extending lists to the right without using the append/3-procedure. There exist some proposals for an automatic transformation of list-programs into difference-list programs. But, we are interested in a construction of difference-list programs by the programmer, avoiding the need of a transformation. In [Ge06] it was demonstrate, how left-recursive procedures with a dangling call of append/3 can be transformed into right-recursion using the unfolding technique. For some types of right-recursive procedures the equivalence of the accumulator technique and difference-list technique was shown and rules for writing corresponding difference-list programs were given. In the present paper an improved and simplified rule is derived which substitutes the formerly given ones. We can show that this rule allows to write difference-list programs which supply result-lists which are either constructed in top-down-manner (elements in append-order) or in bottom-up manner (elements in inverse order) in a simple schematic way.

Keywords: Difference-lists, Logic Programming, Teaching

1. What are difference-lists?

Lists are, on the one hand, a useful modelling construct in logic programming because of their not predefined length, on the other hand the concatenation of lists by `append(L1,L2,L3)` is rather inefficient because it copies the list `L1`. To avoid the invocation of the `append/3`-procedure an alternative possibility is the use of incomplete lists of the form `[el1, ... elk | Var]` in which the variable `Var` describes the tail of the list not yet specified completely. If there is an assignment of a concrete list for the variable `Var` in the program, it will results an efficient (physical) concatenation of the first list elements $el_1,...el_k$ of L and `Var` without copying these first list elements. This physical concatenation does not consist in an extra-logically replacing of a pointer (a memory address) but is a purely logical operation since the reference to the list `L` and its tail Var was already created by their specifications in the program.

From the mathematical point of view, the difference of the two lists $[el_1, ...,el_k$ `| Var]` and `Var` denote the initial piece $[el_1, ..., el_k]$ of the complete list. E.g., the difference `[1,2,3]` arises from the lists `[1,2,3|X ]` and `X` or arises from `[1,2,3,4,5]` and `[4,5]` or may arise `from [1,2,3,a]` and `[a]`. The first-mentioned possibility is the most general representation of the list difference `[1, 2, 3]`. Every list may be presented as a difference-list. The empty list can be expressed as a difference of the two lists `L` and `L`, the list `List` is the difference of the list `List` and the empty list (`[]`).

The combination of the two list components $[el_1, ...,el_k$ `| Var]` and `Var` in a structure with the semantics of the list difference will be denoted as a "difference-list". Since such a combination which is based on the possibility of specifying incomplete lists, is always possible, the Prolog standard does not provide any special notation for this combination. A specification of a difference-list from the two lists `L` and `R` may be given by a list notation `[L, R]` or by the use of a separator, e.g. `L-R` or `L\R` (the used separator must be defined in the concrete Prolog system) or as two separate arguments separated by a comma in the argument list.

In practical programming, the concatenation of lists is very frequently expressed using an invocation of the `append/3`-procedure. The reason for it may be the inadequate explanations of the use of incomplete lists and of difference-list technique which uses such incomplete lists. Very different explanations of difference-lists and very different attitudes to them can be found in well-known manuals and textbooks for Prolog.

Difference-lists in the literature

Definition of difference-lists

The earliest extended description of difference-lists was given by Clark and Tärnlund in [CT77]. They illustrated the difference-list or d-list notation by the graphical visualization of lists as it was used in the programming language LISP. One of their examples is given in Fig. 1. In this example the list `[a, b, c, d]` can be considered as the value of the pointer `U`. A sublist of `U`, the list `[c, d]`, could be represented as the value of the pointer `W`. The list `[a, b]` between both pointers can be considered as the difference of both lists. In [CT77] this difference is denoted by the term `<U, W>` which is formed of both pointers. Other authors denote it by a term `U\W` [StSh91]. We prefer to use the representation `U-W` in programs like it is done in [OK90].

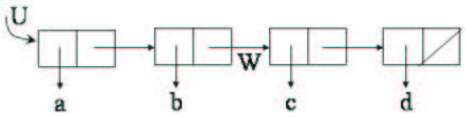


Fig. 1 Visualization of the difference-list <U, W> = [a, b | W] - W = [a, b]

d-list [CT77]

d-list(<U, W>) ↔ U = W
∨ ∃X∃V{U = [X | V] & element(X) & d-list(V, W)}

The pair `<U, W>` represents the difference of the lists `U` and `W` and is therefore called a difference-list (d-list or dl for short). The difference includes all the elements of the list that begins at the pointer `U` and end with the pointer `W` to a list or to `nil`.

In a LISP-like representation the value of the difference of list `U` and `W` can be constructed by substitution of the pointer to `W` in `U` by the empty list. Let us introduce the notation `Val(<U, W>)` for the term which represents a difference-list `<U, W>` - the value of a difference-list `<U, W>`. Of course, the unification procedure in Prolog does not perform the computation of the value like it does not perform arithmetic computations in general.
Clark and Tärnlund have given and proven the definition of construction and definition of the concatenation of difference-lists:

Concatenation of d-lists [CT77]

d-list(<U, W>) ↔∃V{d-list(<U, V>) & d-list(V, W)}

If <[b, c, d | Z], Z> is a difference-list with V=[b, c, d | Z] and W=Z then the difference-list <U, W>= <[a, b, c, d | Z], Z> can be constructed where are X=a and U=[a, b, c, d | Z].

Concatenation of the difference-lists `<U,V>=<[a,b|X],X>` and `<V,W>=<X,nil>` results in the difference-list `<U,W>=<[a,b,c,d],nil>` as soon as `X` is computed to `[c, d]`. The concept of concatenation of difference-lists will strongly used for our rules for programming d-list programs.

Automatic transformation of list-programs

Several authors investigated automatic transformations of list programs into difference-list programs [MS88, ZG88, MS93, AFSV00]. Mariott and Sondergaard [MS93] have deeply analysed the problem and proposed an stepwise algorithm for such a transformation. The problem to solve is to ensure such a transformation is safe. Table 1 shows that semantics of programs may different changing programs from list notation to difference-list

notation. Essentially the missing occur-check in Prolog systems is due to the different behaviour of programs in list and d-list notation. But Table 1 also shows that in case the occur-check would be available different semantics of list-procedures and the corresponding d-list-procedures may occur.
Looking for conditions for a safe automatic generation of difference-lists, Marriot and Søndergaard [MS93] start with the consideration of difference terms.

> Difference term [MS93]
> A difference term is a term of the form <T, T'>, where T and T' are terms. The term T' is the delimiter of the difference term. The denotation of <T, T'> is the term that results when every occurrence of T' in T is replaced by nil.

Fig. 2 represents some examples of difference-lists given in [MS93]. If both U and W point to the list [a], the difference-list <U,W> represents the empty list, i.e. Val(<[a],[a]>)=[]. If U points to a list T and W points to nil, the tail of T, then <U,W> represents T (the pointer to W in T is substituted by nil), i.e. Val(<T,nil>)=T. The difference-list <T,X> is a generalization of the difference-list <[a],[b]> if X is not part of (does not occur in) T. In such cases where the second part of the difference-list does not occur in the first part, there is nothing to change in the first part of the difference-list (no pointer is to substitute by nil) and the first part of the difference-list is the value, i.e. Val(<[a],[b]>)=[a] and Val(<T,X>)=T. In cases where W does not point to a list but to an element of a list, a similar substitution action can be performed to get the value of the difference-list. In <U,W>=<[a],a>, W points to the list element a. If one substitutes the pointer to a by nil one gets Val(<[a],a>=[nil]. A similar, but more complex example is given by the difference-list <g(X,X),X>. All pointers to X has to be substituted by nil which results altogether in g(nil,nil).

Table 1 Semantics of a program in list and difference-list representation

	Intended meaning	Prolog without occur-check	Prolog with occur-check
list-notation	?- quick([], [a]). > no ?- quick([],[a|Z]). > no	quick([],[]). ?- quick([], [a]). > no ?- quick([],[a|Z]). > no	quick([],[]). ?- quick([], [a]). > no ?- quick([],[a|Z]). > no
dl-notation	quick([],X-X). ?-quick([],[a|Z]-Z). > no ?- quick([],[a|Z]-Y). > no	quick([],X-X). ?-quick([],[a|Z]-Z). > Z=[a,a,a…] yes ?- quick([],[a|Z]-Y). > Y=[a|X] yes	quick([],X-X). ?-quick([],[a|Z]-Z). > no ?- quick([],[a|Z]-Y). > Y=[a|X] yes

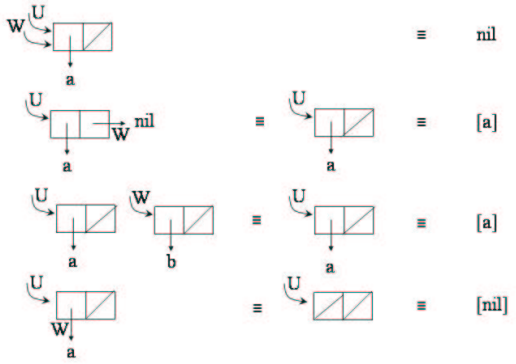


Fig. 2 Difference trees (visualization of examples of [StSh91])
(nil, /, and [] are synonyms for the empty list)

In the context of the transformation of list-procedures into corresponding d-list-procedures, two difference-terms should unify if the corresponding terms unify. This property is valid for simple difference-terms.

> Simple difference term [MS93]
> A difference term is simple iff it is of the form <T,nil> or of the form <T,X> where X represents a variable.
>
> Difference-list [MS93]
> A difference-list is a difference term that represents a list. It is closed iff the list that it represents is closed (that is, the difference-term is of the form <[T1, …, Tn], Tn>. The difference-list is simple, if Tn is nil or a variable.

The definition of difference-lists allows that W could be a pointer not only to a list but to any term. Therefore, each closed list can be represented as a difference-list, but not every difference-list is the representation of a closed list. The automatic transformation of lists to difference-list is safe if the generated difference-list is simple and T, Tn in <T, Tn> have non-interfering variables.

Table 2 should remember that the problem of different semantics of programs is not a special problem for difference-list. Also in pure list processing the programmer has to be aware that bindings of variables to themself may occur during unification and may give rise to different behaviour depended from the programming system.

Table 2 Different semantics of programs

Intended meaning	Prolog without occur-check	Prolog with occur-check

`append([], X, X).`
 	
`?- append([],[a|Z],Z).`
`> no`
	`append([], X, X).`
	
	`?- append([], [a|Z],Z).`
	`> Z=[a,a,a,…]`
	`yes`
		`append([], X, X).`
		
		`?- append([], [a|Z],Z).`
		`> no`

The authors investigate in [MS93] conditions for a save transformation of list-programs into d-list programs and present a algorithm for the transformation.

Difference-lists in Prolog textbooks
Clocksin has described the difference-list technique in a tutorial [Clock02] based on his book "Clause and Effect" [Clock97] as "*probably one of the most ingenious programming techniques ever invented yet neglected by mainstream computer science*". In the contrary, Dodds opinion [Dodd90] concerning a pair of variables *(All, Temp)* for the result *All* of a list operation and the used accumulator *Acc* is less emphatic: "*some people find it helpful to think of* All *and* Temp *as a special data structure called difference-list. We prefer to rest our account of the matter on the logical reading of the variable as referring to any possible tail, rather than to speak of 'lists with variable tails'*".

In the classical Prolog textbook of **Clocksin and Mellish** [CM84] a certain relationship between accumulator-technique and difference-list technique is stated: "with difference-list we also use two arguments (comment: as for the use of accumulators), but with a different interpretation. The first is the *final result*, *and the second argument is for a hole in the final result where further information* can be put".

O'Keefe [1] uses the same analogy: **"**A common thing to do in Prolog is to carry around a partial data structure and some of the holes in it. To extend the data structure, you fill in a hole with a new term which has some holes of its own. Difference-list are a special case of this technique where the two arguments we are passing around the positions a list**."** The represented program code for a subset-relation corresponds in its structure to the bottom-up construction of lists. This structure is identified as "difference-list" notation.

Clark&McCabe [CMcC84] denote a difference-list as difference pair. It is used as substitute for an append/3-call. The effects are described by means of the procedure for the list reversal. If the list to be reversed is described by [X|Y], the following logical reading exists for the recursive clause: „[X|Y] has a reverse represented

by a difference pair [Z, Z1] if Y has a reverse represented by a difference pair [Z, [X|Z1]] ". ...suppose that [Z, [X|Z1]] is [[3,2,1,0], [1,0]] representing the list [3,2], ...then [Z, Z1] of is [[3,2,1,0], [0]] representing the list [3,2,1]". ... you will always get back a most general difference pair representation of the reverse of the list". Besides this example no further guideline for programming with difference-lists is given.

A detailed presentation of the difference-list technique is given by **Sterling and Shapiro** [StSh93]. They stress the tight relationship of accumulator technique and difference-list technique: „...Difference-lists generalize the concept of accumulators" and „... Programs using difference-lists are sometimes structurally similar to programs written using accumulators". A comparison of a flatten_dl/2 procedure, which uses difference-lists and a flatten/3 procedure, that uses the accumulator technique is given (deviating of the representation in [StSh91], the accumulator is not the second but the third argument in Fig. 3, for better comparison reasons)

Flattening of lists using difference-lists	Flattening of lists using accumulators

`flatten_dl([X|Xs], Ys-Zs) :-`
`  flatten_dl(X,Ys-Ys1),`
`  flatten_dl(Xs,Ys1-Zs).`
`flatten_dl(X,[X|Xs]-Xs) :-`
`  atom(X),not X=[].`
`flatten_dl([],Xs-Xs).`
	`flatten([X|Xs],Ys, Zs) :-`
	`  flatten(Xs,Ys1, Zs),`
	`  flatten(X, Ys, Ys1).`
	`flatten(X, [X|Xs], Xs) :-`
	`  atom(X),not X=[].`
	`flatten([],Xs, Xs).`

Fig. 3 Comparison of list flattening in difference-list and accumulator technique

Unfortunately the conclusion on this comparison is no construction rule for the use of difference-lists, but the (partly wrong) remark that the difference between both methods "… is the goal order in the recursive clause of flatten". In fact, the two recursive flatten/3 calls in the flatten/3 procedure can be exchanged without changing result. Instead of trying to formalize the technique it is „mystified": „Declaratively the (comment: flatten_dl/2)... is straightforward. ... The operational behaviour... is harder to understand. The flattened list seems to be built by magic. ... The discrepancy between clear declarative understanding and difficult procedural understanding stems from the power of the logical variable. We can specify logical relationships implicitly, and leave their enforcement to Prolog. Here the concatenation of the difference-lists has been expressed implicitly, and is mysterious when it happens in the program." ([StSh91], S. 242). At the other hand, the authors mention, that using a difference-list form of the call to append/3 and a unfolding operation in respect to the definition of append/3 an append/3-free (difference-list) program can be derived.

Colho&Cotta [CC88] compare two versions for sorting a list using the quicksort algorithm (Fig. 4). They They attempt, as in [StSh91], to explain the difference between both programs with the help of the difference-list interpretation of append/3: the complete difference S-X results from the concatenation of the difference S-Y and the difference Y-X.

Quicksort [CC88]	
Definition with append/3	Definition with difference-lists

`q_sort1([H|T], S) :-`
`  split(H,T,U1,U2),`
`  q_sort1(U1,V1),`
`  q_sort1(U2,V2),`
`  append(V1, [H|V2], S).`
`q_sort1([],[]).`
	`q_sort2([H|T], S-X) :-`
	`  split(H,T,U1,U2),`
	`  q_sort2(U1,S-[H|Y]),`
	`  q_sort2(U2,Y-X).`
	
	`q_sort2([],X-X).`

Fig. 4 Comparsion of algorithms for quicksort [CC88]

Maier&Warren [MW88] mention the effect of avoidance of append/3 calls using difference-lists. It is pointed out that the property of difference-list is to decompose lists in one step in a head and a tail, each of arbitrary length. This property is useful in natural language processing, where phrases consists of one or more words. In order to arrange this processing append/3-free, word information, for example the information that „small" is an adjective, are represented in the difference-list format: adjective([[small|Rest]-Rest]) instead of using the list format adjective([small]).

Dodd [Dodd90] explains how left-recursive programs, in particular such that use as last call append/3, are transformed to more efficient programs using accumulator technique. For this purpose he develops a formalism on the basis of functional compositions which generates a right-recursive procedure from a left-recursive procedure in six transformation steps. The effects that are achieved by the accumulator technique are shown by a

great number of problems and the relationship to the difference-list technique is mentioned and but there is no systematised approach for the use of accumulators or difference-lists.

The different techniques which are possible for the generation of lists can be reduced to two methods which are essential for list processing: top-down- and bottom-up generation of structures. In [Ge06] we have shown that the difference-list notation could be considered as syntactic sugar since it can be derived from procedures programmed with the accumulator technique by a simple syntactic transformation. But, difference-lists are not only syntactic sugar, since a simple modelling technique using difference-lists instead of additional arguments or accumulators can offer advantages constructing result lists with certain order of elements, efficiently. Knowledge of the (simple) programming rules for difference-list procedures can make it easier to use difference-lists and promotes their use in programming.

2. Top-down and Bottom-up construction of lists

The notions of top-down and bottom-up procedures for traversing structures like trees are well established. We will use the notions top-down constructions of lists and bottom-up construction of lists in this paper which should describe the process of building lists with a certain order of elements in relation to the order in which the elements are taken from the corresponding input list.

> Top-down construction of lists
> The order el_1' – el_2' of two arbitrary elements el_1', el_2' in the constructed list corresponds to the order el_1 – el_2 in which the two elements el_1, el_2 are taken from the input term (maybe a list or a tree structure).
>
> Bottom-up construction of lists
> The order el_2' – el_1' of two arbitrary elements el_1', el_2' in the constructed list corresponds to the order el_1 – el_2 in which the two elements el_1, el_2 are taken from the input term (maybe a list or a tree structure).

An input list may be, e.g., [2, 4, 3, 1]. A top-down construction of the result list [2, 4, 3, 1] is given if the elements are taken from left to right from the input list and put into the constructed list in a left to right manner. If the elements are taken from the input list by their magnitude and put into the result list from left to right, the list [1 2 3 4] will be (top-down) constructed.
A bottom-up construction of the result list [1 3 4 2] is given if the elements are taken from left to right from the input list and put into the constructed list in a right to left manner. If the elements are taken from the input list by their magnitude and put into the result list from right to left, the list [4 3 2 1] will be (bottom-up) constructed.

	Top-Down construction of lists	Bottom-Up-construction of lists

Use of of calls of append/3	`copy_td_ap([],A,A).`
	`copy_td_ap([X|Xs],A,R):-`
	`  copy_td_ap(Xs,A,RR),`
	`  append([X],RR,R).`
	`?- copy_td_ap([2,1,3,4],[],X).`
	`X = [2, 1, 3, 4]`
		`copy_bu_ap([],A,A).`
		`copy_bu_ap([X|Xs],A,R) :-`
		`  append([X],A,NA),`
		`  copy_bu_ap(Xs,NA,R).`
		`?- copy_bu_ap([2,1,3,4],[],X).`
		`X = [4, 3, 1, 2]`

append/3-calls unfolded	`copy_td([],A,A).`
	`copy_td([X|Xs],A,[X|R]):-`
	`  copy_td(Xs,A,R).`
	`?- copy_td([2,1,3,4],[],X).`
	`X = [2, 1, 3, 4]`
		`copy_bu([],A,A).`
		`copy_bu([X|Xs],A,R):-`
		`  copy_bu(Xs,[X|A],R).`
		`?- copy_bu([2,1,3,4],[],X).`
		`X = [4, 3, 1, 2]`

Use of difference-lists	`d_copy_td([],A-A).`
	`d_copy_td([X|Xs],[X|R]-A):-`
	`  d_copy_td(Xs,R-A).`
	`?- d_copy_td([2,1,3,4],X-[]).`
	`X = [2, 1, 3, 4]`
		`d_copy_bu([],A-A).`
		`d_copy_bu([X|Xs],R-A):-`
		`  d_copy_bu(Xs,R-[X|A]).`
		`?- d_copy_bu([2,1,3,4],X-[]).`
		`X = [4, 3, 1, 2]`

Fig. 5 Some examples for top-down and bottom-up construction of lists

Fig. 5 shows some definitions for the tasks of copying lists. The procedures copy_td_ap/3 and copy_td/3 are procedures which construct the result list in a top-down manner. The list elements of the result list are supplied in the same order than elements of the input list. The procedures copy_bu_ap/3 and copy_bu/3 are procedures which construct the result list in a bottom-up manner. The elements of the result list occur in an inverse order related to the elements of the input list. The procedures copy_td/3 and copy_bu/3 result by unfolding the append/3-calls in copy_td_ap/3 and copy_bu_ap/3.

Construction in a top-down manner means, the first element chosen from the input list will be the first element of the result list, which is recursively extended to the right. Construction in a bottom-up manner means, the first element chosen from the input list will be the last element in the result list, which is recursively extended to the left. In each case, the extension of the result parameter is performed by a cons-operation [Xl...]. The question is, where to insert this cons-operation? To answer this question, different complex problems are investigated in the next sections.

3. Procedures with one recursive call - reverse of a list

Specification: The difference-list UL-A is the reverse of a list L if the tail T of the list L (without the first element H) is reversed by a recursive call of the same algorithm. The recursive call supplies a difference-list UL-[H|A], which denotation is one element smaller than the difference-list UL-A. The reverse of the empty list is the empty list, i.e. the difference-list L-L.

Fig. 6 shows the definition and some invocations of the difference-list and accumulator-version of the reverse of a list. This procedure corresponds to a bottom-up manner for construction of the result list. The elements of the result list have an inverse order compared with the input list. The cons-operation in the accumulator-version puts the first element H of the input list in front of the accumulator, the list A, of all already transferred elements. The difference-list version is a syntactic variant of the accumulator-version of the procedure.

Difference-list definition (BU) for list reversal	Accumulator-version (BU) for list reversal

`dl_rev([],IL-IL).`
`dl_rev([H|T],IL-A) :-`
`  dl_rev(T,IL-[H|A]).`
 	
`?- dl_rev([a,b,c,d],R-[]).`
`R=[d,c,b,a]`
 	
`?- dl_rev([a,b],R-[e,f]).`
`R = [b,a,e,f]`
 	
`?- dl_rev([a,b],R).`
`R = [b,a|Y]-Y`
 	
`?- dl_rev([a,b],X), X=[b|Z]-[e,f].`
`X= [b,a,e,f]-[e,f]`
`Z=[a,e,f]`
	`acc_rev([],IL,IL).`
	`acc_rev([H|T],A,IL) :-`
	`  acc_rev(T,[H|A],IL).`
	
	`Same as: ?- acc_rev([a,b,c,d],[],R).`
	`R=[d,c,b,a]`
	
	`same as: ?- acc_rev([a,b],[e,f],R).`
	`R=[b,a,e,f]`
	
	`same as:?- acc_rev([a,b],X2,X1),X=X1-X2.`
	`X=[b,a|X2] -X2`
	
	`same as:?- acc_rev([a,b],X2,X1),`
	`X1=[b|Z], X2=[e,f], X=X1-X2.`
	`X= [b,a,e,f]-[e,f]`
	`Z=[a,e,f]`

Fig. 6 Properties of the BU definition for the reverse of a list

The attempt to perform the cons-operation for the first element H not with the accumulator but with IL, the inverse list, would result in a definition dl_a/2 (Fig. 7) which is a TD-procedure. In this case, the application of the TD method is wrong in respect to the specification of list reversal because a TD construction of lists let unchanged the order of list elements. The list elements of the result of dl_a/2 have the same order as the input list – therefore it is a definition of a difference-list procedure dl_append/2 for concatenating lists.

Difference-list definition (TD) for append/2	Properties of the definition

`dl_a([],IL-IL).`
`dl_a([H|T],[H|IL]-A):-`
`  dl_a(T,IL-A).`
 	
`?- dl_a([a,b,c],X).`
`X = [a,b,c|A]-A`
	Same variable for all parts of the difference-list.
	Cons-operation for the first element of input list with output list

Fig. 7 Properties and definition of the TD definition for dl_a/2

In dl_a/2 for the concatenation of lists a dl_a/2-call (i.e. an append-like call) occurs. In order to design the definition of dl_a/2 free of calls to dl_a/2, the difference-list notation is extended onto all arguments of dl_a/2. The lists X and Y are concatenated to the list Z.

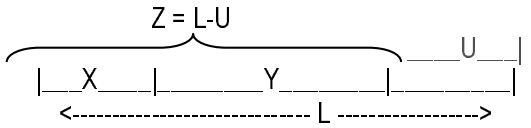


Fig. 8 Difference-list interpretation of the concatenation of two lists

List `X` can be represented as a difference-list in the form `L-Y`. The list `Y` has the difference-list format `Y-[]` or more general: `Y-U`, where `U` is supposed to be an arbitrary further list (maybe the empty list `[]`). Correspondingly the list `Z` may be represented as the difference-list `L-[]` or more general: `L-U`. In this way the procedure for the list concatenation may be represented by:

```
dl_append(L-Y, Y-U, L-U).
```

Fig. 9 Difference-list notation for the concatenation of two lists

The end condition `append([],Y,Y)` of the `append/3`-Prozedur is contained since it may be `Y=L`, so that follows:

```
?- dl_append(L-L,L-U,L-U).
yes
```

Fig. 10 Treatment of the end condition of list concatenation

From the `dl_append/3` example results the programming rule:

From the existence of the difference-lists A−B and B−C can be deduced the existence of the difference-list A−C (and vice versa).

Rule 1 Composition of difference-lists

Important is, that this assertion has a logical reading. The rule does not contain regulations for the order of the proofs (i. e. the order of procedure calls).
This assertion could be considered as the key for the wanted construction rule. We have to integrate the cons-operation [X|L], which in this format does not suit to the assertion above. E.g., in the bottom-up procedure `copy_bu/2` for copying list elements the difference-list R-A for the result is composed by the difference-list `R−[X|A]`. Let us introduce a temporary variable Z and substitute Z = [X|A]. This unification is inserted either directly in the code of the procedure or it is performed indirectly. The indirect invocation of the unification may happen in call of another procedure or in a call `cons(X,Z−A)` of the special cons/2-procedure which is defined as:

```
cons(Element,[Element|Temp]-Temp).
```

Fig. 11 Definition of the cons-operation

Doing the same in the procedure d_copy_bu/2 (Fig. 5), the reading of the resulting procedure `d_copy_bu1/2` corresponds to the above given programming rule: the result `R−A` is derived by computing `R−Z` and `Z−A`. The order of the generation of the parts of the composition is arbitrary and could be changed. The equivalent procedure `d_copy_bu2/2` avoids the loss of right recursion in `d_copy_bu1/2`. Because of the purely syntactic transformation a procedure of the form of `d_copy_bu/2` may be considered as shorthand for a procedure of the form `copy_bu2/2`.

Copying of a list with the BU difference-list method and using the cons/2-procedure	Copying of a list with the BU difference-list method, using the cons/2-procedure, and right recursion

`d_copy_bu1([],A-A).`
`d_copy_bu1([X|Xs],R-A):- % R-A`
`  d_copy_bu1(Xs,R-Z), %= R-Z`
`  cons(X,Z-A). %Z=[X|A] %+ Z-A`
	`d_copy_bu2([],A-A).`
	`d_copy_bu2([X|Xs],R-A):- % R-A`
	`  cons(X,Z-A), % Z=[X|A] %= Z-A`
	`  d_copy_bu2(Xs,R-Z). %+ R-Z`

Fig. 12 BU difference-list definition for copying a list

The rather similar argumentation applies for the interpretation of the top-down procedure `d_copy_td/2`.

Copying of a list with the TD difference-list method and explicit unification	Copying of a list with the TD difference-list method and use of the cons/2-procedure

`d_copy_td1([],AL-AL).`
`d_copy_td1([X|Xs],Z-AL):-`
`  Z=[X|RR],`
`  d_copy_td1(Xs,RR-AL).`
	`d_copy_td2([],AL-AL).`
	`d_copy_td2([X|Xs],Z-AL):- % Z-AL`
	`  cons(X,Z-RR),%≡Z=[X|RR] %= Z-RR`
	`  d_copy_td2(Xs,RR-AL). %+RR-AL`

Fig. 13 TD definition for copying a list

Using this result, the guideline for building TD- and BU difference-list procedures could be formulated as in Fig. 14. Note, that the only difference is the assignment of the parts of the resulting difference-list to the calls dealing with the elements of the input-list (third item in the table for each case). The term “append-order” denotes the result-list with the same order of the elements as in the input-list (or more detailed: the same order in which the elements are chosen from the input-list).

Order of elements in the result list	**Structure of procedures in difference-list notation**
Result with append-order of elements (TD construction of lists)	• The result parameter in the head of the clause for the base case (empty input list, empty result list) is the difference-list L-L. • The resulting difference-list L-L_n in the head of the clause for the general case could be split in difference-lists L-L_1, L_1- …- L_{n-1}, L_{n-1}-L_n, where L_1-…-L_{n-1} denotes an arbitrary set of further difference-lists. • The difference-list L-L_1 is used for dealing with the first element (or first part) of the input list, L_1-L_2 for dealing with the next element(s) of the input list, etc. • the cons/2-operation may be used for dealing with a single element of the input list (unfolding the call of cons/2 supplies a more dense procedure definition)
Result with inverse order of elements (BU construction of lists)	• The result parameter in the head of the clause for the base case(empty input list, empty result list) is a difference-list L-L • The resulting difference-list L-L_n in the head of the clause for the general case could be split in difference-lists L-L_1, L_1- …- L_{n-1}, L_{n-1}-L_n, where L_1-…-L_{n-1} denotes an arbitrary set of further difference-lists. • The difference-list $\mathbf{L_{n-1}}$-$\mathbf{L_n}$ is used for dealing with the first element (or first part) of the input list, $\mathbf{L_{n-1}}$-$\mathbf{L_n}$ for dealing with the next element(s) of the input list, etc. • the cons/2-operation may be used for dealing with a single element of the input list (unfolding the call of cons/2 supplies a more dense procedure definition)

Fig. 14 Rules for difference-list notation of procedures

4. Procedures with several recursive calls - flattening of a list

Specification of flattening lists: The resulting difference-list A-C of flattening a deeply structured list is composed from the difference-list A-B for flattening the first list element (may be deeply structured) and the difference-list B-C for flattening the tail of the list.

Comment: If the order of elements in the result list is supposed to correspond to the order of elements in the input list, the TD-rule for difference-list must be used, that is the first part L-L1 of the complete difference-list L-LN is used for the treatment of the first taken element of the input list.

Flattening of a list with the TD-difference-list method (call to cons/2)	Properties

`d_flat_td([], L-L).`
`d_flat_td([H|T],L-LN):-`
`  cons(H,L-L1),`
`  atom(H), !,`
`  d_flat_td(T, L1-LN).`
`d_flat_td([H|T],L-LN) :-`
`  d_flat_td(H, L-L1),`
`  d_flat_td(T, L1-LN).`
 	
`?-d_flat_td([a,[b],c],L-[]).`
`L = [a, b, c]`
	Base case: DL with same variables.
	Difference-list L-LN denotes the result
	Dealing with first element H (L=[H|A])
	
	Recursive call for the rest of the input list.
	L-LN results from DL's L-L1 and L1-LN.
	Processing of the first part of the input list.
	Processing of the last rest of the input list.
	
	Query with a difference-list for the result.

Fig. 15 Properties of the TD difference-list notation for list flattening

Flattening of a list with the TD-difference-list method (unfolding the call cons(H,L-L1), i.e. L=[H|L1])	Properties

`d_flat_td1([], L-L).`
`d_flat_td1([H|T],[H|L1]-LN):-`
`  atom(H), !,`
`  d_flat_td1(T, L1-LN).`
`d_flat_td1([H|T],L-LN) :-`
`  d_flat_td1(H, L-L1),`
`  d_flat_td1(T, L1-LN).`
 	
`?- d_flat_td1([a,[b],`
`  [[c]]],A-[]).`
`A = [a, b, c]`
	Base case: DL with same variables.
	L substituted by [H|L1]
	
	Recursive call for the rest of the input list.
	L-LN results from DL's L-L1 and L1-LN.
	Processing of the first part of the input list
	Processing of the last part of the input list.
	
	Query with a difference-list for the result..

Flattening of a list with the TD-difference-list method (permutation of the calls in the second clause, compared with d_flat_td1)	Properties (the same as for d_flat_td1/2)

`d_flat_td2([], L-L).`
`d_flat_td2([H|T],[H|L1]-LN):-`
`  atom(H), !,`
`  d_flat_td2(T, L1-LN).`
`d_flat_td2([H|T], L-LN):-`
`  d_flat_td2(T, L1-LN),`
`  d_flat_td2(H, L-L1).`
 	
`?- d_flat_td2([a,[b],[[c]]],E-[]).`
`E = [a, b, c]`
	Base case: DL with same variables.
	L substituted by [H|L1]
	
	Recursive call for the rest of the input list.
	L-LN results from DL's L-L1 and L1-LN.
	Processing of the last part of the input list
	Processing of the first part of the input list
	
	Query with a difference-list for the result.

Fig. 16 Exchanged order of recursive calls compared with Fig. 15

The order of the elements in the result list remains the same if the order of the recursive calls will be exchanged. This assertion is also true for the generation (with the BU-rule) of a result list with elements in inverse order compared to the input list.

Flattening of a list with the BU-difference-list method (using a call to cons/2)	Properties

`d_flat_bu([], Y-Y).`
`d_flat_bu([H|T], L-LN) :-`
`  atom(H), !,`
`  cons(H, L1-LN),`
`  d_flat_bu(T, L-L1).`
 	
`d_flat_bu([H|T], L-LN) :-`
`  d_flat_bu(H, L1-LN),`
`  d_flat_bu(T, L-L1).`
 	
`?- d_flat_bu([a,[b], [[c]]],E-[]).`
`E = [c, b, a]`
	Base case: DL with same variables.
	Difference-list L-LN denotes the result
	
	Dealing with first element H (L1=[H|LN])
	Recursive call for the rest of the input list.
	
	L-LN results from DL's L-L1 and L1-LN.
	Processing of the first part of the input list.
	Processing of the last part of the input list
	
	Query with a difference-list for the result

Fig. 17 BU difference-list definition for flattening lists

Flattening of a list with the BU-difference-list method (folding the call cons(H,L1-LN), i.e. L1=[H|LN])	Properties

`d_flat_bu1([], L-L).`
`d_flat_bu1([H|T], L-LN) :-`
`  atom(H), !,`
`  d_flat_bu1(T, L-[H|LN]).`
`d_flat_bu1([H|T], L-LN) :-`
`  d_flat_bu1(H, L1-LN),`
`  d_flat_bu1(T, L-L1).`
 	
`?- d_flat_bu1([a,[b],`
`  [[c]]],E-[]).`
`E = [c, b, a]`
	Base case: DL with same variables.
	Difference-list L-LN denotes the result
	
	L1 substituted by [H|LN]
	L-LN results from DL's L-L1 and L1-LN.
	Processing of the first part of the input list.
	Processing of the last part of the input list
	
	Query with a difference-list for the result

Fig. 18 BU difference-list definition using an unfolding operation

Flattening of a list with the BU-difference-list method (permutation of the calls in the second clause - compared with d_flat_td1)	Properties

`d_flat_bu2([], L-L).`
`d_flat_bu2([H|T], L-LN) :-`
`  atom(H), !,`
`  d_flat_bu2(T, L-[H|LN]).`
`d_flat_bu2([H|T], L-LN) :-`
`  d_flat_bu2(T, L-L1),`
`  d_flat_bu2(H, L1-LN).`
 	
`?- d_flat_bu2([a,[b],[[c]]],E-[]).`
`E = [c, b, a]`
	Base case: DL with same variables.
	Difference-list L-LN denotes the result
	
	L1 substituted by [H|LN]
	L-LN results from DL's L-L1 and L1-LN.
	Processing of the last part of the input list.
	Processing of the first part of the input list
	
	Query with a difference-list for the result

Fig. 19 Exchanged order of recursive calls compared to Fig. 18

5. Algorithmic computation of the order of elements - quicksort

The construction principles for the case, that an arbitrary element of the input list is taken as the current first or last element of the output list, are the same ones, as already mentioned for the top-down- and. bottom-up construction of lists. This is supposed to be explained at the example of the sorting of lists according to the quicksort algorithm.

Specification of quicksort: The sorted difference-list S-Acc for an input list [H|T] results from the composition of the difference-list for the sorting of all elements that are smaller or equal the current element and the sorted difference-list for all elements which are greater than the current element. The current element fits in the result list between the elements which are greater than and less than the current element.

In the top-down approach for construction of the result the complete difference-list L-LN is composed from the difference-list L1-L2 for sorting the list elements which are smaller than H, from the difference-list for the element H, and from the difference-list L2-LH for sorting the elements of the input list which are greater than H (Fig.20).

In the BU method for the difference-list construction, we can either start with the first part of the input list together with the last part of the difference-list for the result or we can start with the last part of the input list together with the initial part of the difference-list for the result. The corresponding procedures are called d_qsort_bu1/2 and d_qsort_bu3/2 (Fig. 21). The versions with the unfolded call of cons/2 are called d_qsort_bu2/2 and d_qsort_bu4/2.

Quick-Sort, TD-construction of lists	
Same order in which the elements are taken from the input list	
Use of a call to cons/2	Unfolding the call to cons/2
L-LN composed from L-L1, L1-L2, and L1-LN	L-LN composed from L1-LN and L-[H|L1]

`d_qsort_td1([],L-L).`
`d_qsort_td1([H|T],L-LN) :-`
`  split(H,T,A,B),`
`  d_qsort_td1(A,L-L1),`
`  cons(H,L1-L2),%as: L1=[H|L2]`
`  d_qsort_td1(B,L2-LN).`
 	
`?- d_qsort_td1([2,1,4,3], Res).`
`Res = [1, 2, 3, 4|LN] - LN`
	`d_qsort_td2([],L-L).`
	`d_qsort_td2([H|T],L-LN) :-`
	`  split(H,T,A,B),`
	`  d_qsort_td2(A,L-[H|L2]),`
	`  d_qsort_td2(B,L2-LN).`
	
	`?- d_qsort_td2([2,1,4,3], Res).`
	`Res = [1, 2, 3, 4|LN] - LN`

Fig. 20 TD difference-list notation for Quicksort

Quick-Sort, BU-construction of lists	
S-Acc formed from Y-Acc and S-[H|Y]	
Use of the cons/2-operation	Unfolding the call for the cons/2-operation

`d_qsort_bu1([],Y-Y).`
`d_qsort_bu1([H|T],S-Acc) :-`
`  split(H,T,A,B),`
`  d_qsort_bu1(A,Z-Acc),`
`  d_qsort_bu1(B,S-Y),`
`  cons(H,Y-Z). %as Y=[H|Z]`
 	
`?- d_qsort_bu1([2,1,4,3], Res).`
`Res = [4, 3, 2, 1|Acc] - Acc`
	`d_qsort_bu2([],Y-Y).`
	`d_qsort_bu2([H|T],S-Acc) :-`
	`  split(H,T,A,B),`
	`  d_qsort_bu2(A,Y-Acc),`
	`  d_qsort_bu2(B,S-[H|Y]).`
	
	`?- d_qsort_bu2([2,1,4,3], Res).`
	`Res = [4, 3, 2, 1|Acc] - Acc`

Fig. 21 BU-difference-list notation for quicksort

This behaviour in construction the result of a procedure call is independent from the position of the recursive call for the last input element if the body of the clause does not contain calls with side effects or calls which may loop infinitely (Fig. 22).

Quick-Sort, BU-construction of lists	
S-Acc composed from S-[H|Y] and Y-Acc	
Use of a call to cons/2	Folding the call for the cons/2-procedure

`d_qsort_bu3([],Y-Y).`
`d_qsort_bu3([H|T],S-Acc) :-`
`  split(H,T,A,B),`
`  d_qsort_bu3(B,S-Z),`
`  cons(H, Z-Y), %as Z= [H|Y]`
`  d_qsort_bu3(A,Y-Acc).`
 	
`?- d_qsort_bu3([2,1,4,3], Res).`
`Res = [4, 3, 2, 1|Acc] - Acc`
	`d_qsort_bu4([],Y-Y).`
	`d_qsort_bu4([H|T],S-Acc) :-`
	`  split(H,T,A,B),`
	`  d_qsort_bu4(B,S-[H|Y]),`
	`  d_qsort_bu4(A,Y-Acc).`
	
	`?- d_qsort_bu4([2,1,4,3], Res).`
	`Res = [4, 3, 2, 1|Acc] - Acc`

Fig. 22 Exchanged order of calls compared with Fig. 21

The difference-list representation of the quicksort algorithm is more declarative than the corresponding

representation using a call to append/3 (see also Fig. 23). A call to append/3 may loop forever if the first and second argument of append/3 are variables. The behaviour of the corresponding difference-list versions of the procedure is independent from the position of the call to cons/2.

Quicksort		
	Definitions using difference-lists	
Definitions using an append/3-call	Use of a call to cons/2	Call to cons/2 unfolded

`qsort([],[]).`
`qsort([H|T],S) :-`
`  split(H,T,A,B),`
`  qsort(A,Y),`
`  qsort(B,Z),`
`  append(Y,[H|Z],S).`
`?- qsort([3,1,2],X).`
`X = [1,2,3]`
	`d_qsort([],L-L).`
	`d_qsort([H|T],L-LN) :-`
	`  split(H,T,A,B),`
	`  d_qsort(A,L-L1),`
	`  d_qsort(B,L2-LN),`
	`  cons(H,L1-L2).`
	`?-d_qsort([3,1,2],X-[]).`
	`X = [1,2,3]`
		`du_qsort([],L-L).`
		`du_qsort([H|T],L-LN) :-`
		`  split(H,T,A,B),`
		`  du_qsort(A,L-[H|L2]),`
		`  du_qsort(B,L2-LN).`
		
		`?-du_qsort([3,1,2],X-[]).`
		`X = [1,2,3]`

`qsort5([],[]).`
`qsort5([H|T],S) :-`
`  split(H,T,A,B),`
`  qsort5(B,Z),`
`  append(Y,[H|Z],S),`
`  qsort5(A,Y).`
`?-qsort5([3,1,2],X).`
`Stack overflow`
	`d_qsort5([],L-L).`
	`d_qsort5([H|T],L-LN):-`
	`  split(H,T,A,B),`
	`  d_qsort5(B,L2-LN),`
	`  cons(H,L1-L2),`
	`  d_qsort5(A,L-L1).`
	`?- d_qsort5([3,1,2],X-[]).`
	`X = [1,2,3]`
		`du_qsort5([],L-L).`
		`du_qsort5([H|T],L-LN) :-`
		`  split(H,T,A,B),`
		`  du_qsort5(B,L2-LN),`
		`  du_qsort5(A,L-[H|L2]).`
		
		`?-du_qsort5([3,1,2],X-[]).`
		`X = [1,2,3]`

Fig. 23 Exchange of the order of calls

6. Processing trees

There exist, corresponding to [Sterling-Shapiro86], three different possibilities for the linear traversal of trees. Any node X in a binary tree, besides the leave nodes, has a *Left* and a *Right* successor tree. A pre-order traversal visits the tree in the following order: X, Left Right, which can be programmed using a call append([X|Left], Right, Tree). Correspondingly, an in-order traversal is given by the call append(Left, [X|Right],Tree) and a post-order traversal by the sequence of calls append(Right,[X],T), append(Left,T,Tree). A concrete example is shown in Fig. 24.

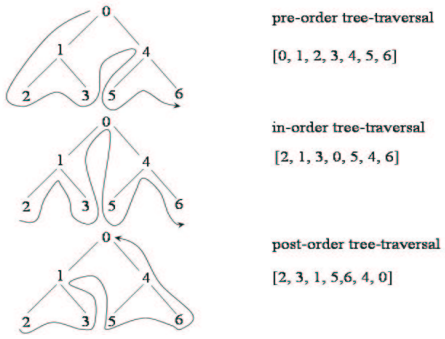


Fig. 24 Examples for linear traversals of binary trees

Specification of a pre-order traversal: The result of a pre-order traversal is the difference-list L-LN. In a top-

down construction of the result, the node X of the structure tree(X,Left,Right) is visited first and supplies the difference-list L-L1, the traversal of the left subtree supplies the difference-list L1-L2, and the traversal of the right subtree supplies L2-LN. In a bottom-down construction of the result, the node X of the structure tree(X,Left,Right) is visited first and supplies the difference-list L2-LN, the traversal of the left subtree supplies the difference-list L1-L2, and the traversal of the right subtree supplies L-L1. The in-order and post-order traversals are specified analogous.

The procedures of pre_order/2, in_order/2 and post_order/2 are different by the different positions of the cons/2-operation. This operation occurs after the head of the main clause for pre_order/2 (cons(H,Ls-L1); see also Fig. 25) , after the first recursive call for in_order/2 (cons(H, L1-l2)) and after the second recursive call for post_order/2 (cons(H, L2–Rs)). The Speedup increases by a factor of about 1.2 to 1.5 if the cons-operation is unfolded. While unfolding results in a relevant difference of speedup factors, there is hardly to recognize a significant difference in the speedup for top-down and bottom-up construction of lists.

Definitions for pre-order tree-traversal	
Top-down construction of result	Bottom-up construction of result

`%use of cons/2`
`d_pre_order_td(tree(X,L,R), Ls-Rs) :-`
`  /*Ls=[X|L1]*/  cons(X, Ls-L1),`
`  d_pre_order_td(L, L1-L2),`
`  d_pre_order_td(R, L2-Rs).`
`%append([X|Ls],Rs,Xs).`
`d_pre_order_td([],L-L).`
	`% use of cons/2`
	`d_pre_order_bu(tree(X,L,R),Ls-Rs) :-`
	`  /*L2=[X|Rs]*/  cons(X, L2-Rs),`
	`  d_pre_order_bu(L, L1-L2),`
	`  d_pre_order_bu(R, Ls-L1).`
	`%append([X|Ls],Rs,Xs).`
	`d_pre_order_bu([],L-L).`

`%call of cons/2 unfolded`
 	
`du_pre_order_td(tree(X,L,R), [X|L1]- Rs):-`
`  du_pre_order_td(L, L1 - L2),`
`  du_pre_order_td(R, L2 - Rs).`
`du_pre_order_td([],L-L).`
	`%call of cons/2 unfolded`
	
	`du_pre_order_bu(tree(X,L,R),Ls - Rs):-`
	`  du_pre_order_bu(L, L1-[X|Rs]),`
	`  du_pre_order_bu(R, Ls - L1).`
	`du_pre_order_bu([],L-L).`

Fig. 25 Different procedure definitions for pre-order tree-traversal

Speedup of order/2-predicates				
Benchmarks	**Speedup**			
	TD-construction of lists		BU-construction of lists	
	dl	dl-unfolded	dl	dl-unfolded
pre-order	2,49	3,34	2,76	3,65
in-order	1,37	2,01	1,45	2,23
post-order	5,03	6,72	5,23	6,40

Fig. 26 Speedup for tree-traversal DL-procedures in relation to the original procedures defined with append/3

7. TD-BU procedures

The almost identical schemes for top-down and bottom-up construction of list allow to construct from an input list a list with elements and a list with the same elements in inverse order in one procedure at the same time. To avoid doubling the calls to cons/2 an extended procedure cons/3 is introduced, which puts the selected element H into two difference-list:

```
cons(El, [El|L1]-L1, [El|L2]-L2).
```

Fig. 27 summarizes the construction rule for generation of result lists in append-order and inverse order in one procedure at the same time.

Rule for TD-BU-construction of lists
• The two result parameter in the clause for the base case (empty input list, empty result list) are two difference-list A-A and R-R where A and R are both different from each other and from other variables in the clause. • The result parameters in the head of the clause for the general case are A-A_n and R-R_n where A, A_n, R and R_n are both different from each other and from other variables in the clause. A-A_n is a difference-list which denotation is a list with elements in append-order, R-R_n is a difference-list which denotation is a list in inverse order. • The resulting difference-lists A-A_n and R-R_n in the body of the clause for the general case could be split in difference-lists A-A_1, A_1- …- A_{n-1}, A_{n-1}-A_n, and R-R_1, R_1- …- R_{n-1}, R_{n-1}-R_n where A_1-…-A_{n-1} and R_1-…-R_{n-1} denote arbitrary sets of further difference-lists. • The difference-lists A-A_1 and R_{n-1}-R_n are used for dealing with the first element (or first part) of the input list, A_1-A_2 and R_{n-2}-R_{n-1} for dealing with the next element(s) of the input list, etc. • the cons/3-operation may be used for dealing with a single element of the input list (unfolding the call of cons/3 supplies a more dense procedure definition)

Fig. 27 Rule for construction of result lists in append-order and reverse order in one procedure at the same time

An example for a TD-BU-procedure is presented in If only one of these result lists are needed, a possible disadvantage of such a TD-BU-procedure is the additional time needed for generation of the second list.

TD-BU-quicksort

```
%qsort(Input, List_in_append_order, List_in_reverse_order)
d_qsort([],L-L, B-B).
d_qsort([H|T],        L-LN,      R-RN) :-
   split(H,T,A,B),
   d_qsort(A,         L-L1,     R2-RN),
   cons(H,            L1-L2,    R1-R2),
   d_qsort(B,         L2-LN,     R-R1).
```

Fig. 28 Quicksort with result-lists in append- and reverse-order

The presented rule for difference-list programming and the tight connection between difference-list procedures and both top-down and bottom-up construction of lists could facilitate very much the application of this technique. In several papers and textbooks the connection between difference-list technique and accumulator technique is mentioned but the complete relationship and its simplicity was not described until now. In [AFSV00], e.g., the authors, who deal with automatic transformation of list into difference-list representation for a functional language, mention as differences between dl-technique and accumulator technique the different positions of arguments and the different goal-order (first and second column in Fig. 29). But, the different goal order is possible but not needed and is therefore no feature of a transformation from an accumulator representation to a difference-list representation. In the contrary, because of possible side effects or infinite loops in the called procedures, an exchange of calls, like it is done in the quicksort-procedure, is generally not possible.

Quicksort - Comparison of dl- and accumulator-representation		
Analogy described in [AFSV02]		Accumulator-procedure analogous to du_qs/2
dl-procedure (with unfolded cons/2-call)	Accumulator-procedure with exchanged call-order	

du_qs([],L-L).
du_qs([H|T],L-LN) :-
  split(H,T,A,B),
  du_qs(A,L-[H|L1]),
  du_qs(B,L1-LN).
 	
?-du_qs([2,3,1],X-[]).
X = [1,2,3]
	acc_qs([],L,L).
	acc_qs([H|T],L-LN) :-
	  split(H,T,A,B),
	  acc_qs(B,LN,L1),
	  acc_qs(A,[H|L1],L).
	
	?-acc_qs([2,3,1],X-[]).
	X = [1,2,3]
		acc_qsort([],L,L).
		acc_qsort5([H|T],L-LN) :-
		  split(H,T,A,B),
		  acc_qsort(A,[H|L1],L),
		  acc_qsort(B,LN,L1),
		
		?-acc_qsort([2,3,1],X,[]).
		X = [1,2,3]

Fig. 29 Comparison of quicksort in difference-list and accumulator representation (du_qs/2 and acc_qs/2 are syntactical adapted but correspond to the procedures given in [AFSV02])

8. Summary and Future work

We have considered kinds of dealing with difference-list in the literature and have investigated simple examples of constructing result lists from input lists. We are interested in a common mechanism to get the result list of a list processing procedure either in top-down or in bottom-up manner. From efficiency reason such mechanism has to avoid calls to append/3, has to avoid left-recursive programs and should avoid unnecessary backtracking steps. The advantages of the use of the difference-list techniques for efficiency of programs were stressed by many authors. In several textbooks examples for the application of difference-lists can be found. Some authors refer to a tight connection of accumulator technique and difference-list technique. But until now there is missing a simple guideline how to program with difference-lists. The matter has stayed to be difficulty. Therefore there are some attempts to transform programs which use calls to append to concatenate lists or to insert elements into lists by an automatic program transformation to difference-list programs.
From all these information we have derived a simple rule for using difference-lists for by the programmer. This rule was verified at various examples. We believe that it is much simpler to program immediately with difference-lists than to use program transformation to get a difference-list program. The programmer is able to maintain and debug his/her original program more easily, because he/she understands it better than a transformed program. Program transformation has to ensure the semantics of the original program. If the transformation engine can not decide, if the variable X in [H|X] denotes the empty list or a variable which does not occur elsewhere, a transformation into the difference-list notation must not be performed. At the other hand, difference-list procedures are more declarative and robust than procedures which use calls of append/3, where an infinite loop may occur if e.g., the first and third arguments are variables. Avoiding calls to append/3 would avoid such errors as it could be demonstrated with the quicksort-procedure (see also Fig. 23). There are six possible permutations of two recursive calls and the call to append/3 in the clause for the main case. For one of these permutations the quicksort-procedure loops infinitely but the corresponding difference-list procedure does not.

We consider as the main advantage, from the programmer point of view, that we have given a simple and schematic rule for using difference-lists. Our rule generalizes both, bottom-up construction of list using accumulators and top-down construction of lists using calls to append/3, to the notion of difference-list. The introduction of the cons/2-operation serves as a didactic means to facilitate and simplifying the use of difference-lists. This operation could be removed easily from the procedures by unfolding.
In our work until now we were exclusively interested in the definition of the rule supporting programming with difference-list. Future work should deal with a detailed analysis of the effects in respect to time and memory consumption. We have already noticed very different speedup rates for difference-list versions of procedures compared with the original procedures. There seem to be also significant differences in implantations of Prolog systems.

References

[AFSV00] Albert, E.; Ferri, C.; Steiner, F.; Vidal, G.: Improving Functional Logic-Programs by difference-lists. In He, J.; Sato, M.: Advances in Computing Sciece – ASIAN 2000. LNCS 1961. pp 237-254. 2000.

[CC88] Colhoe, H.; Cotta, J. C.: Prolog by Example. Springer-Verlag, 1988.

[Clock97] Clocksin, W. F.: Clause and Effect. Prolog Programming for the Working Programmer. Springer-Verlag. 1997.

[Clock02] Clocksin, W. F.: Prolog Programming. Lecture "Prolog for Artificial Intelligence" 2001-02 at University of Cambridge, Computer Laboratory. 2002.

[CM84] Clocksin, W. F.; Mellish, C. S.: Programming in Prolog. Springer-Verlag, 1981, 1984, 1987.

[CMcC84] Clark, K. L.; McCabe, F. G.: micro-Prolog: Programming in Logic. Prentice Hall International, 1984.

[CT77] Clark, K. L.; Tärnlund, S, Å: A First Order Theory of Data and Programs. In: Information Processing (B. Gilchrist, ed.), North Holland, pp. 939-944, 1977.

[Dodd90] Dodd, T.: Prolog. A Logical Approach. Oxford University Press, 1990

[Ge06] Geske, U.: How to teach difference-lists. Tutorial.. http://www.kr.tuwien.ac.at/wlp06/T02-final.ps.gz (Online Proceedings – WLP 2006; last visited: Sept. 06, 2007), 2006.

[MS88] Marriott, K.; Søndergaard, H.: Prolog Transformation by Introduction of Difference-Lists. TR 88/14. Dept. CS, The University of Melbourne, 1988.

[MS93] Marriott, K.; Søndergaard, H.: Prolog Difference-list transformation for Prolog. New Generation Computing, 11 (1993), pp. 125-157, 1993.

[MW88] Maier, D.; Warren, D. S.: Computing with Logic. The Benjamin/Cummings Publisher Company, Inc., 1988.

[OK90] O'Keefe, Richard A.: The Craft of Prolog. The MIT Press. 1990.

[StSh91] Sterling, L; Shapiro, E.: The Art of Prolog. The MIT Press, 1986. Seventh printing, 1991.

[ZG88] Zhang, J.; Grant, P.W.: An automatic difference-list transformation algorithm for Prolog. In: Kodratoff, Y. (ed.): Proc. 1988 European Conf. Artificial Intelligence. Pp. 320-325. Pittman, 1988.

Constraints

Weighted-Task-Sum – Scheduling Prioritized Tasks on a Single Resource*

Armin Wolf[1] and Gunnar Schrader[2]

[1] Fraunhofer FIRST, Kekuléstr. 7, D-12489 Berlin, Germany
Armin.Wolf@first.fraunhofer.de
[2] sd&m AG, Kurfürstendamm 22, D-10719 Berlin, Germany
gunnar.schrader@sdm.de

Abstract. Optimized task scheduling is an NP-hard problem, especially if the tasks are prioritized like surgeries in hospitals. Better pruning algorithms for the constraints within such constraint optimization problems, even for the constraints representing the objectives to be optimized, will result in faster convergence of branch & bound algorithms.
This paper presents new pruning rules for weighted (task) sums where the addends are the start times of tasks to be scheduled on an exclusively available resource and weighted by the tasks' priorities. The presented pruning rules are proven to be correct and the speed-up of the optimization is shown in comparison with well-known general-purpose pruning rules for weighted sums.

1 Motivating Introduction

The allocation of non-preemptive activities (or tasks) on exclusively allocatable resources occurs in a variety of application domains: job-shop scheduling, time tabling as well as in surgery scheduling. Very often, the activities to be scheduled have priorities that have to be respected.

Example 1. Typically, the sequence of surgeries is organized with respect to the patients' *ASA scores*. These values, ranging from 1 to 5, subjectively categorize patients into five subgroups by preoperative physical fitness and are named after the *American Association of Anaesthetists* (ASA). In the clinical practice, it is very common to schedule high risk patients, i.e. with a high ASA score, as early as possible and short surgeries before the long ones: a typical schedule is shown in Figure 1.

In the given example it is easy to respect the priorities of the activities, i.e. the ASA scores of the surgeries and their durations: it is only necessary to establish some ordering constraints stating that all surgeries with ASA score 3 are scheduled before all surgeries with value 2, which have to be scheduled before

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

Operating Room # 1			Tuesday, 15th November 2005			
No.	Patient Code	Surgery	Start Time	End Time	ASA score	Duration
1	3821	gastric banding	7:30 h	8:00 h	3	0:30
2	5751	appendectomy	8:00 h	8:30 h	3	0:30
3	2880	inguinal hernia, left	8:30 h	9:15 h	3	0:45
4	3978	fundoplication	9:15 h	10:45 h	3	1:30
5	7730	appendectomy	10:45 h	11:15 h	2	0:30
6	3881	gastric banding	11:15 h	11:55 h	2	0:40
7	3894	inguinal hernia, left	11:55 h	12:40 h	2	0:45
8	7962	fundoplication	12:40 h	15:10 h	2	2:30
9	8263	inguinal hernia, right	15:10 h	15:40 h	1	0:30
10	8120	inguinal hernia, right	15:40 h	16:25 h	1	0:45
11	8393	umbilical hernia	16:25 h	17:25 h	1	1:00
12	3939	enterectomy	17:25 h	19:25 h	1	2:00

Fig. 1. A typical surgery schedule respecting the ASA scores

the ones with value 1. Further, these ordering constraints have to state that within the surgeries having the same ASA scores the shorter have to be before the longer ones. However, real life is more complicated: sometimes it is better to schedule a patient with low risk and a simple surgery between two complicated surgeries of high and moderate risk. Furthermore, beyond the surgeries' ASA scores and their durations other objectives like the equipment to be used, the operation team members or the sepsis risk often have to be optimized, too. In these cases, the presented optimization criteria cannot be reflected by any "hard" constraints, because otherwise there might be a contradiction with other possibly controversial optimization criteria. Therefore, we propose to represent a prioritization of patients with high ASA scores and long surgery durations by a sum of the surgeries' start times weighted by factors depending on their patients' ASA scores. This sum might be one factor in a combining objective function, which is also influenced by other optimization criteria.

Concerning Example 1, the schedule presented in Figure 1 is suboptimal, if we minimize the sum of start times of the surgeries weighted by their ASA scores directly: the weighted sum's value is 5305 minutes if every start time is coded as the minutes after 7:30 a.m. The sum value of an optimal schedule is 4280 minutes. However, if we use the weights 10^{ASA} (1000, 100, 10 instead of 3, 2, 1), the first schedule is still optimal — its sum value is 315300 minutes.

Scheduling of tasks on exclusively available resources are in general NP-hard problems (cf. [2]), especially if they must be scheduled optimally with respect to an objective function like the sum of some values. Nevertheless, constraint programming (CP) focuses on these problems, resulting in polynomial algorithms for pruning the search space (e.g. [2,3,5,13]) as well as in heuristic and specialized search procedures for finding solutions (e.g. [11,14,15]).

Thus, obvious and straightforward approaches to solve such optimization problems in CP would be based on a constraint for the tasks to be serialized on the one hand and on the other hand on a weighted sum of the tasks' start times.

Example 2. Let three tasks be given which will be allocated to a common exclusively available resource: task no. 1 with duration $d_1 = 3$ and priority $\alpha_1 = 3$, task 2 with $d_2 = 6$ and $\alpha_2 = 3$, and task 3 with $d_3 = 6$ and $\alpha_3 = 2$. The tasks must not start before a given time 0, i.e. their not yet determined start times (variables) S_1, S_2 and S_3, respectively, must be non-negative. Furthermore the objective $R = \alpha_1 \cdot S_1 + \alpha_2 \cdot S_2 + \alpha_3 \cdot S_3$ has to be minimized. An obvious CP model of the problem in any established CP system over finite domains (e. g. CHIP, ECLiPSe, SICStus, etc.) is

$$\begin{aligned} &\mathsf{disjoint}([S_1, S_2, S_3], [3, 6, 6]) \\ &\wedge\ S_1 \geq 0 \wedge S_2 \geq 0 \wedge S_3 \geq 0 \wedge R = 3 \cdot S_1 + 3 \cdot S_2 + 2 \cdot S_3 \end{aligned}$$

or similar.[3] Any pruning of the variables' domains yields that the minimal value of R is 0. However, the minimal admissible value is 27 because no tasks may overlap[4] but this implicit knowledge is not considered in the weighted sum. Now, if we schedule task 2 to be first, then the minimal admissible value of R is 36 — which might contradict an upper bound of the variable (e.g. 35) set by a branch & bound optimization to solve the problem. However, the minimal value of R will be 30 if we add $S_2 = 0$ to the CP model, because the knowledge of the tasks' sequential ordering is no available for the pruning of the weighted sum constraint: In general the pruning algorithms for exclusively available resources (cf. e.g. [2,10,12,13]) will prune the potential start times of the remaining tasks to be at least $6 = s_2 + d_2$. Thus, the earliest possible start times of the tasks 1 and 3 is 6, which computes to $\min(R) = 3 \cdot 6 + 3 \cdot 0 + 2 \cdot 6 = 30$. Thus, the detection of a contradiction with R's upper bound 35 requires the allocation of at least one more task.

"The drawback of these approaches comes from the fact that the constraints are handled independently" [7].

2 Related Work and Contribution

The idea to consider constraints more globally is well known and well established (see e. g. [4]) because it results in better pruning of the search space and thus in better performance. Surprisingly, only a few proposals are made to combine the knowledge of so-called *global* constraints, too. In [1] and extension of the alldifferent constraint is made to consider inter-distance conditions in scheduling. In [7] a new constraint is proposed that combines a sum and a difference

[3] In some CP systems temporal non-overlapping of tasks in represented by the so-called sequence constraint instead of the disjoint constraint.

[4] The computation of this stronger value is the core contribution of this paper.

constraint. To our knowledge, this is the first time a proposal is made that combines a disjoint constraint (cf. [2,10,12]) with a weighted sum (see e. g. [8]) and shows its practical relevance. New techniques for this new *global constraint* are introduced that allows us to tackle the fact that the variables in the addends of the considered weighted sums are the start times of tasks to be sequentially ordered, i.e. allocated to an exclusively available resource. This knowledge yields stronger pruning resulting in earlier detections of dead ends during the search for a minimal sum value. This is shown by experimental results.

3 Definitions

Formally, a *prioritized task scheduling problem* like the scheduling of surgeries presented in Example 1 is a constraint optimization problem (COP) over finite domains. This COP is characterized by a finite task set $T = \{t_1, \ldots, t_n\}$. Each task $t_i \in T$ has an a-priori determined, positive integer-valued duration d_i. However, the start times $s_1, \ldots, s_n$ of the tasks in T are initially finite-domain variables having integer-valued finite potential start times, i.e. the constraints $s_i \in S_i$ hold for $i = 1, \ldots, n$, where S_i is the set of potential start times of the task $t_i \in T$. Furthermore, each task $t_i \in T$ has fixed positive priority $\alpha_i \in \mathbb{Q}$.

Basically, the tasks in T have to be scheduled non-preemptively and sequentially — neither with any break nor with any temporal overlap — on the exclusively available resource (e.g. a machine, an employee, etc. or especially an operating room).

Ignoring other objectives and for simplicity's sake, the tasks in T have to be scheduled such that the sum of prioritized start times is minimal. Thus, the *problem* is to find a *minimal solution*, i.e. some start times $s_1 \in S_1, \ldots, s_n \in S_n$ such that $\bigwedge_{1 \leq i < j \leq n} (s_i + d_i \leq s_j \vee s_j + d_j \leq s_i)$ is satisfied and the weighted sum $\sum_{i=1}^{n} \alpha_i \cdot s_i$ is *minimal*: for any other *solution* $s'_1 \in S_1, \ldots, s'_n \in S_n$ such that either $s'_i + d_i \leq s'_j$ or $s'_j + d_j \leq s'_i$ holds for $1 \leq i < j \leq n$ the weighted sum is suboptimal, i.e. $\sum_{i=1}^{n} \alpha_i \cdot s_i \leq \sum_{i=1}^{n} \alpha_i \cdot s'_i$.

In the following, we call this problem, which is characterized by a task set T, the *Prioritized Task Scheduling Problem* of T, or PTSP(T) for short. Furthermore, we call a PTSP(T) *solvable* if a (minimal) solution exists.

4 Better Pruning for the PTSP

Considering [8], pruning for a weighted sum constraint $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$ with positive rational factors $\alpha_1, \ldots, \alpha_n$ on finite domain variables $s_1 \in S_1, \ldots, s_n \in S_n$ and $r \in R$ works as follows: based on the domains' minimal and maximal values new lower and upper bounds (lwbs and upbs) for the variables' values are computed: $\mathsf{lwb}(s_k) = \left\lceil 1/\alpha_k \cdot \min(R) - \sum_{i=1, i \neq k}^{n} \alpha_i/\alpha_k \cdot \max(S_i) \right\rceil$ and $\mathsf{upb}(s_k) = \left\lfloor 1/\alpha_k \cdot \max(R) - \sum_{i=1, i \neq k}^{n} \alpha_i/\alpha_k \cdot \min(S_i) \right\rfloor$ for $k = 1, \ldots, n$ and especially $\mathsf{lwb}(r) = \lceil \sum_{i=1}^{n} \alpha_i \cdot \min(S_i) \rceil$ and $\mathsf{upb}(r) = \lfloor \sum_{i=1}^{n} \alpha_i \cdot \max(S_i) \rfloor$. Then,

the domains are updated with these bounds[5]: $S'_k = S_k \cap [\mathsf{lwb}(s_k), \mathsf{upb}(s_k)]$ for $k = 1, \ldots, n$ and especially $R' = R \cap [\mathsf{lwb}(r), \mathsf{upb}(r)]$. Finally, this process is iterated until a fix-point is reached, i.e. none of the domains change anymore.

As shown in Example 2, this pruning procedure applied to an PTSP(T) is too general if the minimal start times of all tasks have almost the same value. In this case, the fact that the variables $s_1, \ldots, s_n$ are start times of tasks to be serialized is not considered. Also shown in Example 2, this results in rather poor pruning and rather late detection of dead ends during a branch & bound search for good or even best solutions where the greatest value of r is bound to the actual objective value. We must therefore look for an alternative approximation of the lower bound of the variable r using the knowledge about the variables in the the sum's addends.

A nontrivial approximation of a lower bound of r results from some theoretical examinations using an ordering on the tasks in T:

Lemma 1. *Let a PTSP(T) be given. Further, it is assumed that the tasks in $T = \{t_1, \ldots, t_n\}$ are ordered such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. – It should be noted that it is always possible to sort the tasks in T in such a way.*

Now, if for any permutation $\sigma : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, n\}$ with $\sigma(0) = 0$ and any index $k \in \{1, \ldots, n-1\}$ the inequality $\sigma(k) > \sigma(k+1)$ holds, then swapping the k-th and the $k+1$-th task never increases the total sum, i.e. it holds $\sum_{i=1}^{n} \alpha_{\theta(i)}(\sum_{j=0}^{i-1} d_{\theta(j)}) \leq \sum_{i=1}^{n} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)})$ for the resulting permutation θ with $\theta(i) = \sigma(i)$ for $i \in \{1, \ldots, n\} \setminus \{k, k+1\}$ and $\theta(k) = \sigma(k+1)$ as well as $\theta(k+1) = \sigma(k)$. Here, $d_0 = \min(S_1 \cup \ldots \cup S_n)$ be the earliest potential start time of any task in T.

Proof. It holds that

$$
\begin{aligned}
\sum_{i=1}^{n} \alpha_{\theta(i)}(\sum_{j=0}^{i-1} d_{\theta(j)}) &= \sum_{i=1}^{k-1} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)}) + \alpha_{\sigma(k+1)}(\sum_{j=0}^{k-1} d_{\sigma(j)}) \\
&\quad + \alpha_{\sigma(k)}(\sum_{j=0}^{k-1} d_{\sigma(j)} + d_{\sigma(k+1)}) + \sum_{i=k+2}^{n} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)}) \\
&= \sum_{i=1}^{k-1} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)}) + \alpha_{\sigma(k+1)}(\sum_{j=0}^{k} d_{\sigma(j)}) \\
&\quad + \alpha_{\sigma(k)} d_{\sigma(k+1)} - \alpha_{\sigma(k+1)} d_{\sigma(k)} \\
&\quad + \alpha_{\sigma(k)}(\sum_{j=0}^{k-1} d_{\sigma(j)}) + \sum_{i=k+2}^{n} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)}) \\
&= \sum_{i=1}^{n} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)}) + \alpha_{\sigma(k)} d_{\sigma(k+1)} - \alpha_{\sigma(k+1)} d_{\sigma(k)}
\end{aligned}
$$

[5] The updated domains are quoted for better discrimination.

Further, with $\sigma(k+1) < \sigma(k)$ it also holds that $\alpha_{\sigma(k)} d_{\sigma(k+1)} \leq \alpha_{\sigma(k+1)} d_{\sigma(k)}$. Thus, the inequality

$$\sum_{i=1}^{n} \alpha_{\theta(i)} (\sum_{j=0}^{i-1} d_{\theta(j)}) \leq \sum_{i=1}^{n} \alpha_{\sigma(i)} (\sum_{j=0}^{i-1} d_{\sigma(j)})$$

immediately follows: the addend $\alpha_{\sigma(k)} d_{\sigma(k+1)} - \alpha_{\sigma(k+1)} d_{\sigma(k)}$ has no positive value because of the assumed ordering of the tasks. □

The knowledge deduced in Lemma 1 is used in the following theorem to prove that the non-decreasing ordering of the tasks with respect to their duration/priority quotients results in a minimal sum of accumulated durations:

Theorem 1. *Let a PTSP(T) be given. Further, it is assumed that the tasks in* $T = \{t_1, \ldots, t_n\}$ *are ordered such that* $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ *holds for* $1 \leq i < j \leq n$.

Then, for any permutation $\sigma : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, n\}$ *with* $\sigma(0) = 0$ *it holds*

$$\sum_{i=1}^{n} \alpha_i (\sum_{j=0}^{i-1} d_j) \leq \sum_{i=1}^{n} \alpha_{\sigma(i)} (\sum_{j=0}^{i-1} d_{\sigma(j)}) \ ,$$

where $d_0 = \min(S_1 \cup \ldots \cup S_n)$ *be the earliest potential start time of all tasks in* T.

Proof. The inequality to be shown is proven by induction over n, the numbers of tasks. Obviously, it holds for one task ($n = 1$): there is exactly one permutation σ — the identity.

For the induction step from n to $n+1$ let $n+1$ ordered tasks and an arbitrary permutation $\sigma : \{0, 1, \ldots, n, n+1\} \to \{0, 1, \ldots, n, n+1\}$ with $\sigma(0) = 0$ be given. Now during the induction step the fact is used that the permutation of the first n addends in the sum $\sum_{i=1}^{n+1} \alpha_{\sigma(i)} (\sum_{j=0}^{i-1} d_{\sigma(j)})$ does not influence the value of the last addend $\alpha_{\sigma(n+1)} (\sum_{j=1}^{n} d_{\sigma(j)})$. This results in the following two equalities and by induction in the final inequality:

$$\begin{aligned}
\sum_{i=1}^{n+1} \alpha_{\sigma(i)} (\sum_{j=0}^{i-1} d_{\sigma(j)}) &= \sum_{i=1}^{n} \alpha_{\sigma(i)} (\sum_{j=0}^{i-1} d_{\sigma(j)}) + \ \alpha_{\sigma(n+1)} (\sum_{j=0}^{n} d_{\sigma(j)}) \\
&= \sum_{i=1}^{n} \alpha_{\sigma(i)} (\sum_{j=0}^{i-1} d_{\sigma(j)}) + \ \alpha_{\sigma(n+1)} (\sum_{j=0, j \neq \sigma(n+1)}^{n+1} d_j) \\
&\geq \sum_{i=1, i \neq \sigma(n+1)}^{n+1} \alpha_i (\sum_{j=0}^{i-1} d_j) + \ \alpha_{\sigma(n+1)} (\sum_{j=0, j \neq \sigma(n+1)}^{n+1} d_j) \ .
\end{aligned}$$

Now, let $k = \sigma(n+1)$. If $k \neq n+1$, i.e. $k < n+1$ holds, the successive swaps of the k-th task and the $(n+1)$-th, $\ldots$ $(k+1)$-th tasks yields to the sum

$\sum_{i=1}^{n+1} \alpha_i(\sum_{j=0}^{i-1} d_j)$. Lemma 1 states that during these successive swaps the total sum is never increased. Concluding, the inequality to be proven holds:

$$\sum_{i=1}^{n+1} \alpha_{\sigma(i)}(\sum_{j=0}^{i-1} d_{\sigma(j)}) \geq \sum_{i=1}^{n+1} \alpha_i(\sum_{j=0}^{i-1} d_j) \ .$$

□

This means that Theorem 1 results in a non-trivial lower bound of r: $\mathsf{lwb'}(r) = \left\lceil \sum_{i=1}^{n} \alpha_i(\sum_{j=0}^{i-1} d_j) \right\rceil$, if the tasks in T are ordered such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$.

Corollary 1. *Let a PTSP(T) be given with $T = \{t_1, \ldots, t_n\}$ and the objective $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$. Then, the pruning of the domain R of the sum variable r by updating $R' = R \cap [\mathsf{lwb'}(r), \mathsf{upb}(r)]$ is correct, i.e. all values of r which are part of any solution of the PTSP(T) are greater than or equal to $\mathsf{lwb'}(r)$.* □

5 Practical Application

For an adequate practical application of Theorem 1 in a constraint programming system any change of the potential start times of the tasks has to be considered immediately. This means that the bounds of the variables have to be updated after any change of the variables domains.

Let a PTSP(T) with ordered tasks $T = \{t_1, \ldots, t_n\}$ be given such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. Then, especially during the search for (minimal) solutions of the PTSP some of the tasks will be scheduled. Thus, there will be a partition of already scheduled tasks $S = \{t_{p_1}, \ldots, t_{p_q}\}$ and unscheduled tasks $U = \{t_{u_1}, \ldots, t_{u_v}\}$ such that $S \cup U = T$ and $S \cap U = \emptyset$ holds. For the scheduled tasks the start times are determined, i.e. $|S_{p_1}| = 1, \ldots, |S_{p_q}| = 1$. The start times of the unscheduled tasks are undetermined, i.e. $|S_{u_1}| > 1, \ldots, |S_{u_v}| > 1$. For simplicity's sake let $S_{p_1} = \{w_1\}, \ldots, S_{p_q} = \{w_q\}$.

Following Theorem 1 for all potential start times $s_1 \in S_1, \ldots s_n \in S_n$ it holds that $\sum_{i=1}^{n} \alpha_i \cdot s_i = \sum_{i=1}^{q} \alpha_{p_i} \cdot s_{p_i} + \sum_{j=1}^{v} \alpha_{u_j} \cdot s_{u_j} \geq \sum_{i=1}^{q} \alpha_{p_i} \cdot w_i + \sum_{j=1}^{v} \alpha_{u_j} \cdot (\sum_{k=0}^{j-1} d_{u_k})$ where $d_{u_0} = \min(S_{u_1} \cup \ldots \cap S_{u_v})$ is the earliest potential start time of all unscheduled tasks in U. This means that Theorem 1 results in a third more adequate, non-trivial lower bound of r:

$$\mathsf{lwb''}(r) = \left\lceil \sum_{i=1}^{q} \alpha_{p_i} \cdot w_i + \sum_{j=1}^{v} \alpha_{u_j} \cdot (\sum_{k=0}^{j-1} d_{u_k}) \right\rceil \ .$$

Example 3 (Example 2 continued). Considering again the three tasks to be allocated to a common exclusively available resource. Now, if we schedule task 2 to be first with $s_2 = 0$, then in general the pruning algorithms (cf. e.g. [2,10,12,13]) will prune the potential start times of the remaining tasks to be not less

than $s_2 + d_2 = 6$. Thus, $d_{u_0} = \min(S_1 \cup S_3) \geq 6$ holds and it follows: $\alpha_1 \cdot s_1 + \alpha_2 \cdot s_2 + \alpha_3 \cdot s_3 \geq \alpha_2 \cdot s_2 + \alpha_1 \cdot d_{u_0} + \alpha_3 \cdot (d_{u_0} + d_1) \geq 3 \cdot 0 + 3 \cdot 6 + 2 \cdot (6+3) = 36$, the value stated in Example 2 is now provided with evidence.

Corollary 2. *Let a PTSP(T) be given with $T = \{t_1, \ldots, t_n\}$ and the objective $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$. Then, the pruning of the domain R of the sum variable r by updating $R' = R \cap [\max(\mathsf{lwb'}(r), \mathsf{lwb''}(r)), \mathsf{upb}(r)]$ is correct, i.e. all values of r which are part of any solution of the PTSP(T) are greater than or equal to* $\max(\mathsf{lwb'}(r), \mathsf{lwb''}(r))$. □

6 Empirical Examination

For an empirical examination of the influence of the presented lower bounds during the search for (minimal) solutions of PTSPs we implemented the presented approximations lwb' and lwb'' in a specialized WeightedTaskSum constraint in our Java constraint solving library `firstcs` [6]. WeightedTaskSum specializes `firstcs`' WeightedSum constraints implementing the general pruning rules (cf. [8]) for weighted sums, i. e. $S = \sum_{i=1}^{n} \alpha_i A_i$ on finite domain variables $S, A_1, \ldots, A_n$. Additionally, the implementation of WeightedTaskSum takes into account that the variables $A_1, \ldots, A_n$ are the start times of tasks that must not overlap in time. Thus, the durations of the tasks are additional parameters of this constraint.

For the optimization we used a branch & bound approach with a dichotomic bounding scheme: given a lower bound lwb and an upper bound upb of the objective with lwb $<$ upb an attempt is made to find a solution with an objective not greater than $\mathsf{mid} = \left\lfloor \frac{\mathsf{lwb}+\mathsf{upb}}{2} \right\rfloor$. If such a solution exists, the upper bound is decreased, i.e. upb $=$ mid; if not, the lower bound is increased, i.e. lwb $=$ mid $+1$. The search continues until lwb $=$ upb holds. Then, the most recently found solution is a minimal solution of the PTSP. The used branching is a left-to-right, depth-first incremental tree search which avoids re-traversals of already visited paths in the search tree containing suboptimal solutions (cf. [9]). The applied search strategy is specialized for contiguous task scheduling [14].

The presented pruning rules are implemented in the WeightedTaskSum constraint which is now applied to real life task scheduling problems like the scheduling of surgeries in hospitals. The latter was indeed the ultimate cause to search for better and specialized pruning algorithms: the respond times of an interactive surgery scheduler based on our Java constraint solving library `firstcs` [6] were not satisfactory. With the help of the WeightedTaskSum constraint, we managed to solve a real world problem where the user expectations with regard to quality and speed could be fulfilled.

For the sake of simplicity, we will restrict ourselves to Example 1 in order to demonstrate how the WeightedTaskSum constraint can improve the search process. We compare the scheduling using the WeightedTaskSum and WeightedSum constraints, respectively. Both constraints were combined with an implementation of disjoint constraints, called SingleResource in `firstcs`. Any SingleResource

constraint avoids overlapping of tasks on an exclusively available resource and performs *edge-finding*, *not-first-/not-last-detection* and other pruning strategies (cf. [13]). The results for the reunite comparison[6] are shown in Figure 2. The table there shows how the runtime, the number of choices and backtracks increase by the increasing number of surgeries to be scheduled. In the first row the number of surgeries which have to be scheduled are indicated. For the purpose of comparison by the number of surgeries we shortened the schedule of Example 1 by the last surgeries of each ASA score category. This means, for example, that we left out the surgeries 12, 8 and 4 when we computed the schedule for nine surgeries only.

Obviously, the WeightedTaskSum constraint yields much faster to the first optimal solutions than the WeightedSum constraint. Whereas the runtime using the latter increases exponentially, the runtime using the WeightedTaskSum increases only slightly. During further examination of the search process we found out that the search using the WeightedSum constraint took up most of the time needed for finding the optimum before the proof of the optimum could be executed. However, both phases took much longer than the equivalent phases of the search using the WeightedTaskSum constraint.

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution		
	time/msec.	# choices	# backtracks	time/msec.	# choices	# backtracks
8	110	44	16	312	699	680
9	125	57	21	969	2914	3895
10	156	92	37	4328	12619	12593
11	235	190	88	26703	74007	73964
12	250	238	106	116814	313061	312999

Fig. 2. Comparison of the elapsed runtime, the choices made, and the performed backtracks for finding a minimal solution of a PTSP(T) using the WeightedTaskSum and WeightedSum constraints, respectively.

However, if we want the exact ordering as shown in Figure 1 by using the weighted (task) sum constraints — instead of using "hard" constraints — we can increase the weights of the weighted (task) sum induced by the ASA scores non-proportionally: either as shown in the introductory Example 1 or by representing ASA score 3 by weight 300, 2 by 20 and 1 by itself. In any case an adequate representation — resulting in the desired ordering — can be computed easily: the weights have to be chosen such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. When we increase the weights in the described manner the schedule will be computed not only exactly as shown in Figure 1. In addition, the search which uses the WeightedSum constraint will find the first optimal solution much faster than be-

[6] The results were obtained by using a Pentium IV PC with 3.01 GHz and 2 GB RAM.

fore. Figure 3 shows how the increased weights (300 for 3 etc.) drastically changes the comparison between WeightedTaskSum and WeightedSum based search, especially in the worst case where the surgeries are considered in reversed order (cf. Figure 1). We also compared both sum constraints on the best case where the surgeries are considered in ideal order (results in "( )"). The runtime complexity of both optimizations behave similarly. But with respect to the reversed order the WeightedTaskSum based search still performs better than the WeightedSum based search.

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution		
	time/msec.	# choices	# backtracks	time/msec.	# choices	# backtracks
8	844 (63)	956 (12)	492 (0)	1016 (31)	1076 (29)	678 (21)
9	2344 (78)	2561 (16)	1291 (0)	2953 (47)	2801 (45)	1706 (36)
10	3578 (78)	3430 (18)	1827 (0)	4641 (47)	3816 (65)	2446 (55)
11	5328 (78)	4973 (20)	2737 (0)	7015 (62)	5358 (87)	3485 (76)
12	15422 (94)	10840 (21)	5880 (0)	20422 (94)	11502 (97)	7217 (85)

Fig. 3. Comparison of the elapsed runtime, the choices made and the performed backtracks for finding a minimal solution with regard to the non-proportional increased weights and reversed (ideal) orders of surgeries.

Of course, the runtime depends not only on the total number of surgeries but also on the distribution of the ASA score and the duration. With the WeightedSum constraint the computation of the first optimal solution takes much longer when most (or all) surgeries have the same ASA score and/or the same duration.[7] The reason for this lies in a weaker propagation which results in a much later detection of dead ends during the search.

In Figure 4 we demonstrate how the ASA scores and the durations having the same value each may effect the runtime. We compared the performance of the WeightedTaskSum together with SingleResource against the computation of the sums for all permutations of the considered surgeries. These exponential exhaustive computations ($n!$) are necessary if we use WeightedSum either together with SingleResource or any other, even specialized constraint for scheduling equal length tasks (cf. [1]). However, WeightedTaskSum requires the least number of choices to find an optimal solution: n for the finding and $n-1$ for proving the optimality where n is the number of surgeries.

[7] The examination of such "degenerated" problems is not just an "academic" issue without any practical relevance. On the contrary, we were faced with these kinds of problems by user demands. In addition, it is a fact that systems which deny computing obvious solutions — obvious for the human user only, of course — are less accepted in practice. The best explanation will not help in that case, especially if the user is not familiar with the concepts of optimization.

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution	
n	time/msec.	# choices	# backtracks	time/msec.	# combinations $n!$
8	76	15	0	16	40320
9	78	17	0	156	362880
10	93	19	0	1531	3628800
11	94	21	0	17015	39916800
12	95	23	0	200667	479001600

Fig. 4. Comparison of the elapsed runtime, the made choices and the performed backtracks with the consideration of all combinations for finding a minimal solution with regard to "degenerated" problems where the ASA score and the duration have the same value each.

7 Conclusion

The consideration of prioritized task scheduling problems shows that specialized constraint problems allow specialized and in this case even stronger pruning rules. In this paper, we invented new pruning rules for weighted sums where the addends are the weighted start times of tasks to be serialized. Then, the presented theoretical results are adopted for their practical application in constraint programming. Further, their adequacy was practically shown in a real-world application domain: the optimal sequencing of surgeries in hospitals. There, in contrast to the original rules the new pruning rules behave in very robust way, especially in "degenerated" but real scheduling problems.

References

1. Konstantin Artiouchine and Philippe Baptiste. Inter-distance constraint: An extension of the all-different constraint for scheduling equal length jobs. In Peter van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005, 11th International Conference*, volume 3709 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 2005.
2. Philippe Baptiste, Claude le Pape, and Wim Nuijten. *Constraint-Based Scheduling.* Number 39 in International Series in Operations Research & Management Science. Kluwer Academic Publishers, 2001.
3. Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001, 7th International Conference*, volume 2239 of *Lecture Notes in Computer Science*, pages 377–391. Springer Verlag, 2001.
4. Nicolas Beldiceanu and Evelyne Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.
5. Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In Pascal van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming, ICLP'94*, pages 369–383. MIT Press, 1994.

6. 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. Online available at `uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`.
7. Jean-Charles Régin and Michel Rueher. A global constraint combining a sum constraint and difference constraints. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000, 6th International Conference*, volume 1894 of *Lecture Notes in Computer Science*, pages 384–395, Singapore, September 18–21 2000. Springer Verlag.
8. Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space. In Harald Søndergaard, editor, *Third International Conference on Principles and Practice of Declarative Programming*, pages 115–126, Florence, Italy, September 2001. ACM Press.
9. Pascal van Hentenryck and Thierry le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9(3 & 4):257–275, 1991.
10. Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorical Optimisation Problems – CP-AI-OR'04*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347. Springer Verlag, 2004.
11. Petr Vilím. Computing explanations for the unary resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: Second International Conference, CP-AI-OR 2005, Proceedings*, volume 3524 of *Lecture Notes in Computer Science*, pages 396–409. Springer Verlag, 2005.
12. Petr Vilím, Roman Barták, and Ondřej Čepek. Unary resource constraint with optional activities. In Marc Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004, 10th International Conference*, volume 3258 of *Lecture Notes in Computer Science*, pages 62–76. Springer Verlag, 2004.
13. Armin Wolf. Pruning while sweeping over task intervals. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference*, volume 2833 of *Lecture Notes in Computer Science*, pages 739–753. Springer Verlag, 2003.
14. Armin Wolf. Reduce-to-the-opt — a specialized search algorithm for contiguous task scheduling. In K.R. Apt, F. Fages, F. Rossi, P. Szeredi, and J. Váncza, editors, *Recent Advances in Constraints*, volume 3010 of *Lecture Notes in Artificial Intelligence*, pages 223–232. Springer Verlag, 2004.
15. Armin Wolf. Better propagation for non-reemptive single-resource constraint problems. In B. Faltings, A. Petcu, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2004, Lausanne, Switzerland, June 23-25, 2004, Revised Selected and Invited Papers*, volume 3419 of *Lecture Notes in Artificial Intelligence*, pages 201–215. Springer Verlag, 2005.

Efficient Edge-Finding on Unary Resources with Optional Activities

Sebastian Kuhnert

Fraunhofer FIRST, Kekuléstr. 7, 12489 Berlin, Germany

Abstract. Unary resources play a central role in modeling scheduling problems. Edge-finding is one of the most popular techniques to deal with unary resources in constraint programming environments. Often it depends on external factors if an activity will be included in the final schedule, making the activity optional. Currently known edge-finding algorithms cannot take optional activities into account. This paper introduces an edge-finding algorithm that finds restrictions for enabled *and* optional activities. The performance of this new algorithm is studied for modified job-shop and random-placement problems.

Keywords: constraint-based scheduling, global constraints, optional tasks and activities, unary resources

1 Unary Resources with Optional Activities

Many everyday scheduling problems deal with allocation of resources: Schools must assign rooms to lectures, factories must assign machines to tasks, train companies must assign tracks to trains. Often an activity on a resource must be finished before a deadline and cannot be started before a release time. Formally, an **activity** i is described by its duration p_i and its earliest and latest starting and completion times as shown in Figure 1. In constraint programming systems the start time is usually stored as a variable ranging from est_i to lst_i. The duration is constant in most applications, in that case the completion times can be derived as $\mathrm{ect}_i = \mathrm{est}_i + \mathrm{p}_i$ and $\mathrm{lct}_i = \mathrm{lst}_i + \mathrm{p}_i$. Otherwise let p_i denote the minimum duration for the purposes of this paper.[1]

[1] The values of p_i are used to restrict the domains of other variables. Using larger values than the minimum duration might cause unwarranted restrictions.

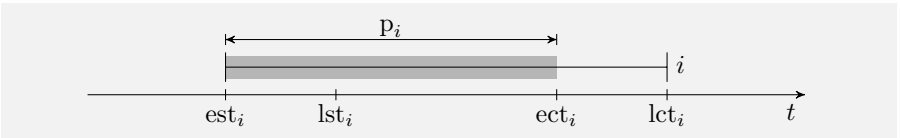


Fig. 1. Attributes of an activity i.

A resource is called **unary resource**, if it can process only one activity (or task) at once and tasks cannot be interrupted. Scheduling on unary resources is a highly complex combinatorial problem: To decide if a set T of activities can be scheduled on an unary resource is NP-complete [4, p. 236]. Constraint programming is a way to approach this problem. Using this programming paradigm, many scheduling problems can be solved [1].

This work is concerned with **optional activities** on unary resources. That are activities for which it is not (yet) known if they should be included in the final schedule. This is modelled by adding another attribute: The variable $status_i$ can take the values 1 for enabled and 0 for disabled. The domain $\{0, 1\}$ thus indicates an optional task. Additionally, let T_{enabled}, T_{optional} and T_{disabled} denote the (disjoint) subsets of T which contain the enabled, optional and disabled tasks.

Optional activities entail additional difficulties for constraint filtering algorithms: As optional activities might become disabled later, they may not influence any other activity, because the resulting restrictions would not be justified. However, it is possible to detect if the inclusion of an optional activity causes an overload on an otherwise non-overloaded unary resource. In this case it is possible to disable this optional task.

Several standard filtering algorithms for unary resources have been extended for optional activities by Vilím, Barták and Čepek [7]. To the best of the author's knowledge, no edge-finding algorithm has been proposed so far that considers optional activities. This paper aims to fill this gap.

2 Edge-Finding

There are two variants of edge-finding: One restricts the earliest starting times, the other restricts the latest completion times of the tasks on an unary resource. This paper only presents the former one, as the latter is symmetric to it.

To state the edge-finding rule and the algorithm, the following notational abbreviations are needed. Given a set Θ of tasks, $\mathrm{ECT}(\Theta)$ is an lower bound[2] of the earliest completion time of all tasks in Θ:

$$\mathrm{ECT}(\Theta) := \max_{\Theta' \subseteq \Theta} \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'}\} \tag{1}$$

Vilím has shown how this value can be computed using a so-called Θ-tree [6], that has $\mathcal{O}(\log n)$ overhead for adding or removing tasks from Θ and offers constant time access to this value in return. Later Vilím, Barták and Čepek extended this data structure to Θ-Λ-trees that retain the time complexity and include up to one task from an additional set Λ, such that (the lower bound of) the earliest completion time $\mathrm{ECT}(\Theta, \Lambda)$ becomes maximal [7]:

$$\mathrm{ECT}(\Theta, \Lambda) := \max_{j \in \Lambda} \{\mathrm{ECT}(\Theta \cup \{j\})\} \overset{(1)}{=} \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'}\} \right\} \tag{2}$$

[2] It can be less than the real earliest completion time because it assumes that activities can be interrupted. This allows faster computation at the cost of slightly weaker domain restrictions.

For this paper another extension is needed: The maximum (lower bound of the) earliest completion time for the tasks in Θ plus exactly one task from Ω plus up to one task from Λ:

$$\mathrm{ECT}(\Theta, \Omega, \Lambda) := \max_{o \in \Omega} \left\{ \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \left\{ \mathrm{est}_{\Theta' \cup \{o\}} + \mathrm{p}_{\Theta' \cup \{o\}} \right\} \right\} \right\} \tag{3}$$

As the algorithm presented below calculates Λ dynamically,[3] this value cannot be pre-computed using a tree structure. Instead, a loop over all tasks is needed, yielding $\mathcal{O}(n)$ time complexity. This loop iterates over all tasks $j \in T$, pre-sorted by their earliest starting time est_j, and keeps track of the earliest completion time of the tasks processed so far. To achieve this, the following fact is used:

Proposition 1 (Calculation of ECT[4]). *Given a set of tasks T, for each partition $L \cup R = T$, $L \cap R = \emptyset$ at an arbitrary earliest starting time t (i. e. $\forall l \in L : \mathrm{est}_l \leq t$ and $\forall r \in R : \mathrm{est}_r \geq t$) holds:*

$$\mathrm{ECT}(T) = \max \left\{ \mathrm{ECT}(R), \mathrm{ECT}(L) + \sum_{j \in R} \mathrm{p}_j \right\}$$

When iterating over the tasks $j \in T$, let L be the set of tasks considered so far (thus $\mathrm{ECT}(L)$ is known) and $R = \{j\}$ the set containing only the current activity (for which $\mathrm{ECT}(R) = \mathrm{ect}_j$ is also known). Furthermore let $L' := L \cup \{j\}$ denote the set of activities considered after the current iteration. This way L grows from the empty set until it includes all activities in T. To compute not only $\mathrm{ECT}(\Theta)$ but $\mathrm{ECT}(\Theta, \Omega, \Lambda)$ on the basis of Proposition 1, the following values must be updated in the loop for each task j that is considered:[5]

1. (The lower bound of) the earliest completion time of the Θ-activities only, i. e. $\mathrm{ECT}(L' \cap \Theta)$:

$$ect_{L'} := \begin{cases} \max\{ect_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \\ ect_L & \text{otherwise} \end{cases}$$

2. (The lower bound of) the earliest completion time when including up to one Λ-activity, i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Lambda)$:

$$ectl_{L'} := \begin{cases} \max\{ectl_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \\ \max\{ectl_L, \mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{if } j \in \Lambda \\ ectl & \text{otherwise} \end{cases}$$

[3] This applies also to one occurrence of $\mathrm{ECT}(\Theta, \Lambda)$.

[4] This is a variant of a proposition proven by Vilím, Barták and Čepek [7, p. 405], that was stated for left and right subtrees in a Θ-tree.

[5] The indices of ect_L, $ectl_L$, $ectl_L$ and $ectol_L$ are only added for conceptual clarity. If the values are updated in the right order (i. e. $ectol_L$ first and ect_L last), only plain variables *ect*, *ectl*, *ecto* and *ectol* (and no arrays) are needed. Before the loop all these variables must be initialised to $-\infty$, which is the earliest completion time of the empty set.

3. (The lower bound of) the earliest completion time when including exactly one Ω-activity (provided there is one), i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Omega, \emptyset)$:

$$ecto_{L'} := \begin{cases} \max\{ecto_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ecto_L + \mathrm{p}_j & \text{if } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{if } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ecto_L\} & \text{if } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ ecto_L & \text{otherwise} \end{cases}$$

4. (The lower bound of) the earliest completion time when including up to one Λ- and exactly one Ω-activity (provided there is one), i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Omega, L' \cap \Lambda)$:

$$ectol_{L'} := \begin{cases} \max\{ectol_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ectol_L + \mathrm{p}_j & \text{if } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ectl_L + \mathrm{p}_j\} & \text{if } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ectl_L + \mathrm{p}_j, ectol_L\} & \text{if } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ \max\{ectol_L, ecto_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Lambda \wedge \Omega \cap L = \emptyset \\ \max\{ectol_L, ecto_L + \mathrm{p}_j\} & \text{if } j \in \Lambda \wedge \Omega \cap L \neq \emptyset \\ ectol_L & \text{otherwise} \end{cases}$$

After the last iteration (i. e. $L' = T$) $ectol_{L'} = \mathrm{ECT}(L' \cap \Theta, L' \cap \Lambda, L' \cap \Omega) = \mathrm{ECT}(\Theta, \Omega, \Lambda)$ contains the result of the computation.

Finally, one more notational abbreviation is needed to state the edge-finding rule: The set of all tasks in a set T (usually all tasks or all enabled tasks), that end not later than a given task j:

$$\Theta(j, T) := \{k \in T \mid \mathrm{lct}_k \leq \mathrm{lct}_j\} \tag{4}$$

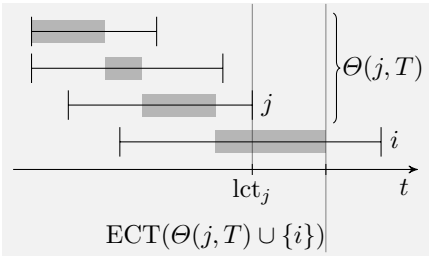
Definition 1 (Edge-finding rule). *For all activities $j \in T$ and $i \in T \setminus \Theta(j, T)$ holds that if*

$$\mathrm{ECT}(\Theta(j, T) \cup \{i\}) > \mathrm{lct}_j \tag{5}$$

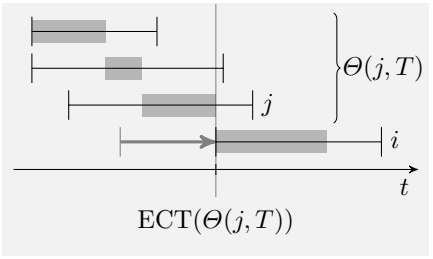
then i must be scheduled after all activities in $\Theta(j, T)$ and it is possible to set

$$\mathrm{est}_i := \max\{\mathrm{est}_i, \mathrm{ECT}(\Theta(j, T))\} \,. \tag{6}$$

This definition was introduced by Vilím, Barták and Čepek [7, p. 412ff] and proven equivalent to the more traditional form of the edge-finding rule. The idea behind this rule is to detect if an activity i has to be the last one within the set $\Theta(j, T) \cup \{i\}$. This is ensured by the precondition (5) of the rule, which is illustrated in Figure 2(a): As the latest completion time lct_j is by (4) also the latest completion time of $\Theta(j, T)$, the precondition states that $\Theta(j, T)$ must be finished before $\Theta(j, T) \cup \{i\}$ can be finished.



(a) The rule is applicable for j and i: $\mathrm{ECT}(\Theta(j,T) \cup \{i\}) > \mathrm{lct}_j$.



(b) After the application est_i is at least $\mathrm{ECT}(\Theta(j,T))$.

Fig. 2. An application of the edge-finding rule

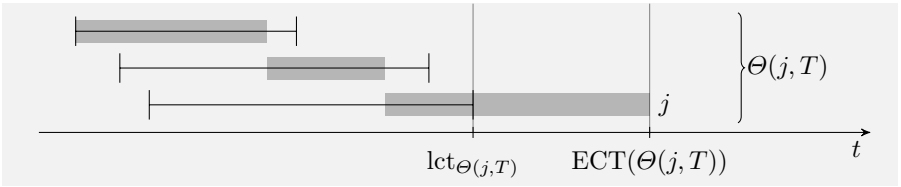


Fig. 3. An application of the overload rule

The resulting restriction (6), which is illustrated in Figure 2(b), ensures that i is not started until all activities in $\Theta(j,T)$ can be finished.

Besides the edge-finding rule, the presented edge-finding algorithm makes use of the following overload rule, that can be checked along the way without much overhead. It is illustrated in Figure 3.

Definition 2 (Overload rule). *For all $j \in T$ holds:*

If $\mathrm{ECT}(\Theta(j,T)) > \mathrm{lct}_j$ then an overload has occurred, i. e. it is not possible to schedule all activities in T without conflict.

2.1 Edge-Finding Algorithm

Now to the central algorithm of this article: Listing 1 shows a program that finds all applications of the edge-finding rule to the set of enabled tasks and furthermore disables optional tasks whose inclusion would cause an overload (i. e. est_i could be raised to a value greater than lst_i for some task i).

Throughout the repeat-loop the set Θ is updated to reflect $\Theta(j, T_{\mathrm{enabled}})$ and Ω is updated to reflect $\Theta(j, T_{\mathrm{optional}})$ – exceptions are only allowed if multiple tasks have the same lct value. As j iterates over Q, lct_j decreases and j is removed from Θ or Ω and added to Λ.

Listing 1. Edge-finding algorithm

```
1  for i ∈ T do // cache changes to est_i −
2      est'_i = est_i // changing it directly would mess up Θ-Λ-trees
3
4  (Θ, Ω, Λ) = (T_enabled, T_optional, ∅) // initialise tree(s)
5  Q = queue of all non-disabled j ∈ T \ T_disabled in descending order of lct_j
6  j = Q.first // go to the first task in Q
7  repeat
8      if status_j ≠ disabled then // move j to Λ
9          (Θ, Ω, Λ) = (Θ \ {j}, Ω \ {j}, Λ ∪ {j})
10     Q.dequeue; j = Q.first // go to the next task in Q
11
12     if status_j = enabled then
13         if ECT(Θ) > lct_j then
14             fail // because overload rule (Definition 2) applies
15
16         while ECT(Θ, Λ) > lct_j do
17             i = grey activity responsible for ECT(Θ, Λ)
18             if ECT(Θ) > est_i then // edge-finding rule is applicable
19                 if ECT(Θ) > lst_i then // inconsistency detected
20                     status_i = disabled // fails if i is enabled
21                 else
22                     est'_i = ECT(Θ) // apply edge-finding rule
23             Λ = Λ \ {i} // no better restriction for i possible [7, p. 416]
24
25         while ECT(Θ, Ω) > lct_j do // overload rule applies
26             o = optional activity responsible for ECT(Θ, Ω)
27             status_o = disabled
28             Ω = Ω \ {o}
29
30         while Ω ≠ ∅ and
31                 ECT(Θ, Ω, Λ'(o)) > lct_j do // Λ'(o) is defined in (7)
32             // edge-finding rule detects overload
33             o = optional activity responsible for ECT(Θ, Ω, ...)
34                                 // already used in line 31 with that meaning
35             status_o = disabled
36             Ω = Ω \ {o}
37     else if status_j = optional then
38         if ECT(Θ ∪ {j}) > lct_j // overload rule applicable ...
39         or ECT(Θ ∪ {j}, {i ∈ T_enabled \ Θ | lst_i < ECT(Θ ∪ {j})}) > lct_j
40         then // ... or edge-finding rule detects overload
41             status_j = disabled
42             Ω = Ω \ {j} // no more restrictions for j possible
43 until the end of Q is reached
44
45 for i ∈ T do
46     est_i = est'_i // apply cached changes
```

Lines 13 to 23 handle enabled activities j and correspond closely to the algorithm by Vilím, Barták and Čepek [7, p. 416], with the only change being the handling of $status_i = \text{optional}$ in lines 18 to 22: If $\text{ECT}(\Theta) > \text{lst}_i$ (line 19) the restriction $\text{est}'_i = \text{ECT}(\Theta)$ (line 22) would cause an inconsistency as no possible start time for i remains. In this case the activity i is set to disabled (line 20), which fails for enabled activities and is the proper restriction for optional ones.

There are several more additions to handle optional activities: Firstly, the case $status_j = \text{optional}$ is handled. It follows the scheme of the enabled case, bearing in mind that j is optional: The overload-condition $\text{ECT}(\Theta) > \text{lct}_j$ on line 13 carries over to $\text{ECT}(\Theta \cup \{j\}) > lct_j$ on line 38, where no immediate failure is generated but the optional activity j which would cause the overload is disabled.

If j is optional, the condition $\text{ECT}(\Theta, \Lambda) > lct_j$ of the while-loop on line 16 can only result in the disabling of j as no optional activity may influence others. For this, no while-loop is required and a simple if-condition suffices. It must however take care to choose an appropriate $i \in \Lambda$ that leads to j being disabled. This is achieved by requiring $i \in T_{\text{enabled}}$ and $\text{lst}_i < \text{ECT}(\Theta \cup \{j\})$ in the condition on line 39.

Secondly, the case $status_j = \text{enabled}$ is extended with two more while-loops. They realise overload-detection and edge-finding for optional activities that are still contained in Ω. The set $\Lambda'(o)$, which is used in the condition of the second while-loop, is defined as follows:

$$\begin{aligned}\Lambda'(o) := \Big\{ i \in T_{\text{enabled}} \setminus \Theta \;\Big|\; & \text{lst}_i < \text{ECT}(\Theta \cup \{o\}) \\ & \wedge\, \nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \\ & \qquad \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big) \Big\} \end{aligned} \tag{7}$$

Like for the algorithm by Vilím, Barták and Čepek, the algorithm in Listing 1 must be repeated until it finds no further restrictions to make it idempotent.

Proposition 2 (Correctness). *The algorithm presented in Listing 1 is correct, i. e. all restrictions are justified.*

Proof. All restrictions are applications of the edge-finding and overload rules (Definitions 1 and 2). The correctness of these rules thus carries over to the algorithm. □

2.2 Complexity

Proposition 3 (Time Complexity). *The time complexity of the edge-finding algorithm presented in Listing 1 is $\mathcal{O}(n^2)$, where n is the number of activities on the unary resource.*

Proof. To see this, observe that each loop is executed at most n times: The two small copy-loops in lines 1–2 and 45–46 directly loop over all tasks, the main loop in lines 12–43 loops over the subset of non-disabled tasks and the three inner loops in lines 16–23, 25–28 and 30–36 each remove a task from Λ or Ω. As each task is added to these sets at most once, each loop is executed at most n times.

Sorting the tasks by lct (line 5) can be done in $\mathcal{O}(n \log n)$. All other lines (this includes all lines within the loops) bear at most $\mathcal{O}(n)$ complexity. Most are $\mathcal{O}(1)$, the more complicated ones are $\mathcal{O}(\log n)$ or $\mathcal{O}(n)$, as discussed for (1) to (3).

Only the calculation of $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ in line 31 needs special consideration, as o already references the task chosen from Ω. The definition of $\Lambda'(o)$ in (7) makes it possible to calculate it in the following way:

- When calculating $ectl_{L'}$ take *all* activities from $T_{\text{enabled}} \setminus \Theta$ in account, as it is not yet known which o will be chosen.
- For the cases with $j \in \Omega$ during the calculation of $ectol_{L'}$, consider $ectl_L + \mathrm{p}_j$ for calculating the maximum only if the activity i chosen for $ectl_L$ satisfies $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$.
- For the cases with $j \in \Lambda$ during the calculation of $ectol_{L'}$, consider $ecto_L + \mathrm{p}_j$ for the maximum only if the activity o chosen for $ecto_L$ satisfies $\mathrm{lst}_j < \mathrm{ECT}(\Theta \cup \{o\})$.

To be able to access $\mathrm{ECT}(\Theta \cup \{o\})$ in constant time for all $o \in \Omega$, three arrays $ectAfter_o$, $pAfter_o$ and $ectBefore_o$ can be used, where after/before refers to the position of o in the list of tasks sorted by est_j and the three values consider only tasks from Θ. They can be computed based on Proposition 1 in linear time by looping over the tasks pre-sorted by est_j. It holds:

$$\mathrm{ECT}(\Theta \cup \{o\}) = \max\{ectAfter_o, \mathrm{ect}_o + pAfter_o, ectBefore_o + \mathrm{p}_o + pAfter_o\}$$

Thus the overall time complexity of the algorithm is $\mathcal{O}(n^2)$. □

2.3 Optimality

For efficient restriction of the domains it is necessary that as many applications of the edge-finding rule as possible are found by the algorithm, while it is equally crucial that it offers fast runtimes.

Proposition 4 (Optimality). *The algorithm presented in Listing 1 finds all applications of the edge-finding rule to enabled activities and disables almost all optional activities that would cause an overload that is detectable by the edge-finding rule.*

Proof. First consider all applications of the edge-finding rule that involve only enabled tasks. Let therefore $j \in T_{\text{enabled}}$ and $i \in T_{\text{enabled}} \setminus \Theta(j, T_{\text{enabled}})$, such that $\mathrm{ECT}(\Theta(j, T_{\text{enabled}}) \cup \{i\}) < \mathrm{lct}_j$. As argued above, the set Θ at some point takes the value of $\Theta(j, T_{\text{enabled}})$ and i is contained in Λ. Then in line 18 i will be chosen (possibly after other i' have been chosen and removed from Λ) and est_i will be adjusted or an overload detected.

If an optional activity $o \in T_{\text{optional}}$ is involved, things are a bit more complicated. If the edge-finding rule is applicable for the set $T' := T_{\text{enabled}} \cup \{o\}$, the following cases can occur:

Case 1: $o \notin \Theta(j, T') \cup \{i\}$
The optional activity o is not involved and the edge-finding rule is equally applicable to the set T_{enabled}. The resulting restriction is found by the algorithm as argued above.

Case 2: $o = i$
This case is handled together with non-optional i in lines 16 to 23.

Case 3: $o = j$
In this case est_i may not be adjusted as the optional activity o may not influence the non-optional i. However, if est_i can be adjusted to a value greater than lst_i, the inclusion of o would cause an overload and o can be disabled. This is done in lines 37 to 42.

Case 4: $o \in \Theta(j, T') \setminus \{j\}$
Like in the previous case the only possible restriction is disabling o if it causes an overload. This case is handled in lines 30 to 36. For optimal results $\Lambda'(o)$ would have to be defined slightly differently, omitting the condition

$$\nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big).$$

However, this would destroy the $\mathcal{O}(n^2)$ complexity of the algorithm. As this case never occurs for any of the problem instances measured in the next section, omitting these restrictions appears to be a good choice. □

2.4 Comparison with Other Edge-Finding Algorithms

The presented algorithm has been implemented for the Java constraint programming library `firstcs` [5] and compared to other algorithms. The following algorithms were considered:

New algorithm: The algorithm presented in Listing 1. It finds almost all possible applications of the edge-finding rule in $\mathcal{O}(n^2)$ time.

Simplified algorithm: Similar to the previous algorithm but without the loop in lines 30 to 36. This saves one of the two most expensive computations, though the asymptotic time complexity is still $\mathcal{O}(n^2)$. Fewer restrictions are found.

Active-only algorithm: The algorithm presented by Vilím, Barták and Čepek [7, p. 416]. It runs at $\mathcal{O}(n \log n)$ time complexity but takes no optional tasks into account.

Iterative algorithm: Run the active-only algorithm once on T_{enabled} and then for each $o \in T_{\text{optional}}$ on $T_{\text{enabled}} \cup \{o\}$, adjusting or disabling o only. This algorithm finds all restrictions to optional activities at the cost of $\mathcal{O}(n^2 \log n)$ runtime.

Table 1. Runtime in milliseconds and number of backtracks for the different algorithms and job-shop-scheduling instances.

	new		**simplified**		**active-only**		**iterative**	
instance	**time**	**bt**	**time**	**bt**	**time**	**bt**	**time**	**bt**
abz5-alt	4843	5859	4484	5859	4515	6441	4964	5859
orb01-alt	55747	56344	53662	56344	49361	56964	57013	56344
orb02-alt	7800	7265	7394	7265	9590	10610	7609	7265
orb07-alt	81856	99471	79063	99471	78309	104201	79786	99471
orb10-alt	136	85	125	85	121	85	132	85
la16-alt	7269	8294	6886	8294	6593	8841	7241	8294
la17-alt	46	9	31	9	35	11	31	9
la18-alt	26780	26846	25147	26846	24877	29039	25897	26846
la19-alt	2566	2022	2406	2022	4609	4632	2574	2022
la20-alt	62	53	63	53	55	53	62	53
abz5	5863	5107	5863	5107	5587	5107	5570	5107
orb01	29612	21569	29706	21569	28198	21569	28336	21569
orb02	10144	7937	10187	7937	9644	7937	9687	7937

For benchmarking, *-alt* variants [7, p. 423] of job-shop problems [3] were used: For each job, exactly one of the fifth and sixth task must be included for an solution. The times are measured in milliseconds and include finding a solution for the known optimal make-span and proving that this make-span is indeed optimal.

The search was performed using a modified resource-labelling technique that at each level in the search tree decides not only the ordering of two tasks on a machine, but also if these tasks are enabled or disabled.[6]

Table 1 shows the results. The number of backtracking steps required is equal for the new, the simplified and the iterative algorithm. So the new algorithm looses none of the possible restrictions, even if the loop in lines 30 to 36 is left out. When comparing the runtimes, the overhead of this loop is clearly visible: The simplified variant saves time, as it has lower overhead.

The active-only algorithm needs more backtracking steps because it finds less restrictions. Often the lower complexity compensates for this when it comes to the runtimes and the active-only variant is slightly better than the simplified one. The largest relative deviations is the other way around, though: For orb02-alt the active-only algorithm is almost 30% slower than the simplified one, for la19-alt it even needs 90% more time. These are also the instances with the largest difference in backtracking steps. It obviously depends on the inner structure of the job-shop instances (or the problems in general), if the higher time complexity

[6] This leads to the following alternatives: (a) both enabled with i before j, (b) both enabled with i after j, (c) only i enabled, (d) only j enabled or (e) both disabled.

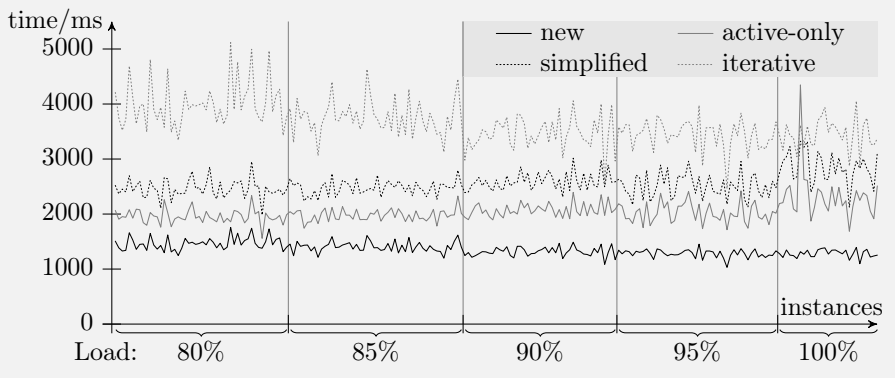


Fig. 4. Runtimes for instances of the Random-Placement-Problem

of the newly proposed algorithms is justified by the backtracking steps saved by the additionally found restrictions.

The runtime of the iterative algorithm is mostly better than the new, but worse than the simplified algorithm.[7] As the tested instances all have relatively few optional activities per machine[8] (which benefits the active-only and iterative algorithms), it is interesting how the new algorithm performs for problems with more activities and a higher proportion of optional ones. One such problem is the random-placement-problem [2], which can be solved using alternative resource constraints which in turn can be implemented using single resources with optional activities [8]. The runtimes of the instances provided on `http://www.fi.muni.cz/~hanka/rpp/` that are solvable in less than ten minutes are shown in Figure 4.[9]

It is obvious that the new algorithm is the fastest for this problem. Surprisingly the simplified algorithm is worse than the active-only one: Obviously it finds no or not enough additional restrictions. The power of the new algorithm thus derives from the loop omitted in the simplified one.[10] The iterative algorithm is the slowest, though it finds as many restrictions as the new one. The reason is the higher asymptotic time complexity, which seems to be prohibitive even for relatively small resources with 50 to 100 tasks which appear in the tested random-placement instances.

[7] An exception are the three last instances, which include no optional tasks.

[8] They have 10 jobs each consisting of 10 tasks, of which 2 are made optional. Thus the machines have 10 tasks each of which 20% are optional on average.

[9] All algorithms need 0 backtracking steps, with the exception of the active-only algorithm for one instance with 100% load.

[10] This is confirmed by the number of choices needed during search: For the iterative and new algorithms it is always equal (the new one again looses no restrictions) and averagely 407, but the active-only and simplified ones both need more than 620 on average.

3 Summary

This paper introduces a new edge-finding algorithm for unary resources. It deals with optional activities correctly, finds all applications of the edge-finding rule to the enabled activities and disables optional activities if they cause an overload detected by the edge-finding rule.

With $\mathcal{O}(n^2)$ asymptotic time complexity it is slower than edge-finding algorithms that cannot take optional activities into account. The additionally found restrictions offer a significant reduction of the search decisions needed to find and prove the solution of a problem. Especially for problems with many activities, of which a high proportion is optional, this leads to faster runtimes.

Future work can study the influence of different labelling strategies when it comes to optional activities: Is it better to first establish an order of tasks and then decide which tasks should be enabled? Another possible direction for future work is studying for which problem structure the newly proposed algorithm yields better results, possibly implementing heuristics to decide which algorithm to use.

References

1. Philippe Baptiste, Claude Le Pape, and Wim Nuijten. 'Constraint-Based Scheduling – Applying Constraint Programming to Scheduling Problems'. Boston: Kluwer Academic Publishers, 2001. ISBN 0792374088.
2. Barták, Roman, Tomáš Müller and Hana Rudová. 'A New Approach to Modeling and Solving Minimal Perturbation Problems'. In: Recent Advances in Constraints, CSCLP 2003. LNCS 3010. Berlin: Springer, 2004. ISBN 978-3-540-21834-0. 233–249.
3. Yves Colombani. 'Constraint programming: an efficient and practical approach to solving the job-shop problem'. In: Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*. LNCS 1118. Berlin: Springer, 1996. ISBN 3-540-61551-2. 149–163.
4. Michael R. Garey and David S. Johnson. 'Computers and Intractability – A Guide to the Theory of NP-Completeness'. New York: W. H. Freeman, 1979. ISBN 0-7167-1045-5.
5. 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. URL: `http://uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`.
6. Petr Vilím. '$\mathcal{O}(n \log n)$ Filtering Algorithms for Unary Resource Constraint'. In: Jean-Charles and Régin Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. LNCS 3011. Berlin: Springer, 2004. ISBN 978-3-540-21836-4. 335–347.
7. Petr Vilím, Roman Barták and Ondřej Čepek. 'Extension of $\mathcal{O}(n \log n)$ Filtering Algorithms for the Unary Resource Constraint to Optional Activities'. In: *Constraints* 10.4 (2005). 403–425.
8. Wolf, Armin, and Hans Schlenker. 'Realising the Alternative Resources Constraint'. In: Applications of Declarative Programming and Knowledge Management, INAP 2004. LNCS 3392. Berlin: Springer, 2005. ISBN 978-3-540-25560-4. 185–199.

Encoding of Planning Problems and their Optimizations in Linear Logic

Lukas Chrpa, Pavel Surynek and Jiri Vyskocil

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
{chrpa,surynek,vyskocil}@kti.mff.cuni.cz

Abstract. Girard's Linear Logic is a useful formalism which can be used to manage a lot of problems with consumable resources. Its expressiveness is quite good for easily understandable encoding of many problems. We concentrated on expressing planning problems by linear logic in this paper. We observed a rich usage of a construct of consumable resource in planning problem formulation. This fact motivates us to develop encoding of planning problems in linear logic. This paper shows how planning problems can be encoded in Linear Logic and how to encode some optimizations of these planning problems which can be used to improve the efficiency of finding solutions (plans).

1 Introduction

Linear Logic is a useful formalism which can be used to formalize many problems with consumable resources [2]. There is a lot of such problems that are handling with consumable and with renewable resources in practice. Linear Logic gives us a good expressiveness that help us with formalizing of these problems. This is because the problems are usually encoded in formulae with linear size with respect to the length of the problems.

Previous research gave us some interesting results. Many of these results are in a theoretical area. One of the important results was connectivity of Linear Logic with the Petri Nets [17] which gave us more consciousness of the good expressiveness of Linear Logic.

Instead of the encoding of Petri nets in Linear Logic, there is a possibility of encoding of planning problems in Linear Logic. Planning problems are an important branch of AI and they are very useful in many practical applications. Previous research showed that planning problems can be simply encoded in Linear Logic and a problem of plan generation can be reduced to a problem of finding a proof in linear logic [16, 10, 3]. This paper shows that Linear Logic is not only able to encode planning problems directly, but Linear Logic is also able to encode many optimizations of planning problems which can be used to increase the efficiency of finding solutions of these planning problems.

We provide a short introduction to Linear Logic and to planning problems in Section 2. Description of the pure encoding of planning problems in Linear

Logic and an extension of this encoding which works with negative predicates is provided in Section 3. Description of the encoding of optimizations such as encoding of static predicates, blocking actions, enforcing actions and assembling actions is provided in Section 4. Section 5 give us the overview of related work in this area and Section 6 will give us possible future work in this area and Section 7 concludes.

2 Preliminaries

This section presents some basic information about Linear Logic and planning problems that will be helpful to the reader to get through this paper.

2.1 Linear Logic

Linear Logic was introduced by Girard in 1987 [7]. Linear Logic is often called 'logic of resources', because unlike the 'classical' logic, Linear Logic can handle with expendable resources. The main difference between 'classical' logic and Linear Logic is that if we say "From A, A imply B we obtain B", in 'classical' logic A is still available, but in Linear Logic A consumed which means that A is no longer available.

In addition to implication ($\multimap$), there are more operators in linear logic. However we mention only those that are relevant for our encoding. One of the operators in multiplicative conjunction ($\otimes$) whose meaning is consuming (on the left side of implication) or obtaining (on the right side of implication) resources together. Another operator is exponential (!), which converts linear fact (expendable resource) to 'classical' fact (not expendable resource). At last, there are additive conjunction (&) and additive disjunction ($\oplus$) whose meaning is close to modalities. Exactly when additive conjunction is used on the right side of implication, then only one alternative must be proved, but when additive disjunction is used on the right side of implication, then all the alternatives must be proved. If the additives are on the left side of implication, proving is quite similar, only the meaning of additive conjunction and additive disjunction is swapped.

Proving in Linear Logic is quite similar to proving in the 'classical' logic, which is based on Gentzen's style. Part of Linear Logic calculus which is needed in this paper is following:

Id	$A \vdash A$	$\Gamma \vdash \top, \Delta$	$\mathbf{R}\top$
$\mathbf{L}\otimes$	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, (A \otimes B) \vdash \Delta}$	$\frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2 \vdash B, \Delta_2}{\Gamma_1, \Gamma_2 \vdash (A \otimes B), \Delta_1 \Delta_2}$	$\mathbf{R}\otimes$
$\mathbf{L}\multimap$	$\frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, (A \multimap B) \vdash \Delta_1 \Delta_2}$	$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash (A \oplus B), \Delta} \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash (A \oplus B), \Delta}$	$\mathbf{R}\oplus$
W !	$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}$	$\frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}$	**C !**
D !	$\frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}$	$\frac{\Gamma, A \vdash \Delta}{\Gamma, (\forall x) A \vdash \Delta}$	**Forall**

The complete calculus of Linear Logic can be found at [7, 8, 15].

2.2 Planning problems

This subsection brings us some basic introduction which is needed to understand basic notions frequently used in a connection to planning problems.

In this paper we consider an action-based planning problems like Block World, Depots etc. These planning problems are based on the worlds containing objects (boxes, robots, etc.) and locations (depots, platforms, etc.). Relationships between objects and places (Box 1 is on Box2, Robot 1 is in Depot 2, etc.) are described by predicates. The worlds can be changed only by preforming actions where each action is deterministic. The next paragraph describes notions and notations used in this paper in connection to planning problems.

State s is a set of all predicates which are true in a corresponding world. Action a is a 3-tuple $(p(a), e^-(a), e^+(a))$ where $p(a)$ is a precondition of the action a, $e^-(a)$ is a set of negative effects of the action a and $e^+(a)$ is a set of positive effects of the action a and $e^-(a) \cap e^+(a) = \oslash$. Action a can be performed on the state s if $p(a) \subseteq s$ and when the action a is performed on the state s, a new state s' is obtained where $s' = (s \setminus e^-(a)) \cup e^+(a)$. Planning domain represents a set of all possible states with respect to a given world and a set of all actions which can transform the given world. Planning problem is represented by the planning domain, by an initial state and a goal state. Plan is an ordered sequence of actions which leads from the initial state to the goal state. To get deeper insight about planning problems, see [6].

3 Planning in Linear Logic

In this section, we show that Linear Logic is useful formalism for encoding planning problems. Main idea of the encoding is based on the fact that predicates in planning problems can be represented as linear facts that can be consumed or obtained depending on performed actions.

3.1 Basic encoding of planning problems

Idea of reducing the problem of plan generation to finding a proof in Linear Logic was previously studied at [16, 10, 3]. As it was mentioned above, predicates in planning problems can be encoded as linear facts. Let $s = \{p_1, p_2, \ldots, p_n\}$ be a state, its encoding in Linear Logic is following:

$$(p_1 \otimes p_2 \otimes \ldots \otimes p_n)$$

Let $a = \{p(a), e^-(a), e^+(a)\}$ is an action, its encoding in Linear Logic is following:

$$\forall p_i \in p(a) \setminus e^-(a), 1 \leq i \leq l; \forall r_j \in e^-(a), 1 \leq j \leq m; \forall s_k \in e^+(a), 1 \leq k \leq n$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes r_1 \otimes r_2 \otimes \ldots \otimes r_m) \multimap (p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes s_1 \otimes s_2 \otimes \ldots \otimes s_n)$$

Performing of the action a can be reduced to a proof in Linear Logic in following way (Γ and Δ represent multiplicative conjunction of literals):

$$\dfrac{\dfrac{\vdots}{\Gamma, p_1, \ldots p_l, s_1, \ldots, s_n \vdash \Delta} \qquad \dfrac{}{p_1, \ldots p_l, r_1, \ldots, r_m \vdash p_1, \ldots p_n, r_1, \ldots, r_m}(Id)}{\Gamma, p_1, \ldots p_l, r_1, \ldots, r_m, ((p_1 \otimes \ldots \otimes p_l \otimes r_1 \otimes \ldots \otimes r_m) \multimap (p_1 \otimes \ldots \otimes p_l \otimes s_1 \otimes \ldots \otimes s_n)) \vdash \Delta}\ (L \multimap)$$

In most of planning problems it is not known how many times are actions performed. That is why exponential ! is used for each rule representing the action. The proof must be modified in dependance of how many times is the action performed. There can be three cases:

- action a is not performed — then use $W!$ rule in a following way:

$$\dfrac{\Gamma \vdash \Delta}{\Gamma, !a \vdash \Delta}\ (W!)$$

- action a is performed just once — then use $D!$ rule in a following way:

$$\dfrac{\Gamma, a \vdash \Delta}{\Gamma, !a \vdash \Delta}\ (D!)$$

- action a is performed more than once — then use $D!$ and $C!$ rule in a following way:

$$\dfrac{\dfrac{\Gamma, !a, a \vdash \Delta}{\Gamma, !a, !a \vdash \Delta}\ (D!)}{\Gamma, !a \vdash \Delta}\ (C!)$$

The last thing, what is needed to explain, is why the constant $\top$ must be used. That is, because the goal state is reached when some state contains all of the goal predicates. The state can certainly contain more predicates. The importance of the constant $\top$ can be seen in following:

$$\dfrac{g_1, \ldots, g_n \vdash g_1 \otimes \ldots \otimes g_n\ (Id) \qquad s_1, \ldots, s_m \vdash \top\ (R\top)}{g_1, \ldots, g_n, s_1, \ldots, s_m \vdash g_1 \otimes \ldots \otimes g_n \otimes \top}\ (R\otimes)$$

Now it is clear that whole planning problem can be reduced to proving in Linear Logic. Let $s_0 = \{p_{0_1}, p_{0_2}, \ldots, p_{0_m}\}$ is an initial state, $g = \{g_1, g_2, \ldots, g_q\})$ is a goal state and $a_1, a_2, \ldots, a_n$ are actions encoded as above. The planning problem can be encoded in a following way:

$$\begin{gathered}
p_{0_1}, p_{0_2}, \ldots, p_{0_m}, \\
!(p_1^1 \otimes p_2^1 \otimes \ldots \otimes p_{l_1}^1 \otimes r_1^1 \otimes r_2^1 \otimes \ldots \otimes r_{m_1}^1) \multimap (p_1^1 \otimes p_2^1 \otimes \ldots \otimes p_{l_1}^1 \otimes s_1^1 \otimes s_2^1 \otimes \ldots \otimes s_{n_1}^1), \\
!(p_1^2 \otimes p_2^2 \otimes \ldots \otimes p_{l_2}^2 \otimes r_1^2 \otimes r_2^2 \otimes \ldots \otimes r_{m_2}^2) \multimap (p_1^2 \otimes p_2^2 \otimes \ldots \otimes p_{l_2}^2 \otimes s_1^2 \otimes s_2^2 \otimes \ldots \otimes s_{n_2}^2), \\
\vdots \\
!(p_1^n \otimes p_2^n \otimes \ldots \otimes p_{l_n}^n \otimes r_1^n \otimes r_2^n \otimes \ldots \otimes r_{m_n}^n) \multimap (p_1^n \otimes p_2^n \otimes \ldots \otimes p_{l_n}^n \otimes s_1^n \otimes s_2^n \otimes \ldots \otimes s_{n_n}^n) \\
\vdash g_1 \otimes g_2 \otimes \ldots \otimes g_q \otimes \top
\end{gathered}$$

The plan exists if and only if the above expression is provable in Linear Logic. Obtaining the plan from the proof can be done by checking the $(L \multimap)$ rules from bottom (the expression) to top (axioms) of the proof.

3.2 Encoding of negative predicates

Previous subsection showed how to encode planning problems in Linear Logic. However, this encoding works with positive precondition only. In planning problems are usually used negative preconditions which means that an action can be performed if some predicate does not belong to current state. However in Linear Logic the negative preconditions cannot be encoded directly. Fortunately, there are some possible approaches how to bypass this problem.

First approach can be used in propositional Linear Logic. The basic encoding of planning problems must be extended with linear facts representing negative predicates (each predicate p will obtain a twin $\overline{p}$ which represents that predicate $p \notin s$). It is clear that either p or $\overline{p}$ is contained in the each part of the proof. The encoding of the state s, where the predicates $p_1, \ldots, p_m \in s$ and the predicates $p_{m+1}, \ldots, p_n \notin s$:

$$p_1 \otimes \ldots \otimes p_m \otimes \overline{p_{m+1}} \otimes \ldots \otimes \overline{p_n}$$

Each action $a = \{p(a), e^-(a), e^+(a)\}$ from a given planning domain can be transformed to the action $a' = \{p(a), e'^-(a'), e'^+(a')\}$, where $e'^-(a') = e^-(a) \cup \{\overline{p} \,|p \in e^+(a)\}$ and $e'^+(a') = e^+(a) \cup \{\overline{p} \,|p \in e^-(a)\}$. The encoding of the action a' is following:

$$\forall p_i \in p(a) \setminus e'^-(a'), 1 \leq i \leq l; \forall \overline{p_{i'}} \in p(a) \setminus e'^-(a'), 1 \leq i' \leq l'$$
$$\forall r_j \in e'^-(a'), 1 \leq j \leq m; \forall s_k \in e'^+(a'), 1 \leq k \leq n$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes \overline{p_1} \otimes \overline{p_2} \otimes \ldots \otimes \overline{p_{l'}} \otimes r_1 \otimes r_2 \otimes \ldots \otimes r_m \otimes \overline{s_1} \otimes \overline{s_2} \otimes \ldots \otimes \overline{s_n}) \multimap$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes \overline{p_1} \otimes \overline{p_2} \otimes \ldots \otimes \overline{p_{l'}} \otimes s_1 \otimes s_2 \otimes \ldots \otimes s_n \otimes \overline{r_1} \otimes \overline{r_2} \otimes \ldots \otimes \overline{r_m})$$

Second approach can be used in predicate Linear Logic. Each linear fact representing the predicate $p(x_1, \ldots, x_n)$ can be extended by one argument representing if the predicate $p(x_1, \ldots, x_n)$ belongs to the state s or not ($p(x_1, \ldots, x_n) \in s$ can be represented as $p(x_1, \ldots, x_n, 1)$ and $p(x_1, \ldots, x_n) \notin s$ can be represented as $p(x_1, \ldots, x_n, 0)$). Encoding of the actions can be done in a similar way like in the first approach. The advantage of this approach is in a fact that the representation of predicates can be generalized to a case that more than one (same) predicate is available which may be helpful in encoding of some other problems (for example Petri Nets).

3.3 Example

In this example we will use a predicate extension of Linear Logic. Imagine the version of "Block World", where we have slots and boxes, and every slot may contain at most one box. We have also a crane, which may carry at most one box.

Objects: 3 slots (1,2,3), 2 boxes (a, b), crane

Initial state: $in(a, 1) \otimes in(b, 2) \otimes free(3) \otimes empty$

Actions:

$$PICKUP(Box, Slot) = \{ \quad p = \{empty, in(Box, Slot)\}, \\ e^- = \{empty, in(Box, Slot)\}, \\ e^+ = \{holding(Box), free(Slot)\}\}$$

$$PUTDOWN(Box, Slot) = \{ \quad p = \{holding(Box), free(Slot)\}, \\ e^- = \{holding(Box), free(Slot)\}, \\ e^+ = \{empty, in(Box, Slot)\}\}$$

Goal: Box a in slot 2, Box b in slot 1.

The encoding of the action $PICKUP(Box, Slot)$ and $PUTDOWN(Box, Slot)$:

$PICKUP(Box, Slot) : empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)$

$PUTDOWN(Box, Slot) : holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)$

The whole problem can be encoded in Linear Logic in the following way:

$in(a, 1), in(b, 2), free(3), empty, !(empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)),$
$!(holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)) \vdash in(b, 1) \otimes in(a, 2) \otimes \top$

4 Encoding optimizations of planning problems in Linear Logic

In the previous section, we showed the pure encoding of planning problems in Linear Logic. To improve efficiency of searching for a plan, it is needed to encode some optimizations which are described in next subsections.

4.1 Handling with static predicates

Static predicates are usually used in the planning problems. Static predicate can be easily detected, because each static predicate in a planning problem is such predicate that belongs to the initial state and does not belong to any effect of any action (static predicates appear only in preconditions). It is possible to encode the static predicates like 'classical' facts using the exponential ! and actions that use the static predicates can be encoded in a following way: (Due to space reason are 'normal' predicates replaced by dots)

$$\ldots \otimes !a \otimes \ldots \multimap \ldots$$

The encoding of static predicates is described in propositional linear logic to better understanding. This encoding has the purpose in predicate Linear Logic.

4.2 Blocking of actions

To increase efficiency of solving planning problems it is quite necessary to use some technique which helps the solver to avoid unnecessary backtracking. Typically the way how to avoid a lot of unnecessary backtracking is blocking of actions of what is known that wont lead to the solution (for example inverse actions).

The idea how to encode the blocking of actions is in adding a new predicate $can(a, x)$, where a is an action and $x \in \{0, 1\}$ is representing a status of the action a. If $x = 0$ then the action a is blocked and if $x = 1$ then the action a is unblocked. Now is clear that the encoding of the action a must be done in a following way: (Due to space reason are 'normal' predicates replaced by dots)

$$can(a, 1) \otimes \ldots \multimap can(a, 1) \otimes \ldots, \text{or}$$
$$can(a, 1) \otimes \ldots \multimap can(a, 0) \otimes \ldots$$

The first expression means that a does not block itself and the second expression means that a block itself.

Assume that an action b can block the action a. Encoding of the action b is following ($can(b, ?)$ means $can(b, 1)$ or $can(b, 0)$ like in the previous encoding of the action a):

$$\forall X : can(b, 1) \otimes can(a, X) \otimes \ldots \multimap can(b, ?) \otimes can(a, 0) \otimes \ldots$$

The predicate $can(a, X)$ from the expression above represents a fact that is not known if a is blocked or not. If a is already blocked then X unifies with 0 and anything remains unchanged ($can(a, 0)$ still holds). If a is not blocked then X unifies with 1 which means that a become blocked, because $can(a, 1)$ is no longer true and $can(a, 0)$ become true.

Now assume that an action c can unblock the action a. Encoding of the action c is done by a similar way like the action b. The encoding of the action c is following ($can(c, ?)$ means $can(c, 1)$ or $can(c, 0)$ like in the previous encoding of the action a):

$$\forall X : can(c, 1) \otimes can(a, X) \otimes \ldots \multimap can(c, ?) \otimes can(a, 1) \otimes \ldots$$

The explanation how this encoding works is similar like the explanation in the previous paragraph.

4.3 Enforcing of actions

Another optimization which can help the solver to find the solution faster is enforcing of actions. Typically, when some action is performed is necessary to perform some other action. It is possible to enforce some action by blocking of the other actions, but this may decrease the efficiency, because each single action must be blocked in the way described in the previous subsection which means that the formula rapidly increases its length.

The idea how to encode the enforcing of actions is also in adding new predicate $can(a)$, where a is an only action which can be performed. Let $can(1)$ represents the fact that all actions can be preformed. The encoding of an action a which does not enforce any other action is following:

$$(can(a) \oplus can(1)) \otimes \ldots \multimap can(1) \otimes \ldots$$

The expression $can(a) \oplus can(1)$ means that action a can be performed if and only if a is enforced by other action ($can(a)$ is true) or all actions are allowed ($can(1)$ is true).

Now assume that an action b enforces the action a. The encoding of the action b is following:

$$(can(b) \oplus can(1)) \otimes \ldots \multimap can(a) \otimes \ldots$$

It is clear that in this encoding are one or all actions allowed in a certain step. This idea can be easily extended by adding some new symbols representing that a certain groups of actions are allowed.

4.4 Assembling actions into a single action

Another kind of optimizations in planning problems can be assembling actions into single action usually called macro action. This approach is based on a fact that some sequences of actions is used several times. Let $a_1, \ldots, a_n$ is a sequence of actions encoded in Linear Logic in a way described before. For further usage will be used shortened notation of the encoding like that:

$$\bigotimes_{\forall l \in \Gamma_i} l \multimap \bigotimes_{\forall r \in \Delta_i} r \quad \forall i \in \{1, \ldots, n\}$$

Assume that an action a is created by a assembling of the sequence of the actions $a_1, \ldots, a_n$. Because a is also an action, the encoding is following:

$$\bigotimes_{\forall l \in \Gamma} l \multimap \bigotimes_{\forall r \in \Delta} r$$

The following algorithm shows how to create the action a from the sequence of the actions $a_1, \ldots, a_n$.

Algorithm 1:
`INPUT:` $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (from the encoding of the actions $a_1, \ldots, a_n$)
`OUTPUT:` Γ, Δ (from the encoding of the action a)

$\Gamma := \Delta := \varnothing$
for $i = 1$ **to** n **do**
    Λ:=$\Delta \cap \Gamma_i$
    Δ:=$(\Delta \setminus \Lambda) \cup \Delta_i$
    $\Gamma := \Gamma \cup (\Gamma_i \setminus \Lambda)$
endfor

Proposition 1. *The algorithm 1 is correct.*

Proof. The correctness of the algorithm 1 can be proved inductively in a following way:

For $n = 1$, it is easy to see that algorithm 1 works correctly for the only action a_1. The for cycle on lines 2-6 runs just once. On the line 3 is easy to see that $\Lambda = \varnothing$, because $\Delta = \varnothing$. Now it can be seen that $\Delta = \Delta_1$ (line 4) and $\Gamma = \Gamma_1$ (line 5), because $\Lambda = \varnothing$ and Γ and Δ before line 4 (or 5) are also empty.

Assume that algorithm 1 works correctly for k actions. From this assumption imply existence of an action a that is created by algorithm 1 (from some sequence of actions $a_1, \ldots, a_k$) after k steps of the for cycle in lines 2-6. Let s is a state and s' is a state which is obtained from s by applying the action a (without loss of generality assume that a can be applied on s). It is true that $s' = (s \setminus \Gamma) \cup \Delta)$. Let a_{k+1} is an action which is applied on the state s' (without loss of generality assume that a can be applied on s'). It is true that $s'' = (s' \setminus \Gamma_{k+1}) \cup \Delta_{k+1} = ((s \setminus \Gamma) \cup \Delta) \setminus \Gamma_{k+1}) \cup \Delta_{k+1}$. After $k+1$-th step of the for cycle (lines 2-6) an action a' is created (from the sequence $a_1, \ldots, a_k, a_k + 1$). When a' is applied on the state s, $s''' = (s \setminus \Gamma') \cup \Delta')$. From lines 3-5 can be seen that $\Gamma' = \Gamma \cup (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$ and $\Delta' = (\Delta \setminus (\Delta \cap \Gamma_{k+1})) \cup \Delta_{k+1}$. Now $s''' = (s \setminus (\Gamma \cup (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1})))) \cup ((\Delta \setminus (\Delta \cap \Gamma_{k+1})) \cup \Delta_{k+1})$. To finish the proof is needed to prove that $s'' = s'''$. $p \in s''$ iff $p \in \Delta_{k+1}$ or $p \in \Delta \wedge p \notin \Gamma_{k+1}$ or $p \in s \wedge p \notin \Gamma \wedge (p \notin \Gamma_{k+1} \vee p \in \Delta)$. $p \in s'''$ iff $p \in \Delta_{k+1}$ or $p \in \Delta \wedge p \notin (\Delta \cap \Gamma_{k+1})$ or $p \in s \wedge p \notin \Gamma \wedge p \notin (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$. It is easy to see that $p \in \Delta \wedge p \notin (\Delta \cap \Gamma_{k+1})$ is satisfied iff $p \in \Delta \wedge p \notin \Gamma_{k+1}$ is satisfied. $p \notin (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$ is satisfied iff $p \notin \Gamma_{k+1}$ or $p \in \Delta$ is satisfied. Now is clear that $s'' = s'''$.

□

The algorithm 1 works with planning problems encoded in propositional Linear Logic. The extension of the algorithm to predicate Linear Logic can be simply done with adding constraints symbolizing which of actions' arguments must be equal. This extension affect only the intersection $(\Delta_i \cap \Gamma_j)$.

Following algorithm for assembling actions into a single action is based on a paradigm divide and conquer which can support parallel implementation.

Algorithm 2:
`INPUT:` $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (from the encoding of the actions $a_1, \ldots, a_n$)
`OUTPUT:` Γ, Δ (from the encoding of the action a)

Function Assemble$(\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots \Delta_n)$:$\Gamma, \Delta$
  if $n = 1$ **then return**(Γ_1, Δ_1) **endif**
  Γ', Δ':=**Assemble**$(\Gamma_1, \ldots, \Gamma_{\lceil \frac{n}{2} \rceil}, \Delta_1, \ldots, \Delta_{\lceil \frac{n}{2} \rceil})$
  Γ'', Δ'':=**Assemble**$(\Gamma_{\lceil \frac{n}{2} \rceil + 1}, \ldots, \Gamma_n, \Delta_{\lceil \frac{n}{2} \rceil + 1}, \ldots, \Delta_n)$
  $\Lambda := \Delta' \cap \Gamma''$
  $\Gamma := \Gamma' \cup (\Gamma'' \setminus \Lambda)$
  $\Delta := \Delta'' \cup (\Delta' \setminus \Lambda)$
  return(Γ, Δ)
endFunction

Proposition 2. *The algorithm 2 is correct.*

Proof. The correctness of the algorithm 2 can be proved in a following way:

$n = 1$ — It is clear that assembling one-element sequence of actions (a_1) is equal to the action a_1 itself.

$n > 1$ — Let $a_1, \ldots a_n$ is a sequence of actions. In the lines 2 and 3 is the sequence split into two sequences $(a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n)$ and the algorithm 2 is recursively applied on them. Because $\lceil \frac{n}{2} \rceil < n$ when $n > 1$, is easy to see that the recursion will finitely terminate (it happens when $n = 1$). Now it is clear that a' and a'' are actions obtained by assembling the sequences $a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n$. These actions are assembled into a single action a at lines 4-6. The proof of correctness of this assembling is done in proof of proposition 1.

□

The algorithm 2 can be extended to predicate Linear Logic in a similar way like algorithm 1.

5 Related work

Previous research showed that planning problems can be encoded in many different formalisms. We consider the formalisms which are related to the logic. It was showed in [11] that planning problems can be solved by reduction to SAT. This approach brings very good results (for example SATPLAN [12] won last two International Planning Competitions (IPC) in optimal deterministic planning). However, there seems to be a problem with extending the SAT based planners to be able to solve planning problems with time and resources. Otherwise, Linear Logic (especially predicate Linear Logic) seems to be expressive enough for the encoding of planning problems with time and resources. The other logic formalism which can be used in connection to planning problems is Temporal Logic (especially Simple Temporal Logic). It is showed in [1] that Temporal Logic can be used to improve searching for plans. This approach is used in several planners (for example TALplanner [5] which had very good results in IPC 2002). Temporal Logic is good for representing relationships between actions, but Linear Logic is able to represent whole planning problems.

First ideas of solving planning problems via theorem proving in Linear Logic has been shown at [16, 10]. These papers showed that M?LL fragment of Linear Logic is strong enough to formalize planning problems. Another interesting work in this area is [4] and it describes a possibility of recursive applying of partial plans. Analyzing planning problems encoded in Linear Logic via partial deduction approach is described in [13].

One of the most important results was an implementation of a planning solver called RAPS [14] which in comparison between the most successful solvers in IPC 2002 showed very interesting results (especially in Depots domain). RAPS showed us that the research in this area can be helpful in searching for the good planning solver.

This paper extends the previous research in this area by fact that many optimizations of planning problems, which helps a planner to find a solution more quickly, can be also easily encoded in Linear Logic.

6 Future work

Next paragraphs show possible directions of our future research in this area.

One of possible directions how to implement the efficient algorithm for solving planning problems encoded in Linear Logic is using the connection between Linear Logic and Petri Nets. It is not difficult to see that the encoding of the planning problems is similar as an encoding of Petri Nets. The implementation of unfolding algorithm for reachability problem in Petri Nets for solving planning problems has been done by [9] and showed very good results. We will study the possibility of extending this algorithm in the way that the extended algorithm will support the encoding of planning problems in predicate Linear Logic.

There is a possibility of extension of the presented encodings of planning problems in Linear Logic into Temporal Linear Logic (to learn more about Temporal Linear Logic, see [18]). It seems to be very useful combination, because Temporal Logic can give us more possibilities for encoding of relationships between actions.

Another possible way of our future research is using of the encodings of the optimizations for transformation of planning domains. Transformed planning domains can be used with existing planners and can reach better results, because the presented optimizations of planning problems can help the planners to prune the search space.

At last, Linear Logic seems to be strong enough to encode planning problems with time and resources. Basic idea of this extension comes from the fact that time and resources can be also represented by linear facts. The usage of predicate Linear Logic seems to be necessary in this case.

7 Conclusion

The previous research showed that Linear Logic is a useful formalism for encoding many problems with expendable resources like planning problems, because in comparison to the other logics, Linear Logic seems to be strong enough to solve also planning problems with time and resources. This paper extended the previous research in this area by the fact that many optimizations of planning problems, which can help a planner to find a solution more quickly, can be encoded in Linear Logic without huge growth of the length of the encoding. Main advantage of this approach is that an improvement of the Linear Logic solver leads to improved efficiency of the planner based on Linear Logic.

8 Acknowledgements

We thank the reviewers for the comments. The research is supported by the Czech Science Foundation under the contract no. 201/07/0205 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

1. Bacchus F., Kabanza F.: Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 22:5-27. 1998.
2. Chrpa L. Linear Logic: Foundations, Applications and Implementations. *In proceedings of workshop CICLOPS 2006*. 110-124. 2006.
3. Chrpa L. Linear logic in planning. *In proceedings of Doctoral Consorcium ICAPS 2006*. 26-29. 2006.
4. Cresswell S., Smaill A., Richardson J. Deductive Synthesis of Recursive Plans in Linear Logic. *In proceedings of ECP 1999*. 252–264. 1999.
5. Doherty P., Kvanstrom J.: TALplanner: A temporal logic based planner. *AI Magazine* 22(3):95-102. 2001.
6. Ghallab M, Nau D., Traverso P. *Automated planning, theory and practice*. Morgan Kaufmann Publishers 2004. 2004.
7. Girard J.-Y. Linear logic. *Theoretical computer science* 50:1–102. 1987.
8. Girard J.-Y. *Linear Logic: Its Syntax and Semantics*. Technical report, Cambridge University Press. 1995.
9. Hickmott S., Rintanen J., Thiebaux S., White L. Planning via Petri Net Unfolding *In proceedings of IJCAI 2007*. 1904-1911. 2007
10. Jacopin E. Classical AI Planning as Theorem Proving: The Case of a Fragment of Linear Logic *In proccedings of AAAI 1993*. 62–66. 1993.
11. Kautz H. A., Selman B.: Planning as Satisfiability. *In proccedings of ECAI 1992*. 359-363. 1992.
12. Kautz H. A., Selman B., Hoffmann J.: SatPlan: Planning as Satisfiability. *In proccedings of 5th IPC 2006*.
13. Küngas P. Analysing AI Planning Problems in Linear Logic - A Partial Deduction Approach. *IN proceedings of SBIA 2004*. 52–61. 2004.
14. Küngas P. Linear logic for domain-independent ai planning. *Proceedings of Doctoral Consorcium ICAPS 2003*. 2003.
15. Lincoln P. Linear logic. *Proceedings of SIGACT 1992*. 1992.
16. Masseron M. Tollu C., Vauzeilles J. Generating plans in linear logic i-ii. *Theoretical Computer Science*. vol. 113, 349-375. 1993.
17. Oliet N. M., Meseguer, J. From petri nets to linear logic. *Springer LNCS 389*. 1989.
18. Tanabe M. Timed petri nets and temporal linear logic. *In proceedings of Application and Theory of Petri Nets*. 156-174. 1997

This article was processed using the LaTeX macro package with LLNCS style

Partial Model Completion in Model Driven Engineering using Constraint Logic Programming

Sagar Sen[1,2], Benoit Baudry[1], Doina Precup[2]

[1] IRISA, Campus de Beaulieu
35042 Rennes Cedex France
[2] School of Computer Science, McGill University,
Montreal, Quebec, Canada
{ssen,bbaudry }@irisa.fr , dprecup@cs.mcgill.ca

Abstract. In *Model Driven Engineering* a model is a graph of objects that conforms to a meta-model and a set of constraints. The meta-model and the constraints declaratively restrict models to a valid set. Models are used to represent the state and behaviour of software systems. They are specified in visual modelling environments or automatically synthesized for program testing. In such applications, a modeller is interested in specifying a partial model or a set of partial models which has a structure and associated properties that interests him/her. Completing a partial model manually can be an extremely tedious or an undecidable task since the modeller has to satisfy tightly-coupled and arbitrary constraints. We identify this to be a problem and present a methodology to solve (if a solution can be found within certain time bounds) it using *constraint logic programming*. We present a transformation from a partial model, its meta-model, and additional constraints to a constraint logic program. We solve/query the CLP to obtain value assignments for undefined properties in the partial model. We then complete the partial model using the value assignments for the rest of the properties.

1 Introduction

Model-driven engineering (MDE) [1] [2] [3] is an emerging trend that promises decrease in development time of complex software systems. For this, MDE makes extensive use of models. A model is specified in a modelling language which is described declaratively using a meta-model and a set of constraints. The Object Management Group (OMG) [4] standard for specifying a meta-model is Meta-object Facility (MOF) [5]. The constraints on the properties described in the meta-model are specified in a constraint language such as Object Constraint Language (OCL) [6].

Several tasks in MDE require building and manipulating models. Models are built in visual modelling environments [7] [8]. Models are automatically synthesized for testing [9] and verification [10]. The design space of models are explored for meeting requirements in embedded systems [11]. In all these cases, a modeller would like to specify a partial model that interests him/her. The task of completing a model manually could be extremely tedious since the modeller will have to satisfy a set of restrictions that are imposed by the meta-model and constraints (for instance, completing models in a visual modelling environment). We intend to automate this process.

In this paper we propose an approach where we use the idea that a partial model, its meta-model, and a set of constraints together result in a large set of constraints that can be represented as a *constraint logic program* (CLP). The CLP is solved/queried to obtain the missing elements in a partial model to give a complete model. We use the simple Hierarchical Finite State Machine (HFSM) modelling language to illustrate our approach. However, our approach is extensible to any modelling language. In Section 2 we describe meta-models, constraints on it and the resulting modelling language for specifying models. Section 3 presents the algorithm for transforming a partial model with its meta-model and constraints to a constraint logic program. In Section 4 we present an example based on the Hierarchical Finite State Machine modelling language. We conclude in Section 5.

2 Meta-models and Constraints of a Modelling Language

In this section we define the concept of a meta-model and constraints on it to specify a modelling language. We present the Hierarchical Finite State Machine (HFSM) modelling language with its meta-model and constraints as an illustrative example. The basic principles for specifying any modelling language follow the same set of procedures we use to discuss the HFSM modelling language.

2.1 Meta-models

A modelling language allows us to specify models. A model consists of objects and relationships between them. The *meta-model* of the modelling language specifies the types of all the objects and their possible inter-relationships. The type of an object is referred to as a *class*. The meta-model for the HFSM modelling language is presented in Figure 1 (a). The classes in the meta-model are **HFSM**, **Transition**, **AbstractState**, **State**, and **Composite**.

In this paper we use the Object Management Group (OMG) [4] standard Meta-object Facility (MOF) [5] for specifying a meta-model. MOF is a modelling language capable of specifying domain-specific modelling languages and its own meta-model. This property of MOF is known as *bootstrapping*. We use the visual language notation in MOF to specify the meta-model for the HFSM modelling language in Figure 1 (a).

Each class in the meta-model has *properties*. A property is either an *attribute* or a *reference*. An attribute is of primitive type which is either Integer, String, or Boolean. For instance, the attributes of the class **State** are isInitial and isFinal both of which are of primitive type Boolean. An example domain of values for the primitive attributes is given in Table 1. The String variable can be a finite set of strings or a regular expression that specifies a set of strings.

Describing the state of a class of objects with only primitive attributes is not sufficient in many cases. Modelling many real-world systems elicits the need to model complex relationships such as modelling that an object contains another set of objects or an object is related to another finite set of objects. This set of related objects is constrained by a *multiplicity*. A reference property of a class allows its objects to be related to a set of objects. For instance, an object of class **HFSM** contains **Transition** objects.

Table 1. Domains for Primitive Datatypes

Type	Domain
Boolean	$\{true\}, \{false\}$
Integer	$\{MinInt, .., -2\},\{-1\},\{0\},\{1\},\{2, ..MaxInt\}$
String	$\{null\},\{""\},\{". + "\}$

The multiplicity constraint for the relationship is shown at the relationship ends in Figure 1 (a). One **HFSM** object can contain * or arbitrary number of **Transition** objects. The reference names for either class are given at opposite relationship ends. The reference hfsmTransitions is a property of class **Transition** while reference transitions is a property of class **HFSM**.

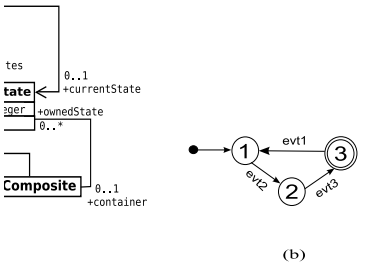


Fig. 1. (a) The Hierarchical Finite State Machine Meta-model (b) A Hierarchical Finite State Machine model in its Concrete Visual Syntax

A special type of relationship is that of *containment*. For instance, in Figure 1 (a) the **HFSM** class is the container for both **AbstractState** and **Transition** objects. The distinguishing visual syntax in comparison to other types of relationships is black rhombus at the container class end of the relationship.

Objects can inherit properties from other classes. The attributes and references of a class called a *super class* are inherited by derived classes. For instance, the classes **State** and **Composite** inherit properties from **AbstractState**. The visual language syntax to represent an inheritance relationship is distinguished by a triangle at the super class end of the relationship.

The visual language of MOF to specify a meta-model is expressive enough to specify a network of objects along with the domain of each of its properties. The multiplicity constraints impose constraints on the number of possible relationships between objects. However, most complex constraints that describe relationships between properties can easily and concisely be specified using a textual language for modelling constraints. In the following section we present examples of such constraints and talk about a language to specify them.

In our implementation the input meta-model for a modelling language is read in from a XML file containing the graph with the representation of the classes and the relationships between them. The cardinality constraints are generated and stored separately as Prolog clauses.

2.2 Constraints on Meta-models

Classes and the domain of its properties in a meta-model specify basic domain and multiplicity constraints on a model. However, some constraints that cannot be expressed directly within the meta-model are better specified in a textual language. For example, it is not possible to use multiplicities to express that a HFSM must contain only one initial state. This has to be defined in an external constraint that is attached to the meta-model.

There exist a number of choices for languages when it comes to expressing constraints on models. The OMG standard for specifying constraints on MOF models is the Object Constraint Language (OCL) [6]. The constraint that there must be only one initial state in a HFSM model is expressed in OCL as:-

$$context\ State\ inv:$$
$$State.allInstances() \rightarrow select(s|s.isInitial = True) \rightarrow size() = 1$$

The OCL is a high-level and complex constraints specification language for constraints. The evolving complexity in the syntax of OCL on the one hand makes it hard for one to specify its formal semantics on the other. We consider only a subset of OCL for manually specifying Prolog constraints on models.

We use the constraint logic programming language called ECLiPSe [12] (based on Prolog) to specify constraints on model properties. Predicates in ECLiPSe equips a search algorithm with results from automatic symbolic analysis and domain reduction in a model to prune its search space. Consider the model of a HFSM in Figure 1 (b). The ECLiPSe predicate that constrains the number of initial states is:

$$ModelisInitial = [HFSMStateObject1isInitial,$$
$$HFSMStateObject2isInitial,$$
$$HFSMStateObject3isInitial],$$
$$ModelisInitial :: [0..1],$$
$$ic\_global : occurrences(1, ModelisInitial, 1),$$

The $ModelisInitial$ list contains unique identifiers for $isInitial$ attributes of three state objects $StateObject1$, $StateObject2$, and $StateObject3$. The domain of all these identifiers in the list $ModelisInitial$ is binary. The predicate $occurences$ in the ic_global library of ECLiPSe states that the number of occurrences of the value 1 in the list is limited to 1. This means that only one of the variables in the list is assigned a value of 1. The choice of 1 for a variable automatically guides the compiler to perform domain reduction and reduce the domain of the rest of the variables in the list. The dynamically assigned domain for the other variables will be 0 and not $0, 1$. This feature drastically reduces the search space for the constraint satisfaction algorithm.

It is interesting to note that predicates in OCL are specified at the meta-model level of the modelling language while the constraints in ECLiPSe are generated for each model. In the next section we present how we generate ECLiPSe predicates for each model automatically.

In our implementation Prolog constraints are generated for a meta-model and stored in a prolog pl file. The constraints operate of lists of variables in the partial model.

3 Transformation: Partial Model, Meta-model, and Constraints to CLP

In this section we describe an algorithm to transform a partial model in a modelling language (meta-model and constraints) to a CLP. Keeping the structure static in the partial model we query the CLP to obtain value assignments for the properties in the model. The query process invokes a back-tracking algorithm to assign values to the variables such that all related predicates are satisfied. In this case the algorithm returns a *Yes*. If the back-tracking method fails to find a set of value assignments that can complete the model the algorithm returns a *No*. We start with describing a partial model in 3.1. In Section 3.3 we present the variables and functions that will be used in the algorithm. The algorithm itself is presented in Section 3.4.

3.1 Partial Model

A *partial model* is a set of objects described in a modelling language. The properties of the objects are either partially specified or they are not specified at all. What is specified is the domain of possible assignments for each and every property. This is derived from the meta-model that specifies the modelling language for the partial model objects.

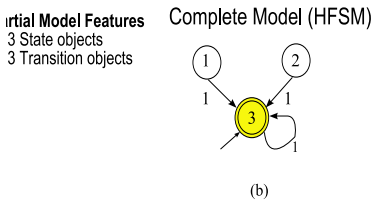


Fig. 2. (a) A Partial Model in HFSM (b) A Complete Model

In Figure 2 (a) we present a partial model that has 3 state objects and 3 transition objects in the HFSM language. Note that the labels, events, and other properties of the objects are not specified at all.

The domain of each property in the model is shown in Figure 3. We need to find a way to assign appropriate values for every property in the model so that it satisfies the structural requirements imposed by the meta-model and also additional constraints specified textually as CLP predicates.

Property	Domain
HFSM.StateObject1.isInitial	{True, False}
HFSM.StateObject1.isFinal	{True, False}
HFSM.StateObject1.label	{1,2,3,...,N}, where N is positive integer
HFSM.StateObject2.isInitial	{True, False}
HFSM.StateObject2.isFinal	{True, False}
HFSM.StateObject2.label	{1,2,3,...,N}, where N is positive integer
HFSM.StateObject3.isInitial	{True, False}
HFSM.StateObject3.isFinal	{True, False}
HFSM.StateObject3.label	{1,2,3,...,N}, where N is positive integer
HFSM.TransitionObject1.event	{1,2,3,4,5}
HFSM.TransitionObject2.event	{1,2,3,4,5}
HFSM.TransitionObject3.event	{1,2,3,4,5}
HFSM.StateObject1.incomingTransition	{HFSM.TransitionObject1.target,HFSM.TransitionObject2.target,HFSM.TransitionObject3.target}
HFSM.StateObject1.outgoingTransition	{HFSM.TransitionObject1.source,HFSM.TransitionObject2.source,HFSM.TransitionObject3.source}
HFSM.StateObject1.hfsmStates	{HFSM.states}
HFSM.StateObject2.incomingTransition	{HFSM.TransitionObject1.target,HFSM.TransitionObject2.target,HFSM.TransitionObject3.target}
HFSM.StateObject2.outgoingTransition	{HFSM.TransitionObject1.source,HFSM.TransitionObject2.source,HFSM.TransitionObject3.source}
HFSM.StateObject2.hfsmStates	{HFSM.states}
HFSM.StateObject3.incomingTransition	{HFSM.TransitionObject1.target,HFSM.TransitionObject2.target,HFSM.TransitionObject3.target}
HFSM.StateObject3.outgoingTransition	{HFSM.TransitionObject1.source,HFSM.TransitionObject2.source,HFSM.TransitionObject3.source}
HFSM.StateObject3.hfsmStates	{HFSM.states}
HFSM.states	{HFSM.StateObject1.hfsmStates,HFSM.StateObject2.hfsmStates,HFSM.StateObject3.hfsmStates}
HFSM.currentState	{HFSM.StateObject1.hfsmCurrentState, HFSM.StateObject2.hfsmCurrentState, HFSM.StateObject3.hfsmCurrentState }
HFSM.transitions	{HFSM.TransitionObject1.hfsmTransitions, HFSM.TransitionObject2.hfsmTransitions, HFSM.TransitionObject3.hfsmTransitions}
HFSM.TransitionObject1.target	{HFSM.StateObject1.incomingTransition,HFSM.StateObject2.incomingTransition,HFSM.StateObject3.incomingTransition}
HFSM.TransitionObject1.source	{HFSM.StateObject1.outgoingTransition,HFSM.StateObject2.outgoingTransition,HFSM.StateObject3.outgoingTransition}
HFSM.TransitionObject1.hfsmTransitions	{HFSM.transitions}
HFSM.TransitionObject2.target	{HFSM.StateObject1.incomingTransition,HFSM.StateObject2.incomingTransition,HFSM.StateObject3.incomingTransition}
HFSM.TransitionObject2.source	{HFSM.StateObject1.outgoingTransition,HFSM.StateObject2.outgoingTransition,HFSM.StateObject3.outgoingTransition}
HFSM.TransitionObject2.hfsmTransitions	{HFSM.transitions}
HFSM.TransitionObject3.target	{HFSM.StateObject1.incomingTransition,HFSM.StateObject2.incomingTransition,HFSM.StateObject3.incomingTransition}
HFSM.TransitionObject3.source	{HFSM.StateObject1.outgoingTransition,HFSM.StateObject2.outgoingTransition,HFSM.StateObject3.outgoingTransition}
HFSM.TransitionObject3.hfsmTransitions	{HFSM.transitions}

Fig. 3. Domain of Properties in the Partial Model

Which value is taken by a property will be decided when we transform the partial model to a CLP and solve it. This transformation process and solution is discussed in the next sections.

3.2 CLP as a Common Language

Requirements in MDE come from many sources. In this paper we focus on three sources of requirements.

1. The meta-model is a diagram specified in a meta-modelling language such as MOF
2. The constraints are textually specified in OCL or directly in CLP
3. The partial model is specified either diagrammatically or textually as an instance of the meta-model.

There also could be many other sources of requirements such as pre-conditions and post-conditions from programs that manipulate models and an objective function that is soft constraint on the properties of a model. These requirements are specified in different languages that are either graphical or textual and are usually expressed in the most convenient specification language such as OCL or MOF. Combining the declarative specifications from different sources into one *common language* is necessary for analysis on a common platform. The following section presents an algorithm that syntactically transforms constraints from various sources (the three that we consider) into a CLP.

3.3 Preliminaries

The input to the transformation consists of a partial model, the meta-model and the constraints specification of a modelling language. In Table 2 we list out symbols and their descriptions we use for describing the algorithm in the next section.

Variable	Description
M	Partial model
$M.properties$	Set of all properties(attributes and references) in M
$Model$	List of CLP variables for all properties in M
$Model\_property$	List of CLP variables for a relationship from a reference called $property$
$property.lowerBound$	Lower bound on multiplicity of a relationship from a reference called $property$
$property.upperBound$	Upper bound on multiplicity of a relationship from a reference called $property$
$property\_to\_reference$	A 0/1 variable for a relationship from a $property$ to one of its $reference$s
$reference\_to\_property$	A 0/1 variable for a relationship from a $reference$ to its $property$
C	Set of non-meta-model constraint predicates
$Model\_constraint$	List of variables that are constrained by the constraint $constraint$
$constraint.predicate$	Name of the CLP predicate for a constraint identifier
$constraint.parameters$	String of additional comma separated parameters for a predicate

Table 2. Symbols we use in the model to CLP transformation algorithm

The transformation algorithm uses some handy functions which we list and describe in Table 3.

Function	Description
$isAttribute(property)$	Returns 1 if the property is an attribute, otherwise 0
$isReference(property)$	Returns 1 if the property is an reference, otherwise 0
$domainSet(property)$	Returns domain set of a property
$relatedVariables(constraint)$	Returns a set of variables that are constrained by $constraint$

Table 3. Functions we use in the model to CLP transformation algorithm

The CLP we generate uses predicate libraries in ECLiPSe. The *ic* library contains predicates for hybrid integer/real interval arithmetic constraints. It also includes a solver for these predicates. The *ic_global* library contains predicates for specifying various global constraints over intervals. The standard CLP predicates synthesized in the code by the algorithm are described in the book [12]. In ECLiPSe each predicate is separated by a ',' indicating that it is a logical and of the predicates.

3.4 Algorithm

We describe the algorithm to transform a partial model to a CLP in six phases. In Phase 1 we synthesize CLP code to import library predicates *ic* and *ic_global*. We then synthe-

size code to start the CLP procedure *solveModel*. In Phase 2 the algorithm synthesizes the list, *Model*, of all the model properties in the partial model.

In Phase 3 the algorithm generates code to assign a domain of values to the attributes and relationships. The function *domainSet* returns a set of possible values for each attribute. For each reference variable, *property*, in the model the *domainSet* returns a set of possible related references in the model. A binary relationship variable $property\_to\_reference$ is created and has the integer domain $[0..1]$. If a relationship variable $property\_to\_reference$ is set to 1 by the back-tracking algorithm then its corresponding relationship $reference\_to\_property$ is also set to 1 by the constraint $property\_to\_reference = reference\_to\_property$. A list $Model\_property$ stores all relationship variables from the property. The multiplicity constraints on a relationship are inserted in the CLP using the *sum* predicate on the list $Model\_property$. The number of possible relationships should be between $property.lowerBound$ and $property.upperBound$

Now that constraints on domains of attributes and basic multiplicity are already included we go on to Phase 4 of the algorithm. In this phase we create a dependent variables list $Model\_constraint$ for each $constraint$. In Phase 5 we insert the CLP predicate $constraint.predicate$ for the $constraint$ with parameters $Model\_constraint$ and additional parameters $constraint.parameters$, if necessary.

In the final Phase 6 we insert the predicate $labeling(Model)$ which performs back-tracking search on the CLP variables. To execute the synthesized CLP we make a query : $-solveModel(Model)$ in ECLiPSe after compiling the program. The result of the query is the set of value assignments that satisfies the constraints.

The pseudo code for the transformation algorithm is presented as follows :-

Phase 1: Import predicate libraries and start the CLP procedure

```
1: print ": −lib(ic)"
2: print ": −lib(ic_global)"
3: print "solveModel(Model) : −"
   Phase 2: Synthesize CLP variables for all model properties
4: printString ←"Model = ["
5: for property in M.properties do
6:    if isAttribute(property) then
7:       printString ← printString + property+","
8:    else if isReference(property) then
9:       for reference in domainSet(property) do
10:         printString ← printString + property+"_to_"+reference+","
11:      end for
12:   end if
13: end for
14: print printString.rstrip(',') + "];" {rstrip strips printString of the last ,}
   Phase 3: Assign property domains and multiplicity constraints
15: for property in M.properties do
16:   if isAttribute(property) then
17:      print property+"::"+"["+domainSet(property)+"],"
```

```
18:    end if
19:    if isReference(property) then
20:       printString ← "Model_property = ["
21:       for reference in domainSet(property) do
22:          printString ← printString + property+"_to_"+reference+","
23:       end for
24:    end if
25:    print printString.rstrip(',')+"];"
26:    print "Model_"+property+":: [0..1]"
27:    print "sum(Model_"+property+") >="+ property.lowerBound+","
28:    print "sum(Model_"+property+") =<"+property.upperBound+","
29: end for
30: for reference in domainSet(property) do
31:    print property+"_to_"+reference = reference+"_to_"+property+","
32: end for
    Phase 4: Create dependent variable lists
33: for constraint in C do
34:    printString ← "Model_constraint=["
35:    for variable in relatedVariables(constraint) do
36:       printString ← printString + variable+","
37:    end for
38:    print printString.rstrip(',')
39: end for
    Phase 5: Impose constraints on lists
40: for constraint in C do
41:    print constraint.predicate+"("+constraint.parameters+","+Model_constraint+"),"
42: end for
    Phase 6: Solve Model
43: print "labeling(Model)."
```

4 An Example

We consider the partial model shown in Figure 2 (a) as our example. The simple HFSM has three states and three transition objects. The algorithm presented in Section 3.4 takes the partial model and the meta-model and constraints of the HFSM modelling language as input. The output is a CLP program. We do not present the structure of the CLP program due to space limitations.

When the CLP is queried or solved the *labeling* predicate invokes a backtracking solver that selects values from the domain of each property such that the conjunction of all the constraints is satisfied. Executing this predicate returns a *Yes* if the model is satisfied else it returns a *No*. If a *Yes* is returned then the compiler also prints a set of valid assignments for the variables. Assigning these values to the partial model results in a complete model. In the simplest case, when no property of the partial model is given the result is the complete model shown in Figure 2 (b). However, when certain values for properties are already specified the synthesized CLP program has new domains for

these properties. These domains have only one value restricting the solver to choose just one assignment.

5 Conclusion

The idea of completing a partially specified model can be extended in many ways. For example, when we want to test a program it is important that we test it with input models that lead to covering most of its execution paths. Specific execution paths can be reached when some partial information about reaching it is already available. Such partial information can be specified in the form of a partial model with value assignments to some properties. The task of obtaining values of other properties is not directly consequential to identifying an error in a program but is necessary to produce a valid input model. Our method to complete such a partial model can be extended to large scale generation of test models to perform *coverage-based* testing of programs that manipulate models.

We solve a CLP using back-tracking as a means to achieve constraint satisfaction. At the moment we donot select different results based on back-tracking. The difference in the result is due to the knowledge already given in the partial model. Nevertheless, simply replacing a constraint satisfaction algorithm such as back-tracking with a constraint optimization algorithm opens many possibilities. One such possibility is to synthesize a model that optimizes an objective by meeting soft constraints or requirements including hard constraints specified in the modelling language. In the domain of software engineering such a framework can be used to synthesize customized software.

References

1. Schmidt, D.C.: Model-driven engineering. IEEE Computer **39**(2) (2006) 25–31
2. France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-driven development using uml 2.0: Promises and pitfalls,. In: IEEE Computer Society Press. (2006)
3. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap,. In: FOSE '07: 2007 Future of Software Engineering. (2007)
4. OMG: OMG Home page. http://www.omg.org (2007)
5. OMG: MOF 2.0 Core Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04 (2005)
6. OMG: The Object Constraint Language Specification 2.0, OMG Document: ad/03- 01-07 (2007)
7. de Lara Jaramillo, J., Vangheluwe, H., Moreno., M.A.: Using meta-modelling and graph grammars to create modelling environments. Electronic Notes in Theoretical Computer Science with editors Paolo Bottoni and Mark Minas **72** (2003) 15
8. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. Computer (2001) 44–51
9. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon., Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proceedings of ISSRE'06, Raleigh, NC, USA (2006)
10. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)

11. Neema, S., Szitpanovits, J., Karsai, G.: Constraint-based design space exploration and model synthesis. In: Proceedings of EMSOFT 2003, Lecture Notes in Computer Science. Number 2855 (2003) 290–305
12. Apt, K.R., Wallace, M.G.: Constraint Logic Programming with ECLiPSe. Cambridge University Press (2007)

Dynamic Parser Cooperation for Extending a Constrained Object-Based Modeling Language

Ricardo Soto[1,2] and Laurent Granvilliers[1]

[1] LINA, CNRS, Université de Nantes, France
[2] Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso, Chile
{ricardo.soto,laurent.granvilliers}@univ-nantes.fr

Abstract. In this paper, we present a simple description language for extending the syntax of a modeling language for constrained objects. The use of this description language will be explained by means of a practical example. The process of extending the modeling language involves an automatic update of the parsing system which will be detailed. Finally, the architecture for supporting this extension process will be presented.

1 Introduction

CP (Constraint programming) allows to describe systems in terms of variables, relations between the variables called constraints, and to calculate values of the variables satisfying the constraints. This is a general approach to tackle constraint-based problems.

In the past years, several programming languages and libraries were designed for CP and CLP (Constraint Logic Programming). For instance, ECLiPSe [17], ILOG SOLVER [12], and OZ [14]. In these approaches, a host language is used to state control operations, a constraint language is used for modeling in terms of variables and constraints, and a strategy language may be used to tune the solving process.

The expertise required by CP languages led to the development of modeling languages such as OPL [16]. Here, a higher-level of abstraction is provided. There is no need for dealing with operational concerns of the host language. The user states the model and the system solves it by means of a fixed underlying solver.

A recent concern is to separate the modeling language from the underlying solver [13, 8]. To this end, a three-layered architecture is proposed, including a modeling language, a solver, and a middle tool mapping models (with a high-level of abstraction) to executable solver code (with a low-level of abstraction). This architecture gives the possibility to plug-in new solvers and to process a same model with different solvers.

An important inconvenience of this architecture is the non-existence of a mechanism for updating the modeling language. For instance, if a new functionality such as a new method, predicate or global constraint is added in the solver, the unique way to use it from the modeling layer is to update the grammar of

the modeling language and to recompile it by hand. Likewise, the mapping tool needs to be modified. The translation of the new functionality from the modeling language to the solver language must be included.

In this paper, we present a simple description language to extend the syntax of a modeling language, in order to make the architecture adaptable to further upgrades of the solvers. In addition we present an interesting parsing system for handling efficiently this extension process.

The extension language has been designed as part of the COMMA [2] system, a three-layered architecture for modeling constrained objects (objects subject to constraints on their attributes [15]). In this architecture models can be translated to three native solver models: ECLiPSe [17], Gecode/J [3] and RealPaver [10].

The COMMA modeling language is based on the constrained object modeling paradigm [11]. Indeed, the COMMA language is built from a combination of a constraint language and an object-oriented framework. In this work, we will focus on extending just the constraint language of COMMA. We see no apparent necessity for extending the object-oriented framework.

The outline of this paper is as follows. In Section 2 we give an overview of the COMMA language. Section 3 describes the extension language. The parsing system is presented in Section 4. The process of updating the architecture is described in Section 5, followed by the related work and conclusions.

2 COMMA Overview

In order to explain the means of extensions and the way to use it, let us first show the components of a COMMA model using the Perfect Squares Problem [9]. The goal of this problem is to place a given set of squares in a square area. Squares may have different sizes and they must be placed in the square area without overlapping each other.

A COMMA model is represented by a set of classes. Each class is defined by attributes and constraints. Attributes may represent decision variables or constrained objects. Decision variables must be declared with a type (Integer, Real or Boolean). Constants are given in a separate data file. A set of constraint zones can be encapsulated into the class with a given name. A constraint zone can contain constraints, loops, conditional statements, optimization statements, and global constraints. Loops can use loop-variables which do not need to be declared (`i` and `j` in the example). There is no need for object constructors to state a class, direct variable assignment can be done in the constraint zone.

Figure 1 shows a COMMA model for the Perfect Squares Problem. Three constant values are imported from an external data file called `PerfectSquares.dat`, `sideSize` represents the side size of the square area where squares must be placed, `squares` is the quantity of squares (8) to place and `size` is an array containing the square sizes. At line 3, the definition of the class begins, `PerfectSquares` is the name given to this class. Then, two integer arrays of decision variables are defined, which represent respectively the x and y coordinates of the square area. So, x[2]=1 and y[2]=1 means that the second of the eight squares must be placed

in row 1 and column 1, indeed in the upper left corner of the square area. Both arrays are constrained, the decision variables must have values into the domain `[1,sideSize]`.

```
//data
sideSize:=5;
squares:=8;
size:=[3,2,2,2,1,1,1,1];

1.  import PerfectSquares.dat;
2.
3.  class PerfectSquares {
4.    int x[squares] in [1,sideSize];
5.    int y[squares] in [1,sideSize];
6.
7.    constraint inside {
8.      forall(i in 1..squares) {
9.        int x[i] <= sideSize - size[i] + 1;
10.       int y[i] <= sideSize - size[i] + 1;
11.     }
12.   }
13.
14.   constraint noOverlap {
15.     forall(i in 1..squares) {
16.       for(j:=i+1;j<=squares;j++) {
17.         int x[i] + size[i] <= x[j] or
18.             x[j] + size[j] <= x[i] or
19.             y[i] + size[i] <= y[j] or
20.             y[j] + size[j] <= y[i];
21.       }
22.     }
23.   }
24.
25.   constraint fitArea {
26.     int (sum(i in 1..squares) (size[i]^2)) = sideSize^2;
27.   }
28. }
```

Fig. 1. A COMMA model for the Perfect Squares Problem

At line 7, a constraint zone called `inside` is declared. In this zone a `forall` loop contains two constraints to ensure that each square is placed inside the area, one constraint about rows and the other about columns. Due to extensibility requirements of the language, constraints must be typed. In fact, the control engine needs to recognize the kind of constraints before sending it to the correct parser (see Section 4). The constraint `noOverlap` declared at line 14 ensures that

two squares do not overlap. The last constraint called `fitArea` ensures that the set of squares fits perfectly in the square area.

3 Extending COMMA

In this section we continue with the same example. Let us consider that a programmer adds to the Gecode/J solver two new built-in functionalities: a constraint called `noOverlap` and a function called `sumArea`. The constraint `noOverlap` will ensure that two squares do not overlap and `sumArea` will sum the square areas of a square set. In order to use these functionalities we can extend the syntax of the constraint language by defining a new file (called extension file) where the rules of the translation are described.

This file is composed by three blocks (see Figure 2): an `Option` block, a `Function` block, and a `Relation` block. In the `Option` block we state the domain and the solver where the new functionalities will be defined, in this case the integer domain (`int`) and the Gecode/J solver are selected. Consequently the integer parser will be updated automatically to include these new functionalities (see Section 5). Functionalities involving more than one domain must be included in the mixed parser.

In the `Function` block we define the new functions to add. The grammar of the rule is as follows: ⟨*COMMA-code*⟩ `(` ⟨*input-parameters*⟩ `) -> "` ⟨*solver-code*⟩ `";`

```
Option {
  domain: int;
  solver: Gecode/J;
}
Function {
  sumArea(s[]) -> "sumArea($s$);";
}
Relation {
  noOverlap(x[],y[],size[],squares)
           -> "noOverlap($x$, $y$, $size$, $squares$);";
}
```

Fig. 2. Extension for Gecode/J

In the example the COMMA code is `sumArea`. This is the name which will be used to call the new function from COMMA. The input parameter of the new COMMA function is an array (`s[]`). Finally, the corresponding Gecode/J code is given to define the translation. The new function will be translated to `sumArea(s);`. This code calls the new built-in method from the solver file. The translator must recognize the correspondence between input parameters in COMMA and input parameters in the solver code. Therefore, variables must be

tagged with $ symbols. In the example, the input parameter of the COMMA function will be translated as the input parameter in the Gecode/J function.

In the `Relation` block we define the new constraints to add. We use the same grammar as for functions. In the example, a new constraint called `noOverlap` is defined, it receives four parameters. The translation to Gecode/J is given. Once the extension file is completed, it can be called by means of an import statement. The resultant COMMA model using extensions is shown in Figure 3.

```
1.  import PerfectSquares.dat;
2.  import PerfectSquaresGecode.ext;
3.
4.  class PerfectSquares {
5.    int x[squares] in [1,sideSize];
6.    int y[squares] in [1,sideSize];
7.
8.    constraint placeSquares {
9.      forall(i in squares) {
10.        int x[i] <= sideSize - size[i] + 1;
11.        int y[i] <= sideSize - size[i] + 1;
12.     }
13.     int noOverlap(x,y,size,squares);
14.     int sumArea(size) = sideSize^2;
15.   }
16. }
```

Fig. 3. Using extensions in the Perfect Squares Problem

4 Dynamic Parser Cooperation

The COMMA system is written in Java and the ANTLR [1] tool has been used for generate lexers, parsers and tree walkers. The system is supported by a three-layered architecture: a modeling layer, a mapping layer and a solving layer (see Figure 4). The compiling system is composed by three compilers. One for the COMMA language, one for the data and another for the extension files. This system is the basis of the mechanism to extend the constraint language.

The COMMA compiler (see Figure 6) is composed of one parser per constraint domain (Integer, Real, Boolean and Objects), one parser for constraints involving more than one domain (Mixed parser) and one base parser for the rest of the language (classes, import and control statements).

In order to get the abstract syntax tree (AST) from the parsing process, several cooperations between these parsers are performed at running time. A control engine manages this cooperation by sending each line of the COMMA model to the correct parser. Lines are syntactically checked by the parser and

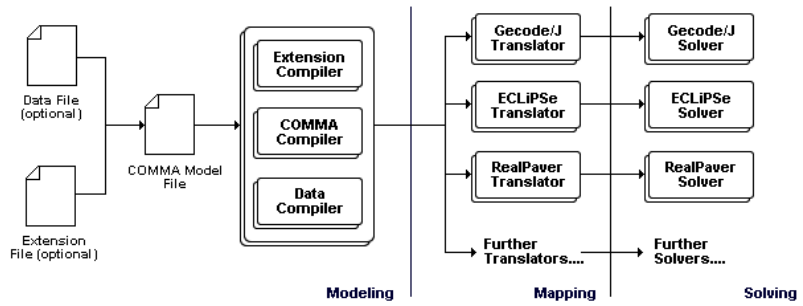


Fig. 4. Architecture of COMMA

then transformed into an AST which is returned to the control engine. The control engine takes this AST and attach it to the AST of previously parsed lines. Let us clarify this process by means of a simple example.

```
1.  class Coop {
2.    int  a;
3.    real b;
4.    constraint ex {
5.      int  a > 2;
6.      real b < 3.5;
7.    }
8.  }
```

Fig. 5. Attributes and constraints from different domains

Figure 5 shows a COMMA class called `Coop` which has attributes and constraints from different domains. The parsing process of this file is as follows: At line `1` the class is declared. This is part of the base language, so the base parser builds the corresponding AST for this line. Then, the control engine detects at line `2` an `int` type, this line is sent to the integer parser which builds the corresponding AST. The control engine takes this AST and attach it at the end of the previous AST. The same process is repeated for lines `3` and `4`. In order to send the constraints to the correct parser they also need to be typed. The constraint at line `5` is sent to the integer parser and then attached to the previous AST. The same process is performed for the rest of the lines.

Once the AST of the model is complete, a semantic checking is performed by means of two tree walkers which check types, class names, variable names, inheritances and compositions. Then, the AST is transformed into a Java object storing the model information, data information and extensions information

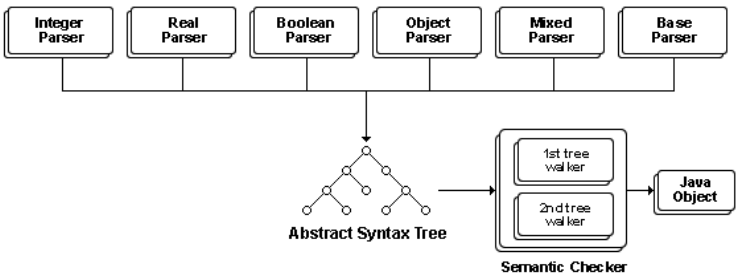


Fig. 6. The compiling process

using efficient representations. Finally, this Java object is translated to the executable solver file.

The independence of parsers has been done for three reasons. (1) It gives us the adequate modularity to easily maintain the parsing engine. (2) It avoids us to recompile the base parser (and parsers not involved in the extension) each time a new extension is added. This leads to a faster extension process since it is not necessary to update and recompile the whole language, we recompile just the updated domain. (3) This is a necessary condition to avoid ambiguities between identifier tokens that may arise from new extensions added. For instance, the same function defined for two different domains.

5 Updating the architecture

A control engine is able to automatically update the necessary parsers when a new relation or function is added as an extension. The process is as follows: when a new extension file is detected by the base parser in a model, the extension compiler is called. The extension file is parsed, and then translated to an ANTLR grammar. This grammar is merged with the previous domain grammar to generate a new grammar from which the ANTLR tool generates the parser in Java code. The new parser for the updated domain is compiled and then it replaces the previous domain parser (See Figure 7).

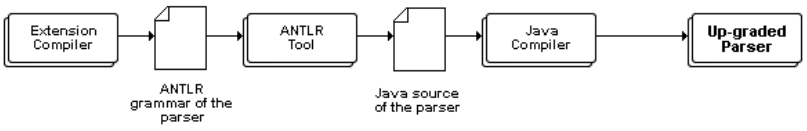


Fig. 7. The extension process

The control engine adds the new tokens to a table of symbols. The rules of translation are stored in a XML file. This file is used to perform the translation of the new functionalities, from the COMMA language to the solver file. The control engine manages ambiguities by checking the new tokens of the extension with the existing tokens in the table of symbols.

6 Related Work

Extensibility has been studied widely in the area of programming languages. Language extensions can define new language constructs, new semantics, new types and translations to the target language. Many techniques support each of these extensions. Some examples are syntactic exposures [4], hygienic macro expansions [6] and adaptable grammars [5].

These approaches are in general dedicated to extend the whole syntax of a language, consequently they provide a bigger framework than we need. For this reason we have chosen term rewriting [7] as the base of our extension language. This technique allows one to define a clear correspondence between two sets of terms, exactly as we use in our extension language: The initial terms are transformed into the target language terms (⟨*initial-terms*⟩ -> ⟨*target-language-terms*⟩).

As far as we know, this is the first attempt to make syntax-extensible a modeling language for constraint-based problems.

7 Conclusion and Future Work

We have shown by means of a practical example how the constraint language of COMMA can be extended using a simple description language. The process of extending the constraint language is handled by a control engine and a set of independent parsers. The parser independence provides us the adequate modularity to avoid recompiling the whole language each time a new extension is added. This leads to a faster extension process since just the updated domain is recompiled.

The work done in this extension language may be improved adding customizable functions. For instance, functions for which the priority and the notation (prefix, infix, postfix) can be defined, such as math operators (+,*,-,/). An extension manager may be useful to control which functionalities could be eliminated or maintained.

References

1. *ANTLR Reference Manual*. http://www.antlr.org.
2. *COMMA Modeling Language*. http://www.inf.ucv.cl/~rsoto/comma.
3. *Gecode System*. http://www.gecode.org.
4. Shigeru Chiba. A metaobject protocol for c++. In *OOPSLA*, pages 285–299, 1995.

5. Henning Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, 1990.
6. William D. Clinger and Jonathan Rees. Macros that work. In *POPL*, pages 155–162, 1991.
7. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.
8. Alan M. Frisch et al. The design of essence: A constraint language for specifying combinatorial problems. In *IJCAI*, pages 80–87, 2007.
9. I. Gambini. *Quant aux carrés carrelés*. PhD thesis, L'Université de la Méditerranée aix-Marseille II, 1999.
10. L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.
11. B. Jayaraman and P.Y. Tambay. Constrained Objects for Modeling Complex Structures. In *OOPSLA*, Minneapolis, USA, 2000.
12. J.F. Puget. A C++ implementation of CLP. In *SCIS*, Singapore, 1994.
13. Reza Rafeh, Maria J. García de la Banda, Kim Marriott, and Mark Wallace. From zinc to design model. In *PADL*, pages 215–229, 2007.
14. G. Smolka. The oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. 1995.
15. R. Soto and L. Granvilliers. Exploring the canopy of constraint modeling languages. In *Constraint Modelling and Reformulation Workshop held in conjunction with CP 2006*, pages 88–101, Nantes, France, 2006.
16. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
17. M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming, 1997.

Constraint-Based Examination Timetabling for the German University in Cairo

Slim Abdennadher and Marlien Edward

German University in Cairo, Department of Computer Science,
5th Settlement New Cairo City, Cairo, Egypt
slim.abdennadher@guc.edu.eg,marlien.nekhela@student.guc.edu.eg
http://cs.guc.edu.eg

Abstract. Constraint programming is mainly used to solve combinatorial problems such as scheduling and allocation, which are of vital importance to modern business. In this paper, we introduce the examination timetabling problem at the German University in Cairo. As manual generation of exam schedules is time-consuming and inconvenient, we show how this problem can be modeled as a constraint satisfaction problem and can be solved using SICStus Prolog with its powerful constraint library.

1 Introduction

Timetabling problems are real life combinatorial problems concerned with scheduling a certain number of events within a specific time frame. If solved manually, timetabling problems are extremely hard and time consuming and hence the need to develop tools for their automatic generation [4].

The German University in Cairo (GUC) started in Winter 2003. Each year the number of students increases and the timetabling problem whether for courses or examinations becomes slightly more difficult. For the past four years, schedules for both courses and exams has been generated manually. The process involves attempting to assign courses/exams to time slots and rooms while satisfying a certain number of constraints. Processed manually the course timetabling took on average three weeks, while the examination timetabling a week. With the growing number of students, the GUC timetabling problems have become too complex and time-consuming to be solved manually.

In this paper, we will only address the examination timetabling problem. Currently, at the GUC, there are 140 courses, 5500 students, and around 1300 seats for which examinations should be scheduled. The examination schedules are generated twice per semester. Once for the midterm examinations, which occur in the middle of the semester and are distributed over a one week period; and again during the final examinations, which occur at the end of the semester and are distributed over a two weeks period.

The examination timetabling problem at the GUC is non-interruptible and disjunctive. It is non-interruptible since once an examination starts it can not be

paused and resumed later. It is disjunctive since an examination room can only contain one exam at a specific time. A solution to the examination timetabling problem aims at allocating a time slot and a room to every examination while satisfying a given set of constraints.

Several approaches have been used to solve such combinatorial problems [6], e.g. using an evolutionary search procedures such as ant colony optimization [3], or using a combination of constraint programming and local search techniques [5], or using graph coloring approach [2].

In this paper, we present an ongoing research to develop a constraint-based system to solve the examination timetabling problem at the GUC. Constraint programming is a problem-solving technique that works by incorporating constraints into a programming environment [1]. Constraints are relations which specify the domain of solutions by forbidding combinations of values. Constraint programming draws on methods from artificial intelligence, logic programming, and operations research. It has been successfully applied in a number of fields such as scheduling, computational linguistics, and computational biology.

In this paper, we show how the GUC examination timetabling problem can be formalized as a constraint satisfaction problem (CSP) and implemented by means of specialized constraint solving techniques that are available in a constraint logic programming language (CLP) like SICStus Prolog. In solving the problem, we have to deal with two types of constraints, hard constraints and soft constraints. Hard constraints are constraints that must be satisfied. They represent regulations for examinations and resource (rooms) constraints. Soft constraints are constraints that should be satisfied but can be violated. Soft constraints can for example represent wishes of lecturers and students.

The paper is organized as follows. The next section introduces the GUC examination timetabling problem and its requirements. Then we show how the problem can be modeled as a CSP problem and how we implemented the different constraints in SICStus Prolog. Finally, we conclude with a summary and future work.

2 Problem Description and Requirements

The GUC examination timetabling problem aims at producing examination schedules for all courses given in a particular term. The GUC has four distinct faculties where each faculty offers a variety of majors. Some of the majors offer the same courses which makes examination scheduling for these courses interdependent.

Students are enrolled in the different courses according to their major. Students in every major are divided into student groups depending on the courses they take. This grouping of students taking the same stream of courses simplifies the scheduling task since a large number of students can be dealt with as a single entity with a certain number of students.

The examination slots at the GUC differ from the midterms to the final examinations. Usually the midterm examinations are distributed over a one week

period where the days are divided into four examination slots and the duration of each examination slot is 2 hours. The final examination period consist of two weeks with three examination slots per day where each examination slot is 3 hours.

The GUC examination timetabling problem consists of a set of courses and a set of students that need to be scheduled within limited resources of time and space. The task is to assign the student groups to examination slots and thus produce examination schedules for the midterm and final examinations. Since scheduling of the student groups from the different faculties is interdependent as time, space and specific courses are shared, schedules for the different faculties can not be produced separately and thus a single schedule for the whole university must be generated.

The GUC examination timetabling problem can be performed in two phases. In the first phase, the different course examinations are assigned to time slots while taking into account only the total number of students enrolled in the courses and the total number of available seats. In the second phase, after producing the examination schedule for all student groups, distribution of students among the rooms is done. In this paper, we will only model the first phase as a constraint satisfaction problem and solve it using constraint programming. The second phase does not require constraint techniques. A simple algorithm was designed and implemented using Java to distribute the students among the examination rooms and seats. In both phases, the different constraints that are imposed by the regulations of the GUC Examination Committee have to be taken into account. Dividing the examination scheduling problem into two parts reduces the complexity since the number of constraints that must be satisfied per phase is decreased. This division is feasible since the total number of seats is considered in the course scheduling (first) phase.

When assigning an examination to a certain time slot, several constraints must be considered.

Time Clash Constraint: A student group cannot have more than one examination at the same time.

Semester Clash Constraint: Student groups from the same major but in different semesters cannot have examinations at the same time. This is needed to enable students who are re-taking a course from a previous semester due to poor performance to sit for the course examination.

Core Exam Constraint: A student group cannot take more than one core examination per day.

Exam Restriction Constraint: A student group cannot have more than a certain number of examinations per day.

Difficulty Constraint: A student group cannot have two difficult examinations in two consecutive days (difficulty rating for a course is assigned by the course lecturer and students).

Capacity Constraint: The total number of students sitting for an examination in a certain time slot cannot exceed a predefined limit. This constraint ensures that the total number of students sitting for an examination at a

certain time slot do not exceed the total number of available seats in the university campus.

Pre-assignment Constraint: An examination should be scheduled in a specific time slot.

Unavailability Constraint: An examination should not be scheduled in a specific time slot.

3 Modeling the Problem as a Constraint Satisfaction Problem

Constraint satisfaction problems have been a subject of research in Artificial intelligence for many years [7].

A constraint satisfaction problem (CSP) is defined by a set of variables each having a specific domain and a set of constraints each involving a set of variables and restricting the values that the variables can take simultaneously.

The solution to a CSP is an assignment that maps every variable to a value. We may want to find just one solution, all possible solutions or an optimal solution given some objective function defined in terms of some or all variables. Here we are speaking about a constraint satisfaction optimization problem.

In modeling the GUC examination timetabling problem as a CSP we use the following notations :

- n is the number of courses to be scheduled.
- $D = \{1, \ldots, m\}$ indicates a set of m time slots.
- $T = \{T_1, \ldots, T_n\}$ represents the time slot variables for all examinations, where T_i indicates the time slot assigned to an examination i.
- l is the number of all student groups in the university.
- The domain of every variable T_i is between 1 and m.

m is the number of available time slots and can be calculated by multiplying the number of days of the examination period by the number of slots per day. For example, if the examination period consists of 6 days and every day consists of 3 time slots therefore the available number of time slots is 18.

Different constraints are applied on T_i eliminating the different values each variable can take and thus reducing the domains of every variable to improve the efficiency of the search thus reaching a solution in an optimal time.

Now we describe the requirements of our problem in terms of SICStus prolog using the constraint solver over finite domains (clpfd).

Time clash constraint stating that no student can have more than one exam at the same time slot can be enforced using the `all_different/1` global constraint. We have to constrain the l sets of variables representing the courses for every group of students to be different.

Let `Tgroup = [T1,...,Tj]` be a list of variables that represent examination time slots of a specific group then applying the `all_different/1` constraint on this list would ensure that all exams for that group are held in different time slots. Note that `Tgroup` is a subset of `T`

```
all_different(Tgroup)
```

In order to abide to the semester clash constraint, we have to take into consideration that all examinations of the same major should be scheduled at different time slots. Let `Y` be the number of courses of a certain major. `Tmajor = [T1,...,TY]` is a list of variables representing examination time slots for these courses. Applying the `all_different/1` constraint on these variables will ensure that all these exams are assigned to different time slots.

```
all_different(Tmajor)
```

Difficulty constraint has to be fulfilled in order to produce a valid examination schedule. We cannot schedule two difficult exams consecutively. Knowing the difficulty rating of every course, all time slot variables for difficult examinations can be known. For every two variables representing difficult examinations of a student group an inequality constraint ($\# \leq$) is imposed on their difference. This difference has to be greater than a certain number.

Enforcing the core exam constraint produces an acceptable examination schedule. A student cannot take more than one core exam per day. We can impose this constraint using the `serialized/2` global constraint. In general, the constraint `Serialized/2` constrains x tasks, each with a starting time and duration so that no tasks overlap. In our case, if we add to the duration of every core exam the number of slots per day, we can get a gap of at least one day between every two core examinations. Let `Z` be the number of core courses of a certain student group and `Tcore = [T1,...,TZ]` is a list variables that represent the time slots for these course then applying the following constraint

```
serialized(Tcore,[Gaps_added_durations])
```

will ensure that the gaps between core examinations is at least one day.

Exam restriction constraint limits the number of examinations held per day for every student to be less than a certain number. The variables that represent the examination time slots for every student group are known. The global constraint `count/4` can be used to limit the number of variables assigned to the same day to be less than a certain number. Let `x` be the number of courses assigned to a specific student group, `T = [T1,...,Tx]` is a list of variables representing the day of every examination slot and $1 \leq$`Y`$\leq$`m`, then `count(Y, T, #<, 3)` should be applied `m` times for every group of students constraining the number of exams held on every day to be less than 3.

In order to enforce the capacity constraint, the number of all students attending an examination at a certain time slot should not exceed a certain limit. The cumulative/4 global constraint can be used to assign a number of examinations to the same time slot constraining the number of students to be less than the specified limit.

```
cumulative(Starts, Durations, Capacities, limit)
```

`Starts` is a list of variables representing all courses. `Durations` is a list of all examination durations. `Capacities` is a list of the number of students enrolled in every course. `limit` is the number of seats available for examination.

Preassignment and unavailability constraint that restricts an exam to be scheduled in a given time slot or should not be scheduled in such time slot can be imposed using the equality and the inequality constraints. For example if a course `Ti` has to be assigned to time slot `j` then the constraint `Ti #= j` is added.

4 Implementation and Testing

Our system is composed of four components.

- A database server that is used to store all needed information about courses, students and rooms.
- A Java engine that is used to generate the CLP code and solves the second phase of the problem by assigning courses and students to examination rooms.
- A Prolog engine that is responsible for running the generated CLP code and getting a solution for the first phase of the problem by assigning a time slot to every examination.
- A graphical user interface implemented in Java that is used to display the results of both phases.

Figure 1 shows our system architecture.

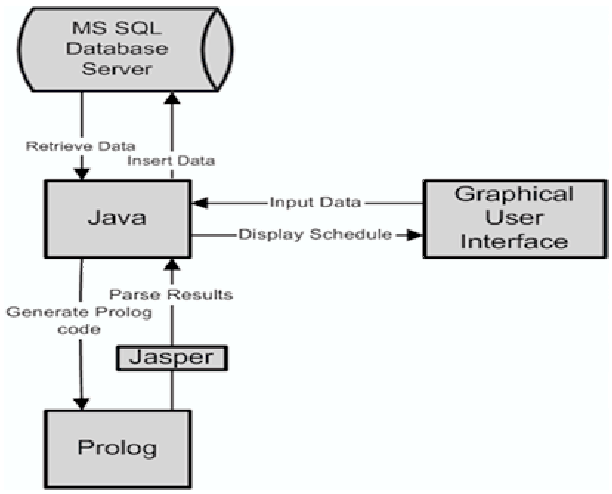


Fig. 1. *System Architecture*

The relational database management system Microsoft SQL server 2000 was used to store all the needed information about the input data such as courses,

student groups and rooms as well as the output data such as the exams with their examination time slots and examination rooms. Relations between the different tables of the database are maintained in order to specify the courses of every student group and the rooms in which every exam of a course is held. An instance of an examination is expressed by a pair `exam(C,T)`, where `C` is a unique course identifier and `T` is the examination time slot assigned to that course. Instances of examinations of the different courses are the inputs to our problem.

In general, a CLP program consists of three basic parts. We begin by associating domains to the different variables we have, then the different constraints of the problem are specified, and finally a labeling strategy is defined to look for a feasible solution via a backtrack search or via a branch-and-bound search. Labeling, which is used to assign values to variables, is needed since constraint solving is in general incomplete. We employed the first-fail strategy to select the course to be scheduled and a domain splitting strategy to select a period from the domain of the selected course. First-fail selects one of the most-constrained variables, i.e. one of the variables with the smallest domain. Domain splitting creates a choice between `X #=< M` and `X #> M`, where `M` is the midpoint of the domain of `X`. This approach yielded a good first solution to our problem. The input to a CLP program is usually in the form of a list. Constraints are expressed using the predefined constraints implemented in the constraint solver over finite domains `clpfd`.

Initially, we query the database using Java to obtain all the information needed to be able to produce a valid assignment. The CLP code is generated such that it restricts the solutions using the constraints already chosen by the user through the graphical interface. The input to our problem is a list of timetable instances. While generating that list we constrain the values of the time slot variables using the membership global constraint `domain/3`. After generating the list of instances, different constraints are applied to the variables eliminating values of their domains until finally a solution is reached.

After generating the SICStus prolog code, Jasper[1] is used to consult the program and parse the the values of the variables that represent the time slots for the different examinations. These values with their associated examination instances are stored in the database. A graphical user interface is then used to display the generated timetable. Figure 2 shows a sample output for a test run of our implementation.

Our system was tested on realistic data and constraints for the generation of the examination timetable for the final examinations of this term, Spring 2007. The test data consisted of 140 courses, 5500 students, and 1300 available seats for them. A solution was generated within few seconds. Our testing environment was a Microsoft Windows Vista Ultimate running on 32-bit processor at 1.73 GHz with 1 GB of RAM.

[1] a prolog interface to the java virtual machine

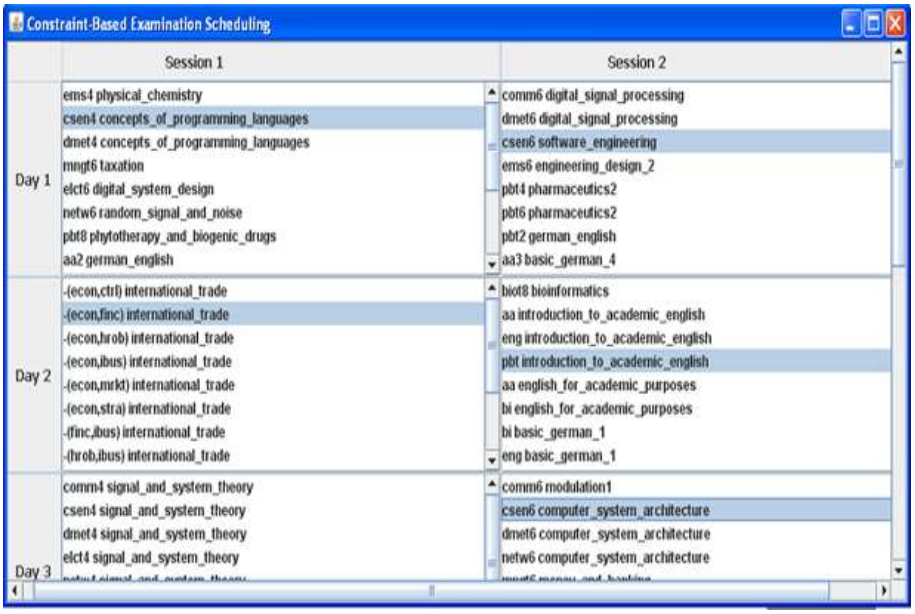


Fig. 2. *Sample Examination Timetable*

5 Conclusion and Future Work

In this paper, the GUC examination timetabling problem has been discussed and a possible solution using constraint logic programming has been presented. Due to the declarativity of our approach, our tool can be easily maintained and modified to accommodate new requirements and additional constraints.

One direction of future work is to use the constraint programming technology presented in this paper within a local search framework to optimize and repair the solutions found. This will lead to a hybrid method which was proved to be a powerful method to solve examination timetabling problems [5].

Another enhancement would be to integrate this system with the invigilation timetabling system implemented at the GUC. This system schedules invigilators such as professors, doctors and teaching assistant to proctor the scheduled examinations.

References

1. S. Abdennadher and T. Frühwirth. *Essentials of Constraint Programming*. Springer, 2002.
2. S. Petrovic E. Burke, M. Dror and R. Qu. Hybrid graph heuristics within a hyper-heuristic approach to exam timetabling problems. In *The Next Wave in Computing, Optimization, and Decision Technologies*. 2005.

3. M. Eley. Ant algorithms for the exam timetabling problem. In *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling*, 2006.
4. B. McCollum and N. Ireland. University timetabling: Bridging the gap between research and practice. In *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling*, 2006.
5. L. T.G. Merlot, N. Boland, B. D. Hughes, and P. J. Stuckey. A hybrid algorithm for the examination timetabling problem. In *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling*, 2002.
6. Andrea Schaerf. A survey of automated timetabling. In *115*, page 33. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 30 1995.
7. Edward Tsang. *Foundation of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

Constraint-Based University Timetabling for the German University in Cairo

Slim Abdennadher and Mohamed Aly

German University in Cairo, Department of Computer Science,
5th Settlement New Cairo City, Cairo, Egypt
slim.abdennadher@guc.edu.eg,mohamed.abdulazim@student.guc.edu.eg
http://cs.guc.edu.eg

Abstract. The course timetabling problem at the German University in Cairo has been in existence ever since the establishment of the university in 2003. Courses offered at the university, academic staff resources, and room resources make the problem over-constrained. Currently timetables are manually designed. The process is effort and time consuming and the results are often not satisfactory. In this paper, we provide a description of the problem and we show how it can be modeled as a constraint satisfaction problem. Our system, implemented in SICStus Prolog, generates a good timetable within some minutes instead of by hand some weeks.

1 Introduction

University course timetabling problems are combinatorial problems which consist in scheduling a set of courses within a given number of rooms and time periods. Generic solutions implemented and designed for such problems are not always compatible with all universities due to the wide variety of constraints. Number of courses, academic resources, room resources, and university-specific requirements are all factors which make the problem variant and hard. Therefore, universities design timetables manually or attempt to implement a tailored system that suits their requirements and constraints. Generating a timetable manually often requires a lot of time and effort. In addition, manually generated schedules usually fail to satisfy all essential constraints which are critical for the operation of the university. Thus, automating the generation of course timetables is still a challenging task [1].

The German University in Cairo (GUC), established in 2003, currently has 5500 students, and offers 140 courses through an academic staff of 300 members. Course timetabling has been a problem at the GUC ever since it started. Timetables are manually generated. The main goal is to generate a schedule which is personalized for all students in different faculties at the university. Many problems arise with the resulting schedules like teaching assistant allocation to different sessions in the same time slot or single room allocation to multiple sessions at the same instance. With an increasing number of students, academic staff and room resources, the timetabling problem complexity is getting more

arduous. The amount of time, work, and effort invested in making a timetable is very tremendous. Designing a valid timetable requires many iterations between the person in charge and academic department heads. At the end, the schedule "does the job" but not satisfactorily.

Several approaches have been proposed to solve course timetabling problems [2]. The most popular technique is based on graph coloring algorithms [3]. The main disadvantage of this approach is the difficulty of incorporating application specific constraints into the problem formulation. Other methods include local search techniques, e.g. simulated annealing [4] and genetic algorithms [5, 6]. In this paper, we present a constraint-based system to solve the course timetabling problem at the German University in Cairo. Constraint Programming draws on methods from artificial intelligence, logic programming, and operations research. It has been successfully applied in scheduling problems and proved to have promising results [7–11].

The paper is organized as follows. Section 2 introduces the course timetabling problem at the GUC and all constraints associated with it. In Section 3, we describe how the problem can be modeled as a constraint satisfaction problem. In Section 4, we give an overview of our implemented system. Finally, we conclude with a summary and directions for future work.

2 Problem Description

The GUC consists of four faculties, namely the faculty of Engineering, the faculty of Pharmacy and Biotechnology, the faculty of Management, and the faculty of Applied Arts. Each faculty offers a set of majors. For every major, there is a set of associated courses. Courses are either compulsory or elective. Faculties do not have separate buildings, therefore all courses from all faculties should be scheduled taking into consideration shared room resources. Students registered to a major are distributed among groups for lectures (lecture groups) and groups for tutorials or labs (study groups). A study group consists of maximum 25 students. Students registered to study groups are distributed among language groups. In each semester study groups are assigned to courses according to their corresponding curricula and semester. Due to the large number of students in a faculty and lecture hall capacities, all study groups cannot attend the same lecture at the same time. Therefore, groups of study groups are usually assigned to more than one lecture group. For example, if there are 27 groups studying Mathematics, then 3 lecture groups will be formed.

The timetable at the GUC spans a week starting from Saturday to Thursday. A day is divided to five time slots, where a time slot corresponds to 90 minutes. An event can take place in a time slot. This can be either a lecture, exercise, or practical session and it is given by either a lecturer or a teaching assistant. Usually, lectures are given by lecturers and exercise and practical sessions are given by teaching assistants. In normal cases, lectures take place at lecture halls, exercise sessions at exercise rooms and practical sessions take place in specialized laboratories depending on the requirements of a course. In summary, an event is

given by a lecturer or an assistant during a time slot in a day to a specific group using a specific room resource. This relationship is represented by a timetable for all events provided that hard constraints are not violated. Hard constraints are constraints that cannot be violated and which is considered to be of great necessity to the university operation. These constraints can be also added by some lecturers like part timers or visiting professors. Hard constraints will be outlined and discussed later in this section. The timetable tries to satisfy other constraints which are not very important or critical. Such constraints are known as soft constraints that should be satisfied but maybe violated. For example, these constraints can come in form of wishes from various academic staff.

Courses have specific credit hours. The credit hours are distributed on lecture, exercise, or practical sessions. Every two credit hours correspond to a teaching hour. A teaching hour corresponds to a time slot on the timetable. Sessions with one credit hour corresponds to a biweekly session. These sessions are usually scheduled to take place with other biweekly sessions if possible. For example a course with 3 lecture credit hours, 4 exercise credit hour and 0 practical credit hours can be interpreted as having two lectures per week one being scheduled weekly and the other is biweekly, two exercise sessions per week and no practical sessions. Lecture groups and study groups take the corresponding number of sessions per week. Courses might have hard constraints. Some courses require specialized laboratories or rooms for their practical or exercise sessions. For example, for some language courses a special laboratory with audio and video equipment is required. The availability of required room resources must be taken into consideration while scheduling. Some lecturers have specific requirements on session precedences. For example, in a computer science introductory course a lecturer might want to schedule exercise sessions before practical sessions.

Furthermore, some constraints should be taken into consideration to improve the quality of education. An important constraint is that no lectures should be scheduled in the last slot of any day. Another constraint requires that a certain day should have an activity slot for students, and a slot where all university academics can meet. For those slots no sessions should be scheduled. A study group should avoid spending a full day at the university. In other words, the maximum number of slots that a group can attend per day is 4. Therefore, if a group starts its day on the first slot, then it should end its day at most on the fourth slot, and if it starts its day on the second slot, then it should end its day at most on the fifth slot. Furthermore, The number of days off of a group depends on the total number of credit hours a group has. For example, if a group has a total of 24 course credit hours in a semester, the total number of slots needed would be $\frac{24}{2} = 12$ slots, which corresponds to 3 study days, then a group may have $6 - 3 = 3$ days off in a week.

A certain number of academics are assigned to a course at the beginning of a semester. Teaching assistants are assigned to one course at a time. For courses involving practical and exercise sessions, a teaching assistant can be assigned to both an exercise and a practical session or one of them. This should be taken into consideration when scheduling to avoid a possible clash. Furthermore, the

total numbers of teaching assistants assigned to a session should not exceed the maximum number of assistants assigned to the corresponding course at any time. A lecturer can be assigned to more than once course. This should be considered when scheduling in order to avoid a possible overlap. Academics assigned to courses have a total number of working and research hours per day and days off that need to be considered when scheduling. Courses should be scheduled in a manner such that the number of session hours given by an academic should not exceed his or her teaching load per day taking into consideration days off. For some courses, part timer lecturers are assigned. Those lecturers impose hard constraints most of the time. They might have special timing requirements or room requirements. These constraints are considered of a high priority and must be taken into consideration.

Another problem that resides when scheduling is the GUC transportation problem. The GUC provides bus transportation services for both students and academics. Two bus rounds take place in the beginning of the day to transport students and academics from various meeting points in Cairo. Other bus rounds take place, one after the fourth slot and the other after the fifth slot to transport students and academics back home. The first round delivers students just before the first slot and the second round delivers students just before the second slot. The number of students coming each day at the GUC by bus should be as balanced as possible in a way such that no limitation as possible is encountered with bus resources.

The GUC problem is too complex since a lot of constraints should be taken into account when scheduling a session. Due to the complexity of the problem, a good heuristic and a good model is needed to solve the problem. Any generated solution would be acceptable as it would save weeks of manual work. The solution can then be manually edited and modified later on. After getting to a good schedule minor amendments can be done to improve it.

3 Modeling the Problem as a Constraint Satisfaction Problem

In our model we have 5 slots in a day and 6 days in a week, our domain is chosen to be from 1 to 30 where each number corresponds to a slot. For example, slot 1 would correspond to the first slot on Saturday, and slot 23 would correspond to the third slot on Wednesday. Consequently, slots from 1 to 5 correspond to Saturday, 6 to 10 correspond to Sunday, 11 to 15 correspond to Monday, and so on. Figure 1 visualizes the timetable.

At each slot a group attends an instance of a course (lecture, tutorial, or practical session). We define this relationship by the tuple $(COURSE, GROUP, TYPE, SEQ, BI)$, where $COURSE$ is a unique identifier of a certain course, $GROUP$ is a unique identifier of a certain group, $TYPE$ declares the course instance (LECT for a lecture session, EXER for an exercise session, and PRAC for a practical session), SEQ enumerates the instance number (for example if two lectures are offered within the week, then two sequences are

DAY\SLOT	1st	2nd	3rd	4th	5th
SAT	1	2	3	4	5
SUN	6	7	8	9	10
MON	11	12	13	14	15
TUE	16	17	18	19	20
WED	21	22	23	24	25
THU	26	27	28	29	30

Fig. 1. *CSP Model Timetable*

defined, namely 1 and 2), and BI is a boolean value that defines if an instance should be scheduled as biweekly session. We define our variables to the problem by the tuple ($x_n = (COURSE, GROUP, TYPE, SEQ, BI)$) and the associated domain is a value from 1 to 30 ($D = \{1, ..., 30\}$).

Considering the relationship represented by the tuple, our variables to correspond to lecture, exercise, and practical sessions taken by groups. The model expresses the problem of finding a slot for every course instance associated with a group such that constraints on the variables are not violated. In the next section we describe how we can model our set of constraints. We apply constraints on our variables such that we end up with a solution to find when would every session for each group be scheduled taking into consideration the constraints of room, lecturers, and teaching assistant resources.

To implement the problem, we have chosen the constraint logic programming (CLP) paradigm. This paradigm has been successful in tackling CSPs as it extends the logic paradigm hence enhancing the solutions by making them more declarative and readable, and it supports the propagation of constraints for a specific domain providing an efficient implementation of computationally expensive procedures [12]. We have used the finite domain constraint library of SICStus prolog (clpfd) to model the problem. In the following, the different types of constraints are described [13]:

Domain Constraints: Domain constraints are used to restrict the range of values variables can have. The constraint `domain` is used to restrict the domain of a variable or a list of variables. For example, the constraint `domain([s1,s2],5,25)` restricts the domain of `s1` and `s2` to be between 5 and 25.

Arithmetic Constraints: Many arithmetic constraints are defined in SICStus prolog constraint library. Arithmetic constraints are in the form `Expr RelOp Expr`, where `Expr` generally corresponds to a variable or a normal arithmetic expression, and `RelOp` is an arithmetic constraint operator like `#=`, `#\=`, `#<`, `#>`, `#>=`, and `#=<`. For example, `X#=Y` constraints the variables `X` and `Y` to be equal to each other, and `X+Y#>=3` constraints that the sum of `X` and `Y` should be greater than or equal to `3`.

Propositional Constraints: Propositional constraints enables the expressions of logical constraints. The can be used along with arithmetic constraints to express a useful constraint. Two of the useful propositional constraints are the conjunction `X#/\Y` and disjunction `X#\/Y`. These constraints represent the conjunction and the disjunction of two constraints. For example, adding

the constraint `X#=Y #/\ X#=Z` constrains that `X` can either be equal to `Y` or `Z`.

Combinatorial Constraints: Combinatorial constraints vary in form. Two of the widely used constraints are `all_different/1` and `count/4`. The input for the global constraint `all_different` constraint is a list. The constraint ensures that all elements of the list are distinct. For example if the input list is a list of sessions from `s1..N`, then the constraint will be added as `all_different([s1,s2,...,sN])`. Another combinatorial constraint used is the `count/4`. The input to the constraint is in the form `Val, List, RelOp, Count` , where `Val` is a number, `List` is a list of variables, `RelOp` is an arithmetic constraint operator, and Count is a number. The constraint is satisfied if `N` is the number of elements of `List` that are equal to `Val` and `N RelOp Count`. For example, `count(1,[A,B,C],#<,2)` constraints that the total number of elements in the list `[A,B,C]` having the value `1` should be less than `2`.

The following describes the real constraints and how we used SICStus clpfd library to model them:

Domain of Sessions is from 1 to 30: To restrict the domain of the sessions, we use the global constraint `domain`. In order to constraint the list `L` of all variables, we add the constraint `domain(L,1,30)`.

Group Session Overlap: The sessions of a study group should not overlap; i.e. a study group cannot attend two different courses at the same time. The global constraint `all_different` is used to model constraints on sessions where no overlap should be encountered. For example, if sessions `s1,s2,s3` belong to a group, then we add a constraint `all_different([s1,s2,s3])`.

Lecturer giving more than a course: Academic members who teach more than one course should not have their sessions overlapped while scheduling. To ensure this, we add an `all_different` constraint to constrain the sessions given by an academic member.

Full Day Prevention Constraint: To avoid bringing a group for a full day, then their associated sessions in a day should not be scheduled either in the first slot or in the last slot. To model this we use the inequality constraint with the conjunction `#/\` and the disjunction `#\/` constraints to exclude the first session and the last session of each day for every group. For example, if the following sessions `s1, s2, s3` are for a group and we would like to constraint the first day (slots 1,2,3,4,5), we add the constraint
`(s1#\=1#/\ s2 #\= 1 #/\s3 #\= 1)#\/`
`(s1 #\= 5 #/\s2 #\=5 #/\ s3 #\=5)`.

Course Precedence Requirements: For all courses that have special precedence requirements on sessions (i.e. if an exercise session must scheduled before a practical session), we use the less than constraint `#<`. For example, if `s2` is before `s1` we add the constraint `s2#<s1`.

Part Timer Constraints: If a part timer lecturer selects a certain slot to give a session, then all session variables given by this lecturer will be constrained,

using the equality constraint, `#=`, to the required slot number. For example, consider a part timer giving the session `s1`, and can only come on the third and the fourth slots on Saturday (slots correspond to 3 and 4). We add the constraint `s1#=4 #\/ s1#=5`.

Lecturer Holiday Constraint: In order to express a lecturer holiday constraint and avoiding scheduling any courses taught by him on that day, we constrain all sessions given by the lecturer not to have the values of the day using the inequality constraint `#\=` with the disjunction constraint. For example, if a lecturer giving the session `s1` is off on Saturday, then we add the constraint `s1#\=1 #/\ s1#\=2 ... #/\ s1#\=5`.

Activity Slots, Free Sessions and Holidays: Slots that are required to remain empty with no scheduled sessions (as for the student activities slot) can be easily modeled by using the combinatorial constraint `count`. This is achieved by constraining the count of all variables in a list having the slot value is equal to zero. For example, if sessions `s1...sn` are not to be scheduled on the first session on Sunday, then we add the constraint `count(6,[s1...sn],#=,0)`.

Room Resources Constraint: In order to avoid scheduling exercise sessions in a slot more than the number of rooms available we constraint all the exercise sessions for every slot to be less than or equal the maximum number of available rooms using the `count` constraint with the arithmetic constraint operator `#=<`. For example if we cannot schedule more than 50 exercises per slot due to for sessions s1...sN we simply add `count(Slot,[s1,s2..sN],#=<,50)`, where `Slot` indicates the slot number (we have to define this constraint for all slots).

Teaching Assistant Resources: The `count` constraint is also used to limit the number of sessions scheduled per slot with the number of available teaching assistants assigned to the course or with the number of available room resources for a specific session (like a practical session).

Number of Lecture Sessions in a Day: We use the `count` constraint to constraint the total number of lectures per day. In order to model this we map all sessions to their corresponding day. This is done by subtracting 1 from the slot value and dividing by 5. By that we end with a value between 0 and 5 indicating the day. For example, Saturday the third slot has the value 3. Subtracting one from the value results in 2, dividing by 5 gives 0 which indicates day 0 (Saturday). Another example, Monday the second slot has the value 12. Performing the above operations, we get the value 2 which indicates the third day (Monday). We constraint the variables after applying the indicated operations (subtraction of one followed by division by 5) with the count constraint such that we have at most two lectures per day. For example, if lecture sessions `s1...sn` are taken by a group, we add the constraint `count(0,[(s1-1)/5,(s2-1)/5 ,...,(sn-1)/5],#=<,2)` for Saturday, and similarly for the rest of the week (Saturday is 0, Sunday is 1, Monday is 2, and so forth).

4 Implementation, Testing and Evaluation

A CLP program contains three basic parts. In the beginning, variables and their associated domains are defined, the constraints of the problem are laid out, and finally a labeling strategy is define to look for a feasible solution via a backtrack search or via a branch-and-bound search. Labeling, which is used to assign values to variables, is needed since constraint solving is in general incomplete.We employed the most-constrained strategy to select the course to be scheduled and a domain splitting strategy to select a period from the domain of the selected course. Most-constrained selects one of the variables with the smallest domain, breaking the ties by selected the one which has the most constraints suspended on it and selecting the left most one [13]. Domain splitting creates a choice between `X #=< M` and `X #> M`, where `M` is the midpoint of the domain of `X`. This approach yielded a good first solution to our problem.

Our system is composed of four main parts. An engine that controls the program logic and communicates with different parts of the system. A database to hold all data about courses, groups, academics, room resources and constraints. An interface between the engine and the constraint solver. A graphical user interface to help the user scheduler to interact with the system, input constraints and data, and output the resulting schedule. Figure 2 illustrates the overall system architecture.

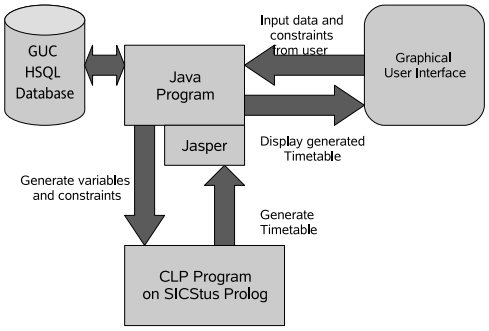


Fig. 2. *System Architecture*

In order to manage data about courses, groups, academics, room resources, and constraints, we have used the open source relational database engine HSQL[1]. database entity relationship diagram. We define relationships between course instances (lecture, exercise, and practical sessions) and groups. Data including course credit hours, precedence constraints, relationships between language and study groups, academic and room resources is stored in the database. Group

[1] Hypersonic SQL `http://www.hsqldb.org`

and course instances have unique integer identifiers. A timetable instance is uniquely identified by the pair $S(GID, IID)$, where GID is the unique group identifier and IID is the unique course instance identifier. Timetable instances are the input variables to the problem. Depending on the constraint, we query the database. For example, for a certain group we would like to add the constraint that no sessions should overlap. For that, we need to choose all the sessions which correspond to the tutorials and lectures from the `TUTPRAC_INSTANCE` table (exercise and practical instances) and the `LECT_INSTANCE` lecture instances. This is done by querying the database with following query:

```
SELECT GID, IID FROM TUTPRAC_INSTANCE WHERE GID = ##
UNION
SELECT LGID AS GID, IID FROM LECT_INSTANCE WHERE LGID = ##;
```

, where GID is a group identifier and LGID is a lecture group identifier (since study groups belong to lecture groups).

We used Java[2] to query the database and generate our SICStus prolog code. We use a simple file writer which outputs the constraints in a prolog file. After the prolog program is generated, we use Jasper (a prolog interface to the Java virtual machine) to run the CLP program. The input to our CLP program is a prolog list of schedule instances (group with course instance). The list identically corresponds to the schedule instances of the database. After the constraint solver is done with labeling and verifying the correctness of all values of the variables, we import the resulting schedule using Jasper back to the database. A graphical user interface, is designed and implemented to ease the constraint and data management, and to display and export the generated timetables to various formats.

We have tested the system on three main different study groups majoring in their second semester with a total of 473 course instance variables (corresponding to lecture, exercise, and practical session) to be scheduled and we generated a good solution within a few seconds. Our testing environment was a Microsoft Windows Vista Ultimate running on a 32-bit processor at 1.73 GHz with 1 GB of RAM. With our test bench we had 623029 resumptions, 77689 constraint entailments, 167925 prunings, 2 backtracks and a total of 77915 constraints created.

5 Conclusion and Future Work

In this paper, we have presented how the course timetabling problem at the German University in Cairo can be modeled as a constraint satisfaction problem. To implement it, we have used the constraint logic programming language SICStus Prolog. The time taken to implement, debug and test the software was about 14 weeks. The use of a declarative paradigm means that the program can be easily maintained and modified which is crucial for a young university like the GUC. Until now, we dealt only with hard constraints. One direction of future

[2] `http://www.java.com`

work is to use the constraint programming technology presented in this paper within a local search framework to optimize and repair the solutions found. This will lead to a hybrid method which was proved to be a powerful method to solve timetabling problems, e.g. examination timetabling [14].

References

1. McCollum, B., Ireland, N.: University timetabling: Bridging the gap between research and practice. In: Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling. (2006)
2. Schaerf, A.: A survey of automated timetabling. In: 115. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X (30 1995) 33
3. de Werra, D.: An introduction to timetabling. European Journal of Operational Research **19** (1985) 151 – 162
4. Abramson, D.: Constructing school timetables using simulated annealing: Sequential and parallel algorithms. Management Science **37, 1** (1991) 98–113
5. Colorni, A., Dorigo, M., Maniezzo, V.: Genetic algorithms and highly constrained problems: the time-table case. In Schwefel, H.P., Männer, R., eds.: Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1. Volume 496., Dortmund, Germany, Springer-Verlag, Berlin, Germany (1-3 1991) 55–59
6. Perzina, R.: Solving the university timetabling problem with optimized enrolment of students by a parallel self-adaptive genetic algorithm. In: Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling. (2006)
7. Goltz, H., Matzke, D.: University timetabling using constraint logic programming. In: First International Workshop on Practical Aspects of Declarative Languages. (1999)
8. Rudov, H., Matyska, L.: Constraint-based timetabling with student schedules (2000)
9. Deris, S., Omatu, S., Ohta, H.: Timetable planning using the constraint-based reasoning. Comput. Oper. Res. **27**(9) (2000) 819–840
10. Abdennadher, S., Schlenker, H.: Nurse scheduling using constraint logic programming. In: AAAI/IAAI. (1999) 838–843
11. Azevedo, F., Barahona, P.: Timetabling in constraint logic programming. In: Proceedings of 2nd World Congress on Expert Systems, Estoril, Portugal (1994)
12. Fernández, A.J., Hill, P.M.: A comparative study of eight constraint programming languages over the boolean and finite domains. Constraints **5**(3) (2000) 275–301
13. Intelligent Systems Laboratory: SICStus Prolog User's Manual. Swedish Institute of Computer Science, PO Box 1263, SE-164 29 Kista, Sweden. (April 2001)
14. Merlot, L.T., Boland, N., Hughes, B.D., Stuckey, P.J.: A hybrid algorithm for the examination timetabling problem. In: Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling. (2002)

Databases and Data Mining

Compiling Entity-Relationship Diagrams into Declarative Programs*

Bernd Braßel Michael Hanus Marion Müller

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

Abstract. This paper proposes a framework to support high-level database programming in a declarative programming language. In order to ensure safe database updates, all access and update operations related to the database are generated from high-level descriptions in the form of entity-relationship diagrams (ERDs). We propose a representation of ERDs in the declarative language Curry so that they can be constructed by various tools and then translated into this representation. Furthermore, we have implemented a compiler from this representation into a Curry program that provides access and update operations based on a high-level API for database programming.

1 Motivation

Many applications in the real world need databases to store the data they process. Thus, programming languages for such applications must also support some mechanism to organize the access to databases. This can be done in a way that is largely independent on the underlying programming language, e.g., by passing SQL statements as strings to some database connection. However, it is well known that such a loose coupling is the source of security leaks, in particular, in web applications [9]. Thus, a tight connection or amalgamation of the database access into the programming language should be preferred.

In principle, logic programming provides a natural framework for connecting databases (e.g., see [3, 5]) since relations stored in a relational database can be considered as facts defining a predicate of a logic program. Unfortunately, the well-developed theory in this area is not accompanied by practical implementations. For instance, distributions of Prolog implementations rarely come with a standard interface to relational databases. An exception is Ciao Prolog that has a persistence module [2] that allows the declaration of predicates where the facts are persistently stored, e.g., in a relational database. This module supports a simple method to query the relational database, but updates are handled by predicates with side effects and transactions are not explicitly supported. A similar concept but with a clear separation between queries and updates has been proposed in [7] for the multi-paradigm declarative language Curry [8]. This will be the basis for the current framework which has the following objectives:

– The methods to access and update the database should be expressed by language features rather than passing SQL strings around.

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

– Queries to the database should be clearly separated from updates that might change the outcome of queries.
– Safe transactions, i.e., sequence of updates that keep some integrity constraints, should be supported.
– The necessary code for these operations should be derived from specifications whenever possible in order to obtain more reliable applications.

For this purpose, we define an API for database programming in Curry that abstracts from the concrete methods to access a given database by providing abstract operations for this purpose. In particular, this API exploits the type system of Curry in order to ensure a strict separation between queries and updates. This is described in detail in Section 2. To specify the logical structure of the database, we use the entity-relationship (ER) model [1], which is well established for this purpose. In order to be largely independent of concrete specification tools, we define a representation of ER diagrams in Curry so that concrete ER specification tools can be connected by defining a translator from the format used in these tools into this Curry representation. This representation is described in Section 3. Finally, we develop a compiler that translates an ER specification into a Curry module that contains access and update operations and operations to check integrity constraints according to the ER specification. The generated code uses the database API described in Section 2. The compilation method is sketched in Section 4. Finally, Section 5 contains our conclusions.

2 Database Programming in Curry

We assume basic familiarity with functional logic programming (see [6] for a survey) and Curry [8] so that we sketch in the following only the basic concepts relevant for this paper.

Functional logic languages integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic programming features like computing with partial information (logic variables), constraint solving, and non-deterministic search for solutions.

As a concrete functional logic language, we use Curry in our framework. From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [10], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are first-class citizens as in Haskell and are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. Function calls with free variables are evaluated by a possibly nondeterministic instantiation of demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule.

Example 1. The following Curry program defines the data types of Boolean values, "possible" (maybe) values, union of two types, and polymorphic lists (first four lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool       = True    | False
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data List a     = []      | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last :: [a] -> a
last xs | conc ys [x] =:= xs   = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants, `Nothing` and `Just` as the constructors for possible values (where `Nothing` is considered as no value), `Left` and `Right` to inject values into a union (`Either`) type, and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` and `b` are type variables ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

Curry also offers other standard features of functional languages, like higher-order functions (e.g., "`\`x `->` e" denotes an anonymous function that assigns to each x the value of e), modules, or monadic I/O [11] (e.g., an operation of type "`IO t`" is an I/O action, i.e., a computation that interacts with the "external world" and returns a value of type `t`).

Logic programming is supported by admitting function calls with free variables (see "`conc ys [x]`" above) and constraints in the condition of a defining rule. Conditional program rules have the form $l \mid c \;=\; r$ specifying that l is reducible to r if the condition c is satisfied (see the rule defining `last` above). A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable constructor terms.

A relational programming style, which is typical for database applications, is supported by considering predicates as functions with result type `Success`. For instance, a predicate `isPrime` that is satisfied if the argument (an integer number) is a prime can be modeled as a function with type

```
isPrime :: Int -> Success
```

The following rules define a few facts for this predicate:

```
isPrime 2 = success
isPrime 3 = success
isPrime 5 = success
```

Apart from syntactic differences, any pure logic program has a direct correspondence to a Curry program. For instance, a predicate `isPrimePair` that is satisfied if the arguments are primes that differ by 2 can be defined as follows:

```
isPrimePair :: Int -> Int -> Success
isPrimePair x y = isPrime x & isPrime y & x+2 =:= y
```

In order to deal with information that is persistently stored outside the program (e.g., in databases), [7] proposed the concept of dynamic predicates. A *dynamic predicate* is a predicate where the defining facts (see `isPrime`) are not part of the program but stored outside.

Moreover, the defining facts can be modified (similarly to dynamic predicates in Prolog). In order to distinguish between definitions in a program (that do not change over time) and dynamic entities, there is a distinguished type `Dynamic` for the latter. For instance, in order to define a dynamic predicate `prime` to store prime numbers persistently, we have to define it by

```
prime :: Int -> Dynamic
prime persistent "store"
```

where *store* specifies the storage mechanism, e.g., a directory for a lightweight file-based implementation or a database specification [4].

There are various primitives that deal with dynamic predicates. First, there are combinators to construct complex queries from basic dynamic predicates. For instance, the combinator

```
(<>) :: Dynamic -> Dynamic -> Dynamic
```

joins two dynamic predicates, and the combinator

```
(|>)  :: Dynamic -> Bool    -> Dynamic
```

restricts a dynamic predicate with a Boolean condition. Thus, the expression

```
prime x <> prime y |> x+2==y
```

specifies numbers `x` and `y` that are prime pairs. On the one hand, such expressions can be translated into corresponding SQL statements [4] so that the programmer is freed of dealing with details of SQL. On the other hand, one can use all elements and libraries of a universal programming language for database programming due to its conceptual embedding in the programming language.

Since the contents of dynamic predicates can change over time, one needs a careful concept of evaluating dynamic predicates in order to keep the declarative style of programming. For this purpose, we introduce a concept of queries that are evaluated in the I/O monad, i.e., at particular points of time in a computation.[1] Conceptually, a *query* is a method to compute solutions w.r.t. dynamic predicates. Depending on the number of requested solutions, there are different operations to construct queries, e.g.,

```
queryAll :: (a -> Dynamic) -> Query [a]
queryOne :: (a -> Dynamic) -> Query (Maybe a)
```

`queryAll` and `queryOne` construct queries to compute all and one (if possible) solution to an abstraction over dynamic predicates, respectively. For instance,

```
qPrimePairs :: Query [(Int,Int)]
qPrimePairs = queryAll (\(x,y) -> prime x <> prime y |> x+2==y)
```

is a query to compute all prime pairs. In order to access the currently stored data, there is an operation `runQ` to execute a query as an I/O action:

```
runQ :: Query a -> IO a
```

For instance, executing the main expression "`runQ qPrimePairs`" returns prime pairs that can be derived from the prime numbers currently stored in the dynamic predicate `prime`.

In order to change the data stored in dynamic predicates, there are operations to add and delete knowledge about dynamic predicates:

[1] Note that we only use the basic concept of dynamic predicates from [7]. The following interface to deal with queries and transactions is new and more abstract than the concepts described in [7].

```
addDB    :: Dynamic -> Transaction ()
deleteDB :: Dynamic -> Transaction ()
```

Typically, these operations are applied to single ground facts (since facts with free variables cannot be persistently stored), like "`addDB (prime 13)`". In order to embed these update operations into safe transactions, the result type is "`Transaction ()`" (in contrast to the proposal in [7] where these updates are I/O actions). A *transaction* is basically a sequence of updates that is completely executed or ignored (following the ACID principle in databases). Similarly to the monadic approach to I/O, transactions also have a monadic structure so that transactions can be sequentially composed by a monadic bind operator:

```
(|>>=) :: Transaction a -> (a -> Transaction b) -> Transaction b
```

Thus, "`t1 |>>= \x -> t2`" is a transaction that first executes transaction `t1`, which returns some result value that is bound to the parameter `x` before executing transaction `t2`. If the result of the first transaction is not relevant, one can also use the specialized sequential composition "`|>>`":

```
(|>>) :: Transaction a -> Transaction b -> Transaction b
t1 |>> t2 = t1 |>>= \_ -> t2
```

A value can be mapped into a trivial transaction returning this value by the usual return operator:

```
returnT :: a -> Transaction a
```

In order to define a transaction that depend on some data stored in a database, one can also embed a query into a transaction:

```
getDB :: Query a -> Transaction a
```

For instance, the following expression exploits the standard higher-order functions `map`, `foldr`, and "`.`" (function composition) to define a transaction that deletes all known primes that are smaller than `100`:

```
getDB (queryAll (\i -> prime i |> i<100)) |>>=
foldr (|>>) (returnT ()) . map (deleteDB . prime)
```

To apply a transaction to the current database, there is an operation `runT` that executes a given transaction as an I/O action:

```
runT :: Transaction a -> IO (Either a TError)
```

`runT` returns either the value computed by the successful execution of the transaction or an error in case of a transaction failure. The type `TError` of possible transaction errors contains constructors for various kinds of errors:

```
data TError = TError TErrorKind String

data TErrorKind = KeyNotExistsError | DuplicateKeyError
  | UniqueError | MinError | MaxError | UserDefinedError
```

(the string argument is intended to provide some details about the reason of the error). `UserDefinedError` is a general error that could be raised by the application program whereas the other alternatives are typical errors due to unsatisfied integrity constraints. An error is raised inside a transaction by the operation

```
errorT :: TError -> Transaction a
```

where the specialization

```
failT :: String -> Transaction a
```

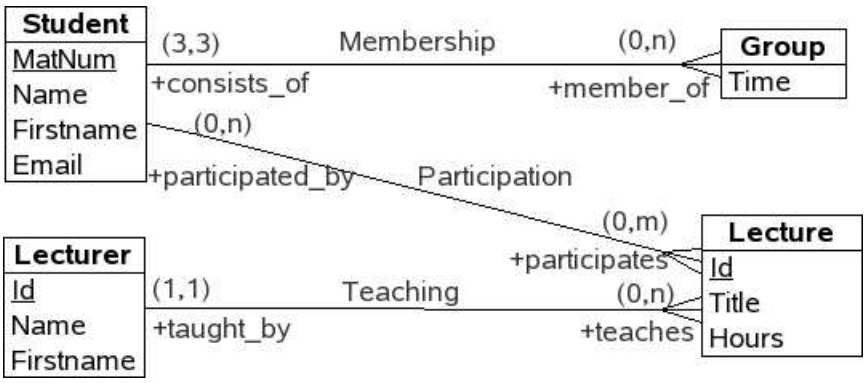


Fig. 1. A simple ER diagram for university lectures

```
failT s = errorT (TError UserDefinedError s)
```

is useful to raise user-defined transaction errors. If an error is raised inside a transaction, the transaction is aborted and the changes to the database performed in this transaction are undone.

All features for database programming are contained in a specific `Database` library[2] so that they can be simply used in the application program by importing it. This will be the basis to generate higher-level code from ER diagrams that are described in the following.

3 ER Diagrams

The entity-relationship model is a framework to specify the structure and specific constraints of data stored in a database. It uses a graphical notation, called ER diagrams (ERDs) to visualize the conceptual model. In this framework, the part of the world that is interesting for the application is modeled by entities that have attributes and relationships between the entities. The relationships have cardinality constraints that must be satisfied in each valid state of the database, e.g., after each transaction.

There are various tools to support the data modeling process with ERDs. In our framework we want to use some tool to develop specific ERDs from which the necessary program code based on the `Database` library described in the previous section can be automatically generated. In order to become largely independent of a concrete tool, we define a representation of ERDs in Curry so that a concrete ERD tool can be applied in this framework by implementing a translator from the tool format into our representation. In our concrete implementation, we have used the free software tool Umbrello UML Modeller[3], a UML tool part of KDE that also supports ERDs. Figure 1 shows an example ERD constructed with this tool. The developed ERDs are stored in XML files so that the Umbrello format can be easily translated into our ERD format.

The representation of ERDs as data types in Curry is straightforward. A complete ERD consists of a name (that is later used as the module name for the generated code) and lists of entities and relationships:

[2] `http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/Database.html`
[3] `http://uml.sourceforge.net`

```
data ERD = ERD String [Entity] [Relationship]
```

An entity has a name and a list of attributes, where each attribute has a name, a domain, and specifications about its key and null value property:

```
data Entity = Entity String [Attribute]

data Attribute = Attribute String Domain Key Null

data Key = NoKey | PKey | Unique

type Null = Bool

data Domain = IntDom    (Maybe Int)    | FloatDom (Maybe Float)
            | StringDom (Maybe String) | ...
```

Thus, each attribute is part of a primary key (`PKey`), unique (`Unique`), or not a key (`NoKey`). Furthermore, it is allowed that specific attributes can have null values, i.e., can be undefined. The domain of each attribute is one of the standard domains or some user-defined type. For each kind of domain, one can also have a default value (modeled by the `Maybe` type in Curry).

Finally, each relationship has a name and a list of connections to entities (`REnd`), where each connection has the name of the connected entity, the role name of this connection, and its cardinality as arguments:

```
data Relationship = Relationship String [REnd]

data REnd = REnd String String Cardinality

data Cardinality = Exactly Int | Range Int (Maybe Int)
```

The cardinality is either a fixed integer or a range between two integers (where `Nothing` as the upper bound represents an arbitrary cardinality).

4 Compiling ER Diagrams into Curry Programs

The transformation of ERDs into executable Curry code is done in the following order:

1. Translate an ERD into an `ERD` term.
2. Represent the relationships occurring in an `ERD` term as entities.
3. Map all entities into corresponding Curry code based on the `Database` library.

The first step depends on the format used in the ERD tool. As mentioned above, we have implemented a translation from the Umbrello UML Modeller into `ERD` terms. This part is relatively easy thanks to the presence of XML processing tools.

4.1 Transforming ERDs

The second step is necessary since the relational model supports only relations (i.e., database tables). Thus, entities as well as relationships must be mapped into relations. The mapping of entities into relations is straightforward by using the entity name as the name of the relation and the attribute names as column names. The mapping of relationships is more subtle. In principle, each relationship can be mapped into a corresponding relation. However, this simple approach might cause the creation of many relations or database tables. In order to reduce them, it is sometimes better to represent specific relations as foreign

keys, i.e., to store the key of entity e_1 referred by a relationship between e_1 and e_2 in entity e_2. Whether or not this is possible depends on the kind of the relation. The different cases will be discussed next. Note that the representation of relationships as relations causes also various integrity constraints to be satisfied. For instance, if an entity has an attribute which contains a foreign key, the value of this attribute must be either null or an existing key in the corresponding relation. Furthermore, the various cardinalities of each relationship must be satisfied. Each transaction should ensure that the integrity constraints are valid after finishing the transaction.

Now we discuss the representation of the various kinds of relationships in the ER model. For the sake of simplicity, we assume that each relationship contains two ends, i.e., two roles with cardinality ranges (min, max) so that we can characterize each relationship by their related cardinalities $(min_A, max_A) : (min_B, max_B)$ between entities A and B (where max_i is either a natural number greater than min_i or ∞, $i \in \{A, B\}$).

Simple-simple (1:1) relations: This case covers all situations where each cardinality is at most one. In the case $(0, 1) : (1, 1)$, the key of entity B is added as an attribute to entity A containing a foreign key since there must be exactly one B entity for each A entity. Furthermore, this attribute is `Unique` to ensure the uniqueness of the inverse relation. The case $(0, 1) : (0, 1)$ can be similarly treated except that null values are allowed for the foreign key.

Simple-complex (1:n) relations: In the case $(0, 1) : (min_B, max_B)$, the key of entity A is added as a foreign key (possibly null) to each B entity. If $min_B > 0$ or $max_B \neq \infty$, the integrity constraints for the right number of occurrences must be checked by each database update. The case $(1, 1) : (0, max_B)$ is similarly implemented except that null values for the foreign key are not allowed.

Complex-complex (n:m) relations: In this case a new relation representing this relationship is introduced. The new relation is connected to entities A and B by two new relationships of the previous kinds.

Note that we have not considered relationships where both minimal cardinalities are greater than zero. This case is excluded by our framework (and rarely occurs in practical data models) since it causes difficulties when creating new entities of type A or B. Since each entity requires a relation to an existing entity of the other type and vice versa, it is not possible to create the new entities independently. Thus, both entities must be created and connected in one transaction which requires specific complex transactions. Therefore, we do not support this in our code generation. If such relations are required in an application (e.g., cyclic relationships), then the necessary code must be directly written with the primitives of the `Database` library.

Based on this case distinction, the second step of our compiler maps an `ERD` term into a new `ERD` term where foreign keys are added to entities and new entities are introduced to represent complex-complex relations. Furthermore, each entity is extended with an internal primary key to simplify the access to each entity by a unique scheme.

4.2 Code Generation for ERDs

After the mapping of entities and relationships into relations as described above, we can generate the concrete program code to organize the database access and update. As already mentioned, we base the generated code on the functionality provided by the library

`Database` described in Section 2. The schemas for the generated code are sketched in this section. We use the notation *En* for the name of an entity (which starts by convention with an uppercase letter) and en for the same name where the first letter is lowercase (this is necessary due to the convention in Curry that data constructors and functions start with uppercase and lowercase letters, respectively).

The first elements of the generated code are data types to represent relations. For each entity *En* with attributes of types $at_1, \ldots, at_n$, we generate the following two type definitions:

```
data En = En Key at_1 ... at_n

data EnKey = EnKey Key
```

`Key` is the type of all internal keys for entities. Currently, it is identical to `Int`. Thus, each entity structure contains an internal key for its unique identification. The specific type *En*`Key` is later used to distinguish the keys for different entities by their types, i.e., to exploit the type system of Curry to avoid confusion between the various keys. For each relation that has been introduced for a complex-complex relationship (see above), a similar type definition is introduced except that it does not have an internal key but only the keys of the connected entities as arguments. Note that only the names of the types are exported but not their internal structure (i.e., they are *abstract* data types for the application program). This ensures that the application program cannot manipulate the internal keys.

In order to access or modify the attributes of an entity, we generate corresponding functions where we use the attribute names of the ERD for the names of the functions. If entity *En* has an attribute A_i of type at_i ($i = 1, \ldots, n$), we generate the following getter and setter functions and a function to access the key of the entity:

```
enA_i :: En -> at_i
enA_i (En _ ... x_i ... _) = x_i

setEnA_i :: En -> at_i -> En
setEnA_i (En x_1 ... _ ... x_n) x_i = En x_1 ... x_i ... x_n

enKey :: En -> EnKey
enKey (En k _ ... _) = EnKey k
```

As described in Section 2, data can be persistently stored by putting them into a dynamic predicate. Thus, we define for each entity *En* a dynamic predicate

```
enEntry :: En -> Dynamic
enEntry persistent "..."
```

Since the manipulation of all persistent data should be done by safe operations, this dynamic predicate is not exported. Instead, a dynamic predicate *en* is exported that associates a key with the data so that an access is only possible to data with an existing key:

```
en :: EnKey -> En -> Dynamic
en key obj | key =:= enKey obj = enEntry obj
```

Note that the use of a functional logic language is important here. For instance, the access to an entity with a given key `k` can be done by solving the goal "*en* `k o`" where `o` is a free variable that will be bound to the concrete instance of the entity.

For each role with name rn specified in an ERD, we generate a dynamic predicate of type

```
rn :: E_1Key -> E_2Key -> Dynamic
```

where E_1 and E_2 are the entities related by this role. The implementation of these predicates depend on the kind of relationship according to their implementation as discussed in Section 4.1. Since complex-complex relationships are implemented as relations, i.e., persistent predicates (that are only internal and not exported), the corresponding roles can be directly mapped to these. Simple-simple and simple-complex relationships are implemented by foreign keys in the corresponding entities. Thus, their roles are implemented by accessing these keys. We omit the code details that depend on the different cases already discussed in Section 4.1.

Based on these basic implementations of entities and relationships, we also generate code for transactions to manipulate the data and check the integrity constraints specified by the relationships of an ERD. In order to access an entity with a specific key, there is a generic function that delivers this entity in a transaction or raises a transaction error if there is no entry with this key:

```
getEntry :: k -> (k -> a -> Dynamic) -> Transaction a
getEntry key pred =
  getDB (queryOne (\info -> pred key info)) |>>=
  maybe (errorT (KeyNotExistsError "no entry for..."))
        returnT
```

This internal function is specialized to an exported function for each entity:

```
getEn :: EnKey -> Transaction En
getEn key = getEntry key en
```

In order to insert new entities, there is a "new" transaction for each entity. If the ERD specifies no relationship for this entity with a minimum greater than zero, there is no need to provide related entities so that the transaction has the following structure (if *En* has attributes of types $at_1, \ldots, at_n$):

```
newEn :: at_1 -> ··· -> at_n -> Transaction En
newEn a_1 ... a_n = check_1 |>> ... |>> check_n |>> newEntry ...
```

Here, $check_i$ are the various integrity checks (e.g., uniqueness checks for attributes specified as `Unique`) and `newEntry` is a generic operation (similarly to `getEntry`) to insert a new entity. If attribute A_i has a default value or null values are allowed for it, the type at_i is replaced by `Maybe` at_i in `new`*En*. If there are relationships for this entity with a minimum greater than zero, than the keys (in general, a list of keys) must be also provided as parameters to `new`*En*. The same holds for the "new" operations generated for each complex-complex relationship. For instance, the new operation for lectures according to the ERD in Figure 1 has the following type:

```
newLecture :: LecturerKey -> Int -> String -> Maybe Int
              -> Transaction Lecture
```

The first argument is the key of the lecturer required by the relationship `Teaching`, and the further arguments are the values of the `Id`, `Title` and `Hours` attributes (where the attribute `Hours` has a default value so that the argument is optional).

Similarly to `new`*En*, we provide also operations to update existing entities. These operations have the following structure:

```
updateEn :: En -> Transaction ()
updateEn e = check_1 |>> ... |>> check_n |>> updateEntry ...
```

Again, the various integrity constraints must be checked before an update is finally performed. In order to get an impression of the kind of integrity constraints, we discuss a few checks in the following.

If an attribute of an entity is `Unique`, this property must be checked before a new instance of the entity is inserted. For this purpose, there is a generic transaction

```
unique :: a -> (b -> a) -> (b -> Dynamic) -> Transaction ()
```

where the first argument is the attribute value, the second argument is a getter function for this attribute, and the third argument is the dynamic predicate representing this entity, i.e., a typical call to check the uniqueness of the new value a_i for attribute A_i of entity *En* is (`unique` a_i *en*A_i *En*). This transaction raises a `UniqueError` if an instance with this attribute value already exists.

If an entity contains a foreign key, each update must check the existence of this foreign key. This is the purpose of the generic transaction

```
existsDBKey :: k -> (a -> k) -> (a -> Dynamic) -> Transaction ()
```

where the arguments are the foreign key, a getter function (*en*`Key`) for the key in the foreign entity and the dynamic predicate of the foreign entity. If the key does not exist, a `KeyNotExistsError` is raised. Furthermore, there are generic transactions to check minimum and maximum cardinalities for relationships and lists of foreign keys that can raise the transaction errors `MinError`, `MaxError`, or `DuplicateKeyError`. For each `new` and `update` operation generated by our compiler, the necessary integrity checks are inserted based on the ER specification.

Our framework does not provide delete operations. The motivation for this is that safe delete operations require the update of all other entities where this entity could occur as a key. Thus, a simple delete could cause many implicit changes that are difficult to overlook. It might be better to provide only the deletion of single entities followed by a global consistency check (discussed below). A solution to this problem is left as future work.

Even if our generated transactions ensure the integrity of the affected relations, it is sometimes useful to provide a global consistency check that is regularly applied to all data. This could be necessary if unsafe delete operations are performed or the database is modified by programs that do not use the safe interface but directly accesses the data. For this purpose, we also generate explicit global consistency test operations that check all persistent data w.r.t. the ER model (we omit the details here).

5 Conclusions

We have presented a framework to compile conceptual data models specified as entity-relationship diagrams into executable code for database programming in Curry. This compilation is done in three phases: translate the specific ERD format into a tool-independent representation, transform the relationships into relations according to their complexity, and generate code for the safe access and update of the data.

Due to the importance of ERDs to design conceptual data models, there are also other tools with similar objectives. Most existing tools support only the generation of SQL code, like the free software tools DB-Main[4] or DBDesigner4[5]. The main motivation for our de-

[4] `http://www.db-main.be`

[5] `http://www.fabforce.net/dbdesigner4`

velopment was the seamless embedding of database programming in a declarative programming language and the use of existing specification methods like ERDs as the basis to generate most of the necessary code required by the application programs. The advantages of our framework are:

- The application programmer must only specify the data model in a high-level format (ERDs) and all necessary code for dealing with data in this model is generated.
- The interface used by the application programs is type safe, i.e., the types specified in the ERD are mapped into types of the programming language so that ill-typed data cannot be constructed.
- Updates to the database are supported as transactions that automatically checks all integrity constraints specified in the ERD.
- Checks for all integrity constraints are derived from the ERD for individual tables and the complete database so that they can be periodically applied to verify the integrity of the current state of the database.
- The generated code is based on an abstract interface for database programming so that it is readable and well structured. Thus, it can be easily modified and adapted to new requirements. For instance, integrity constraints not expressible in ERDs can be easily added to individual update operations, or specific deletion operations can be inserted in the generated module.

References

1. P. P.-S. Chen. The Entity-Relationship Model—Toward a Unifi ed View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, pp. 9–36, 1976.
2. J. Correas, J.M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 104–119. Springer LNCS 3057, 2004.
3. S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
4. S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.
5. H. Gallaire and J. Minker, editors. *Logic and Databases*, New York, 1978. Plenum Press.
6. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
7. M. Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, Vol. 2004, No. 5, 2004.
8. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
9. S.H. Huseby. *Innocent Code: A Security Wake-Up Call for Web Programmers*. Wiley, 2003.
10. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
11. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Squash: A Tool for Designing, Analyzing and Refactoring Relational Database Applications ⋆

Andreas M. Boehm[1], Dietmar Seipel[2], Albert Sickmann[1] and Matthias Wetzka[2]

[1] University of Würzburg, Rudolf-Virchow-Center for Experimental Biomedicine
Versbacher Strasse 9, D-97078 Würzburg, Germany
Emails: ab@andiboehm.de, albert.sickmann@virchow.uni-wuerzburg.de

[2] University of Würzburg, Department of Computer Science
Am Hubland, D-97074 Würzburg, Germany
Emails: matthias.wetzka@web.de, seipel@informatik.uni-wuerzburg.de

Abstract. The *performance* of a large biological application of relational databases highly depends on the quality of the database schema design, the resulting structure of the tables, and the logical relations between them. In production reality, the performance mainly depends on the semantics comprised in the stored data and the corresponding queries.
We have developed a tool named Squash (SQL Query Analyzer and Schema EnHancer) for designing, analyzing and refactoring relational database applications. Squash parses the SQL definition of the database schema and the queries into an XML representation called SquashML. This information can be managed using the declarative XML *query* and *transformation* language FNQuery, which we have implemented in SWI-PROLOG, thus allowing modification and dynamic processing of the schema data. Visualization of relationships and join paths of queries is integrated, too. Using an online connection, Squash can determine characteristics from the current database, such as join selectivities or table sizes.
Squash comes with a set of predefined methods for tuning the database schema according to the load profile induced by the application, and with methods for proposing changes in the database schema such as the creation of indexes, partitioning, splitting, or further normalization. SQL statements are adapted simultaneously upon modification of the schema, and they are rewritten in consideration of database-specific statement processing rules including the integration of optimizer hints. Moreover, the Squash *framework* is very flexible and extensible; *user-defined methods* based on ad-hoc FNQuery statements can be plugged in.

1 Introduction

During productive use, enterprise-class databases undergo a lot of changes in order to keep up with ever-changing requirements. The growing space-requirements and the growing complexity of a productive database increase the complexity of maintaining a good performance of the database query execution. Performance is highly dependent on the database schema design [12]. In addition, a complex database schema is more prone to design errors.

⋆ This work was supported by the Deutsche Forschungsgemeinschaft (FZT 82).

Increasing the performance and the manageability of a database usually requires restructuring the database schema and has effects on the application code. Additionally, tuning query execution is associated with adding secondary data structures such as indexes or horizontal partitioning [8, 15, 2]. Because applications depend on the database schema, its modification implies the adaption of the queries used in the application code. The tuning of a complex database is usually done by specially trained experts having good knowledge of database design and many years of experience with tuning databases [21, 17]. Due to the lot of possibilities, the optimization of a database is a very complex and time-consuming task, even for professional database experts. In the process of optimization, many characteristic values for the database need to be calculated and assessed in order to determine an optimal configuration. Special tuning tools can help the database administrator to focus on the important information necessary for the optimization and support the database administrator in complex database schema manipulations [21, 17]. More sophisticated tools can even propose optimized database configurations by using the database schema and a characteristic workload log as input [1, 6, 13].

During the last two decades, approaches towards self-tuning database management systems have been developed [7, 10, 17, 24–26]. These approaches have inherent access to the most current load profile of the application. But they are limited to changes only in the database that are transparent for the application code. Thus, they lack the possiblity to automatically update the underlying source code and are not capable of refactoring applications completely. The approach presented in this paper aims at refactoring the database schemas and the queries in the SQL code of the application at the same time.

The rest of the paper, which describes our system Squash, is organized as follows: Section 2 describes the management of SQL data with the query and transformation language FNQuery. Section 3 is on the analysis and tuning of existing database applications. Examples dealing with foreign key constraints and join conditions are presented in Section 4 to illustrate how user-defined methods can be written. Finally, Section 5 will briefly report on a case study for physical database optimization, that was conducted with Squash.

2 Managing SQL Data with FNQuery

We have developed an extensible XML representation called SquashML for SQL schema definitions and queries. The core of SquashML is predetermined by the SQL standard, but it also allows for system-specific constructs of different SQL dialects. For example, some definitions and storage parameters from the Oracle™ database management system were integrated. SquashML is able to represent schema objects, as well as database queries obtained from application code or from logging data. This format allows for easily processing and evaluating the database schema information. Currently, supported schema objects include table and index definitions. The Squash parser was implemented in Perl. Other existing XML representations of databases like SQL/XML usually focus on representing the database contents, i.e. the tables, and not the schema definition it-

self [16]. SquashML was developed specifically to map only the database schema and queries, without the current contents of the database.

The management of the SquashML data is implemented in PROLOG [9, 22]. The representation of XML documents was realized by using the *field notation* data structure [19]. In addition, the XML query and transformation language FNQuery [19, 20] for XML documents in field notation was used. The language resembles XQuery [5], but it is implemented in and fully interleaved with PROLOG. The usual axes of XPath are provided for selection and modification of XML documents. Moreover, FNQuery embodies transformation features, which go beyond XSLT, and update features.

Squash is part of the DisLog Developers' Toolkit (DDK), a large SWI-PROLOG library [23] which can be loaded into any application on either Windows, UNIX or Linux. The DDK is available for download via `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`. Squash comes with a library of predefined methods, which are accessible through a graphical user interface. The system can be flexibly extended by plugging in further, used-defined methods. In the following, we will show how such methods can be defined using FNQuery on SquashML to adapt and specialize the system to further needs. As an example we will show how to detect and visualize foreign key constraints in SQL create statements.

Detection and Visualization of Foreign Key (FK) Constraints. Consider the following abbreviated and simplified statement from a biological database for Mascot™ [18] data:

```
CREATE TABLE USR_SEARCH.
    TBL_SEARCH_RESULTS_MASCOT_PEPTIDE (
  ID_SEARCH_RESULTS_MASCOT_PEPTIDE NUMBER DEFAULT -1
     NOT NULL primary key,
  ID_RES_SEARCH NOT NULL,
  ID_RES_SPECTRA NOT NULL,
  ID_RES_SEARCH_PEPTIDES NOT NULL,
  ID_DICT_PEPTIDE NOT NULL,
  foreign key (ID_DICT_PEPTIDE)
    references TBL_DICT_PEPTIDES,
  foreign key (ID_RES_SEARCH_PEPTIDES)
    references TBL_RES_SEARCH_PEPTIDES ... );
```

The Squash parser transforms this to SquashML; for simplicity we leave out some closing tags:

```
<create_table name="TBL_SEARCH_RESULTS_MASCOT_PEPTIDE"
  schema="USR_SEARCH">
  <relational_properties>
    <column_def name="ID_SEARCH_RESULTS_MASCOT_PEPTIDE"> ...
    <column_def name="ID_RES_SEARCH">
      <inline_constraint>
        <inline_null_constraint type="not_null" /> ...
    <column_def name="ID_RES_SPECTRA"> ...
    <column_def name="ID_RES_SEARCH_PEPTIDES"> ...
    <column_def name="ID_DICT_PEPTIDE"> ...
```

```
<out_of_line_constraint name="...">
  <out_of_line_foreign_key_constraint>
    <column name="ID_DICT_PEPTIDE" />
    <references_clause object="TBL_DICT_PEPTIDES"/> ...
<out_of_line_constraint name="...">
  <out_of_line_foreign_key_constraint>
    <column name="ID_RES_SEARCH_PEPTIDES" />
    <references_clause object="TBL_RES_SEARCH_PEPTIDES"/> ...
```

To get an overview of the complex SquashML element, Squash can visualize the structure of the foreign key (FK) constraints, see Figure 1.

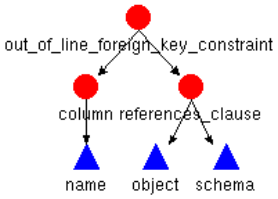


Fig. 1. Foreign Key Constraint in SquashML

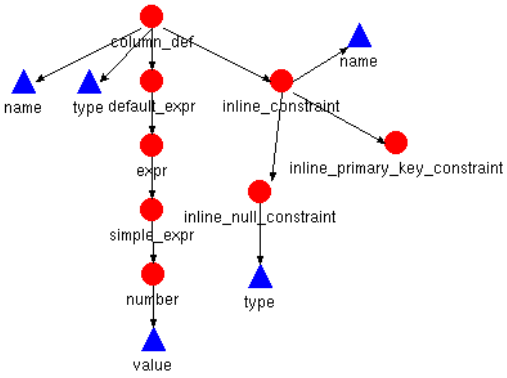


Fig. 2. Primary Key Constraint in SquashML

The following rules determine a FK constraint and the primary key (PK) of the referenced table from a SquashML element `Xml`. The first PROLOG predicate selects an `out_of_line_foreign_key_constraint` within a create table statement for a table `T1`, and then it selects the referencing column `A1` as well as the referenced table

T2; subsequently, a call to the second PROLOG predicate determines the PK A2 of the referenced table T2; Figure 2 shows the structure of PK constraints in SquashML:

```
xml_to_fk_constraint(Xml, T1:A1=T2:A2) :-
   C := Xml/create_table::[@name=T1]/_
      /out_of_line_foreign_key_constraint,
   A1 := C/column@name,
   T2 := C/references_clause@object,
   xml_to_pk_constraint(Xml, T2:A2).

xml_to_pk_constraint(Xml, T:A) :-
   _ := Xml/create_table::[@name=T]/_/column_def::[@name=A]
      /inline_constraint/inline_primary_key_constraint.
```

There exist 18 FK constraints in the whole Mascot™ database schema. Figure 3 visualizes the computed FK constraints of a fragment of the database; e.g., the FK constraint from the column ID_RES_SEARCH_PEPTIDES of the table A=TBL_SEARCH_RESULTS_MASCOT_PEPTIDE to the primary key attribute ID_RES_SEARCH_PEPTIDES of the table B=TBL_RES_SEARCH_PEPTIDES is shown as the rhombus labelled by fk4.

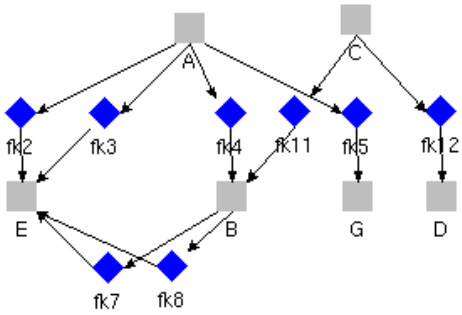


Fig. 3. Foreign Key Constraints in the Database

3 Analysis and Tuning of Database Applications

Squash is divided into 5 components used for parsing, visualization, analysis, manipulation and optimization. Using the visualization and analysis components, a database administrator can gain a quick overview of the most important characteristics of the database and use this information for manual tuning decisions. The optimization component accesses the data of the analysis component and uses heuristic algorithms to automatically determine an optimized database schema. Then, the manipulation component of Squash applies the suggested, potentially complex changes to the XML representation of the database schema. It can automatically propagate schema changes to the queries of the application.

The Squash system provides functions to view the different aspects of the database schema with visual markup of important properties. In addition, the system is able to analyze a database schema and a corresponding workload for semantic errors, flaws and inconsistencies.

Visualization. Comprehending the structure of a complex database schema just by reading the SQL create statements is a very demanding task, and design errors can be missed easily. Squash provides a number of different visualization methods for the database schema, cf. Figure 3, and the queries, cf. Figure 4. Complex select statements tend to include many tables and use them in join operations. Therefore, Squash uses a graph representation for query visualization. If a select statement contains nested subqueries, then these queries can be included if desired.

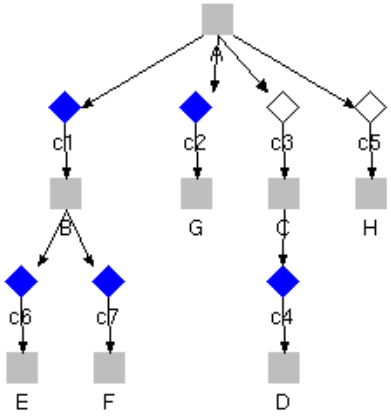


Fig. 4. Join Conditions in a Query

Analysis. The analysis component of Squash performs multiple tasks. Firstly, in addition to a first graphical overview, more advanced Squash methods check the database schema and the queries for possible design flaws or semantic inconsistencies. Using this information, the database administrator can manually decide on optimization methods for the database schema. Secondly, when using the automatic optimization methods of Squash, these data are used as input for heuristic algorithms. Additionally, the analysis component collects schema properties and displays them. These functions are used in order to create reports of the database schema and the queries.

Especially within queries, *semantic errors* are often introduced by inexperienced database programmers [14, 4]. The predefined methods of Squash include the detection of constant output columns, redundant output columns, redundant joins, incomplete column references, and the detection of joins lacking at least one FK relationship. The database schema can also be checked for *design flaws (anomalies)* by Squash. The available Squash methods include detection of isolated schema parts, cyclic FK references, unused columns as well as unused tables, unique indexes that are defined as non-unique,

indexes of low quality, datatype conflicts, anonymous PK or FK constraints, missing primary keys, and the detection (and improvement) of bad join orders.

Refactoring. The refactoring component allows for manipulating the database schema as well as the corresponding application code based on the transformation and update features of FNQuery. Besides trivial manipulations such as adding or removing colums, the system also supports complex schema manipulations that affect other parts of the database schema and the queries, such as vertical partitioning and merging of tables. Squash is able to provide all queries in the workload with the best set of indexes according to the heuristic function of the system, and it generates appropriate optimizer hints.

4 Analysis and Visualization of Join Conditions in SQL Queries

Squash can visualize and analyse complex SQL statements. E.g., the following SQL select statement from the Mascot™ database joins 8 tables:

```
SELECT *
FROM TBL_SEARCH_RESULTS_MASCOT_PEPTIDE A,
   TBL_RES_SEARCH_PEPTIDES B,
   ... C, ... D, ... E, ... F, ... G, ... H
WHERE A.ID_DICT_PEPTIDE IN (
   SELECT B0.ID_PEPTIDE
   FROM TBL_RES_SEARCH_PEPTIDES B0
   WHERE B0.ID_RES_SEARCH = 2025
   GROUP BY ID_PEPTIDE
   HAVING COUNT(ID_RES_SEARCH_PEPTIDES) >=1 )
AND A.ID_RES_SEARCH = 2025
AND c1 AND c2 AND A.FLAG_DELETE = 0
AND c3 AND c6 (+) AND c7 (+)
AND c4 AND c5 AND E.LEN >= 6
AND A.SCORENORMALIZED >= 1.5
ORDER BY D.ID_SEQ_SEQUENZ, B.ACC_ID_CLS, ...;
```

It uses the following 7 join conditions:

```
c1: A.ID_RES_SEARCH_PEPTIDES = B.ID_RES_SEARCH_PEPTIDES
c2: A.ID_RES_SPECTRA = G.ID_RES_SPECTRA
c3: A.ID_RES_SEARCH_PEPTIDES = C.ID_PEPTIDE
c4: C.ID_SEQUENCE = D.ID_SEQ_SEQUENZ
c5: A.ID_RES_SEARCH = H.ID_RES_SEARCH
c6: B.ID_PEPTIDE = E.ID_DICT_PEPTIDE
c7: B.ID_PEPTIDE_MOD = F.ID_DICT_PEPTIDE
```

We wanted to know which of the join conditions of the query are valid FK constraints, cf. Figure 3, since other join conditions might be design errors. The query is parsed into the following SquashML element; for simplicity we leave out closing tags:

```
<select>
  <subquery id="subquery_1">
    <select_list>
      <expr>
        <simple_expr>
          <object table_view="A" column="IDRESSEARCH"/> ...
    <from>
      <table_reference>
        <query_table_reference>
          <query_table_expression>
            <simple_query_table_expression
              object="TBL_SEARCH_RESULTS_MASCOT_PEPTIDE"
              schema="USRSEARCH"/> ...
          <alias name="A"/> ...
    <where> ...
    <order_by> ...
```

The conditions in the WHERE part look like follows:

```
<condition>
  <simple_comparison_condition operator="=">
    <left_expr>
      <expr>
        <simple_expr>
          <object table_view="A"
            column="ID_RES_SEARCH_PEPTIDES"/>
      ...
    <right_expr> ...
            table_view="B" column="ID_RES_SEARCH_PEPTIDES"/>
      ...
```

The following PROLOG rule selects a simple comparison condition with the operator "=", or an outer join condition:

```
xml_to_join_condition(Xml, T1:A1=T2:A2) :-
   S := Xml/select,
   ( Cond := S/_
        /simple_comparison_condition::[@operator=(=)]
   ; Cond := S/_
        /outer_join_condition ),
   cond_to_pair(Cond, left_expr, T1:A1),
   cond_to_pair(Cond, right_expr, T2:A2).
```

Within the selected conditions, the following PROLOG rule determines the left and the right table/column-pair:

```
cond_to_pair(Cond, Tag, T:A) :-
   [T, A] := Cond/Tag/expr/simple_expr
      /object@[table_view, column].
```

In Figure 4, the join conditions which are valid FK constraints – cf. Figure 3 (note that the labels of the constraints in the two figures don't correspond) – are depicted as blue (dark) rhombs, whereas the others are coloured in white. E and F are aliases for the same table. It turned out, that the join condition c3 between A and C does not correspond to a valid FK constraint, but there exist two other FK constraints which link A and C through the table B:

```
 fk4: A:ID_RES_SEARCH_PEPTIDES
        -> B:ID_RES_SEARCH_PEPTIDES
fk11: C:ID_PEPTIDE
        -> B:ID_RES_SEARCH_PEPTIDES
```

Also, the join condition c5 between A and H does not correspond to a valid FK constraint. In this example, both join conditions c3 and c5 are correct – and no design errors or flaws.

5 Physical Database Optimization

The optimization component of Squash supports the proposal and the creation of indexes as well as of horizontal partitions. The system analyzes the database schema in conjunction with the workload and generates an optimized configuration. Many problems in database optimization, such as determining an optimal set of indexes, are known to be $\mathcal{NP}$-complete. Since exact optimality is much too complex to compute, heuristic algorithms are used in order to determine a new configuration of the database schema which is expected to perform better.

Squash uses a *heuristic multi-step approach* to find good configurations. In the first step, statistical data are collected from the database. This information is used to minimize the set of columns that contribute to an optimized configuration. In the second step, the workload is analyzed, and the quality of each column is calculated. All columns whose quality is below a threshold are removed from the set of candidate columns. Finally, the remaining columns are evaluated in detail, and a solution is generated. The algorithms have been developed without a special database management system in mind, but they are parameterized for application to Oracle™.

Index Proposal. Squash is able to propose and generate multi-column indexes. The solution-space is reduced to a manageable size by a partitioned multi-step approach in combination with a scoring function. Finally, the algorithm sorts the columns in an order being most useful for the index. Squash offers three different sorting methods. The first one orders the columns by decreasing selectivity. The second method sorts them according to the type of condition they are used in. The third method takes the reversed order in which the columns appear in the query. Which one produces the best results, depends on the query optimizer of the database management system. In the case study, sorting according to the WHERE clause was found to yield good results for Oracle™.

Horizontal Partitioning. The core task for horizontal partitioning of a table consists in suggesting a column that is suitable for partitioning. In the case of range partitioning,

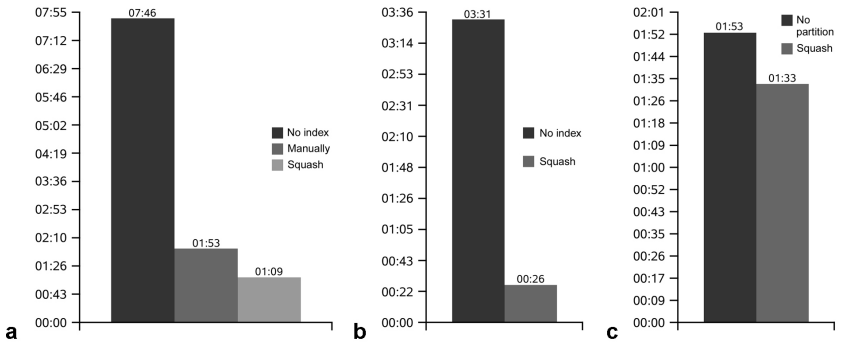


Fig. 5. Runtimes of Characteristic SQL Statements:

- **a)** Sequest™ statement with indexes proposed by Squash. The duration was 7:46 min without any indexes. This could be reduced by manual tuning to 1:53 min, whereas tuning by Squash achieved a further reduction of about 49% down to 1:09 min, i.e. 14% of the original time.
- **b)** Mascot™ statement with indexes proposed by Squash. The execution time was 3:31 without any indexes. This was not tuned manually before. Tuning by indexes proposed by Squash yielded a reduction to 0:26 min.
- **c)** Sequest™ statement with partitions proposed by Squash. The execution time was 1:53 min (manually tuned), whereas tuning by Squash and application of partitions achieved further reduction of about 19% to 1:33 min.

the borders of each partition are calculated by analysing the histograms of the partition key. The number of partitions for hash partitioning is calculated from the table volume.

Case Study on Index Proposal. A case study was conducted using a database system designed for use in mass spectrometry-based proteomics, that provides support for large scale data evaluation and data mining. This system, which is composed of the two subsystems `seqDB` and `resDB` [3, 27], was temporarily ported to Oracle™ for being analyzed by Squash. The complete database schema consists of 46 tables requiring about 68.5 GB disk space, and it fulfills the definition of a data warehouse system.

Some characteristic SQL statements of the data evaluation part `resDB` were analyzed, that were obtained from an application log and were submitted to Squash for analysis in conjunction with the schema creation script. The statements perform the grouping of peptide results into protein identification results that were obtained by Sequest™ [11] and Mascot™ [18], respectively.

For demonstration, the Sequest™ statements were previously tuned manually by an experienced database administrator, the Mascot™ queries were left completely untuned. The improvements resulting from the tuning are depicted in Figure 5; they were between 20% and 80%.

6 Conclusion

We have presented an extensible tool named Squash that applies methods from artificial intelligence to relational database design and tuning. Input and ouput are XML-based, and operations on these data are performed using the declarative XML query and transformation language FNQuery.

The Squash tool supports the refactoring of database applications and considers the interactions between application code and database schema definition; both types of code can be handled and manipulated automatically. Thus, application maintenance is simplified, and the chance for errors is reduced, in sum yielding time and resource savings.

In the future we are planning to analyse and refactor database applications where the SQL code can be *embedded* in other programming languages – such as Java and PHP – as well, based on XML representations of these languages, which we have already developed.

References

1. AGRAWAL, S.; NARASAYYA, V. ; YANG, B.: Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. *Proc. ACM SIGMOD Intl. Conference on Management of Data*. ACM Press, 2004, pp. 359-370
2. BELLATRECHE, L.; KARLAPALEM, K.; MOHANIA, M. K. ; SCHNEIDER, M.: What Can Partitioning Do for Your Data Warehouses and Data Marts? *Proc. Intl. Symposium on Database Engineering and Applications (IDEAS 2000)*. IEEE Computer Society, 2000, pp. 437-446
3. BOEHM, A. M.; SICKMANN, A.: A Comprehensive Dictionary of Protein Accession Codes for Complete Protein Accession Identifier Alias Resolving. In: *Proteomics*, Vol. 6(15), 2006, pp. 4223-4226
4. BRASS, S.; GOLDBERG, C.: Proving the Safety of SQL Queries. *Proc. 5th Intl. Conference on Quality of Software 2005*
5. CHAMBERLIN, D.: XQuery: a Query Language for XML. *Proc. ACM SIGMOD Intl. Conference on Management of Data 2003*. ACM Press, 2003, pp. 682-682
6. CHAUDHURI, S.; NARASAYYA, V.: Autoadmin What-If Index Analysis Utility. *Proc. Intl. Conference on Management of Data Archives. Proc. ACM SIGMOD Intl. Conference on Management of Data 1988*. ACM Press, 1998, pp. 367-378
7. CHAUDHURI, S.; WEIKUM, G.: Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. *Proc. 26th Intl. Conference on Very Large Data Bases (VLDB)*, 2000, pp. 1-10
8. CHOENNI, S.; BLANKEN, H. M. ; CHANG, T.: Index Selection in Relational Databases. *Proc. 5th Intl. Conference on Computing and Information (ICCI)*, IEEE, 1993, pp. 491-496
9. CLOCKSIN, W. F.; MELLISH, C. S.: *Programming in Prolog*. 5th Edition, Springer, 2003
10. DIAS, K.; RAMACHER, M.; SHAFT, U.; VENKATARAMANI, V.; WOOD, G:. Automatic Performance Diagnosis and Tuning in Oracle. *Proc. 2nd Biennal Conference on Innovative Data Systems Research (CIDR)*, 2005, pp. 84-94
11. ENG, J. K.; MCCORMACK, A. L. ; YATES, J. R.: An Approach to Correlate Tandem Mass Spectral Data of Peptides with Amino Acid Sequences in a Protein Database. *Journal of the American Society for Mass Spectrometry*, Vol. 5(11), 1994, pp. 976-989

12. FAGIN, R.: Normal Forms and Relational Database Operators. *Proc. ACM SIGMOD Intl. Conference on Management of Data* 1979
13. FINKELSTEIN, S.; SCHKOLNICK, M. ; TIBERIO, P.: Physical Database Design for Relational Databases. *ACM Transactions on Database Systems (TODS)*, Vol. 13(1), 1988, pp. 91-128
14. GOLDBERG, C.; BRASS, S.: Semantic Errors in SQL Queries: A Quite Complete List. *Proc. 16. GI-Workshop Grundlagen von Datenbanken*, 2004, pp. 58-62
15. GRUENWALD, L.; EICH, M.: Selecting a Database Partitioning Technique. *Journal of Database Management*, Vol. 4(3), 1993, pp. 27-39
16. INTL. ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9075-14:2003 Information Technology – Database Languages – SQL – Part 14: XML-Related Specifications (SQL/XML)*. 2003
17. KWAN, E.; LIGHTSTONE, S.; SCHIEFER, B.; STORM, A. ; WU, L.: Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. *10. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW, Datenbanksysteme für Business, Technologie und Web)*, Bd. 26. Lecture Notes in Informatics (LNI), 2003, pp. 620–629
18. PERKINS, D. N.; PAPPIN, D. J. C.; CREASY, D. M. ; COTTRELL, J. S.: Probability-Based Protein Identification by Searching Sequence Databases Using Mass Spectrometry Data. *Electrophoresis*, Vol. 20(18), 1999, pp. 3551-3567
19. SEIPEL, D.: *Processing XML-Documents in Prolog*. *Proc. 17th Workshop on Logic Programmierung (WLP) 2002*
20. SEIPEL, D.; BAUMEISTER, J. ; HOPFNER, M.: Declarative Querying and Visualizing Knowledge Bases in XML. *Proc. 15th Intl. Conference on Declarative Programming and Knowledge Management (INAP)*, 2004, pp. 140-151
21. TELFORD, R.; HORMAN, R.; LIGHTSTONE, S.; MARKOV, N.; O'CONNELL, S.; LOHMAN, G.: Usability and Design Considerations for an Autonomic Relational Database Management System. *IBM Systems Journal*, Vol. 42(4), 2003, pp. 568-581
22. WIELEMAKER, J.: An Overview of the SWI-Prolog Programming Environment. *Proc. 13th Intl. Workshop on Logic Programming Environments (WLPE), 2003*, pp. 1-16
23. WIELEMAKER, J.: *SWI-Prolog*. Version: 2007. `http://www.swi-prolog.org/`
24. VALENTIN, G.; ZULIANI, M.; ZILIO, D. C.; LOHMAN, G.;SKELLEY, V.: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. *Proc. 16th Intl. Conference on Data Engineering*, 2000, pp. 101-110
25. WEIKUM, G.; HASSE, C.; MÖNKEBERG, A.; ZABBACK, P.: The Comfort Automatic Tuning Project. *Information Systems*, Vol. 19(5), 1994, pp. 381-432
26. WEIKUM, G.; MÖNKEBERG, A.; HASSE, C.; ZABBACK, P.: Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. *Proc. 28th Intl. Conference on Very Large Data Bases (VLDB)*, 2002, pp. 20-31
27. ZAHEDI, R. P.; SICKMANN, A.; BOEHM, A. M.; WINKLER, C.; ZUFALL, N.; SCHÖNFISCH, B.; GUIARD, B.; PFANNER, N. ; MEISINGER, C.: Proteomic Analysis of the Yeast Mitochondrial Outer Membrane Reveals Accumulation of a Subclass of Preproteins. *Molecular Biology of the Cell*, Vol. 17(3), 2006, pp. 1436-1450

Tabling Logic Programs in a Database

Pedro Costa, Ricardo Rocha, and Michel Ferreira

DCC-FC & LIACC
University of Porto, Portugal
c0370061@dcc.fc.up.pt {ricroc,michel}@ncc.up.pt

Abstract. Resolution strategies based on tabling are considered to be particularly effective in Logic Programming. Unfortunately, when faced with applications that store large and/or many answers, memory exhaustion is a potential problem. A common approach to recover space is table deletion. In this work, we propose a different approach, storing tables externally in a relational database. Subsequent calls to stored tables import answers from the database, rather than performing a complete re-computation. To validate this approach, we have extended the YapTab tabling system, providing engine support for exporting and importing tables to and from the MySQL RDBMS. Two different relational schemes for data storage and two data-set retrieval strategies are compared.

1 Introduction

Tabling is an implementation technique where intermediate answers for subgoals are stored and then reused when a repeated call appears. Resolution strategies based on tabling [1, 2] have proved to be particularly effective in logic programs, reducing the search space, avoiding looping and enhancing the termination properties of Prolog models based on SLD resolution [3].

The performance of tabling largely depends on the implementation of the table itself; being called upon often, fast look up and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is *tries* [4]. Tries are trees in which there is one node for every common prefix [5]. Tries meet the previously enumerated criteria of compactness and efficiency quite well. The YapTab tabling system [6] uses tries to implement tables.

Used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and very large tables, quickly filling up memory. In general, there is no choice but to throw away some of the tables, ideally, the least likely to be used next. The common control mechanism implemented in most tabling systems is to have a set of tabling primitives that the programmer can use to dynamically abolish some of the tables. A more recent proposal has been implemented in YapTab, where a memory management strategy, based on a *least recently used* algorithm, automatically recovers space from the least recently used tables when memory runs out [7]. With this approach, the programmer can still force the deletion of particular tables, but can also

transfer to the memory management algorithm the decision of what potentially useless tables to delete. Note that, in both situations, the loss of stored answers within the deleted tables is unavoidable, eventually leading to re-computation.

In this work, we propose an alternative approach and instead of deleting tables, we store them using a relational database management system (RDBMS). Later, when a repeated call appears, we load the answers from the database, hence avoiding re-computation. With this approach, the YapTab's memory management algorithm can still be used, this time to decide what tables to move to the database when memory runs out, rather than what tables to delete. To validate this approach we propose DBTAB, a relational model for representing and storing tables externally in tabled logic programs. In particular, we use YapTab as the tabling system and MySQL [8] as the RDBMS. The initial implementation of DBTAB handles only atomic terms such as atoms and numbers.

The remainder of the paper is organised as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we introduce our model and discuss how tables can be represented in a RDBMS. We then describe how we extended YapTab to provide engine support to handle database stored answers. Finally, we present initial results and outline some conclusions.

2 The Table Space

Tabled programs are evaluated by storing all computed answers for current subgoals in a proper data space, the *table space*. Whenever a subgoal $\mathcal{S}$ is called for the first time, a matching entry is allocated in the table space, under which all computed answers for the call are stored. Variant calls to $\mathcal{S}$ are resolved by consumption of these previously stored answers. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. When all possible resolutions are performed, $\mathcal{S}$ is said to be *completely evaluated*.

The table space can be accessed in a number of ways: **(i)** to look up if a subgoal is in the table, and if not insert it; **(ii)** to verify whether a newly found answer is already in the table, and if not insert it; and, **(iii)** to load answers to variant subgoals.

For performance purposes, tables are implemented using two levels of tries, one for subgoal calls, other for computed answers. In both levels, stored terms with common prefixes branch off each other at the first distinguishing symbol. The table space is organized in the following way. Each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the *subgoal trie*. Each unique path in this trie represents a different subgoal call, with the argument terms being stored within the internal nodes. The path ends when a *subgoal frame* data structure is reached. When inserting new answers, substitution terms for the unbound variables in the subgoal call are stored as unique paths into the *answer trie* [4].

An example for a tabled predicate $f/2$ is shown in Fig. 1. Initially, the subgoal trie contains only the root node. When the subgoal $f(X, a)$ is called, two internal nodes are inserted: one for the variable X, and a second for the con-

stant a. Notice that variables are represented as distinct constants, as proposed by Bachmair *et al.* [9]. The subgoal frame is inserted as a leaf, waiting for the answers. Then, the subgoal $f(Y, 1)$ is inserted. It shares one common node with $f(X, a)$, but the second argument is different, so a new subgoal frame needs to be created. Next, the answers for $f(Y, 1)$ are stored in the answer trie as their values are computed.

Each internal node is a four field data structure. The first field stores the symbol for the node. The second and third fields store pointers respectively to the first child node and to the parent node. The fourth field stores a pointer to the sibling node, in such a way that the outgoing transitions from a node can be collected by following its first child pointer and then the list of sibling pointers. In YapTab, terms are tagged accordingly to their type and the non-tagged part of a term, which cannot be used for data storage purposes, is always less than the usual 32 or 64-bit C representation (in what follows, we shall refer to integer and atom terms as *short atomic terms* and to floating-point and integers larger than the maximum allowed non-tagged integer value as *long atomic terms*). Since long atomic terms do not fit into the node's specific slot, these terms are split in pieces, if needed, and stored surrounded by additional nodes consisting of special markers. This representation particularity is visible in Fig. 1, where the 2^{30} integer is surrounded by a *long integer functor* (LIF) marker.

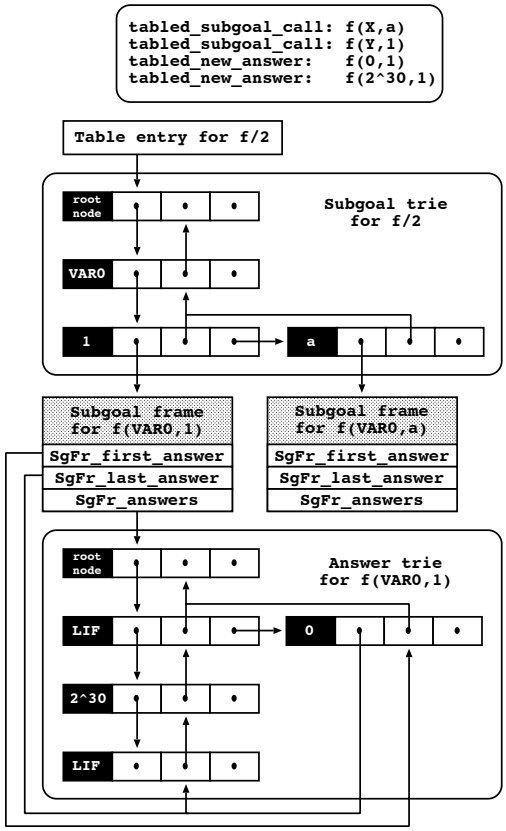


Fig. 1. The table space organization

When adding answers to the trie, the leaf answer nodes are chained in a linked list in insertion time order (using the child field), so that recovery may happen the same way. The subgoal frame internal pointers `SgFr_first_answer` and `SgFr_last_answer` point respectively to the first and last answer of this list. When consuming answers, a variant subgoal only needs to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. Answers are loaded by traversing the trie nodes bottom-up.

3 Extending the YapTab Design

The main idea behind DBTAB is straightforward. Every data transaction occurs in the context of a specific execution session. In that context, a relational table is assigned to each tabled predicate. The relation's name encloses the predicate's functor and arity, the relation's attributes equal the predicate's arguments in number and name. The dumping of a complete tabled subgoal answer set to the database is triggered when the respective table is chosen for destruction by the *least recently used* algorithm.

Data exchange between the YapTab engine and the RDBMS is mostly done through MySQL C API for prepared statements. Two major table space data structures, *table entries* and *subgoal frames*, are expanded with new pointers to `PreparedStatement` data structures. Table entries are expanded with a pointer to an INSERT prepared statement. This statement is prepared to insert a full record at a time into the predicate's relational table, so that all subgoals hanging from the same table entry may use the same INSERT statement when storing their computed answers. Subgoal frames, on the other hand, are expanded with a pointer to a SELECT prepared statement. This statement is used to speed up the data retrieval, while reducing the resulting record-set at the same time. Ground terms in the respective subgoal trie branch are used to refine the statement's WHERE clause - the corresponding fields in the relational representation need not to be selected for retrieval since their values are already known.

3.1 The Relational Storage Model

The choice of an effective representation model for the tables is a hard task to fulfill. The relational model is expected to quickly store and retrieve answers, thus minimizing the impact on YapTab's performance. With this concern in mind, two different database schemes were developed.

Multiple Table Schema To take full advantage of the relational model, data regarding the computed subgoal's answers is stored in several tables, aiming to keep the table space representation as small as possible in the database. Figure 2 shows the multiple table relational schema for the $f/2$ tabled predicate introduced back in Fig. 1.

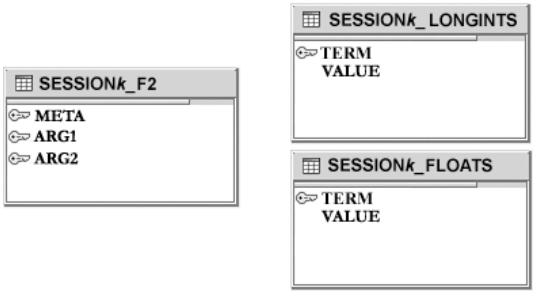


Fig. 2. Multiple table schema

The tabled predicate $f/2$ is mapped into the relational table SESSIONk_F2, where k is the current session id. Predicate arguments become the ARGi integer fields and the META field is used to tell apart the three kinds of possible records: a zero value signals an answer trie

branch; a one value signals a full bound subgoal trie branch and a positive value greater than one signals a subgoal with unbound variables within its arguments. Notice that with this representation, only short atomic terms can be directly stored within the corresponding ARGi integer fields. Long atomic terms are stored in the SESSIONk_LONGINTS and SESSIONk_FLOATS auxiliary tables. Each long atomic value appears only once and is uniquely identified by the key value stored in the TERM integer field.

Single Table Schema The multiple table schema may require several operations to store a single subgoal answer. For instance, for a subgoal such as $f/2$ with two floating-point bindings, five transactions may be required if the floating-point values have not been previously stored. To avoid over-heads in the storage operation, a simpler database schema has been devised (see Fig. 3).

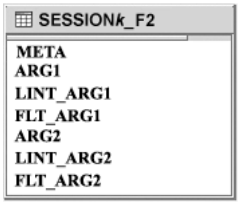


Fig. 3. Single table schema

Table SESSIONk_F2's design now considers the possibility of storage for long atomic terms. For that purpose, specifically typed fields are placed after each ARGi argument field. Regardless of this, each triplet is still considered a single argument for record-set manipulation purposes, hence a single field may be initialised to a value other than NULL; the others must remain unset.

3.2 The DBTAB API

The DBTAB's API, embedded in YapTab, provides a middleware layer between YAP and MySQL. We next present the developed API functions and briefly describe their actions.

`dbtab_init_session(MYSQL *handle, int sid)` uses the database handle to initialise the session identified by the `sid` argument.
`dbtab_kill_session(void)` kills the currently opened session.
`dbtab_init_table(TableEntry tab_ent)` initialises the INSERT prepared statement associated with `tab_ent` and creates the corresponding relational table.
`dbtab_free_table(TableEntry tab_ent)` frees the INSERT prepared statement associated with `tab_ent` and drops the corresponding table if no other instance is using it.
`dbtab_init_view(SubgoalFrame sg_fr)` initialises the specific SELECT prepared statement associated with `sg_fr`.
`dbtab_free_view(SubgoalFrame sg_fr)` frees the SELECT prepared statement associated with `sg_fr`.
`dbtab_store_answer_trie(SubgoalFrame sg_fr)` traverses both the subgoal trie and the answer trie, executing the INSERT prepared statement placed at the table entry associated with the subgoal frame passed by argument.
`dbtab_fetch_answer_trie(SubgoalFrame sg_fr)` starts a data retrieval transaction executing the SELECT prepared statement for `sg_fr`.

3.3 Top-Level Predicates

Two new predicates were added and two pre-existing ones were slightly changed to act as front-ends to the developed API functions. To start a session we must call the `tabling_init_session/2` predicate. It takes two arguments, the first being a database connection handler and the second being a *session identifier*. This identifier can be either a free variable or an integer term meaning, respectively, that a new session is to be initiated or a previously created one is to be reestablished. These arguments are then passed to the `dbtab_init_session()` function, which will return the newly (re)started session identifier. The `tabling_kill_session/0` terminates the currently open session by calling `dbtab_kill_session()`.

YapTab's directive `table/1` is used to set up the predicates for tabling. The DBTAB expanded version of this directive calls the `dbtab_init_table()` function for the corresponding table entry data structure. Figure 4 shows, labeled as **(1)** and **(2)**, the INSERT statements generated, respectively, to each storage schema by the `dbtab_init_table()` function for the call `':- table f/2'`.

(1) INSERT IGNORE INTO SESSIONk_F2(META,ARG1,ARG2) VALUES (?,?,?);
(2) INSERT IGNORE INTO SESSIONk_F2(META,ARG1,LINT1,FLT1,ARG2,LINT2,FLT2)
VALUES (?,?,?,?,?,?,?);
(3) SELECT F2.ARG1 AS ARG1, L.VALUE AS LINT1 FROM SESSIONk_F2 AS F2
LEFT JOIN SESSIONk_LONGINTS AS L ON (F2.ARG1=L.TERM)
WHERE F2.META=0 AND F2.ARG2=22;
(4) SELECT DISTINCT ARG1,LINT1 FROM SESSIONk_F2 WHERE META=0 AND ARG2=22;
(5) SELECT ARG1 FROM SESSIONk_F2 WHERE META>1 AND ARG2=22;

Fig. 4. Prepared statements for $f(Y, 1)$

The `abolish_table/1` built-in predicate can be used to abolish the tables for a tabled predicate. The DBTAB expanded version of this predicate calls the `dbtab_free_table()` function for the corresponding table entry and the `dbtab_free_view()` function for each subgoal frame under this entry.

3.4 Exporting Answers

Whenever the `dbtab_store_answer_trie()` function is called, a new data transaction begins. Given the subgoal frame to store, the function begins to climb the subgoal trie branch, binding every ground term it finds along the way to the respective parameter in the INSERT statement. When the root node is reached, all parameters consisting of variable terms will be left NULL. The attention is then turned to the answer trie and control proceeds cycling through the terms stored within the answer trie nodes. The remaining NULL parameters are bound repeatedly, and the prepared statement is executed for each present branch.

Next, a single record of meta-information is stored. The META field value is set to a bit field structure that holds the total number of variables in the subgoal call. The least significant bit is reserved to differentiate answers generated by full

ground subgoal trie branches from answer trie branches. The ARGi fields standing for variable terms present in the subgoal trie branch are bitwise masked with special markers, that identify each one of the possible types of long terms found in the answer trie and were meant to be unified with the original variable.

Figure 5 illustrates the final result of the described process using both storage schemes. When the subgoal trie is first climbed, ARG2 is bound to the integer term of value 1 (internally represented as 22). All values for ARG1 are then bound cycling through the leafs of the answer trie. The branch for the integer term of value 0 (internally represented as 6) is stored first, and the branch for the long integer term 2^{30} is stored next. Notice how, in the multiple table schema, the ARG1 field of the second record holds the key for the auxiliary table record. At last, the meta-information is inserted. This consists of a record holding in the META field the number of variables in the subgoal call (1 in this case, internally represented by 2) and in the ARGi fields the different terms found in the answer trie for the variables in the subgoal call along with the other ground arguments.

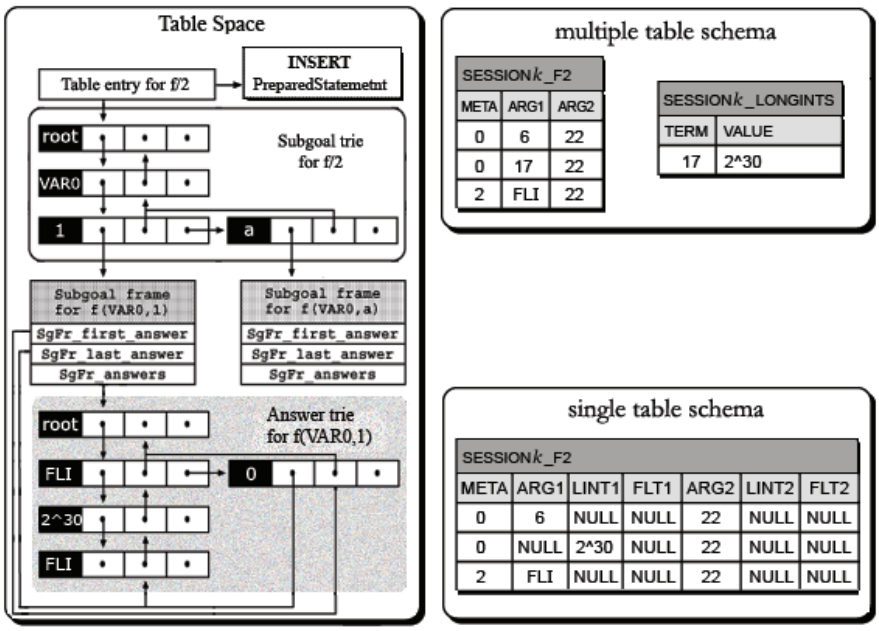


Fig. 5. Exporting $f(Y, 1)$ using both storage schemes

3.5 Importing Answers

To import answers from the database, we first call `dbtab_init_view()` in order to construct the specific SELECT statement used to fetch the answers for the subgoal. Function `dbtab_init_view()` first retrieves the meta-information from

the database and then it uses the ground terms in the meta-information record to refine the search condition within the WHERE part of the SELECT statement.

Figure 4 shows, labeled as **(3)** and **(4)**, the SELECT statements generated to each storage schema by the call to `dbtab_init_view()`. Notice that statement **(4)** bears a DISTINCT option. This is the way to prune repeated answers. Statement **(5)** is used by both schemes to obtain the meta-information record.

The storage schemes differ somewhat in the way the returned result-set is interpreted. The multiple table schema sets the focus on the ARGi fields, where no NULL values can be found. Additional columns, placed immediately to the right of the ARGi fields, are regarded as possible placeholders of answer terms only when these *main* fields convey long atomic term markers. In such a case, the non-NULL additional field value is used to create the specific YapTab term. The single table schema, on the other hand, requires no sequential markers for long atomic terms, hence, it makes no distinction what so ever between ARGi and its possibly following auxiliary fields. For each argument (single field, pair or triplet), the first non-NULL value is considered to be the correct answer term. Figure 6 shows, in the right boxes, the resulting *views* for each storage schema.

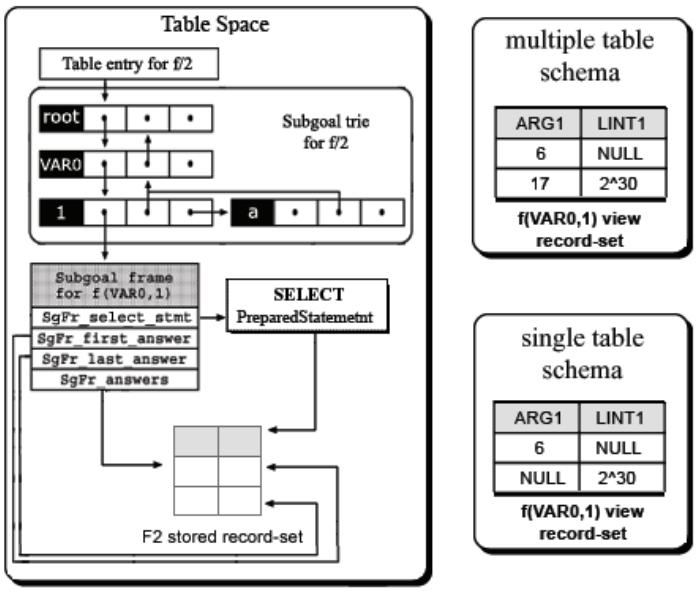


Fig. 6. Importing $f(Y, 1)$ using both storage schemes

3.6 Handling the Resulting Record-Sets

After the SELECT statement execution, two possible strategies may be used to supply the stored record-set with the answers back to the YapTab engine.

Rebuilding the Answer Trie In this scenario, the stored record-set is only used for answer trie rebuilding purposes. The set of retrieved values is sequentially traversed and inserted in the respective subgoal call, exactly as when the `tabled_new_answer` operation occurred. By the end of the process, the entire answer trie resides in the table space and the record-set can then be released from memory. This approach requires no alteration to the YapTab's implemented API.

Browsing the Record-Set In this approach, the stored record-set is kept in memory. Since the answer tries will not change once completed, all subsequent subgoal calls may fetch their answers from the obtained record-set. This is expected to lead to gains in performance since: **(i)** retrieval transaction occurs only once; **(ii)** no time and memory are spent rebuilding the answer trie; and **(iii)** long atomic term representation required down to one fourth of the usually occupied memory. Figure 6 illustrates how the ancillary YapTab constructs are used to implement this idea. The left side box presents the state of the subgoal frame after answer collection for $f(Y, 1)$. The internal pointers are set to the first and last rows of the record-set. When consuming answers, the first record's offset along with the subgoal frame address are stored in a *loader choice point*[1]. The fetched record and its field values are then used to bind the free variables found for the subgoal in hand. If backtracking occurs, the choice point is reloaded and the last recorded offset is used to step through to the next answer. When, at the end of the record-set, an invalid offset is reached, the loader choice point is discarded and execution fails, thus terminating the ongoing evaluation.

4 Initial Experimental Results

For comparison purposes, three main series of tests were performed both in YapTab and DBTAB environments (DBTAB with MySQL 5.0 running an InnoDB engine [8]) using a simple path discovery algorithm over a graph with 10,000, 50,000 and 100,000 possible combinations among nodes. In each series, two types of nodes were considered: integer and floating-point terms. Each setup was executed 10 times and the mean of measured times, in milliseconds, is presented next in Table 1. The environment for our experiments was an Intel Pentium®4 2.6GHz processor with 2 GBytes of main memory and running the Linux kernel-2.6.18.

The table shows two columns for YapTab, measuring the generation and browsing times when using tries to represent the table space, two columns for each of DBTAB storage schemes, measuring the times to export and import the respective number of answers and one last column, measuring the time to recover answers when using the approach that browses through the stored dataset. Some preliminary observations: **(i)** export and import times exclude the table generation time; **(ii)** when the trie is rebuilt after importing, this operation

[1] A loader choice point is a WAM choice point augmented with a pointer to the subgoal frame data structure and with the offset for the last consumed record.

Answers	Terms	YapTab		DBTAB				
				Multiple Table		Single Table		
		Generate	Browse	Export	Import	Export	Import	Browse
10,000	integers	65	1	1055	16	1048	34	2
	floats	103	2	10686	44	1112	47	6
50,000	integers	710	6	4911	76	5010	195	12
	floats	1140	8	83243	204	5012	282	27
100,000	integers	1724	11	9576	153	9865	392	20
	floats	1792	14	215870	418	11004	767	55

Table 1. Execution times, in milliseconds, for YapTab and DBTAB

duration is augmented with generation time; **(iii)** when using tries, YapTab and DBTAB spend the same amount of time browsing them.

As expected, most of DBTAB's execution time is spent in data transactions (export and import). Long atomic terms (floats) present the most interesting case. For storage purposes, the single table approach is clearly preferable. Due to the extra search and insertion on auxiliary tables in the multiple table approach, the export time of long atomic terms (floats) when compared with their short counter-part (integers) increases as the number of answers also increases. For 10,000 answers the difference is about 10 times more, while for 1000,000 the difference increases to 20 times more. On the other hand, the single table approach seems not to improve the import time, since it is, on average, the double of the time spent by the multiple table approach. Nevertheless, the use of LEFT JOIN clauses in the retrieval SELECT statement (as seen in Fig. 4) may become a heavy weight when dealing with larger data-sets. Further experiments with larger tables are required to provide a better insight on this issue.

Three interesting facts emerge from the table. First, the browsing times for tries and record-sets are relatively similar, with the later requiring, on average, the double of time to be completely scanned. Secondly, when the answer trie becomes very large, re-computation requires more time, almost the double, than the fetching (import plus browse) of its relational representation. DBTAB may thus become an interesting approach when the complexity of re-calculating the answer trie largely exceeds the amount of time required to fetch the entire answer record-set. Third, an important side-effect of DBTAB is the attained gain in memory consumption. Recall that trie nodes possess four fields each, of which only one is used to hold a symbol, the others being used to hold the addresses of parent, child and sibling nodes (please refer to section 2). Since the relational representation dispenses the three pointers and focus on the symbol storage, the size of the memory block required to hold the answer trie can be reduced by a factor of four. This is the worst possible scenario, in which all stored terms are integers or atoms. For floating-point numbers the reducing factor raises to eight because, although this type requires four trie nodes to be stored, one floating-point requires most often the size of two integers. For long integer terms, memory gains go up to twelve times, since three nodes are used to store them in the trie.

5 Conclusions and Further Work

In this work, we have introduced the DBTAB model. DBTAB was designed to be used as an alternative approach to the problem of recovering space when the tabling system runs out of memory. By storing tables externally instead of deleting them, DBTAB avoids standard tabled re-computation when subsequent calls to those tables appear. Another important aspect of DBTAB is the possible gain in memory consumption when representing answers for floating-point and long integer terms. Our preliminary results show that DBTAB may become an interesting approach when the cost of recalculating a table largely exceeds the amount of time required to fetch the entire answer record-set from the database. As further work we plan to investigate the impact of applying DBTAB to a more representative set of programs. We also plan to cover all possibilities for tabling presented by YapTab and extend DBTAB to support lists and application terms.

Acknowledgments

This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Pluri-anual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
3. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)
4. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
5. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
6. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
7. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
8. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
9. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74

Integrating XQuery and Logic Programming*

Jesús M. Almendros-Jiménez, Antonio Becerra-Terón
and Francisco J. Enciso-Baños

Dpto. Lenguajes y Computación.
Universidad de Almería. {jalmen,abecerra,fjenciso}@ual.es

Abstract. In this paper we investigate how to integrate the XQuery language and logic programming. With this aim, we represent XML documents by means of a logic program. This logic program represents the document schema by means of rules and the document itself by means of facts. Now, XQuery expressions can be integrated into logic programming by considering a translation from for-let-where-return expressions into logic rules and a goal.

1 Introduction

XQuery [W3C07b,CDF+04,Wad02,Cha02] is a typed functional language devoted to express queries against XML documents. It contains *XPath 2.0* [W3C07a] as a sublanguage. *XPath 2.0* supports navigation, selection and extraction of fragments from XML documents. *XQuery* also includes expressions to construct new XML values and to join multiple documents. The design of *XQuery* has been influenced by group members with expertise in the design and implementation of other high-level languages. *XQuery* has static typed semantics and a formal semantics which is part of the *W3C* standard [CDF+04,W3C07b].

The integration of *declarative programming* and *XML data processing* is a research field of increasing interest in the last years (see [BBFS05] for a survey). There are proposals of new languages for XML data processing based on functional, and logic programming. In addition, *XPath* and *XQuery* have been also implemented in declarative languages.

The most relevant contribution is the *Galax* project [MS03,CDF+04], which is an implementation of *XQuery* in functional programming, using *OCAML* as host language. There are also proposals for new languages based on functional programming rather than implementing *XQuery*. This is the case of *XDuce* [HP03] and *CDuce* [BCF05], which are languages for XML data processing, using regular expression pattern matching over XML trees, subtyping as basic mechanism, and *OCAML* as host language. The *CDuce* language does fully statically-typed transformation of XML documents, thus guaranteeing correctness. In addition, there are proposals around *Haskell* for the handling of XML documents, such as *HaXML* [Thi02] and [WR99].

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02.

There are also contributions in the field of use logic programming for the handling of XML documents. For instance, the *Xcerpt project* [SB02] proposes a pattern and rule-based query language for XML documents, using the so-called query terms including logic variables for the retrieval of XML elements. For this new language a specialized unification algorithm for query terms has been studied. Another contribution of a new language is *XPathLog* (the *LOPIX* system) [May04] which is a *Datalog*-style extension of *XPath* with variable bindings. This is also the case of *XCentric* [CF03], which can represent XML documents by means of logic programming, and handles XML documents by considering terms with functions of flexible arity and regular types. Finally, *FNPath* [Sei02] is a proposal for using Prolog as query language for XML documents based on a field-notation, for evaluating *XPath* expressions based on *DOM*. The Rule Markup Language (*RULEML*) [Bol01,Bol00] is a different kind of proposal in the research area. The aim of the approach is the representation of Prolog facts and rules into XML documents, and thus, the introduction of *rule systems* into the *Web*. Finally, some well-known Prolog implementations include libraries for loading and querying XML documents, such as *SWI-Prolog* [Wie05] and *CIAO* [CH01].

In this paper, we investigate how to integrate the *XQuery* language and logic programming. With this aim:

1. A XML document can be seen as a logic program, by considering *facts* and *rules* for expressing both the XML schema and document. This approach was already studied in our previous work [ABE07,ABE06].
2. A *XQuery expression can be translated into logic programming* by considering a set of rules and a specific goal. Taking as starting point the translation of *XPath* of our previous work [ABE07,ABE06], the translation of *XQuery* introduces *new rules* for the *join* of documents, and for the translation of *for-let-where-return* expressions into logic programming. In addition, a *specific goal* is generated for obtaining the answer to an *XQuery* query.
3. Our technique allows the handling of XML documents as follows. Firstly, the XML documents are loaded. It involves the translation of the XML documents into a logic program. For efficiency reasons the rules, which correspond to the XML document structure, are loaded in *main memory*, but facts, which represent the values of the XML document, are stored in *secondary memory*, whenever they do not fit in main memory (using appropriate *indexing techniques*). Secondly, the user can now write queries against the loaded documents. Now, *XQuery* queries are translated into a logic program and a specific goal. The evaluation of such goal uses the indexing in order to improve the efficiency of query solving. Finally, the answer of the goal can be represented by means of an output XML document.

As far as we know, this is the first time that *XQuery* is implemented in logic programming. Previous proposals either define new query languages for XML documents in logic and functional programming or implement *XQuery*, but in functional programming. The advantages of such proposal is that *XQuery* is embedded into logic programming, and thus *XQuery* can be combined with

logic programs. For instance, logic programming can be used as inference engine, one of the requirements of the so-called *Semantic Web* (`http://www.w3.org/2001/sw/`), in the line of *RuleML*.

Our proposal requires the representation of XML documents into logic programming, which can be compared with those ones representing XML documents in logic programming (for instance, [SB02,CF03]) and, with those ones representing XML documents in relational databases (for instance, [BGvK$^+$05]). In our case, rules are used for expressing the structure of well-formed XML documents, and XML elements are represented by means of facts. Moreover, our handling of XML documents is more "database-oriented" since we use secondary memory and file indexing for selective reading of records. The reason for such decision is that XML documents can usually be too big for main memory [MS03]. Our proposal uses as basis the implementation of *XPath* in logic programming studied in our previous work [ABE07]. In addition, we have studied how to consider a bottom-up approach to the same proposal of [ABE07] in [ABE06].

The structure of the paper is as follows. Section 2 will present the translation of XML documents into Prolog; section 3 will review the translation of *XPath* into logic programming; section 4 will provide the new translation of *XQuery* expressions into logic programming; and finally, section 5 will conclude and present future work. A more complete version of our paper (with a bigger set of examples) can be found in `http://www.ual.es/~jalmen`.

2 Translating XML Documents into Logic Programming

In order to define our translation we need to number the nodes of the XML document. A similar numbering has been already adopted in some proposals for representing XML in relational databases [OOP$^+$04,TVB$^+$02,BGvK$^+$05].

Given an XML document, we can consider a new XML document called *node-numbered XML document* as follows. Starting from the root element numbered as 1, the node-numbered XML document is numbered using an attribute called **nodenumber**[1] where each j-th child of a tagged element is numbered with the sequence of natural numbers $i_1.....i_t.j$ whenever the parent is numbered as $i_1.....i_t$: $< tag\ att_1 = v_1, \ldots, att_n = v_n, \mathbf{nodenumber}= \mathbf{i_1}..... \mathbf{i_t.j} > elem_1, \ldots, elem_s < /tag >$. This is the case of tagged elements. If the j-th child is of a basic type (non tagged) and the parent is an inner node, then the element is labeled and numbered as follows: $< unlabeled\ \mathbf{nodenumber} = \mathbf{i_1}.....\mathbf{i_t.j} > elem < /unlabeled >$; otherwise the element is not numbered. It gives to us a *hierarchical and left-to-right numbering* of the nodes of an XML document. An element in an XML document is further left in the XML tree than another when the node number is smaller w.r.t. the lexicographic order of sequence of natural numbers. Any numbering that identifies each inner node and leaf could be adapted to our translation.

In addition, we have to consider a new document called *type and node-numbered XML document* numbered using an attribute called **typenumber** as

[1] It is supposed that "nodenumber" is not already used as attribute in the tags of the original XML document.

follows. Starting the numbering from 1 in the root of the node-numbered XML document, each tagged element is numbered as: $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i_1 \ldots, i_t.j, \mathbf{typenumber = k} > elem_1, \ldots, elem_s < /tag >$. The type number k of the tag is equal to $l + n + 1$ whenever the type number of the parent is l, and n is the number of tagged elements weakly distinct [2] occurring in leftmost positions at the same level of the XML tree [3].

Now, the translation of the XML document into a logic program is as follows. For each inner node in the type and node numbered XML document $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i, typenumber = k > elem_1, \ldots, elem_s < /tag >$ we consider the following rule, called *schema rule*:

Schema Rule in Logic Programming

$$
\begin{array}{l}
tag(tagtype(Tag_{i_1}, \ldots, Tag_{i_t}, [Att_1, \ldots, Att_n]), NTag, k, Doc) \text{:-} \\
\qquad tag_{i_1}(Tag_{i_1}, [NTag_{i_1}|NTag], r, Doc), \\
\qquad \ldots, \\
\qquad tag_{i_t}(Tag_{i_t}, [NTag_{i_t}|NTag], r, Doc), \\
\qquad att_1(Att_1, NTag, r, Doc), \\
\qquad \ldots, \\
\qquad att_n(Att_n, NTag, r, Doc).
\end{array}
$$

where *tagtype* is a new function symbol used for building a Prolog term containing the XML document; $\{tag_{i_1}, \ldots, tag_{i_t}\}$, $i_j \in \{1, \ldots, s\}$, $1 \leq j \leq t$, is the *set of tags* of the tagged elements $elem_1, \ldots, elem_s$; $Tag_{i_1}, \ldots, Tag_{i_t}$ are variables; $att_1, \ldots, att_n$ are the attribute names; $Att_1, \ldots, Att_n$ are variables, one for each attribute name; $NTag_{i_1}, \ldots, NTag_{i_t}$ are variables (used for representing the last number of the node number of the children); $NTag$ is a variable (used for representing the node number of *tag*); k is the type number of *tag*; and finally, r is the type number of the tagged elements $elem_1, \ldots, elem_s$ [4].

In addition, we consider facts of the form: $att_j(v_j, i, k, doc)$ $(1 \leq j \leq n)$, where *doc* is the name of the document. Finally, for each leaf in the type and node numbered XML document: $< tag\ nodenumber = i, typenumber = k > value < /tag >$, we consider the *fact*: $tag(value, i, k, doc)$. For instance, let us consider the following XML document called "books.xml":

XML document

```
<books>
    <book year="2003">
        <author>Abiteboul</author>
        <author>Buneman</author>
        <author>Suciu</author>
        <title>Data on the Web</title>
        <review>A <em>fine</em> book.</review>
    </book>
```

[2] Two elements are weakly distinct whenever they have the same tag but not the same structure.

[3] In other words, type numbering is done by levels and in left-to-right order, but each occurrence of weakly distinct elements increases the numbering in one unit.

[4] Let us remark that since *tag* is a tagged element, then $elem_1, \ldots, elem_s$ have been tagged with "unlabeled" labels in the type and node numbered XML document when they were not labeled; thus they must have a type number.

```
    <book year="2002">
        <author>Buneman</author>
        <title>XML in Scotland</title>
        <review><em>The <em>best</em> ever!</em></review>
    </book>
</books>
```

Now, the previous XML document can be represented by means of a logic program as follows:

Translation into Prolog of an XML document

Rules (Schema):

```
books(bookstype(Book, []), NBooks,1,Doc) :-
        book(Book, [NBook|NBooks],2,Doc).
book(booktype(Author, Title, Review, [Year]),
        NBook ,2,Doc) :-
        author(Author, [NAu|NBook],3,Doc),
        title(Title, [NTitle|NBook],3,Doc),
        review(Review, [NRe|NBook],3,Doc),
        year(Year, NBook,3,Doc).
review(reviewtype(Un,Em,[]),NReview,3,Doc):-
    unlabeled(Un,[NUn|NReview],4,Doc),
    em(Em,[NEm|NReview],4,Doc).
review(reviewtype(Em,[]),NReview,3,Doc):-
    em(Em,[NEm|NReview],5,Doc).
em(emtype(Unlabeled,Em,[]),NEms,5,Doc) :-
    unlabeled(Unlabeled,[NUn|NEms],6,Doc),
    em(Em, [NEm|NEms],6,Doc).
```

Facts (Document):

```
year('2003', [1, 1], 3,"books.xml").
author('Abiteboul', [1, 1, 1], 3,"books.xml").
author('Buneman', [2,1, 1], 3,"books.xml").
author('Suciu', [3,1,1], 3,"books.xml").
title('Data on the Web', [4, 1, 1], 3,"books.xml").
unlabeled('A', [1, 5, 1, 1], 4,"books.xml").
em('fine', [2, 5, 1, 1], 4,"books.xml").

unlabeled('book.', [3, 5, 1, 1], 4,"books.xml").
year('2002', [2, 1], 3,"books.xml").
author('Buneman', [1, 2, 1], 3,"books.xml").
title('XML in Scotland', [2, 2, 1], 3,"books.xml").
unlabeled('The', [1, 1, 3, 2, 1], 6,"books.xml").
em('best', [2, 1, 3, 2, 1], 6,"books.xml").
unlabeled('ever!', [3, 1, 3, 2, 1], 6,"books.xml").
```

Here we can see the translation of each tag into a predicate name: *books*, *book*, etc. Each predicate has four arguments, the first one, used for representing the XML document structure, is encapsulated into a function symbol with the same name as the tag adding the suffix *type*. Therefore, we have *bookstype*, *booktype*, etc. The second argument is used for numbering each node; the third argument of the predicates is used for numbering each type; and the last argument represents the document name. The key element of our translation is to be able to recover the original XML document from the set of rules and facts.

3 Translating XPath into Logic Programming

In this section, we present how *XPath* expressions can be translated into a logic program. Here we present the basic ideas, a more detailed description can be found in [ABE07].

We restrict ourselves to *XPath* expressions of the form $xpathexpr = /expr_1 \ldots /expr_n$ where each $expr_i$ can be a tag or a *boolean condition* of the form $[xpathexpr = value]$, where $value$ has a basic type. More complex *XPath* queries [W3C07a] will be expressed in *XQuery*, and therefore they will be translated in next section.

With the previous assumption, each *XPath* expression $xpathexpr = /expr_1 \ldots /expr_n$ defines a *free of equalities XPath expression*, denoted by $FE(xpathexpr)$. This free of equalities *XPath* expression defines a subtree of the XML document, in which is required that some paths exist (occurences of boolean conditions $[xpathexpr]$). For instance, with respect to the *XPath* expression */books/book*

$[author = Suciu]/title$, the free of equalities *XPath* expression is $/books/book\ [author]\ /title$ and the subtree of the type and node numbered XML document which corresponds with the expression $/books/book\ [author]/title$ is as follows:

Subtree defined by a Free of Equalities XPath Expression

```
<books nodenumber=1, typenumber=1>
<book year="2003", nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Abiteboul</author>
<author nodenumber=1.1.2 typenumber=3>Buneman</author>
<author nodenumber=1.1.3 typenumber=3>Suciu</author>
<title nodenumber=1.1.4 typenumber=3>Data on the Web</title>
</book>
<book year="2002" nodenumber=1.2, typenumber=2>
<author nodenumber=1.2.1 typenumber=3>Buneman</author>
<title nodenumber=1.2.2 typenumber=3>XML in Scotland</title>
</book>
</books>
```

Now, given a type and node numbered XML document $\mathcal{D}$, a program $\mathcal{P}$ representing $\mathcal{D}$, and an *XPath* expression *xpathexpr* then the *logic program obtained from xpathexpr* is $\mathcal{P}^{xpathexpr}$, obtained from $\mathcal{P}$ taking the schema rules and facts for the subtree of $\mathcal{D}$ defined by $FE(xpathexpr)$. For instance, with respect to the above example, the schema rules defined by $/books/book\ [author]/title$ are:

Translation into Prolog of an XPath Expression

```
books(bookstype(Book, []), NBooks, 1,Doc):-
        book(Book, [NBook|NBooks], 2,Doc).
book(booktype(Author,Title,Review,[Year]),NBook,2,Doc) :-
        author(Author,[NAuthor|NBook],3,Doc),
        title(Title,[NTitle|NBook],3,Doc).
```

and the facts, the set of facts for *title* and *author*. Let us remark that in practice, these rules can be obtained from the schema rules by removing predicates, that is, removing the predicates in the schema rules which are not tags in the free of equalities *XPath* expression.

Now, given a type and node numbered XML document, and an *XPath* expression *xpathexpr*, the *goals obtained from xpathexpr* are defined as follows. Firstly, each *XPath* expression *xpathexpr* can be mapped into a set of prolog terms, denoted by $PT(xpathexpr)$, representing the *pattern* of the query [5]. Basically, the pattern represents the required structure of the record. Now, the *goals* are defined as: $\{: -tag(Tag, Node, r, doc)\ \{Tag \rightarrow t\ \} |\ t \in PT(xpathexpr), r\ is\ a\ type\ number\ of\ tag\ for\ t\}$ where *tag* is the leftmost tag in *xpathexpr* with a boolean condition (and it is the rightmost tag whenever boolean conditions do not exist); Tag and $Node$ are variables; and *doc* is the document name.

For instance, with respect to $/books/book\ [author = Suciu]/title$, $PT(/books/book\ [author = Suciu]/title) = \{booktype('Suciu', Title, Review, [Year])\}$, and therefore the (unique) goal is : $-book(booktype('Suciu', Title, Review, Year), Node, 2, "books.xml")$.

We will call to the leftmost tag with a boolean condition the *head tag* of *xpathexpr* and is denoted by $htag(xpathexpr)$. In the previous example, $htag(/books/book[author = Suciu]/title) = book$.

[5] Due to XML records can have different structure, one pattern is generated for each kind of record.

In summary, the handling of an *XPath* query involves the *"specialization"* of the schema rules of the XML document and the *generation* of one or more goals. The goals are obtained from the leftmost tag with a boolean condition on the *XPath* expression. Obviously, instead of a set of goals for each *XPath* expression, a unique goal can be considered by adding new rules. In the following we will assume this case.

4 Translating XQuery into Logic Programming

Similarly to *XPath*, an *XQuery* expression is translated into a logic program and generates a specific goal. We focus on the *XQuery* core language, whose grammar can be defined as follows.

Core XQuery

xquery:= $dxpfree| < tag >' \{'xquery, \ldots, xquery'\}' < /tag > |flwr.$
dxpfree:= document(doc) '/' xpfree.
flwr:= **for** *$var* **in** *vxpfree [***where** *constraint]* **return** *xqvar*
| **let** *$var* **:=** *vxpfree [***where** *constraint]* **return** *xqvar.*
xqvar:= $vxpfree| < tag >' \{'xqvar, \ldots, xqvar'\}' < /tag > |flwr.$
vxpfree:= $var | $var '/' xpfree | dxpfree.
Op:= $<= | >= | < | > | =.$
constraint := vxpfree Op value | vxpfree Op vxpfree
| constraint **'or'** *constraint | constraint* **'and'** *constraint.*

where *value* is an XML document, *doc* is a document name, and $xpfree$ is a free of equalities *XPath* expression. Let us remark that *XQuery* expressions use free of equalities *XPath* expressions, given that equalities can be always introduced in *where* expressions. Finally, we will say that an *XQuery* expression *ends with attribute name* in the case of the *XQuery* expression has the form *vxpfree* and the rightmost element has the form *@att*, where *att* is an attribute name. The translation of an *XQuery* expression consists of three elements.

- Firstly, for each *XQuery* expression *xquery*, we can define analogously to *XPath* expressions, the so-called *head tag*, denoted by $htag(xquery)$, which is the *predicate name* used for the building of the goal (or subgoal whenever the expression *xquery* is nested).
- Secondly, for each *XQuery* expression *xquery*, we can define the so-called *tag position*, denoted by $tagpos(xquery)$, representing the argument of the head tag (i.e. the argument of the predicate name) in which the answer is retrieved.
- Finally, for each *XQuery* expression *xquery* we can define a logic program $\mathcal{P}^{xquery}$ and a specific goal.

In other words, each *XQuery* expression can be mapped in the translation into a program $\mathcal{P}^{xquery}$ and into a goal of the form : $-tag(Tag_1, \ldots, Tag_n, Node, Type, Docs)$ where *tag* is the head tag, and Tag_{pos} represents the answer of the query, where $pos = tagpos(xquery)$. In addition, $Node$ and $Type$ represent the node and type numbering of the output document, and $Docs$ represents the documents involved in the query. The above elements are defined in Tables 2 and 3 for each case, assuming the notation of Table 1.

Table 1. Notation

$Vars(\Gamma) = \{\$var \mid (\$var, let, vxpfree, C) \in \Gamma \text{ or } (\$var, for, vxpfree, C) \in \Gamma\}$;
$Doc(\$var, \Gamma) = doc$ whenever $\Gamma_{\$var} = document(doc)/xpfree$;
$DocVars(\Gamma) = \{\$var \mid (\$var, let, dxpfree, C) \in \Gamma \text{ or } (\$var, for, dxpfree, C) \in \Gamma\}$;
$\Gamma_{\$var} = vxpfree$ whenever $(\$var, let, vxpfree, C)$ or $(\$var, for, vxpfree, C) \in \Gamma$;
$\Gamma_{\$var} = vxpfree[\lambda_1 \cdot \ldots \cdot \lambda_n]$ where $\lambda_i = \{\$var_i \rightarrow \Gamma_{\$var_i}\}$ and $\{\$var_1, \ldots, \$var_n\} = Vars(\Gamma)$;
$Root(\$var) = \$var'$ whenever $\$var \in DocVars(\Gamma)$ and $\$var = \$var'$ or $(\$var, let, \$var''/xpfree, C) \in \Gamma$ *or* $(\$var, for, \$var''/xpfree, C) \in \Gamma$ and $Root(\$var'') = \$var'$;
$Rootedby(\$var, \mathcal{X}) = \{xpfree \mid \$var/xpfree \in \mathcal{X}\}$;
$Rootedby(\$var, \Gamma) = \{xpfree \mid \$var/xpfree\ Op\ \ vxpfree \in C$ or $\$var/xpfree\ \ Op\ value \in C, C \in Constraints(\$var, \Gamma)\}$; and
$Constraints(\$var, \Gamma) = \{C_i \mid C \equiv C_1\ Op\ \ldots\ Op\ C_n,$ $(\$var, let, vxpfree, C) \in \Gamma$ *or* $(\$var, for, vxpfree, C) \in \Gamma\}$

4.1 Examples

Let us suppose a query requesting the year and title of the books published before 2003.

```
xquery = for $book in document ('books.xml')/books/book
         return let $year := $book/@year
         where $year<2003
         return <mybook>{$year, $book/title}</mybook>
```

For this query, the translation is as follows:

$$\mathcal{P}^{xquery} = \mathcal{P}^{xquery2}_{(book,for,document('books.xml')/books/book,\emptyset)} =$$
$$\mathcal{P}^{xquery3}_{(book,for,document('books.xml')/books/book,\emptyset),(\$year,let,\$book/@year,\$year<2003)} =$$
$$\{\mathcal{R}\} \cup \mathcal{P}^{\$year,\$book/title}_{(book,for,document('books.xml')/books/book,\emptyset),(\$year,let,\$book/@year,\$year<2003)}$$
$$\mathcal{R} =$$
$$\boxed{mybook(mybooktype(Title,[Year]),[Node],[Type],[Doc]) :- \\ join(Title,Year,Node,Type,Doc).}$$
$$\%\ \{join\} = \{htag(\$year), htag(\$book/title)\};$$
$$\%\ tagpos(\$year) = 1\ and\ tagpos(\$book/title) = 2$$
$$\mathcal{P}^{\$year,\$book/title}_{(\$book,for,document('books.xml')/books/book,\emptyset),(\$year,let,\$book/@year,\$year<2003)} =$$
$$\{\mathcal{J}^{\Gamma}\} \cup \mathcal{C}^{\Gamma} \cup \{\mathcal{R}^{\$book}\}$$

$$\cup \mathcal{P}^{document('books.xml')/books/book/@year} \cup \mathcal{P}^{document('books.xml')/books/book/title}$$
$$\mathcal{J}^{\Gamma} =$$
$$\boxed{join(Title,Year,[Node],[Type],[Doc]) :- \\ vbook(Title,Year,Node,Type,Doc), \\ constraints(vbook(Title,Year)).}$$
$$\%\ DocVars(\Gamma) = \{\$book\},\ \$year, \$book/title \in \mathcal{X}$$
$$\%\ Root(\$year) = \$book,\ Root(\$book) = \$book.$$
$$\mathcal{C}^{\Gamma} =$$
$$\boxed{constraints(Vbook) : -lc^1_1(Vbook). \\ lc^1_1(Vbook) : -c^1_1(Vbook). \\ c^1_1(vbook(Title,Year)) : -leq(Year,2003).}$$
$$\%\ C^1 \equiv c^1_1,\ c^1_1 \equiv \$year < 2003$$
$$\%\ C^1 \in constraints(\$year, \Gamma)\ and\ Root(\$year) = \$book$$
$$\mathcal{R}^{\$book} =$$
$$\boxed{vbook(Title,Year,[Node,Node],[TTitle,TYear],'books.xml') :- \\ title(Title,[NTitle|Node],TTitle,'books.xml'), \\ year(Year,Node,TYear,'books.xml').}$$

Table 2. Translation of XQuery into Logic Programming

$\mathcal{P}^{document(doc)/xpfree} =_{def} \mathcal{P}^{xpfree}$ $htag(document(doc)/xpfree) =_{def} htag(xpfree)$ $tagpos(document(doc)/xpfree) =_{def} tagpos(xpfree)$	
$\mathcal{P}^{<tag>\{xquery\_1,\ldots,xquery\_n\}</tag>} =_{def}$ $\{\mathcal{R}\} \cup_{1\le i\le n} \mathcal{P}^{xquery_i}$ $\mathcal{R} \equiv$ $tag(tagtype(Tag^1_{p_1}, \ldots, Tag^k_{p_k}, [Att^1_{q_1}, \ldots, Att^s_{q_s}]),$ $[NTag_1, \ldots, NTag_k, NAtt_1, \ldots, NAtt_s],$ $[TTag_1, \ldots, TTag_k, TAtt_1, \ldots, TAtt_s],$ $[DTag_1, \ldots, DTag_k, DAtt_1, \ldots, DAtt_s]) :-$ $tag_1(\overline{Tag^1}, NTag_1, TTag_1, DTag_1),$ $\ldots$ $tag_k(\overline{Tag^k}, NTag_k, TTag_k, DTag_k),$ $att_1(\overline{Att^1}, NAtt_1, TAtt_1, DAtt_1),$ $\ldots$ $att_s(\overline{Att^s}, NAtt_s, TAtt_s, DAtt_s).$ $htag(xquery) =_{def} tag,\ tagpos(xquery) =_{def} 1$	$\overline{Tag^t}\ 1 \le t \le k,$ *denotes* $Tag^t_1, \ldots, Tag^t_r$ *where* r *is the arity of* tag_t*;* $\overline{Att^j}\ 1 \le j \le s,$ *denotes* $Att^j_1, \ldots, Att^j_s$ *where* s *is the arity of* att_j*;* $htag(xquery_j) = att_i,\ 1 \le i \le s,$ *for some* $j \in \{1, \ldots, n\}$ *which ends with attribute names,* $htag(xquery_j) = tag_t,\ 1 \le t \le k,$ *otherwise* $tagpos(xquery_j) = q_i$ *and* $tagpos(xquery_j) = p_t$ *in the same cases.*
$\mathcal{P}^{\textbf{for } \$var \textbf{ in } vxpfree\ [\textbf{where } C]\ \textbf{return } xqvar} =_{def}$ $\mathcal{P}^{xqvar}_{\{(\$var,for,vxpfree,C)\}}$ $htag\ (xquery) =_{def} htag(xqvar)$ $tagpos(xquery) =_{def} tagpos(xqvar)$	
$\mathcal{P}^{\textbf{let } \$var := vxpfree\ [\textbf{where } C]\ \textbf{return } xqvar} =_{def}$ $\mathcal{P}^{xqvar}_{\{(\$var,let,vxpfree,C)\}}$ $htag\ (xquery) =_{def} htag(xqvar)$ $tagpos(xquery) =_{def} tagpos(xqvar)$	
$\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def}$ $\{\mathcal{R}\} \cup \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\} \cup_{1\le i\le n}\{xqvar_i/xpfree_0\}}_{\Gamma}$ $\mathcal{R} \equiv$ $tag(tagtype(Tag_1, \ldots, Tag_r, [Att^1, \ldots, Att^m]),$ $[Node_1, \ldots, Node_s],$ $[Type_1, \ldots, Type_s],$ $[Doc_1, \ldots, Doc_s]) :-$ $tag_1(\overline{Tag^1}, Node_1, Type_1, Doc_1),$ $\ldots$ $tag_s(\overline{Tag^s}, Node_s, Type_s, Doc_s).$ $htag(xquery/xpfree) =_{def} tag$ $tagpos(xquery/xpfree) =_{def} 1$	$xquery \equiv$ $< tag > \{xqvar_1, \ldots,$ $xqvar_n\} < /tag >$ $xquery/xpfree \in \mathcal{X}$ $xpfree \equiv /tag/xpfree_0$ $\{tag_1, \ldots, tag_s\} =$ $\{htag\ (xqvar_1\ /xpfree_0),$ $\ldots,\ htag\ (xqvar_n\ /xpfree_0)\};$ $Tag_i = Tag^j_{p_j},\ 1 \le i \le r,$ *whenever* $tagpos(xqvar_p\ /xpfree_0) = p_j,$ $htag(xqvar_p\ /xpfree_0) = tag_j,$ $p \in \{1, \ldots, n\};$ $Att_l = Tag^j_{p_j}\ 1 \le l \le s,$ *whenever* $tagpos(xqvar_p\ /xpfree_0) = p_j,$ $htag(xqvar_p\ /xpfree_0) = tag_j$ $p \in \{1, \ldots, n\}$ $xqvar_p\ /xpfree_0$ *ends with attribute names*
$\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\} \cup \{xqvar/xpfree\}}_{\Gamma \cup \{(\$var,for,vxpfree,C)\}}$ $htag(xquery/xpfree) =_{def} htag(xqvar/xpfree)$ $tagpos(xquery/xpfree) =_{def} tagpos(xqvar/xpfree)$	$xquery \equiv$ **for** $\$var$ **in** $vxpfree$ [**where** C] **return** $xqvar$ $xquery/xpfree \in \mathcal{X}$
$\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\} \cup \{xqvar/xpfree\}}_{\Gamma \cup \{(\$var,let,vxpfree,C)\}}$ $htag(xquery/xpfree) =_{def} htag(xqvar/xpfree)$ $tagpos(xquery/xpfree) =_{def} tagpos(xqvar/xpfree)$	$xquery \equiv$ **let** $\$var := vxpfree$ [**where** C] **return** $xqvar$ $xquery/xpfree \in \mathcal{X}$

$\%$ "$books.xml$" $= Doc(\$book, \Gamma)$
$\%\ htag(document\ ('books.xml')\ /books/book\ /@year) = year$
$\%\ htag(document\ ('books.xml')\ /books/book\ /title) = title$
$\%\ \overline{\Gamma}_{\$year} = document\ ('books.xml')\ /books/book/$
$\%\ \overline{\Gamma}_{\$book} = document\ ('books.xml')\ /books/book/$
$\%\ \$year, \$book/title \in \mathcal{X}$
$\mathcal{P}^{document('books.xml')/books/book/@year} = \emptyset$
$\mathcal{P}^{document('books.xml')/books/book/title} = \emptyset$

Table 3. Translation of XQuery into Logic Programming (cont'd)

$$\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \{\mathcal{J}^{\Gamma}\} \cup \mathcal{C}^{\Gamma} \cup \{\mathcal{R}^{\$var} | \$var \in DocVars(\Gamma)\} \cup \bigcup_{\substack{\$var \in DocVars(\Gamma),\\ \$var = Root(\$var'),\\ xpfree \in Rootedby(\$var', \mathcal{X}) \cup Rootedby(\$var', \Gamma)}} \mathcal{P}^{\overline{\Gamma}_{\$var/xpfree}} \quad (1)$$

(1) – $\mathcal{X}$ does not includes tagged elements and flwr expressions

$$\mathcal{J}^{\Gamma} \equiv join(Tag_1, \ldots, Tag_m, [Node_1, \ldots, Node_n], [Type_1, \ldots, Type_n], [Doc_1, \ldots, Doc_n]) :- vvar_1(\overline{Tag^1}, Node_1, Type_1, Doc_1), \ldots vvar_n(\overline{Tag^n}, Node_n, Type_n, Doc_n), constraints(vvar_1(\overline{Tag^1}), \ldots, vvar_n(\overline{Tag^n})). \quad (2)$$

(2)
- $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$;
- $Tag_j = Tag^i_{p_j}$ $\$var'/xpfree_j \in \mathcal{X}$ $Root(\$var') = \$var_i$ one p_j for each $\$var'/xpfree_j$
- $\overline{Tag^i} = Tag^i_1 \ldots Tag^i_s$ Tag^i_r, $1 \leq r \leq s$ one Tag^i_r for each $\$var'/xpfree_r \in \mathcal{X}$ $Root(\$var') = \$var_i$ and one Tag^i_r for each $\$var'/xpfree_r$ in Γ $Root(\$var') = \$var_i$

$$\mathcal{R}^{\$var} \equiv vvar(Tag_1, \ldots, Tag_n, Node, [Type_1, \ldots, Type_n], doc) :- tag_1(Tag_1, [Node_{11}, \ldots, Node_{1k_1} | NTag], Type_1, doc), \ldots, tag_n(Tag_n, [Node_{n1}, \ldots, Node_{nk_n} | NTag], Type_n, doc). \quad (3)$$

(3)
- $doc = Doc(\$var, \Gamma)$
- $tag_i = htag(\overline{\Gamma}_{\$var/xpfree})$ $\$var' /xpfree \in \mathcal{X}$ $\$var = Root(\$var')$
- $Node = [N_1, \ldots, N_n]$ $N_i = [Node_{ik_i} | NTag]$ *if* $(\$var', for, vxpfree, C) \in \Gamma$ $N_i = NTag$ *otherwise*

$$\mathcal{C}^{\Gamma} \equiv \{ constraints(Vvar_1, \ldots, Vvar_n) :- lc^1_1(Vvar_1, \ldots, Vvar_n), \ldots lc^n_1(Vvar_1, \ldots, Vvar_n). \} \cup_{\$var \in Vars(\Gamma), C^j \in constraints(\$var, \Gamma)} \mathcal{C}^j \quad (4)$$

(4) $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$;

$$\mathcal{C}^j \equiv \{lc^j_i(Vvar_1, \ldots, Vvar_n) :- c^j_i(Vvar_1, \ldots, Vvar_n), lc^j_{i+1}(Vvar_1, \ldots, Vvar_n). \mid 1 \leq i \leq n, Op_i = \textbf{and}\} \cup \{lc^j_i(Vvar_1, \ldots, Vvar_n) : -c^j_i(Vvar_1, \ldots, Vvar_n). lc^j_i(Vvar_I, \ldots, Vvar_n) :- lc^j_{i+1}(Vvar_I, \ldots, Vvar_n). \mid 1 \leq i \leq n, Op_i = \textbf{or}\} \cup_{\{c^j_i | C^j \equiv c^j_1 Op_1 \ldots Op_n c^j_n\}} \{\mathcal{C}^j_i\} \quad (5)$$

(5) $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$.

$$\mathcal{C}^j_i \equiv c^j_i(vvar_1(\overline{Tag^1}), \ldots, vvar_n(\overline{Tag^n})) :- Op(Tag^k_j, value). whenever\ c^j_i \equiv \$var'/xpfree_j\ Op\ value\ and\ Root(\$var') = \$var_k$$

$$\mathcal{C}^j_i \equiv c^j_i(vvar(\overline{Tag^1}), \ldots, vvar(\overline{Tag^n})) :- Op(Tag^k_j, Tag^m_r). whenever\ c^j_i \equiv \$var'/xpfree_j\ Op\ \$var'/xpfree_r, Root(\$var') = \$var_k\ and\ Root(\$var') = \$var_m \quad (6)$$

(6) $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$.

$$htag(\$var/xpfree_j) =_{def} join \quad (7)$$
$$tagpos(\$var/xpfree_j) =_{def} j \quad (7)$$

(7) for every $\$var \in Vars(\Gamma)$, $xpfree_j \in Rootedby(\$var, \mathcal{X}) \cup Rootedby(\$var, \Gamma)$

Basically, the translation of *XQuery* expressions produces new rules (in the example *mybook*) having the form of "*views*" in which a *"join"* of documents is achieved (the *join* predicate makes the join). The join combines the values for *local variables whose value is the root of the input documents* (in the example $*book* whose value is computed by *vbook*). The join also takes into account the constraints on these local variables (predicate *constraints*). Finally, for these local variables the set of required paths is computed. In the example, there is a local variable $*book* whose value is the root of the document, and *title* and *year* are the required paths computed by *vbook*.

Now, we can build the goal for obtaining the answer for *xquery* as follows. Taking $htag(xquery) = mybook$ and $tagpos(xquery) = 1$ then the goal is $: -mybook(MyBook, Node, Type, Doc)$ and the answer is:

$MyBook = mybooktype("XML\ in\ Scottland", ["2002"])$, $Node = [[[1,2],[1,2]]]$
$Type = [[[3,3]]]$, $Doc = [["books.xml"]]$

This answer represents the XML document:

Answer as an XML document

```
<mybook year="2002">
    <title>XML in Scotland</title>
</mybook>
```

Let us remark that the output document is not numbered as the source documents. The join of several documents with different node and type numbering produces an unique output document. However, the output document is still indexed and typed by considering the list of node and type numbers of the input documents. In the example the first $[1,2]$ represents the node number of the book titles, and the second $[1,2]$ represents the node number of the book years. Analogously, the first "3" represents the type number of book titles and the second "3" the type number of book years. The numbering of output documents still allows the recovering of the hierarchical structure by considering the lexicographic order in lists. Due to the lack of space we omit here the details about the reconstruction of output documents.

5 Conclusions and Future Work

In this paper, we have studied how to translate *XQuery* expressions into logic programming. It allow us to evaluate *XQuery* expressions against XML documents using logic rules. As future work we would like to implement our technique. We have already implemented *XPath* in logic programming (see `http://indalog.ual.es/Xindalog`). Taking as basis this implementation we would like to extend it to *XQuery* expressions.

References

[ABE06] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Magic sets for the XPath language. *Journal of Universal Computer Science*, 12(11):1651–1678, 2006.

[ABE07] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Querying XML documents in logic programming. *To appear in Theory and Practice of Logic Programming, available at* `http://www.ual.es/~jalmen`, 2007.

[BBFS05] James Bailey, Franois Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web Query Languages: A Survey. In *Reasoning Web, First International Summer School*, volume 3564, pages 35–133. LNCS, 2005.

[BCF05] Veronique Benzaken, Giuseppe Castagna, and Alain Frish. CDuce: an XML-centric general-purpose language. In *Procs of the ACM ICFP*, pages 51–63. ACM Press, 2005.

[BGvK+05] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery - The Relational Way. In *Procs. of the VLDB*, pages 1322–1325. ACM Press, 2005.

[Bol00] H. Boley. Relationships between logic programming and XML. In *Proceedings of the Workshop on Logic Programming*, 2000.

[Bol01] H. Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Proc. of International Conference on Applications of Prolog, INAP*, pages 124–139. Prolog Association of Japan, 2001.

[CDF+04] D. Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, and Philip Wadler. *XQuery from the Experts*. Addison Wesley, 2004.

[CF03] Jorge Coelho and Mario Florido. Type-based XML processing in logic programming. In *Proceedings of the PADL 2003*, pages 273–285. LNCS 2562, 2003.

[CH01] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *TPLP*, 1(3):251–282, 2001.

[Cha02] D. Chamberlin. XQuery: An XML Query Language. *IBM Systems Journal*, 41(4):597–615, 2002.

[HP03] H. Hosoya and B. C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology, TOIT*, 3(2):117–148, 2003.

[May04] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming, TPLP*, 4(3):239–287, 2004.

[MS03] A. Marian and J. Simeon. Projecting XML Documents. In *Procs. of VLDB*, pages 213–224. Morgan Kaufmann, 2003.

[OOP+04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *Procs. of the ACM SIGMOD*, pages 903 – 908. ACM Press, 2004.

[SB02] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of RuleML*, 2002.

[Sei02] D. Seipel. Processing XML-Documents in Prolog. In *Procs. of the Workshop on Logic Programming 2002*, 2002.

[Thi02] Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, 2002.

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *Procs of the ACM SIGMOD*, pages 204–215. ACM Press, 2002.

[W3C07a] W3C. XML Path language (XPath) 2.0. Technical report, www.w3.org, 2007.

[W3C07b] W3C. XQuery 1.0: An XML Query Language. Technical report, www.w3.org, 2007.

[Wad02] P. Wadler. XQuery: A Typed Functional Language for Querying XML. In *Advanced Functional Programming, 4th International School, AFP*, LNCS 2638, pages 188–212. Springer, 2002.

[Wie05] J. Wielemaker. SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam, March 2005.

[WR99] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the International Conference on Functional Programming*, pages 148–159. ACM Press, 1999.

Causal Subgroup Analysis for Detecting Confounding

Martin Atzmueller and Frank Puppe

University of Würzburg,
Department of Computer Science VI
Am Hubland, 97074 Würzburg, Germany
{atzmueller, puppe}@informatik.uni-wuerzburg.de

Abstract. This paper presents a causal subgroup analysis approach for the detection of confounding: We show how to identify (causal) relations between subgroups by generating an extended causal subgroup network utilizing background knowledge. Using the links within the network we can identify relations that are potentially confounded by external (confounding) factors. In a semi-automatic approach, the network and the discovered relations are then presented to the user as an intuitive visualization. The applicability and benefit of the presented technique is illustrated by examples from a case-study in the medical domain.

1 Introduction

Subgroup discovery (e.g., [1–4]) is a powerful approach for explorative and descriptive data mining to obtain an overview of the interesting dependencies between a specific target (dependent) variable and usually many explaining (independent) variables. The interesting subgroups can be defined as subsets of the target population with a (distributional) unusualness concerning a certain property we are interested in: The risk of coronary heart disease (target variable), for example, is significantly higher in the subgroup of smokers with a positive family history than in the general population.

When interpreting and applying the discovered relations, it is often necessary to consider the patterns in a causal context. However, considering an association as having a causal interpration can often lead to incorrect results, due to the basic tenet of statistical analysis that association does not imply causation (cf. [5]): A subgroup may not be causal for the target group, and thus can be suppressed by other causal groups. Then, the suppressed subgroup itself is not interesting, but the other subgroups are better suited for characterizing the target concept. Furthermore, the estimated *effect*, i.e., the quality of the subgroup may be due to associations with other *confounding* factors that were not considered in the quality computation. For instance, the quality of a subgroup may be confounded by other variables that are associated with the independent variables, and are a direct cause of the (dependent) target variable. Then, it is necessary to identify potential confounders, and to measure or to control their influence concerning the subgroup and the target concept. Let us assume, for example, that ice cream consumption and murder rates are highly correlated. However, this does not necessarily mean that ice cream incites murder or that murder increases the demand for ice cream. Instead, both ice cream and murder rates might be joint effects of a common cause or confounding factor, namely, hot weather.

In this paper, we present a semi-automatic causal subgroup analysis approach for the detection of confounding. We use known subgroup patterns as background knowledge that can be incrementally refined: These patterns represent subgroups that are *acausal*, i.e., have no causes, and subgroup patterns that are known to be directly causally related to other (target) subgroups. Additionally, both concepts can be combined, for example, in the medical domain certain variables such as *Sex* have no causes, and it is known that they are causal risk factors for certain diseases. Using the patterns contained in the background knowledge, and a set of subgroups for analysis, we can construct a causal net containing relations between the subgroups. This network can be interactively inspected and analyzed by the user: It directly provides a visualization of the (causal) relations between the subgroups, and also provides for a possible explanation of these. By traversing the relations in the network, we can then identify potential confounding.

The rest of the paper is organized as follows: First, we discuss the background of subgroup discovery, the concept of confounding, and basic constraint-based causal analysis methods in Section 2. After that, we present the causal analysis approach for detecting confounding in Section 3. Exemplary results of the application of the presented approach are given in Section 4 using data from a fielded system in the medical domain. Finally, Section 5 concludes the paper with a discussion of the presented work.

2 Background

In this section, we first introduce the necessary notions concerning the used knowledge representation, before we define the setting for subgroup discovery. After that, we introduce the concept of confounding, criteria for its identification, and describe basic constraint-based techniques for causal subgroup analysis.

2.1 Basic Definitions

Let Ω_A denote the set of all attributes. For each attribute $a \in \Omega_A$ a range $dom(a)$ of values is defined; $\mathcal{V}_A$ is assumed to be the (universal) set of attribute values of the form $(a = v)$, where $a \in \Omega_A$ is an attribute and $v \in dom(a)$ is an assignable value. We consider nominal attributes only so that numeric attributes need to be discretized accordingly. Let CB be the case base (data set) containing all available cases (instances): A case $c \in CB$ is given by the n-tuple $c = ((a_1 = v_1), (a_2 = v_2), \ldots, (a_n = v_n))$ of $n = |\Omega_A|$ attribute values, $v_i \in dom(a_i)$ for each a_i.

2.2 Subgroup Discovery

The main application areas of subgroup discovery (e.g., [1–4]) are exploration and descriptive induction, to obtain an overview of the relations between a (dependent) target variable and a set of explaining (independent) variables. As in the *MIDOS* approach [1], we consider subgroups that are, for example, as large as possible, and have the most unusual (distributional) characteristics with respect to the concept of interest given by a binary target variable. Therefore, not necessarily complete relations but also partial relations, i.e., (small) subgroups with "interesting" characteristics can be sufficient.

Subgroup discovery mainly relies on the subgroup description language, the quality function, and the search strategy. Often heuristic methods (e.g., [3]) but also efficient exhaustive algorithms (e.g., the SD-Map algorithm [4]) are applied. The description language specifies the individuals belonging to the subgroup. For a common single-relational propositional language a subgroup description can be defined as follows:

Definition 1 (Subgroup Description). *A subgroup description $sd = e_1 \wedge e_2 \wedge \cdots \wedge e_k$ is defined by the conjunction of a set of selectors $e_i = (a_i, V_i)$: Each of these are selections on domains of attributes, $a_i \in \Omega_A, V_i \subseteq dom(a_i)$. We define Ω_E as the set of all possible selectors and Ω_{sd} as the set of all possible subgroup descriptions.*

A quality function measures the interestingness of the subgroups and is used to rank these. Typical quality criteria include the difference in the distribution of the target variable concerning the subgroup and the general population, and the subgroup size.

Definition 2 (Quality Function). *Given a particular target variable $t \in \Omega_E$, a quality function $q : \Omega_{sd} \times \Omega_E \rightarrow R$ is used in order to evaluate a subgroup description $sd \in \Omega_{sd}$, and to rank the discovered subgroups during search.*

Several quality functions were proposed (cf. [1–4]), e.g., the functions q_{BT} and q_{RG}:

$$q_{BT} = \frac{(p - p_0) \cdot \sqrt{n}}{\sqrt{p_0 \cdot (1 - p_0)}} \cdot \sqrt{\frac{N}{N - n}}\,, \quad q_{RG} = \frac{p - p_0}{p_0 \cdot (1 - p_0)}, n \geq \mathcal{T}_{Supp}\,,$$

where p is the relative frequency of the target variable in the subgroup, p_0 is the relative frequency of the target variable in the total population, $N = |CB|$ is the size of the total population, and n denotes the size of the subgroup.

In contrast to the quality function q_{BT} (the classic binomial test), the quality function q_{RG} only compares the target shares of the subgroup and the total population measuring the *relative gain*. Therefore, a support threshold $\mathcal{T}_{Supp}$ is necessary to discover significant subgroups.

The result of subgroup discovery is a set of subgroups. Since subgroup discovery methods are not necessarily covering algorithms the discovered subgroups can overlap significantly and their estimated quality (effect) might be confounded by external variables. In order to reduce the redundancy of the subgroups and to identify potential confounding factors, methods for causal analysis can then be applied.

2.3 The Concept of Confounding

Confounding can be described as a bias in the estimation of the effect of the subgroup on the target concept due to attributes affecting the target concept that are not contained in the subgroup description [6]. An extreme case for confounding is presented by *Simpson's Paradox*: The (positive) effect (association) between a given variable X and a variable T is countered by a negative association given a third factor F, i.e., X and T are negatively correlated in the subpopulations defined by the values of F [7]. For the example shown in Figure 1, let us assume that there is a positive correlation between the event X that describes *people that do not consume soft drinks* and T specifying the diagnosis *diabetes*. This association implies that people not consuming soft drinks are

affected more often by diabetes (50% non-soft-drinkers vs. 40% soft-drinkers). However, this is due to age, if older people (given by F) consume soft drinks less often than younger people, and if diabetes occurs more often for older people, inverting the effect.

Combined	T	¬T	∑	Rate (T)
X	25	25	50	50%
¬X	20	30	50	40%
∑	45	55	100	

Restricted on F	T	¬T	∑	Rate (T)
X	24	16	40	60%
¬X	8	2	10	80%
∑	32	18	50	

Restricted on ¬F	T	¬T	∑	Rate (T)
X	1	9	10	10%
¬X	12	28	40	30%
∑	13	37	50	

Fig. 1. Example: Simpson's Paradox

There are three criteria that can be used to identify a **confounding factor** F [6, 8], given the factors X contained in a subgroup description and a target concept T:

1. A confounding factor F must be a *cause* for the target concept T, that is, an independent risk factor for a certain disease.
2. The factor F must be associated/correlated with the subgroup (factors) X.
3. A confounding factor F must *not* be (causally) affected by the factors X.

However, these criteria are only necessary but not sufficient to identify confounders. If purely automatic methods are applied for detecting confounding, then such approaches may label some variables as confounders incorrectly, e.g., if the real confounders have not been measured, or if their contributions cancel out. Thus, user interaction is rather important for validating confounded relations. Furthermore, the identification of confounding requires causal knowledge since confounding is itself a causal concept [8].

Proxy Factors and Effect Modification There are two phenomena that are closely related to confounding. First, a factor may only be associated with the subgroup but may be the real cause for the target concept. Then, the subgroup is only a **proxy factor**. Another situation is given by **effect modification**: Then, a third factor F does not necessarily need to be associated with the subgroup described by the factors X; F can be an additional factor that increases the effect of X in a certain subpopulation only, pointing to new subgroup descriptions that are interesting by themselves.

2.4 Constraint-Based Methods for Causal Subgroup Analysis

In general, the philosophical concept of causality refers to the set of all particular 'cause-and-effect' or 'causal' relations. A subgroup is causal for the target group, if in an ideal experiment [5] the probability of an object not belonging to the subgroup to be a member of the target group increases or decreases when the characteristics of the object are changed such that the object becomes a member of the subgroup. For example, the probability that a patient survives (target group) increases if the patient received a special treatment (subgroup). Then, a redundant subgroup that is, e.g., conditionally independent from the target group given another subgroup, can be suppressed.

For causal analysis, the subgroups are represented by binary variables that are true for an object (case) if it is contained in the subgroup, and false otherwise. For constructing a causal subgroup network, constraint-based methods are particularly suitable

because of scalability reasons (cf. [5, 9]). These methods make several assumptions (cf. [5]) w.r.t. the data and the correctness of the statistical tests. The crucial condition is the *Markov condition* (cf. [5]) depending on the assumption that the data can be expressed by a Bayesian network: Let X be a node in a causal Bayesian network, and let Y be any node that is not a descendant of X in the causal network. Then, the Markov condition holds if X and Y are independent conditioned on the parents of X.

The CCC and CCU rules [9] described below constrain the possible causal models by applying simple statistical tests: For subgroups s_1, s_2, s_3 represented by binary variables the χ^2-test for independence is utilized for testing their independence $ID(s_1, s_2)$, dependence $D(s_1, s_2)$ and conditional independence $CondID(s_1, s_2|s_3)$, shown below (for the tests user-selectable thresholds are applied, e.g., $\mathcal{T}_1 = 1, \mathcal{T}_2 = 3.84$, or higher):

$$ID(s_1, s_2) \longleftrightarrow \chi^2(s_1, s_2) < \mathcal{T}_1\,, \qquad D(s_1, s_2) \longleftrightarrow \chi^2(s_1, s_2) > \mathcal{T}_2\,,$$
$$CondID(s_1, s_2|s_3) \longleftrightarrow \chi^2(s_1, s_2|s_3 = 0) + \chi^2(s_1, s_2|s_3 = 1) < 2 \cdot \mathcal{T}_1$$

Thus, the decision of (conditional) (in-)dependence is threshold-based, which is a problem causing potential errors if very many tests are performed. Therefore, we propose a semi-automatic approach featuring interactive analysis of the inferred relations.

Definition 3 (CCC Rule). *Let X, Y, Z denote three variables that are pairwise dependent, i.e., $D(X, Y), D(X, Z), D(Y, Z)$; let X and Z become independent when conditioned on Y. In the absence of hidden and confounding variables we may infer that one of the following causal relations exists between X, Y and Z: $X \rightarrow Y \rightarrow Z$, $X \leftarrow Y \rightarrow Z$, $X \leftarrow Y \leftarrow Z$. However, if X has no causes, then the first relation is the only one possible, even in the presence of hidden and confounding variables.*

Definition 4 (CCU Rule). *Let X, Y, Z denote three variables: X and Y are dependent ($D(X, Y)$), Y and Z are dependent ($D(Y, Z)$), X and Z are independent ($ID(X, Z)$), but X and Z become dependent when conditioned on Y ($CondD(X, Z|Y)$). In the absence of hidden and confounding variables, we may infer that X and Z cause Y.*

3 Extended Causal Analysis for Detecting Confounding

Detecting confounding using causal subgroup analysis consists of two main steps that can be iteratively applied:

1. First, we generate a causal subgroup network considering a target group T, a user-selected set of subgroups U, a set of confirmed (causal) and potentially confounding factors C for any included group, a set of unconfirmed potentially confounding factors P given by subgroups significantly dependent with the target group, and additional background knowledge described below. In addition to causal links, the generated network also contains (undirected) associations between the variables.
2. In the next step, we traverse the network and extract the potential confounded and confounding factors. The causal network and the proposed relations are then presented to the user for subsequent interpretation and analysis. After confounding factors have been confirmed, the background knowledge can then be extended.

In the following we discuss these steps in detail, and also describe the elements of the background knowledge that are utilized for causal analysis.

3.1 Constructing an Extended Causal Subgroup Network

Algorithm 1 summarizes how to construct an extended causal subgroup network, based on a technique for basic causal subgroup analysis described in [2, 10]. When applying the algorithm, the relations contained in the network can be wrong due to various statistical errors (cf. [5]), especially for the CCU rule (cf. [9]). Therefore, after applying the algorithm, the resulting causal net is presented to the user for interactive analysis.

The first step (lines 1-5) of the algorithm determines for each subgroup pair (including the target group) whether they are independent, based on the inductive principle that the (non in-)dependence of subgroups is necessary for their causality.

In the next step (lines 6-10) we determine for any pair of subgroups whether the first subgroup s_1 is suppressed by a second subgroup s_2, i.e., if s_1 is conditionally independent from the target group T given s_2. The χ^2-measure for the target group and s_1 is calculated both for the restriction on s_2 and its complementary subgroup. If the sum of the two test-values is below a threshold, then we can conclude that subgroup s_1 is conditionally independent from the target group. Conditional independence is a sufficient criterion, since the target distribution of s_1 can be explained by the target distribution in s_2, i.e., by the intersection. Since very similar subgroups could symmetrically suppress each other, the subgroups are ordered according to their quality, and then subgroups with a nearly identical extension (and a lower quality) can be eliminated.

The next two steps (lines 11-18) check conditional independence between each pair of subgroups given the target group or a third subgroup, respectively. For each pair of conditionally independent groups, the separating (conditioning) group is noted. Then, this separator information is exploited in the next steps, i.e., independencies or conditional independencies for pairs of groups derived in the first steps are used to exclude any causal links between the groups. The conditioning steps (lines 6-18) can optionally be iterated in order to condition on combinations of variables (pairs, triples). However, the decisions taken further (in the CCU and CCC rules) may become statistically weaker justified due to smaller counts in the considered contingency tables (e.g., [5, 10]).

Direct causal links (line 19) are added based on background knowledge, i.e., given subgroup patterns that are directly causal for specific subgroups. In the last step (lines 24-26) we also add conditional associations for dependent subgroups that are not conditionally independent and thus not suppressed by any other subgroups. Such links can later be useful in order to detect the (true) associations considering a confounding factor.

Extending CCC and CCU using Background Knowledge The CCU and CCC steps (lines 20-23) derive the directions of the causal links between subgroups, based on information derived in the previous steps: We extend the basic CCC and CCU rules including background knowledge both for the derivation of additional links, and for inhibiting links that contradict the background knowledge. The applicable background consists of two elements: We consider subgroups consisting of factors that have no causes, and subgroups that are not only dependent but (directly) causal for the target group/target variable. The subgroup *Age* $\geq$ *70*, for example, has no causes, whereas the subgroup *Body-Mass-Index (BMI)=overweight* is directly causal for the subgroup *Gallstones=probable*. We introduce associations instead of causal directions if these are wrong, or if not enough information for their derivation is available. The rationale

Algorithm 1 Constructing a causal subgroup net

Require: Target group T, user-selected set of subgroups U, potentially confounding groups P, background knowledge B containing acausal subgroup information, and known subgroup patterns $C \subseteq B$ that are directly causal for other subgroups. Define $\mathcal{S} = U \cup P \cup C$

for all $s_i, s_j \in \mathcal{S} \cup T, s_i \neq s_j$ **do**
  if $approxEqual(s_i, s_j)$ **then**
    Exclude any causalities for the subgroup s_k with smaller correlation to T: Remove s_k
  if $ID(s_i, s_j)$ **then**
    Exclude causality: $ex(s_i, s_j) = true$
for all $s_i, s_j \in \mathcal{S}, s_i \neq s_j$ **do**
  if $\neg ex(s_i, T)$, $\neg ex(s_j, T)$, or $\neg ex(s_i, s_j)$ **then**
    if $CondID(s_i, T|s_j)$ **then**
      Exclude causality: $ex(s_i, T) = true$, and include s_j into $separators(s_i, T)$
      If conditional independencies are symmetric, then select the strongest relation
for all $s_i, s_j \in \mathcal{S}, i < j$ **do**
  if $\neg ex(s_i, T)$, $\neg ex(s_j, T)$, or $\neg ex(s_i, s_j)$ **then**
    if $CondID(s_i, s_j|T)$ **then**
      Exclude causality: $ex(s_i, s_j) = true$, and include T into $separators(s_i, s_j)$
for all $s_i, s_j, s_k \in \mathcal{S}, i < j, i \neq k, j \neq k$ **do**
  if $\neg ex(s_i, s_j)$, $\neg ex(s_j, s_k)$, or $\neg ex(s_i, s_k)$ **then**
    if $CondID(s_i, s_j|s_k)$ **then**
      Exclude causality: $ex(s_i, s_j) = true$, and include s_k into $separators(s_i, s_j)$
Integrate direct causal links that are not conditionally excluded considering the sets C and B
for all $s_i, s_j, s_k \in \mathcal{S}$ **do**
  Apply the extended CCU rule, using background knowledge
for all $s_i, s_j, s_k \in \mathcal{S}$ **do**
  Apply the extended CCC rule, using background knowledge
for all $s_i, s_j, s_k \in \mathcal{S} \cup \{T\}, i < j, i \neq k, j \neq k$ **do**
  if $\neg CondID(s_i, s_j|s_k)$ **then**
    Integrate association between dependent s_i and s_j that are not conditionally excluded

behind this principle is given by the intuition that we want to utilize as much information as possible considering the generated causal net.

For the **extended** CCU **rule** we use background knowledge for inhibiting acausal directions, since the CCU rule can be disturbed by confounding and hidden variables. The causal or associative links do not necessarily indicate direct associations/causal links but can also point to relations enabled by hidden or confounding variables [9].

For the **extended** CCC **rule**, we can use the relations inferred by the extended CCU rule for disambiguating between the causal relations, if the CCU rule is applied in all possible ways: The non-separating condition (conditional dependence) of the relation identified by the CCU rule is not only a sufficient but a necessary condition [9], i.e., considering $X \rightarrow Y \leftarrow Z$, with $CondID(X, Z|Y)$. Additionally, we can utilize background knowledge for distinguishing between the causal relations. So, for three variables X, Y, Z with $D(X, Y), D(X, Z), D(Y, Z)$, and $CondID(X, Z|Y)$, if there exists an (inferred) causal link $X \rightarrow Y$ between X and Y, we may identify the relation $X \rightarrow Y \rightarrow Z$ as the true relation. Otherwise, if Y or Z have no causes, then we select the respective relation, for example, $X \leftarrow Y \rightarrow Z$ for an acausal variable Y.

3.2 Identifying Confounded Relations

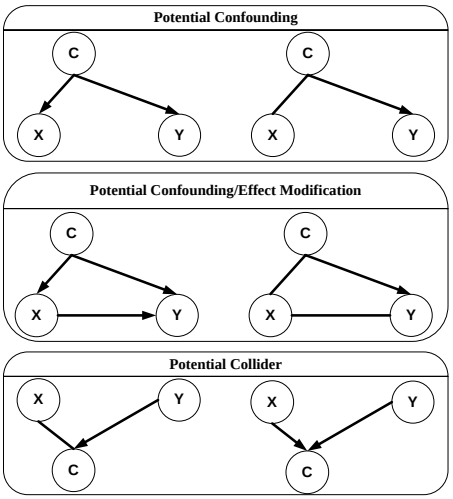


Fig. 2. Examples for Confounded Relations

A popular method for controlling confounding factors is given by *stratification* [6]: For example, in the medical domain a typical confounding factor is the attribute *age*: We can stratify on age groups such as $age < 30$, $age\ 30 - 69$, and $age \geq 70$. Then, the subgroup – target relations are measured within the different strata, and compared to the (crude) unstratified measure.
It is easy to see, that in the context of the presented approach stratification for a binary variables is equivalent to conditioning on them: If we assess a conditional subgroup – target relation and the subgroup factors become independent (or dependent), then this indicates potential confounding. After constructing a causal net, we can easily identify such relations. Since the causal directions derived by the extended CCC and CCU rules may be ambiguous, user interaction is crucial: In the medical domain, for example, it is often difficult to provide non-ambiguous directed relationships between certain variables: One disease can cause another disease and vice versa, under different circumstances. The network then also provides an intuitive visualization for the analysis.

In order to identify potentially confounded relations and the corresponding variables, as shown in Figure 2, and described below, we just need to traverse the network:

- **Potential Confounding**: If there is an association between two variables X and Y, and the network contains the relations $C \rightarrow X$, $C \rightarrow Y$, and there is no link between X and Y, i.e., they are conditionally independent given C, then C is a confounder that inhibits the relation between X and Y. This is also true if there is no causal link between C and X but instead an association.
- **Potential Confounding/Effect Modification**: If the network contains the relations $C \rightarrow X$, $C \rightarrow Y$, and there is also either an association or a causal link between X and Y, then this points to confounding and possible effect modification of the relation between the variables X and Y.
- **Potential Collider (or Confounder)**: If there is no (unconditioned) association between two variables X and Y and the network contains the relations $X \rightarrow C$ and $Y \rightarrow C$, then C is a potential collider: X and Y become dependent by conditioning on C. The variable C is then no confounder in the classical sense, if the (derived) causal relations are indeed true. However, such a relation as inferred by the CCU rule can itself be distorted by confounded and hidden variables. The causal directions could also be inverted, if the association between X and Y is just not strong enough as estimated by the statistical tests. In this case, C is a potential confounder. Therefore, manual inspection is crucial in order to detect the true causal relation.

4 Examples

We applied a case base containing about 8600 cases taken from the SONOCONSULT system [11] – a medical documentation and consultation system for sonography. The system is in routine use in the DRK-hospital in Berlin/Köpenick, and the collected cases contain detailed descriptions of findings of the examination(s), together with the inferred diagnoses (binary attributes). The experiments were performed using the VIKAMINE system [12] implementing the presented approach.

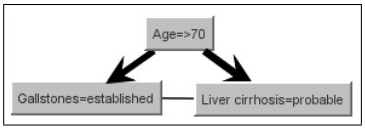


Fig. 3. Confounded Relation: Gallstones and Liver cirrhosis

In the following, we provide some (simplified) examples considering the diagnosis *Gallstones=established* as the target variable. After applying a subgroup discovery method, several subgroups were selected by the user in order to derive a causal subgroup network, and to check the relations w.r.t. possible confounders. These selected subgroups included, for example, the subgroup *Fatty liver=probable or possible* and the subgroup *Liver cirrhosis=probable*.

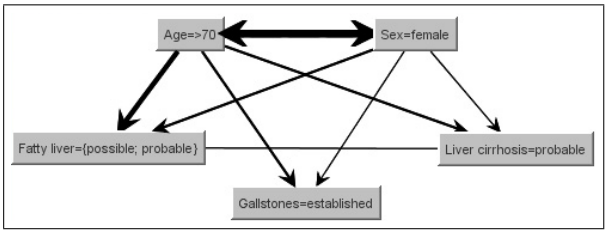


Fig. 4. Confounded Relation: Gallstones, Liver cirrhosis and Fatty Liver

A first result is shown in Figure 3: In this network, the subgroup *Liver cirrhosis=probable* is confounded by the variable *Age*$\geq$*70*. However, there is still an influence on the target variable considering the subgroup *Liver cirrhosis=probable* shown by the association between the subgroups. This first result indicates confounding and effect modification (the strengths of the association between the nodes is also visualized by the widths of the links). A more detailed result is shown in Figure 4: In this network another potential confounder, i.e., *Sex=female* is included. Then, it becomes obvious, that both the subgroup *Fatty liver=probable or possible* and the subgroup *Liver cirrhosis=probable* are confounded by the variables *Sex* and *Age*, and the association (shown in Figure 3) between the subgroup *Liver cirrhosis=probable* and the target group is no longer present (In this example the removal of the gall-bladder was not considered which might have an additional effect concerning a medical interpretation).

It is easy to see that the generated causal subgroup network becomes harder to interpret, if many variables are included, and if the number of connections between the nodes increases. Therefore, we provide filters, e.g., in order to exclude (non-causal) associations. The nodes and the edges of the network can also be color-coded in order to increase their interpretability: Based on the available background knowledge, causal subgroup nodes, and (confirmed) causal directions can be marked. Since the network is first traversed and the potentially confounded relations are reported to the user, the analysis can also be focused towards the respective variables, as a further filtering condition. The user can then analyze selected parts of the network in more detail.

5 Conclusion

In this paper, we have presented a causal subgroup analysis approach for the semi-automatic detection of confounding: Since there is no purely automatic test for confounding [8] the analysis itself depends on background knowledge for establishing an extended causal model/network of the domain. Then, the constructed network can be used to identify potential confounders. In a semi-automatic approach, the network and the potential confounded relations can then be evaluated and validated by the user.

In the future, we are planning to consider an efficient approach for detecting confounding that is directly embedded in the subgroup discovery method. Related work in that direction was described (e.g., [13]). Another interesting direction for future work is given by considering further background knowledge for causal analysis.

Acknowledgements

This work has been partially supported by the German Research Council (DFG) under grant Pu 129/8-1.

References

1. Wrobel, S.: An Algorithm for Multi-Relational Discovery of Subgroups. In: Proc. 1st Europ. Symp. Principles of Data Mining and Knowledge Discovery, Berlin, Springer (1997) 78–87
2. Klösgen, W.: 16.3: Subgroup Discovery. In: Handbook of Data Mining and Knowledge Discovery. Oxford University Press, New York (2002)
3. Lavrac, N., Kavsek, B., Flach, P., Todorovski, L.: Subgroup Discovery with CN2-SD. Journal of Machine Learning Research **5** (2004) 153–188
4. Atzmueller, M., Puppe, F.: SD-Map - A Fast Algorithm for Exhaustive Subgroup Discovery. In: Proc. 10th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2006), Berlin, Springer (2006) 6–17
5. Cooper, G.F.: A Simple Constraint-Based Algorithm for Efficiently Mining Observational Databases for Causal Relationships. Data Min. Knowl. Discov. **1**(2) (1997) 203–224
6. McNamee, R.: Confounding and Confounders. Occup. Environ. Med. **60** (2003) 227–234
7. Simpson, E.H.: The Interpretation of Interaction in Contingency Tables. Journal of the Royal Statistical Society **18** (1951) 238–241
8. Pearl, J.: 6.2 Why There is No Statistical Test For Confounding, Why Many Think There Is, and Why They Are Almost Right. In: Causality: Models, Reasoning and Inference. Cambridge University Press (2000)
9. Silverstein, C., Brin, S., Motwani, R., Ullman, J.D.: Scalable Techniques for Mining Causal Structures. Data Mining and Knowledge Discovery **4**(2/3) (2000) 163–192
10. Kloesgen, W., May, M.: Database Integration of Multirelational Causal Subgroup Mining. Technical report, Fraunhofer Institute AIS, Sankt Augustin, Germany (2002)
11. Huettig, M., Buscher, G., Menzel, T., Scheppach, W., Puppe, F., Buscher, H.P.: A Diagnostic Expert System for Structured Reports, Quality Assessment, and Training of Residents in Sonography. Medizinische Klinik **99**(3) (2004) 117–122
12. Atzmueller, M., Puppe, F.: Semi-Automatic Visual Subgroup Mining using VIKAMINE. Journal of Universal Computer Science **11**(11) (2005) 1752–1765
13. Fabris, C.C., Freitas, A.A.: Discovering Surprising Patterns by Detecting Occurrences of Simpson's Paradox. In: Research and Development in Intelligent Systems XVI, Berlin, Springer (1999) 148–160

Declarative Specification of Ontological Domain Knowledge for Descriptive Data Mining

Martin Atzmueller and Dietmar Seipel

University of Würzburg
Department of Computer Science
Am Hubland, 97074 Würzburg, Germany
{atzmueller, seipel}@informatik.uni-wuerzburg.de

Abstract. Domain knowledge is a valuable resource for improving the quality of the results of data mining methods. In this paper, we present a methodological approach for providing ontological domain knowledge in a *declarative manner*: We utilize Prolog facts and rules for the specification of properties of ontological concepts and for the derivation of further ad-hoc relations of the ontological concepts. This enhances the *documentation*, *extendability*, and *standardization* of the applied knowledge. Furthermore, the presented approach also provides for *automatic verification* and *improved maintenance* options with respect to the used domain knowledge.

1 Introduction

Domain knowledge is a natural resource for knowledge-intensive data mining methods (e.g., [9, 12]), and can be exploited for improving the quality of the data mining results significantly. Appropriate domain knowledge can increase the representational expressiveness and also focus the algorithm on the relevant patterns. Furthermore, for increasing the efficiency of the search method the search space can often be constrained, e.g., [2, 4]. A prerequisite for the successful application and exploitation of domain knowledge is given by a concise description and specification of the domain knowledge. A concise specification also provides for better documentation, extendability, and standardization. Furthermore, maintenance issues and verification of the applied domain knowledge are also enhanced.

In this paper, we present a methodological approach using a declarative specification of ontological domain knowledge: Our context is given by a descriptive data mining method, i.e., we focus on methods that provide descriptive patterns for later inspection by the user. These patterns need to be easy to comprehend and to interpret by the user. Thus, besides the objective quality of the patterns as rated by a quality function such as, e.g., *support* or *confidence* for association rules (e.g., [1]), subjective measures are also rather important: The interestingness of the patterns is also affected by the representational expressiveness of the patterns, and by the focus of the pattern discovery method on the relevant concepts of the domain ontology. In this way, patterns (associations) containing irrelevant and non-interesting concepts can be surpressed in order to increase the overall interestingness of the set of the mined patterns.

Knowledge acquisition is often challenging and costly (knowledge acquisition bottleneck). Thus, an important idea is to ease knowledge acquisition by reusing existing domain knowledge, i.e., knowledge that is contained in existing ontologies or knowledge bases. Furthermore, we can apply high-level knowledge such as properties of ontological objects for deriving simpler constraint knowledge that can be directly included in the data mining step, as discussed, e.g., in [2, 4]. For specifying such properties and relations of the concepts contained in the domain ontology, we utilize Prolog rules as a versatile representation: using these, we obtain a suitable representation formalism for ontological knowledge. Furthermore, we can automatically derive ad-hoc relations between ontological concepts using simple rules, and thus also provide a comprehensive overview and summary for the domain specialist. Then, these rules can also be checked with respect to their consistency, i.e., the provided knowledge base and the derivation knowledge can be verified for its semantic integrity. Altogether, the resulting knowledge base in terms of Prolog rules and facts automatically serves as a specification and documentation of the domain knowledge that is easy to comprehend, to interpret and to extend. Furthermore, new task-specific knowledge can be derived by extending and/or modifying the knowledge base.

The rest of the paper is organized as follows: We first briefly introduce the context of association patterns for descriptive data mining in Section 2. After that, we summarize several types of domain knowledge in Section 3. Next, we describe how the ontological knowledge can be formalized and used for deriving ad-hoc relations, i.e., ontological constraint knowledge in Section 4, and illustrate the approach with practical examples. Section 5 discusses a representation and transformation of the (derived) domain knowledge to XML. Finally, we conclude the paper with a summary in Section 6, and point out interesting directions for future work.

2 Discovering Association Patterns using Domain Knowledge

In the context of this work, we consider descriptive data mining methods for discovering interesting association patterns. Prominent approaches include subgroup discovery methods, e.g., [3, 11, 14], and methods for learning association rules, e.g., [1, 7].

2.1 Association Patterns

Subgroup patterns [4, 10], often provided by conjunctive rules, describe 'interesting' subgroups of cases, e.g., "the subgroup of 16-25 year old men that own a sports car are more likely to pay high insurance rates than the people in the reference population." The main application areas of subgroup discovery [11, 14] are exploration and descriptive induction, to obtain an overview of the relations between a target variable and a set of explaining variables, where variables are attribute/value assignments. The exemplary subgroup above is then described by the relation between the independent (explaining) variables (Sex = male, Age $\leq$ 25, Car = sports car) and the dependent (target) variable (Insurance Rate = high). The independent variables are modeled by selection expressions on sets of attribute values. A subgroup pattern is thus described by a subgroup description in relation to a specific target variable.

Let Ω_A be the set of all attributes. For each attribute $a \in \Omega_A$ a range $dom(a)$ of values is defined. An attribute/value assignment $a = v$, where $a \in \Omega_A, v \in dom(a)$, is called a *feature*. We define the feature space $\mathcal{V}_A$ to be the (universal) set of all features. A single-relational propositional *subgroup description* is defined as a conjunction

$$sd = e_1 \wedge e_2 \wedge \cdots \wedge e_n$$

of (extended) features $e_i \subseteq \mathcal{V}_A$, which are then called selection expressions, where each e_i selects a subset of the range $dom(a)$ of an attribute $a \in \Omega_A$. We define Ω_{sd} as the set of all possible subgroup descriptions.

An *association rule* (e.g., [1, 8]) is given by a rule of the form $sd_B \rightarrow sd_H$, where sd_B and sd_H are subgroup descriptions; the rule body sd_B and the rule head sd_H specify sets of items. E.g., for the insurance domain we can consider an association rule showing a combination of potential risk factors for high insurance rates and accidents:

$$\text{Sex} = \text{male} \wedge \text{Age} \leq 25 \wedge \text{Car} = \text{sports car} \rightarrow \text{Insurance Rate} = \text{high} \wedge \text{Accident Rate} = \text{high}$$

A *subgroup pattern* is a special association rule, namely a horn clause $sd \rightarrow e$, where $sd \in \Omega_{sd}$ is a subgroup description and the feature $e \in \mathcal{V}_A$ is called the target variable.

In general, the quality of an association rule is measured by its support and confidence, and the data mining process searches for association rules with arbitray rule heads and bodies. For subgroup patterns there exist various (more refined) quality measures (e.g., [3, 11]): Since an arbitrary quality function can be applied, the anti-monotony property of support used in association rule mining cannot be utilized in the general case. E.g., the used quality function can combine the difference of the confidence and the apriori probability of the rule head with the size of the subgroup. Since mining for interesting subgroup patterns is more complicated, usually a fixed, atomic rule head is given as input to the search process.

2.2 Application of Domain Knowledge for Descriptive Data Mining

In the context of this paper we focus on pattern mining methods that discover descriptive patterns in the form of conjunctive rules, as outlined above. The knowledge classes described in Section 3 can be directly integrated in the respective pattern discovery step: While exclusion constraints restrict the search space by construction, aggregation constraints do not necessarily restrict it, since new values are introduced, but they help to find more understandable results. Also, combination constraints inhibit the examination of specified sets of concepts and can prune large (uninteresting) areas of the search space. For increasing the representational expressiveness, modifications of the value range of an attribute can be utilized to infer values that are more meaningful for the user. E.g., in the medical domain different aggregated age groups could be considered.

In contrast to existing approaches, e.g., [12, 15] we focus on domain knowledge that can be easily declared in symbolic form. Furthermore, the presented approach features the ability of deriving 'simpler' low-level knowledge (constraints) from high-level ontological knowledge. Alltogether, the complete knowledge base can be intuitively declared, validated, and inspected by the user, as shown in the following sections.

In general, the search space considered by the data mining methods can be significantly reduced by *shrinking* the value ranges of the attributes. Furthermore, the search can often be focused if only *meaningful* values are taken into account. This usually depends on the considered ontological domain. In the examples below, we consider ordinality and normality information.

Let A be an ordinal attribute with the range $dom(A) = \{ v_1, v_2, v_3, v_4, v_5, v_6, v_7 \}$, where $v_1 < v_2 < \ldots < v_7$. E.g., A could be the (discretized) attribute *body weight* with the 7 values massive underweight, strong underweight, underweight, normal weight, overweight, strong overweight, and massive overweight. Ordinality information can be easily applied to derive a restricted domain of aggregated values, where each aggregated value is a meaningful subset of $dom(A)$.

Firstly, we could consider only sub-intervals $\langle v_i, v_j \rangle = \{ v_i, v_{i+1}, \ldots, v_j \}$ of consecutive attribute values, where $v_i \leq v_j$, since subsets with holes, such as $\{v_1, v_3\}$, are irrelevant. Then we obtain only $\binom{7}{2} + 7 = 28$ intervals, instead of all possible $2^7 - 1 = 127$ non-empty subsets:

$$\{v_1\}, \langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \ldots, \langle v_1, v_7 \rangle, \{v_2\}, \langle v_2, v_3 \rangle, \ldots, \langle v_2, v_7 \rangle, \ldots, \langle v_6, v_7 \rangle, \{v_7\}.$$

Secondly, in the medical domain we often know that a certain attribute value denotes the *normal* value; in our example the normal weight is v_4. This value is often not interesting for the analyst, who might focus on the *abnormal* value combinations. Combining normality and ordinality information, we need to consider only 10 subsets:

$$\begin{aligned} \text{below } v_4: &\ \{v_1\}, \{v_1, v_2\}, \{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3\}, \\ \text{above } v_4: &\ \{v_5\}, \{v_5, v_6\}, \{v_5, v_6, v_7\}, \{v_6, v_7\}, \{v_7\}. \end{aligned}$$

Thirdly, if we are interested only in combinations including the most extreme values v_1 and v_7, which is typical in medicine, then we can further reduce to 6 meaningful subsets:

$$\{v_1\}, \{v_1, v_2\}, \{v_1, v_2, v_3\}, \{v_5, v_6, v_7\}, \{v_6, v_7\}, \{v_7\}.$$

As a variant, we could add one of the intervals $\{v_1, v_2, v_3, v_4\}$ or $\{v_4, v_5, v_6, v_7\}$ with the *normal* value v_4.

For the third case with 6 intervals, the savings of such a reduction of value combinations, which can be derived using ordinality, normality information and interestingness assumptions, are huge: If there are 10 ordinal attributes A with a normal value and with seven values each, then the size of the search space considering all possible combinations is reduced from $127^{10} \approx 10^{21}$ to $6^{10} \approx 6 \cdot 10^8$.

3 Types and Classes of Domain Knowledge

The considered classes of domain knowledge include ontological knowledge and (derived) constraint knowledge, as a subset of the domain knowledge described in [2, 4]. Figure 1 shows the knowledge hierarchy, from the two knowledge classes to the specific types, and the objects they apply to. In the following, we first introduce ontological knowledge. After that, we discuss how ad-hoc constraints (i.e., ontological constraint knowledge) can be derived using ontological knowledge.

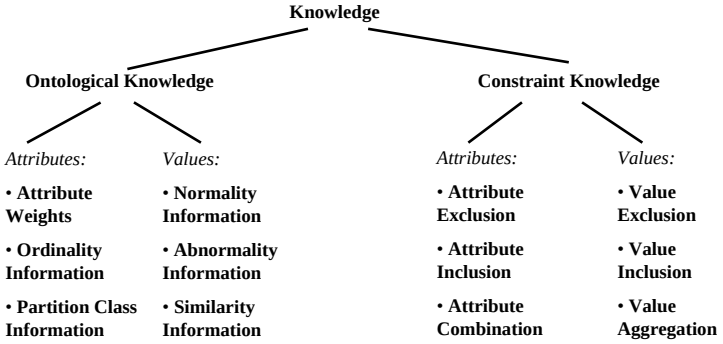


Fig. 1. Hierarchy of (abstract) knowledge classes and specific types

3.1 Ontological Knowledge

Ontological knowledge describes general (ad-hoc) properties of the ontological concepts and can be used to infer additional constraints, that can be considered as ad-hoc relations between the domain objects. The applied ontological knowledge (cf. [4]) consists of the following types that can be easily formalized as Prolog ground facts.

Attribute weights denote the relative importance of attributes, and are a common extension for knowledge-based systems [6]. E.g., in the car insurance domain we can state that the attribute *Age* is more important than the attribute *Car Color*, since its assigned weight is higher:

```
weight(age, 4).
weight(car_color, 1).
```

Abnormality/Normality information is usually easy to obtain for diagnostic domains. E.g., in the medical domain the set of *normal* attribute values contains the expected values, and the set of *abnormal* values contains the unexpected ones, where the latter are often more interesting for analysis. Each attribute value is attached with a label specifying a normal or an abnormal state. Normality information only requires a binary label. E.g., for the attribute *temperature* we could have

```
normal(temperature='36.5-38').
```

Abnormality information defines several categories. For the range $dom(temperature) = \{ t_1, t_2, t_3, t_4 \}$, the value $t_2 =$ '$36.5 - 38$' denotes the normal state, while the values $t_1 =$ '< 36.5', $t_3 =$ '$38 - 39$', and $t_4 =$ '> 39' describe abnormal states. The category 5 indicates the maximum abnormality:

```
abnormality(temperature='>39', 5).
```

Similarity information between attribute values is often applied in case-based reasoning: It specifies the relative similarity between the individual attribute values. For example, for a nominal attribute *Color* with $dom(Color) = \{ white, gray, black \}$, we can state that the value *white* is more similar to *gray* than it is to *black*:

```
similarity(color=white, color=gray, 0.5).
similarity(color=white, color=black, 0).
```

Here, the respective similarity value is $s \in [0; 1]$.

Ordinality information specifies if the value range of a nominal attribute can be ordered. E.g., the qualitative attributes *Age* and *Car Size* are ordinal, while *Color* is not:

```
ordinal_attribute(age).
ordinal_attribute(car_size).
```

Partition class information provides semantically distinct groups of attributes. These disjoint subsets usually correspond to certain problem areas of the application domain. E.g., in the medical domain such partitions are representing different organ systems like *liver*, *kidney*, *pancreas*, *stomach*, *stomach*, and *intestine*. For each organ system a list of attributes is given:

```
attribute_partition(inner_organs, [
   [fatty_liver, liver_cirrhosis, ...],
   [renal_failure, nephritis, ...], ... ]).
```

3.2 Constraint Knowledge

Constraint knowledge can be applied, e.g., for filtering patterns by their quality and for restricting the search space, as discussed below. We distinguish the following types of constraint knowledge, that can be explicitly provided as ground facts in Prolog. Further basic constraints can be derived automatically as discussed in Section 4.

Exclusion constraints for attributes or features are applied for filtering the domains of attributes and the feature space, respectively. The same applies for *inclusion constraints* for attributes or features, that explicitly state important values and attributes that should be included:

```
dsdk_constraint(exclude(attribute), car_color).
dsdk_constraint(include(attribute), age).
dsdk_constraint(exclude(feature), age=30-50).
```

Value aggregation constraints can be specified in order to form abstracted disjunctions of attribute values, e.g., intervals for ordinal values. For example, for the attribute *Age* with $dom(Age) = \{ \text{'}< 40\text{'}, \text{'}40 - 50\text{'}, \text{'}50 - 70\text{'}, \text{'}> 70\text{'} \}$ we can derive the aggregated values '≤ 50' (corresponding to the list `[0-40,40-50]`) and '> 50' (corresponding to the list `[50-70,70-100]`). In general, aggregated values are not restricted to intervals, but can cover any combination of values.

```
dsdk_constraint(aggregated_values,
   age = [ [0-40,40-50], [50-70,70-100] ]).
```

Attribute combination constraints are applied for filtering/excluding certain combinations of attributes, e.g., if these are already known to the domain specialist.

```
dsdk_constraint(exclude(attribute_pair),
   [car_color, car_size]).
dsdk_constraint(include(attribute_pair),
   [age, car_size]).
```

The data mining process should not compute association rules that contain one of the attributes of an excluded attribute pair in the body and the other attribute in the head.

4 Deriving Constraint Knowledge

Figure 2 shows how ontological knowledge can be used to derive further 'basic' constraints.

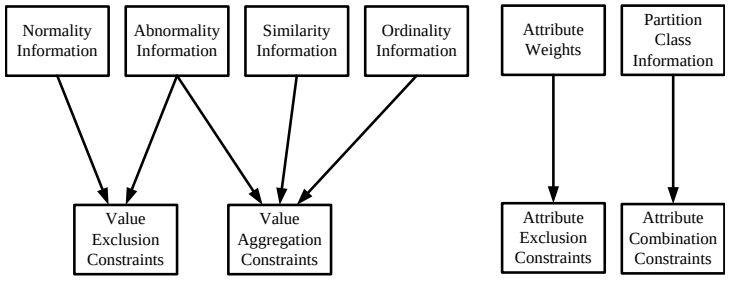


Fig. 2. Deriving constraints using ontological knowledge

Below, we summarize how new constraints can be inferred using ontological knowledge, and show the corresponding Prolog code for deriving the new constraints. The user can generate a summary of all facts derivable by the provided derivation rules for manual inspection by querying `dsdk_constraint(X,Y)` using backtracking.

4.1 Exclusion and Inclusion Constraints

We can construct attribute exclusion constraints using attribute weights to filter the set of relevant attributes by a weight threshold or by subsets of the weight space.

```
dsdk_constraint(exclude(attribute), A) :-
   weight(A, N), N =< 1.
dsdk_constraint(include(attribute), A) :-
   weight(A, N), N > 1.
```

We can apply complex rules for deriving constraints that can be specified by the user:

```
dsdk_constraint(include(attribute), A) :-
   weight(A, N), N >= 2,
   abnorm(A=_, 5).
```

Using abnormality/normality knowledge we can specify global exclusion constraints for the normal features, i.e. the features whose abnormality is at most 1 (the lowest category 0 denotes the don't care value, while the category 1 explicitly indicates a normal value):

```
dsdk_constraint(exclude(feature), A=V) :-
   abnormality(A=V, N), N =< 1.
```

Partition class information can be used to infer attribute combination constraints in order to prevent the combination of individual attributes that are contained in separate partition classes. Alternatively, inverse constraints can also be derived, e.g., to specifically investigate inter-organ relations in the medical domain.

```
dsdk_constraint(exclude(attribute_pair), [A1, A2]) :-
   attribute_partition(_, P),
   member(As1, P), member(As2, P), As1 \= As2,
   member(A1, As1), member(A2, As2).
```

Finally, we can use a generic Prolog rule for detecting conflicts w.r.t. these rules and the derived knowledge:

```
dsdk_constraint(error(include_exclude(X)), Y) :-
   dsdk_constraint(include(X), Y),
   dsdk_constraint(exclude(X), Y).
```

4.2 Aggregation Constraints

Aggregation w.r.t. Abnormality. Using similarity or abnormality/normality information we can filter and model the value ranges of attributes. Global abnormality groups can be defined by *aggregating values* with the same abnormality:

```
dsdk_constraint(aggregated_values, A=Values) :-
   attribute(A),
   findall( Vs,
      abnormality_aggregate(A=Vs),
      Values ).

abnormality_aggregate(A=Vs) :-
   setof( V,
      abnormality(A=V, _),
      Vs ).
```

For a given attribute `A` the helper predicate `abnormality_aggregate/2` computes the set `Vs` (without duplicates) of all values `V` which have the same abnormality `N`; this grouping is done using the Prolog meta-predicate `setof/3`.

Aggregation w.r.t. Similarity. Also, if the similarity between two attribute values is very high, then they can potentially be analyzed as an aggregated value; this will form a disjunctive selection expression on the value range of the attribute. For example, in the medical domain similar attribute values such as *probable* and *possible* (with different abnormality degrees) can often be aggregated.

The following predicate finds the list `Values` of all sublists `Ws` of the domain `Vs` of an attribute `A`, such that the values in `Ws` are similar in the following sense: if `Ws` = $[w_1, w_2, \ldots, w_n]$, then `similarity`$(A = w_i, A = w_{i+1}, s_i)$ must be given, and it must hold $\Pi_{i=1}^{n-1} s_i \geq 0.5$:

```
dsdk_constraint(aggregated_values, A=Values) :-
   ordinal_attribute(A), aggregate_attribute(A),
   domain(A, Vs),
   findall( Ws,
      similar_sub_sequence(A, 0.5, Vs, Ws),
      Values ).
```

Aggregation w.r.t. Normal Value. Ordinality information can be easily used to construct aggregated values, which are often more meaningful for the domain specialist. We can consider all adjacent combinations of attribute values, or all ascending/descending combinations starting with the minimum or maximum value, respectively. Whenever abnormality information is available, we can partition the value range by the given *normal* value and only start with the most extreme value. For example, for the ordinal attribute *liver size* with the values

> *1:smaller than normal*, *2:normal*, *3:marginally increased*, *4:slightly increased*, *5:moderately increased*, and *6:highly increased*,

we partition by the *normal* value 2, and we obtain the following aggregated values:

```
[1], [3,4,5,6], [4,5,6], [5,6], [6]
```

Given the domain `Vs` of an ordinal attribute `A`, that should be aggregated. If the normal value `v` for `A` is given, and `Vs` = $[v_1, \ldots, v_{n-1}, v, v_{n+1}, \ldots, v_m]$, then all starting sequences `Ws` = $[v_1, v_2, \ldots, v_i]$, where $1 \leq i \leq n-1$, for the sub-domain `Vs1` = $[v_1, \ldots, v_{n-1}]$ of all values below `v` and all ending sequences `Ws` = $[v_i, \ldots, v_m]$, where $n+1 \leq i \leq m$, for the sub-domain `Vs2` = $[v_{n+1}, \ldots, v_m]$ of all values above `v` are constructed by backtracking. If the normal value for `A` is not given, then all starting (ending) sequences for the full domain `Vs` are constructed.

```
dsdk_constraint(aggregated_values, A=Values) :-
   ordinal_attribute(A), aggregate_attribute(A),
   domain(A, Vs),
   dsdk_split_for_aggregation(A, Vs, Vs1, Vs2),
   findall( Ws,
      ( starting_sequence(Vs1, Ws)
      ; ending_sequence(Vs2, Ws) ),
      Values ).
```

```
dsdk_split_for_aggregation(A, Vs, Vs1, Vs2),
   ( normal(A=V),
     append(Vs1, [V|Vs2], Vs)
   ; \+ normal(A=_),
     Vs1 = Vs, Vs2 = Vs ).
```

This type of aggregation constraints using Prolog meta predicates usually has to be implemented by Prolog experts.

5 XML Representation of the Knowledge

We can also formalize the factual ontological and constraint knowledge in XML – as a standard data representation and exchange format – and transform it to Prolog facts using the XML query and transformation language FNQuery, cf. [13]. FNQuery has been implemented in Prolog and is fully interleaved with the reasoning process of Prolog.

For example, we are using the following XML format for representing constraint knowledge about attributes and classes of attributes:

```
<attribute name="temperature" id="a1" weight="2"
   normal="t2" ordinal="yes" aggregate="yes">
   <domain>
      <value id="t1">&lt;36.5</value>
      <value id="t2">36.5-38</value>
      <value id="t3">38-39</value>
      <value id="t4">&gt;39</value>
   </domain>
   <similarity value="0.2" val1="t3" val2="t4"/>
   ...
</attribute>

<attribute name="fatty_liver" id="io1" ...
<attribute name="liver_cirrhosis" id="io2" ...
<attribute name="renal_failure" id="io3" ...
<attribute name="nephritis" id="io4" ...

<class name="inner_organs">
   <attribute_partition>
      <attributes>
         <attribute id="io1"/> <attribute id="io2"/>
      </attributes>
      <attributes>
         <attribute id="io3"/> <attribute id="io4"/>
      </attributes>
   </attribute_partition>
</class>
```

The derived Prolog facts can be transformed to XML using FNQuery. Then, using the *Field Notation Grammars* of FNQuery we can generate HTML reports from XML for an intuitive assessment of the (derived) knowledge by the user. Of course, by transforming the XML data we can also provide other specialized output formats, which can, e.g., be used for the integration with other tools.

6 Conclusions

In this paper we presented a methodological approach for the declarative specification of domain knowledge for descriptive data mining. The respective knowledge classes and types can be comprehensively declared, validated, and inspected by the user. The different types of knowledge can be directly integrated in the pattern discovery step of the data mining process: Applying the domain knowledge the search space can be significantly pruned, uninteresting results can be excluded, and both the representational expressiveness of the ontological objects and the generated results can be increased.

The structured ontological domain knowledge and the rules for deriving constraint knowledge can be represented nicely in Prolog term structures using function symbols; the (factual) ontological domain knowledge can also be represented in XML. For reasoning we make essential use of the following features of Prolog: backtracking, meta-predicates (such as findall, setof, etc.), ease of symbolic computations for term structures. Moreover, using the XML query and transformation language FNQuery we can provide a standard input/output format for high-level ontological and explicitly specified or derived constraint knowledge. Thus, other declarative languages, such as answer set programming [5], could not be used.

We can imagine three kinds of users of the presented approach, depending on their level of experience with the system and/or the presented types of knowledge: Unexperienced users just use the tool as a black box, possibly supported by a (simple) graphical user interface. For these users the applicable knowledge needs to be pre-formalized by a domain specialist or knowledge engineer. Experienced users can also enter factual knowledge, usually directly as Prolog facts, but also as knowledge formalized in XML, or supported by specialized (graphical) knowledge acquisition tools. Expert users can program further rules and flexibly extend the system for inferring (specialized) constraints in Prolog.

In the future, we plan to extend the applicable set of domain knowledge by further knowledge elements. Furthermore, developing advanced graphical tools supporting inexperienced users is a worthwhile direction for further increasing the overall user experience. Since the factual knowledge can either be represented in XML or Prolog, appropriate integration, extension, transformation and presentation of the (derived) knowledge is usually easy to accomplish for (external) tools. Another interesting direction for future work is given by an incremental approach for integrating ontological knowledge bases with the obtained data mining results (patterns). E.g., the knowledge bases could be extended in a boot-strapping manner, or they could be directly validated using the discovered patterns.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proc. 20th Int. Conf. Very Large Data Bases, (VLDB), Morgan Kaufmann (1994) 487–499
2. Atzmueller, M., Puppe, F.: A Methodological View on Knowledge-Intensive Subgroup Discovery. In Staab, S., Svátek, V., eds.: Managing Knowledge in a World of Networks, Proc. 15th EKAW. Volume 4248 of LNCS., Springer (2006) 318–325
3. Atzmueller, M., Puppe, F.: SD-Map - A Fast Algorithm for Exhaustive Subgroup Discovery. In: Proc. 10th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD 2006), Berlin, Springer (2006) 6–17
4. Atzmueller, M., Puppe, F., Buscher, H.P.: Exploiting Background Knowledge for Knowledge-Intensive Subgroup Discovery. In: Proc. 19th Intl. Joint Conference on Artificial Intelligence (IJCAI-05), Edinburgh, Scotland (2005) 647–652
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003.
6. Baumeister, J., Atzmueller, M., Puppe, F.: Inductive Learning for Case-Based Diagnosis with Multiple Faults. In: Advances in Case-Based Reasoning. Volume 2416 of LNAI., Berlin, Springer (2002) 28–42 Proc. 6th European Conference on Case-Based Reasoning.
7. Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns Without Candidate Generation. In Chen, W., Naughton, J., Bernstein, P.A., eds.: 2000 ACM SIGMOD Intl. Conference on Management of Data, ACM Press (05 2000) 1–12
8. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, 2001.
9. Jaroszewicz, S., Simovici, D.A.: Interestingness of Frequent Itemsets using Bayesian Networks as Background Knowledge. In: Proc. 10th Intl. Conference on Knowledge Discovery and Data Mining (KDD '04), New York, NY, USA, ACM Press (2004) 178–186
10. Klösgen, W.: 16.3: Subgroup Discovery. In: Handbook of Data Mining and Knowledge Discovery. Oxford University Press, New York (2002)
11. Klösgen, W.: Explora: A Multipattern and Multistrategy Discovery Assistant. In Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R., eds.: Advances in Knowledge Discovery and Data Mining. AAAI Press (1996) 249–271
12. Richardson, M., Domingos, P.: Learning with Knowledge from Multiple Experts. In: Proc. 20th Intl. Conference on Machine Learning (ICML-2003), AAAI Press (2003) 624–631
13. Seipel, D.: Processing XML-Documents in Prolog. In: Proc. 17th Workshop on Logic Programming (WLP 2002). (2002)
14. Wrobel, S.: An Algorithm for Multi-Relational Discovery of Subgroups. In: Proc. 1st Europ. Symp. Principles of Data Mining and Knowledge Discovery, Berlin, Springer (1997) 78–87
15. Zelezny, F., Lavrac, N., Dzeroski, S.: Using Constraints in Relational Subgroup Discovery. In: Intl. Conference on Methodology and Statistics, University of Ljubljana (2003) 78–81

Logic Languages

LTL Model Checking with Logic Based Petri Nets

Tristan M. Behrens and Jürgen Dix

Department of Informatics, Clausthal University of Technology
Julius-Albert-Straße 4, 38678 Clausthal, Germany
{behrens,dix}@in.tu-clausthal.de

Abstract. In this paper we consider *unbounded model checking* for systems that can be specified in *Linear Time Logic*. More precisely, we consider the model checking problem "$\mathcal{N} \models \alpha$", where $\mathcal{N}$ is a generalized Petri net (SLPN) (which we have introduced in previous work), and α is an LTL formula. We solve this problem by using results about the equivalence of LTL formulae and Büchi automata.

1 Introduction

Model checking [1] is the problem to decide $M \models \alpha$: Is the system M a model of a logical formula α or not? Model checking together with *theorem proving* are two major techniques for *design validation at compile time*. Whereas systems based on theorem provers often rely heavily on user interaction, model checkers are fully automatic and deliver in addition a counter-example if an invalid execution or state of a system is detected.

Systems M are usually represented as *Kripke structures*. When dealing with concurrent systems formulæ are often expressed in *Computational Tree Logic* (CTL) or *Linear Time Logic* (LTL) [2]. Established model checkers rely on different theoretical foundations. For example *automata theory* has been investigated in the past and the model checking problem has been reduced to certain algorithms from graph theory.

Concurrent systems M can be described very nicely with *Petri nets*. Petri nets are mathematical structures based on simple syntax and semantics. They have been applied to model and visualize *parallelism*, *concurrency*, *synchronization* and *resource-sharing*. Different Petri net formalisms share several basic principles so that many theoretical results can be transferred between them [3]. In previous work [4] the authors have introduced a new type of Petri nets, *Simple Logic Petri Nets* (*SLPN*), to model concurrent systems that are *based on logical atoms* (e.g. multiagent systems specified in AgentSpeak).

In this paper we firstly recall the classical results about model checking and Büchi automata (Section 2). Our own work starts with Section 3, where we recall the class *SLPN* of *Simple Logic Petri nets* introduced in [4]. Then we show how to construct a Büchi automaton from an *SLPN* (Section 4), via a Kripke structure. In Section 5 we use the results from Sections 2–4 to solve the model checking problem. Finally, Section 6 discusses related and future work. We conclude with Section 7.

2 LTL and Büchi Automata

Linear time logic (LTL) is a propositional logic with an additional operator for time. There is a very useful equivalence between LTL formula and Büchi automata—automata that operate on infinite words—a relationship that is crucial for model checking.

Definition 1 (LTL Syntax [2]). *Let AP be a set of* atomic propositions*. The language LTL_{AP} is constructed as follows:*

- $AP \subset LTL_{AP}$ *and* $\top, \bot \in LTL_{AP}$*,*
- *if* $\alpha \in LTL_{AP}$ *then* $\neg\alpha \in LTL_{AP}$*,*
- *if* $\alpha, \beta \in LTL_{AP}$ *then* $\alpha \wedge \beta \in LTL_{AP}$ *and* $\alpha \vee \beta \in LTL_{AP}$*,*
- *if* $\alpha, \beta \in LTL_{AP}$ *then* $\bigcirc\alpha \in LTL_{AP}$*,* $[\alpha U \beta] \in LTL_{AP}$ *and* $[\alpha R \beta] \in LTL_{AP}$*.*

Definition 2 (Semantics of LTL). *Let $\pi : \ s_0, s_1, s_2, \ldots$ be an (infinite) sequence of states of a system, generally π_i shall represent the sequence $s_i, s_{i+1}, s_{i+2}, \ldots$. Furthermore let $p\alpha$ be an atomic proposition and α and β LTL-formulæ. Then:*

- $\pi \models ap$ *iff* ap *is true in* s_0 *and* $\pi \models \neg\alpha$ *iff not* $\pi \models \alpha$
- $\pi \models \alpha \wedge \beta$ *iff* $\pi \models \alpha$ *and* $\pi \models \beta$*;* $\pi \models \alpha \vee \beta$ *iff* $\pi \models \alpha$ *or* $\pi \models \beta$
- $\pi \models \bigcirc\alpha$ *iff* $\pi_1 \models \alpha$
- $\pi \models [\alpha U \beta]$ *iff* $\exists i \geq 0 : \ \pi_i \models \beta$ *and* $\forall j, 0 \leq j \leq i : \ \pi_j \models \alpha$
- $\pi \models [\alpha R \beta]$ *iff* $\pi \models \neg[\neg\alpha U \neg\beta]$

The unary temporal operator $\bigcirc\alpha$ denotes that the formula α is true in the *next* state, whereas the binary temporal operator $\alpha U \beta$ (*until*) denotes that the formula α holds *until* β becomes true and $\alpha R \beta$ denotes that a α holds until it is *released* by β. We can express other modalities as follows: $\Diamond\alpha$ (*finally*, a formula will be true in the future) is equivalent to $[\top U \alpha]$ and $\Box\alpha$ (*globally*, a formula is true in all time moments from now on) is equivalent to $[\bot U \alpha]$.

Büchi automata are finite automata that read infinite words. Infinite words are suitable when dealing with nonterminating systems (operating systems, agents).

Definition 3 (Büchi Automaton [5]). *The tuple $\mathcal{B} = \{\Sigma, Q, \Delta, Q_0, A\}$ with*

Σ *a finite alphabet,*
Q *a finite set of states,*
$\Delta \subseteq Q \times \Sigma \times Q$ *a transition relation,*
$Q_0 \subseteq Q$ *a set of initial states,*
$A \subseteq Q$ *a set of accepting states,*

is called Büchi automaton*. It accepts an infinite word w if in the course of parsing it at least one accepting state is visited* infinitely often*.*

Büchi automata are well suited for LTL:

Theorem 1 (Gerth et al. ([6])). *For each LTL formula α there exists a Büchi automaton B_α that generates exactly those infinite sequences Π of states for which α holds.*

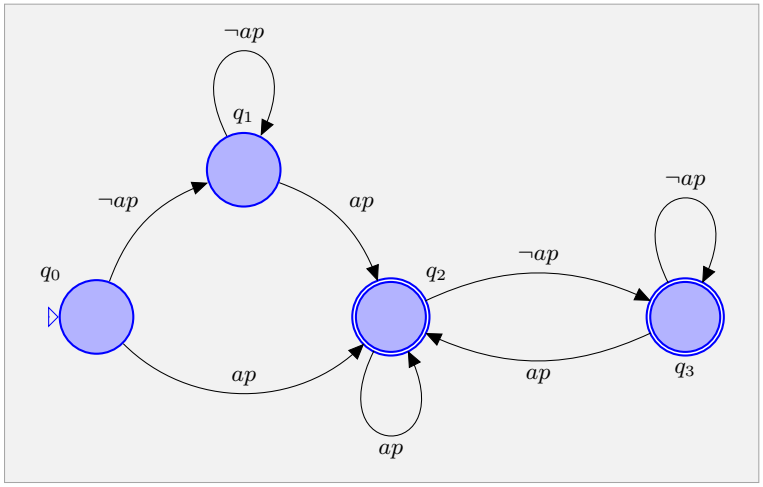


Fig. 1. The Büchi automaton $\mathcal{B}_{\Diamond ap}$ derived from the formula $\Diamond ap$.

The proof is by Gerth's algorithm [6] which takes an LTL formula in positive normal form and generates an equivalent Büchi automaton. This is but one of many algorithms for deriving Büchi automata from LTL-formulæ [7]. It has two main advantages: the automaton can be generated simultaneously with the generation of a model and the algorithm proved to have a good average case complexity.

Example 1 (A Simple Büchi Automaton).
Figure 1 shows the Büchi automaton generated from the LTL formula $\Diamond ap \equiv [\top U ap]$.

3 Simple Logic Petri Nets

In this section we introduce the class *SLPN* of Simple Logic Petri Nets. Let VAR be a finite set of *variables*, $CONST$ be a finite set of *constants*, $FUNC$ be a finite set of *function-symbols*, $TERM$ be the (infinite) set of *terms* constructed from terms, constants and atoms. Let $PRED$ be a set of *predicate symbols*, A^+ be the set of *positive atoms* and A^- be the set of *negative atoms* constructed using predicates and terms. Let $L = A^+ \cup A^-$ be the set of literals, X_{grnd} be the set of all *ground atoms* in each subset $X \subseteq L$. Finally, let $var\text{-}of : 2^L \rightarrow 2^{VAR}$ be the function that selects all variables that are contained in a given set of literals.

Petri nets are often referred to as *token games* and are usually defined by first providing the topology and then the semantics. The topology is a net structure, i.e. a bipartite digraph consisting of *places* and *transitions*. Defining the semantics means (1) defining what a state is, (2) defining when a transition is enabled and, finally, (3) to define how tokens are consumed/created/moved in the net. We refer to the notion that tokens are consumed and created by *firing transitions*—moving is made possible by *consuming* and *creating*. Our most basic definition is as follows:

Definition 4 (Simple Logic Petri Net). *The tuple* $\mathcal{N} = \langle P, T, F, C \rangle$ *with*

$P = \{p_1, \ldots, p_m\}$ *the set of places,*
$T = \{t_1, \ldots, t_n\}$ *the set of transitions,*
$F \subseteq (P \times T) \cup (T \times P)$ *the inhibition relation,*
$C : F \to 2^L$ *the capacity function,*

is called a Simple Logic Petri Net *(SLPN).*

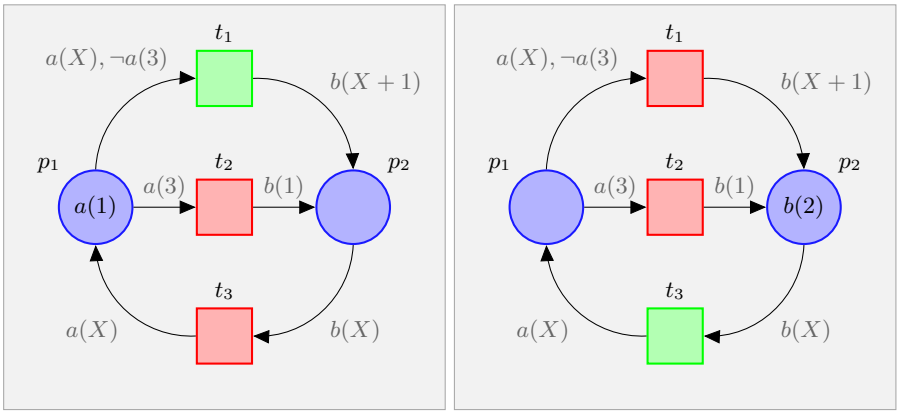


Fig. 2. A Simple Logic Petri Net (*SLPN*) $\mathcal{N}$. On the left is the initial state, t_1 is enabled. The right side shows the state after t_1 has fired. Now t_3 is enabled. This *SLPN* is deadlock-free.

Example 2 (Running example).
Figure 2 shows an *SLPN* in two states of execution. Places are depicted as circles, transitions as boxes, arcs as arrows. Enabled transitions are grey, black otherwise.

Our main notion is that of a *valid net*. To this end we introduce

Definition 5 (Preset, Postset). *For each* $p \in P$ *and each* $t \in T$ *we define the preset and postset of* p *and* t *as follows*

$$\begin{aligned} {\bullet p} &= \{t \in T \mid (t,p) \in F\} \\ p{\bullet} &= \{t \in T \mid (p,t) \in F\} \\ {\bullet t} &= \{p \in P \mid (p,t) \in F\} \\ t{\bullet} &= \{p \in P \mid (t,p) \in F\} \end{aligned}$$

We assume the following two properties: Firstly, if a variable occurs in the label of an **outgoing** arc from a transition, then this variable must also occur in an **ingoing** arc to this transition. Secondly, we do not allow negative atoms as labels of arcs between transitions and places. Otherwise parts of the net would not be executable or would make no sense at all.

Definition 6 (Valid Net). *A SLPN* $\mathcal{N} = \langle P, T, F, C \rangle$ *is* valid *iff the following hold:*

$$\forall t \in T : \ var\text{-}of\left(\bigcup_{p \in t\bullet} C(t,p)\right) \subseteq var\text{-}of\left(\bigcup_{p \in \bullet t} C(p,t)\right) \tag{1}$$

$$\forall (t,p) \in F : \ C(p,t) \in A^+ \tag{2}$$

This means that all variables that are in the labels of arcs between each transition and its postset are also in the labels of the arcs between each transition and its preset. Otherwise we would have uninitialized variables.

Now we have places and transitions and the arcs between them, together with a function which labels each arc with a literal. We would like to have ground atoms inside the places and the ability to move them through the net. Thus we define the state of the net:

Definition 7 (State). *A* state *is a function* $s : P \to 2^{A^+_{grnd}}$.

We denote the *initial state* by s_0. A state s can be written as a vector as $s = \left[s(p_1) \dots s(p_{|P|})\right]^t$ whereas we might leave out the parentheses of the sets $s(p_i)$ iff the representation is clear.

Before describing the transition between states we need the notion of an *enabled transition*. In each state a Petri net might have none, one or several enabled transitions. This holds for our *SLPN*'s as well as for Petri nets in general:

Definition 8 (Enabling of Transitions, Bindings). *A transition* $t \in T$ *is* enabled *if there is* $B \subseteq VAR \times CONST$:

$\forall p \in \bullet t \ \forall a(\boldsymbol{t}) \in \ C(p,t) \cap A^+ : \ a(\boldsymbol{t})_{[B]} \in s(p)$ *and*
$\forall p \in \bullet t \ \forall a(\boldsymbol{t}) \in \ C(p,t) \cap A^- : \neg a(\boldsymbol{t})_{[B]} \notin s(p)$

B is a (possibly empty) set of variable substitutions. We denote wlog by $Subs(t) = \{B_1, \dots, B_n\}$ with $n \in \mathbb{N}$ the set of all sets of such variable substitutions with respect to the transition t for which the above holds.

This means that a transition t is enabled if (1) all positive literals that are labels of arcs between t and its preset $\bullet t$ are unifiable with the literals in the respective places, and (2) that there is no unification for all the negative literals that are labels of arcs between the transition and its preset. Note that we need $\neg a$ in the second formula, because the places never contain negative atoms (closed world assumption).

We are now ready to define transitions of states. *Firing transitions* absorb certain atoms from the places in the preset and put new atoms into the places in the postset using the variable substitutions:

Definition 9 (State Transition).

$$\forall p \in P : \ s'(p) = s(p) \ \setminus \ \left(\bigcup_{t \in p\bullet, tfires} C(p,t)_{[Bind(t)]}\right) \cup \ \left(\bigcup_{t \in \bullet p, tfires} C(t,p)_{[Bind(t)]}\right)$$

Thus each place receives ground literals from each firing transition in its preset and ground literals are absorbed by all the firing transitions in the postset, thus leading to a new state of the whole system.

Example 3 (Running example cont'd).
The initial state is $s_0 = [a(1) \ \emptyset]^t$ After firing t_1 we have the state $s_1 = [\emptyset \ \ b(2)]^t$.

4 SLPNs and Büchi Automata

Theorem 1 states that LTL formulæ are equivalent to Büchi automata. How can we construct a Büchi automaton from a *SLPN*? We construct the Kripke structure representing the state-space of the *SLPN*.

Definition 10 (Kripke Structure $\mathcal{K}$). *The tuple $\mathcal{K} = \langle S, \Delta, S_0, L \rangle$ with*

- $S = \{s_1, \ldots, s_n\}$ *the set of states,*
- $S_0 \subseteq S$ *the set of initial states,*
- $\Delta \subseteq S \ \times S$ *the transition-relation with* $\forall s \in S \ \exists s' \in S : (s, s') \in \Delta$,
- $L : S \rightarrow 2^{AP}$ *the labeling-function, where* $AP = \{a \bullet p \ \mid \ a \in A^+_{grnd}, p \in P\}$ *is the set of* atomic propositions *over SLPN*

is called a Kripke structure.

For each *SLPN* we can generate a respective Kripke structure by exhaustive enumeration of the state-space of the net. The derived Kripke structure is similar to the *reachability graph* of the *SLPN*:

Algorithm 1 $\mathbf{KSfromSLPN}(\mathcal{N})$

$S := \{q_1\}$;
$S_0 := \{q_1\}$;
$\Delta := \emptyset$;
$L := \{(q_1, s_0)\}$;
$queue := \{q_1\}$;
while $queue \neq \emptyset$ ***do***
  $q := removeFirst(queue)$;
  $T := getEnabledTransitions(\mathcal{N}, L(q))$;
  for all $S' \subseteq T, S \neq \emptyset$ ***do***
    $s' := fireTransitions(\mathcal{N}, s, T')$;
    if $\exists q' \in S$ *with* $L(q') = s'$ ***then***
      $\Delta := \Delta \cup \{(q, q')\}$;
    else
      $q' = newNode()$;
      $S := S \cup \{q'\}$;
      $\Delta := \Delta \cup \{(q, q')\}$;
      $L := L \cup \{(q', s')\}$;
      $queue := addLast(queue, a')$;
    end if

end for
end while
return $\langle S, S_0, \Delta, L \rangle$;

Lemma 1 **($\mathcal{K}_\mathcal{N}$).** *Algorithm 1 generates for each SLPN $\mathcal{N}$ a Kripke structure $\mathcal{K}_\mathcal{N}$.*

This algorithm is not guaranteed to terminate.

Example 4 (Running example cont'd.).
Figure 3 shows the Kripke structure generated from $\mathcal{N}$ using our algorithm. The structure is as follows:

$$S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$$
$$S_0 = \{s_1\}$$
$$\Delta = \{(s_i, s_j) \mid j \equiv i + 1 \bmod 6\}$$
$$L(s_1) = [a(1)\ \ \emptyset]^t, L(s_2) = [\emptyset\ \ b(2)]^t, L(s_3) = [a(2)\ \ \emptyset]^t,$$
$$L(s_4) = [\emptyset\ \ b(3)]^t, L(s_5) = [a(3)\ \ \emptyset]^t, L(s_6) = [\emptyset\ \ b(1)]^t$$

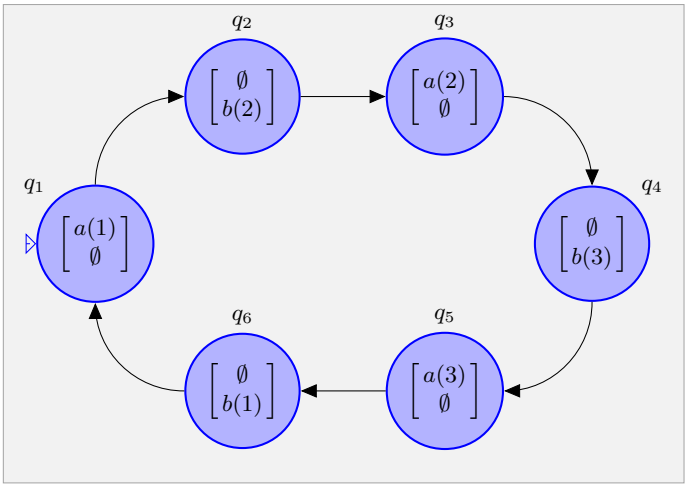


Fig. 3. Kripke structure for the *SLPN* in Fig. 2.

Kripke structures can be easily transformed into Büchi automata:

Theorem 2 **(Büchi automaton $\mathcal{B}_\mathcal{K}$).** *For each Kripke structure $\mathcal{K}$ there exists a Büchi automaton $\mathcal{B}_\mathcal{K}$ that generates exactly those infinite sequences that are equivalent to the possible paths in the Kripke structure.*

Proof (sketch). Transforming a Kripke structure into a Büchi automaton is straightforward. Firstly the states and the arcs of the Kripke structure constitute the states and arcs

of the Büchi automaton. Secondly introduce the initial state and connect it to the states that were initial states in the Kripke structure. Finally put the labels of the states to the incoming arcs and declare all states *accepting* states.

Example 5 (Running example cont'd).
Figure 4 shows the Büchi automaton $\mathcal{B}_{\mathcal{N}}$ generated from the *SLPN* $\mathcal{N}$ in Fig. 2. We are only interested in the truth value of the atomic proposition $b(3) \bullet p_2$ which we denote by ap and $\neg ap$ respectively.

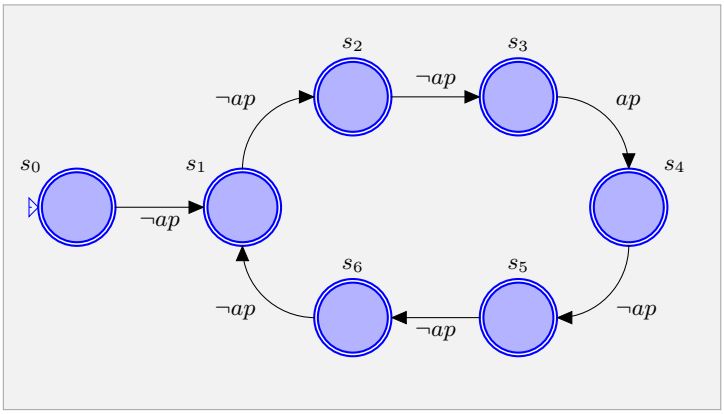


Fig. 4. Büchi automaton for the *SLPN* in Fig. 2 with filtered alphabet: the atomic proposition ap represents $b(3) \bullet p_2$. Note that the automaton is filtered: other atoms in p_2 are ignored as well as p_1 is ignored completely.

5 Model Checking SLPNs

We reduce model checking to the emptiness check of a Büchi automaton.

Theorem 3 (Reducing Model Checking to Büchi Automata). *The model checking problem $\mathcal{N} \models \alpha$ for a SLPN $\mathcal{N}$ and an LTL formula α is equivalent to the emptiness problem of the intersection automaton $\mathcal{B}_{\mathcal{N}} \cap \mathcal{B}_{\neg\alpha}$.*

Proof. Since $\mathcal{B}_{\mathcal{N}}$ generates all possible states of $\mathcal{N}$ (Lemma 1 and Theorem 2) and $\mathcal{B}_{\neg\alpha}$ generates all sequences in which α does not hold, the intersection automaton generates all states in which the formula does not hold. If the generated language is empty α holds in all states.

Lemma 2 (Folklore). *Solving the emptiness problem of a Büchi automaton is equivalent to finding a strongly connected component of the automaton-graph that contains at least one initial and one final state.*

Putting our lemmas and theorems together, we get the following

Algorithm 2 *Given an LTL-formula α and an SLPN $\mathcal{N}$ the* model checking problem *$\mathcal{N} \models \alpha$ can be solved as follows:*

1. *generate the Büchi-Automaton $\mathcal{B}_{\mathcal{N}}$ from the SLPN by considering the Kripke structure representing the state-space of $\mathcal{N}$*
2. *generate the Büchi-Automaton $\mathcal{B}_{\neg\alpha}$ from the formula α using Gerth's algorithm*
3. *solve the emptiness problem for the Büchi-Automaton $\mathcal{B}_{\mathcal{N}} \cap \mathcal{B}_{\neg\alpha}$ by searching a strongly connected component that contains an initial state and a final state.*

If a strongly connected component of the intersection-automaton is found it can serve as a counterexample. A powerful feature of the above algorithm is that it works *on the fly*: the parts of the automata $\mathcal{B}_{\mathcal{N}}$ and $\mathcal{B}_{\neg\alpha}$ need to be generated *on demand only*. Thus a strongly connected component can be found *without complete generation of the automata by expanding them in parallel*.

Unfortunately Algorithm 1 has a drawback: some *SLPN*'s have infinite Kripke structures. Figure 5 shows two such *SLPN*'s. This leads to semi-decidability of our model-checking algorithms. Only in the case when a counterexample exists does the algorithm terminate.

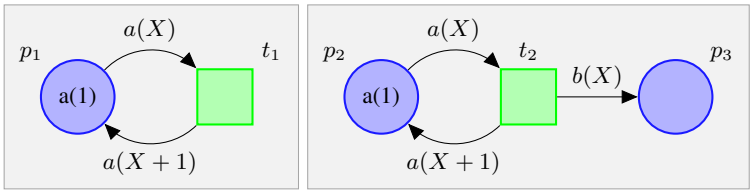


Fig. 5. Two *SLPN*'s with infinite Kripke structures. The left side shows how an *SLPN* $\mathcal{N}_1$ that enumerates the natural numbers. The right side shows an even worse scenario: the cardinality of $s(p_3)$ of $\mathcal{N}_2$ increases in each step and is unbounded.

6 Related and Future Work

As this is an important area of research, there are many competing approaches. Among the most well-known model checkers, there is SPIN: An efficient verification system for models of distributed software systems [8, 9]. The core is the specification language PROMELA—quite similar to structural programming languages—which is to be used to specify concurrent systems. This specification is translated into an automaton together with the correctness claims expressed in LTL. In contrast to our approach SPIN generates an automaton for each asynchronous process in the system and then generates a new automaton for each possible interleaving of the execution of the processes, whereas we generate the interleavings on the fly. Thus our algorithm might find a counterexample without checking all possible interleavings.

- Recently we have made use of the Petri net infrastructure *Petri Net Kernel* [10] (PNK). It is very useful for developers and scientists, because it allows a powerful architecture for basic Petri net tools like editors and simulators and it also permits the extension by versatile plugins. We use PNK for designing Petri nets and because it comes along with the standardized data exchange format *Petri Net Markup Language* (PNML) based on XML.
- We have implemented an *SLPN*-to-*smodels*-converter which takes an *SLPN* and generates a logic program, whose answer sets represent all possible bounded executions of the net. With it we are able to detect deadlocks [4].
- We are working on an AgentSpeak (F)-to-*SLPN*-converter with which we will test our model-checking-framework.
- We plan to examine the relationship between *SLPN* and the Petri-net-classes P/T nets and Colored Petri nets as well as the operational semantics of *SLPN*.

7 Conclusion

SLPN is a class of specialized Petri nets with logical atoms as tokens and logical literals as labels. It is well suited to describe agent languages based on logical atoms (like the family of AgentSpeak languages).

We gave a short summary on LTL model checking with automata: LTL formulæ as well as Kripke structures can be transformed into Büchi automata, finding a final state that is reachable from an initial state and from itself solves the model checking problem.

Finally we showed how to generate a Büchi automaton from a given *SLPN* and concluded with how to perform LTL model checking on *SLPN*'s.

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000) ISBN 0262032708.
2. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). (1990) 995–1072
3. David, R., Alla, H.: Discrete, Continuous, and Hybrid Petri Nets. Springer Verlag (2005) ISBN 3540224807.
4. Behrens, T., Dix, J.: Model checking with logic based petri nets. Technical Report IfI-07-02, Clausthal University of Technology, Dept of Computer Science (May 2007)
5. Thomas, W.: Automata on infinite objects. In Leeuwen, V., ed.: Handbook of Theoretical Computer Science. (1990) 133–164
6. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification Testing and Verification, Warsaw, Poland, Chapman & Hall (1995) 3–18
7. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS, IEEE Computer Society (1986) 332–344
8. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (2003) ISBN 0321228626.
9. Holzmann, G.J.: Software model checking with spin. Advances in Computers **65** (2005) 78–109
10. Kindler, E., Weber, M.: The petri net kernel - an infrastructure for building petri net tools. International Journal on Software Tools for Technology Transfer **3**(4) (2001) 486–497

Integrating Temporal Annotations in a Modular Logic Language

Vitor Nogueira and Salvador Abreu

Universidade de Évora and CENTRIA, Portugal
{vbn,spa}@di.uevora.pt

Abstract Albeit temporal reasoning and modularity are very prolific fields of research in Logic Programming (LP) we find few examples of their integration. Moreover, in those examples, time and modularity are considered orthogonal to each other. In this paper we propose the addition of temporal annotations to a modular extension of LP such that the usage of a module is influenced by temporal conditions. Besides illustrative examples we also provide an operational semantics together with a compiler, allowing this way for the development of applications based on such language.

1 Introduction

The importance of representing and reasoning about temporal information is well known not only in the database community but also in the artificial intelligence one. In the past decades the volume of temporal data has grown enormously, making modularity a requisite for any language suitable for developing applications for such domains. One expected approach in devising a language with modularity and temporal reasoning is to consider that these characteristics co-exist without any direct relationship (see for instance the language MuTACLP [BMRT02] or [NA06]). Nevertheless we can also conceive a scenario where modularity and time are more integrated, for instance where the usage of a module is influenced by temporal conditions. In this paper we follow the later approach in defining a temporal extensions to a language called Contextual Logic Programming (CxLP) [MP93]. This language is a simple and powerful extension of logic programming with mechanisms for modularity. Recent work not only presented a revised specification of CxLP together with a new implementation for it but also explained how this language could be seen as a shift into the Object-Oriented Programming paradigm [AD03]. Finally, CxLP structure is very suitable for integrating with temporal reasoning since its quite straightforward to add the notion of *time of the context* and let that time help to decide if a certain module is eligible or not to solve a goal.

For temporal representation and reasoning we chose Temporal Annotated Constraint Logic Programming (TACLP) [Frü94,Frü96] since this language supports qualitative and quantitative (metric) temporal reasoning involving both time points and time periods (time intervals) and their duration. Moreover, it allows one to represent definite, indefinite and periodical temporal information.

The remainder of this article is structured as follows. In Sects. 2 and 3 we briefly overview CxLP and TACLP, respectively. Section 4 presents the temporal extension of CxLP and Sect. 5 relates it with other languages. Conclusions and proposals for future work follows.

2 An Overview of Contextual Logic Programming

For this overview we assume that the reader is familiar with the basic notions of Logic Programming. Contextual Logic Programming (CxLP) [MP93] is a simple yet powerful language that extends logic programming with mechanisms for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Using the syntax of GNU Prolog/CX (recent implementation for CxLP [AD03]) consider a unit named `employee` to represent some basic facts about university employees, using `ta` and `ap` as an abbreviation of `t`eaching `a`ssistant and `a`ssociate `p`rofessor, respectively:

```
:-unit(employee(NAME, POSITION)).

item :- employee(NAME, POSITION).
employee(bill, ta).
employee(joe, ap).

name(NAME).
position(POSITION).
```

The main difference between the example above and a plain logic program is the first line that declares the unit name (`employee`) along with the unit arguments (`NAME, POSITION`). Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. A unit argument can be interpreted as a "unit global" variable, i.e. one which is shared by all clauses defined in the unit. Therefore, as soon as a unit argument gets instantiated, all the occurrences of that variable in the unit are replaced accordingly.

Suppose also that each employee's position has an associated integer rhat will be used to calculate the salary. Such relation can be easily expressed by the following unit `index`:

```
 :- unit(index(POSITION, INDEX)).

 item :-
      index(POSITION, INDEX).

 index(ta, 12).
 index(ap, 20).

index(INDEX).
position(POSITION).
```

A set of units is designated as a *contextual logic program*. With the units above we can build the program `P = {employee, index}`.

Given that in the same program we can have two or more units with the same name but different arities, to be more precise besides the unit name we should also refer its arity i.e. the number of arguments. Nevertheless, since most of the times there is no ambiguity, we omit the arity of the units. If we consider that `employee` and `index` designate sets of clauses, then the resulting program is given by the union of these sets.

For a given CxLP program, we can impose an order on its units, leading to the notion of *context*. Contexts are implemented as lists of unit designators and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. Moreover, we resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation denoted by `:>` . The goal U `:>` G extends the *current context* with unit `U` and resolves goal `G` in the new context. For instance to obtain Bill's position we could do:

```
| ?- employee(bill, P) :> item

P = ta
```

In this query we extend the initial empty context `[]` [1] with unit employee obtaining context `[employee(bill, P)]` and then resolve query `item`. This leads to `P` being instantiated with `ta`.

Suppose also that the employee's salary is obtained by multiplying the index of its position by the base salary. To implement this rule consider the unit `salary`:

```
:-unit(salary(SALARY)).

item :-
   position(P),
   [index(P, I)] :< item,
   base_salary(B),
   SALARY is I*B.

base_salary(100).
```

The unit above introduces a new operator (`:<`) called *context switch*: goal `[index(P, I)] :< item` invokes `item` in context `[index(P, I)]`. To better grasp the definition of this unit consider the goal:

```
| ?- employee(bill, P) :> (item, salary(S) :> item).
```

[1] In the GNU Prolog/CX implementation the empty context contains all the standard Prolog predicates such as `=/2`.

Since we already explained the beginning of this goal, lets see the remaining part. After `salary/1` being added, we are left with the context `[salary(S), employee(bill,ta)]`. The second `item` is evaluated and the first matching definition is found in unit `salary`. Goal `position(P)` is called and since there is no rule for this goal in the current unit (`salary`), a search in the context is performed. Since `employee` is the topmost unit that has a rule for `position(P)`, this goal is resolved in the (reduced) context `[employee(bill, ta)]`. In an informal way, we queried the context for the position of whom we want to calculate the salary, obtaining `ta`. Next, we query the index corresponding to such position, i.e. `[index(ta, I)] :< item`. Finally, to calculate the salary, we just need to multiply the index by the base salary, obtaining `S = 1200` with the final context `[salary(1200), employee(bill, ta)]`.

3 Temporal Annotated Constraint Logic Programming

This section presents a brief overview of Temporal Annotated Constraint Logic Programming (TACLP) that follows closely Sect. 2 of [RF00]. For a more detailed explanation of TACLP see for instance [Frü96].

We consider the subset of TACLP where time points are totally ordered, sets of time points are convex and non-empty, and only atomic formulae can be annotated. Moreover clauses are free of negation.

Time can be discrete or dense. Time points are totally ordered by the relation $\leq$. We call the set of time points D and suppose that a set of operations (such as the binary operations $+, -$) to manage such points is associated with it. We assume that the time-line is left-bounded by the number 0 and open the future (∞). A *time period* is an interval $[r, s]$ with $0 \leq r \leq s \leq \infty,\ r \in D,\ s \in D$ and represents the convex, non-empty set of time points $\{t \mid r \leq t \leq s\}$. Therefore the interval $[0, \infty]$ denotes the whole time line.

Definition 1 (Annotated Formula). *An annotated formula is of the form $A\alpha$ where A is an atomic formula and α an annotation. Let t be a time point and I be a time period:*

(at) *The annotated formula* **A at t** *means that A holds at time point t.*

(th) *The annotated formula* **A th I** *means that A holds throughout I, i.e. at every time point in the period I.*
A th–annotated formula can be defined in terms of at as: A th $I \Leftrightarrow\ \forall t\ (t \in I \rightarrow A$ at $t)$

(in) *The annotated formula* **A in I** *means that A holds at some time point(s) in the time period I, but there is no knowledge when exactly. The in annotation accounts for indefinite temporal information.*
An in–annotated formula can also be defined in terms of at: A in $I \Leftrightarrow\ \exists t\ (t \in I \wedge A$ at $t)$.

The set of annotations is endowed with a partial order relation $\sqsubseteq$ which turns into a lattice. Given two annotations α and β, the intuition is that $\alpha \sqsubseteq \beta$ if α is "less informative" than β in the sense that for all formulae A, $A\beta \Rightarrow A\alpha$.

In addition to *Modus Ponens*, TACLP has the following two inference rules:

$$\frac{A\alpha \quad \gamma \sqsubseteq \alpha}{A\gamma} \text{ rule } (\sqsubseteq) \qquad \frac{A\alpha \quad A\beta \quad \gamma = \alpha \sqcup \beta}{A\gamma} \text{ rule } (\sqcup)$$

The rule ($\sqsubseteq$) states that if a formula holds with some annotation, then it also holds with all annotations that are smaller according to the lattice ordering. The rule ($\sqcup$) says that if a formula holds with some annotation and the same formula holds with another annotation then it holds in the least upper bound of the annotations. Assuming $r_1 \leq s_1$, $s_1 \leq s_2$ and $s_2 \leq r_2$, we can summarize the axioms for the lattice operation $\sqsubseteq$ by:

$$in[r_1, r_2] \sqsubseteq in[s_1, s_2] \sqsubseteq in[s_1, s_1] = at\ s_1 = th[s_1, s_1] \sqsubseteq th[s_1, s_2] \sqsubseteq th[r_1, r_2]$$

The axioms of the least upper bound $\sqcup$ can be restricted to [2]:

$$th[s_1, s_2] \sqcup th[r_1, r_2] = th[s_1, r_2] \Leftrightarrow s_1 \leq r1, r_1 \leq s_2, s_2 \leq r_2$$

A TACLP *program* is a finite set of TACLP clauses. A TACLP *clause* is a formula of the form $A\alpha \leftarrow C_1, \ldots, C_n, B_1\alpha_1, \ldots, B_m\alpha_m$ $(m, n \geq 0)$ where A is an atom, α and α_i are optional temporal annotations, the C_j's are the constraints and the B_i's are the atomic formulae. Moreover, besides an interpreter for TACLP clauses there is also a compiler that translates them into its CLP form.

4 Temporal Annotations and Contextual Logic Programming

In CxLP with overriding semantics, to solve a goal G in a context C, a search is performed until the topmost unit of C that contains clauses for the predicate of G is found. We propose to adapt this basic mechanism of CxLP (called *context search*) in order to include the temporal reasoning aspects. To accomplish this we add temporal annotations to contexts and to units and it will be the relation between those two annotations that will help to decide if a given unit is eligible to match a goal during a context search.

The addition of time to a context is rather intuitive: instead of a list of unit designators $[u_1, \ldots, un]$ we now have a temporally annotated list of units designators $[u_1, \ldots, un]\alpha$. This annotation α is called the *time of the context* and by default, contexts are implicitly annotated with the current time.

We could follow an approach for units similar to the one proposed for contexts, i.e. to add a temporal annotation to a unit's declaration. Hence we could have units definitions like `:- unit(foo(X)) th [1,4].`

Nevertheless, units and more specifically, units with arguments allow for a refinement of the temporal qualification, i.e. instead of a qualifying the entire unit, we can have several qualifications, one for each possible argument instantiation. For the unit `foo` above we could have:

[2] The least upper bound only has to be computed for overlapping `th` annotations.

```
:- unit(foo(X)).
foo(a) th [1,2].
foo(b) th [3,4].
```

Where the first annotated fact states that unit `foo` with its argument instantiated to `a` has the annotation `th [1,2]`. With these annotations, unit `foo` will be eligible to match a goal in the context `[..., foo(a), ...] in [1,4]` but its not eligible in the context `[..., foo(b), ...] th [3,6]` since $in[1,4] \sqsubseteq th[1,2]$ and $th[3,6] \not\sqsubseteq th[3,4]$. We call those annotated facts the *temporal conditions* of the unit [3].

Each unit defines one temporally annotated predicate with the same name as the unit and arity equal to the number of the unit arguments. For the case of atemporal (timeless) units, it is assumed by default that we have the most general unit designator annotated with the complete time line.

We decided that these temporal annotations can only appear as heads of rules whose body is true, i.e. facts. Such restriction is motivated by efficiency reasons since this way we can compute the least upper bound ($\sqcup$) of the `th` annotated facts before runtime and this way checking the units temporal conditions during a context search is simplified to the verification of partial order ($\sqsubseteq$) between annotations. Moreover, as we shall see in the examples, such restrictions are not limitative since the expressiveness of contexts allow us to simulate TACLP clauses.

Revisiting the employee example, units `employee` and `index` can be written as:

```
:- unit(employee(NAME, POSITION)).

employee(bill, ta) th [2004, inf].
employee(joe, ta) th [2002, 2006].
employee(joe, ap) th [2007, inf].
item.
position(POSITION).
name(NAME).
```

```
:- unit(index(POSITION, INDEX)).
index(ta, 10) th [2000, 2005].
index(ta, 12) th [2006, inf].
index(ap, 19) th [2000, 2005].
index(ap, 20) th [2006, inf].
item.
position(POSITION).
index(INDEX).
```

As an exemplification, consider the goal:

```
?- at 2005 :> employee(joe, P) :> item.
```

In this goal, after asserting that the context temporal annotation is `at 2005`, unit employee is added to the context and goal `item` invoked. The evaluation of `item` is true as long as the unit is eligible in the current context, and this is true if `P` is instantiated with `ta` (teaching assistant), therefore `P = ta`.

Unit `salary` can be defined as:

[3] The reader should notice that this way its still possible to annotate the entire unit, since we can annotate the unit most general designator, for instance we could have $foo(\_)th[1,10]$.

```
:- unit(salary(SALARY)).
item :-
       position(P), index(P, I) :> item,
       base_salary(B), SALARY is B*I.

base_salary(100).
```

There is no need to annotate the goals `position(P)` or `index(P, I) :> item` since they are evaluated in a context with the same temporal annotation. To find out `joe`'s salary in 2005 we can do:

```
?- at 2005 :> employee(joe, P) :> salary(S) :> item.
P = ta
S=1000
```

In the goal above `item` is evaluated in the context `[salary(S), employee(joe, P)] (at 2005)`. Since `salary` is the topmost unit that defines it, the body of the rule for such predicate is evaluated. In order to use the unit `employee(joe, P)` to solve `position(P)`, such unit must satisfy the temporal conditions (`at 2005`), that in this case stands for instantiating `P` with `ta`, therefore we obtain `position(ta)`. A similar reasoning applies for goal `index(ta, I) :> item`, i.e. this `item` is resolved in context `[index(ta, 10), salary(S), employee(joe, ta)] (at 2005)`. The remainder of the rule body is straightforward, leading to the answer `P = ta` and `S = 1000`.

4.1 Compiler

The compiler for this language can be obtained by combining a program transformation with the compiler for TACLP [Frü96]. Given a unit `u`, such transformation rewrites each predicate `P` in the head of a rule by `P'` and add the following clauses to `u`:

```
P :- Temporal_Conditions, !, P'.
P :- :^ P.
```

stating the resolving `P` is equivalent to invoke `P'`, if the temporal conditions are satisfied. Otherwise `P` must be solved in the supercontext (`:^ P`), i.e. `P` is called in the context obtained from removing `u`.

The temporal condition can be formalized as the conjunction $:< [U|\_]\ \alpha,\ U\alpha$, where the first conjunct queries the context for its temporal annotation (α) and its topmost unit (U), i.e. the current unit. The second conjunct checks if the current unit satisfies the time of the context.

As it should be expected, the compiled language is CxLP with constraints. Finally, since GNU Prolog/CX besides the CxLP primitives also has a constraint solver for finite domains (CLP(FD)), the implementation of this language is direct on such a system.

4.2 Application to Legal Reasoning

Legal reasoning is a very productive field to illustrate the application of these languages. Not only a modular approach is very suitable for reasoning about laws but also time is pervasive in their definition.

The following example was taken from the British Nationality Act and it was presented in [BMRT02] to exemplify the usage of the language MuTACLP. The reason to use an existing example is twofold: not only we consider it to be a simple and concise sample of legal reasoning but also because this way we can give a more thorough comparison with MuTACLP. The textual description of this law can be given as a person `X` obtains the British Nationality at time `T` if:

- `X` is born in the UK at the time `T`
- `T` is after the commencement
- `Y` is a parent of `X`
- `Y` is a British citizen or resident at time `T`.

Assuming that the temporal unit `person` represents the name and the place where a person was born:

```
:- unit(person(Name, Country)).
person(john, uk) th ['1969-8-10', inf].
```

The temporal annotation of this unit can be interpreted as the person time frame, i.e. when she was born and when she died (if its alive, we represent it by `inf`).

Before presenting the rule for the nationality act we still need to represent some facts about who is a British citizen along with who is parent of whom:

```
:- unit(british_citizen(Name)).
british_citizen(bob)
     th ['1940-9-6', inf].

:- unit(parent(Parent, Son)).
parent(bob, john)
     th ['1969-8-10', inf].
```

Considering that the commencement date for this law is '1955-1-1', one formalization of this law in our language is [4]:

```
th [L, _] :> person(X, uk) :> item, fd_min(L, T),
'1955-1-1' #=< T,
at T :> (parent(Y, X) :> item,
                        (british_citizen(Y) :> item;
                         british_resident(Y) :> item)).
```

The explanation of the goal above is quite simple since each line correspond to one condition of the textual description of the law.

[4] `fd_min(X, N)` succeeds if `N` is the minimal value of the current domain of `X`.

5 Related Work

Since [BMRT02] relates MuTACLP with proposals such as Temporal Datalog [OM94] and the work on amalgamating knowledge bases [Sub94], we decided to confine ourselves to the comparison between MuTACLP and our language. MuTACLP (Multi-Theory Temporal Annotated Constraint Logic Programming) is a knowledge representation language that provides facilities for modeling and handling temporal information, together with some basic operators for combining different knowledge bases. Although both MuTACLP and the language here proposed use TACLP (Temporal Annotated Constraint Logic Programming) for handling temporal information, it is in the way that modularity is dealt that they diverge: we follow a dynamic approach (also called *programming-in-the-small*) while MuTACLP engages a static one (also called *programming-in-the-large*).

Moreover, the use of contexts allows for a more compact writing where some of the annotations of the MuTACLP version are subsumed by the annotation of the context. For instance, one of the rules of the MuTACLP version of the example of legal reasoning is:

```
get_citizenship(X) at T <- T >= Jan 1 1955, born(X, uk) at T,
                           parent(Y, X) at T,
                           british_citizen(Y) at T.
```

In [NA06] a similar argument was used when comparing with *relational frameworks* such as the one proposed by Combi and Pozzi in [CP04] for workflow management systems, where relational queries were more verbose that its contextual version.

6 Conclusion and Future Work

In this paper we presented a temporal extension of CxLP where time influences the eligibility of a module to solve a goal. Besides illustrative examples we also provided a compiler, allowing this way for the development of applications based on these languages. Although we provided (in the Appendix A) the operational semantics we consider that to obtain a more solid theoretical foundation there is still need for a fixed point or declarative definition.

Besides the domain of application exemplified we are currently applying the language proposed to other areas such as medicine and natural language. Finally, it is our goal to continue previous work [AN06,ADN04] and show that this language can act as the backbone for constructing and maintaining temporal information systems.

References

AD03. Salvador Abreu and Daniel Diaz. Objective: In minimum context. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2003.

ADN04. Salvador Abreu, Daniel Diaz, and Vitor Nogueira. Organizational information systems design and implementation with contextual constraint logic programming. In *IT Innovation in a Changing World – The 10th International Conference of European University Information Systems*, Ljubljana, Slovenia, June 2004.

AN06. Salvador Abreu and Vitor Nogueira. Towards structured contexts and modules. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 436–438. Springer, 2006.

BMRT02. Paolo Baldan, Paolo Mancarella, Alessandra Raffaetà, and Franco Turini. Mutaclp: A language for temporal reasoning with multiple theories. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 2002.

CP04. Carlo Combi and Giuseppe Pozzi. Architectures for a temporal workflow management system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 659–666, New York, NY, USA, 2004. ACM Press.

Frü94. T. Frühwirth. Annotated constraint logic programming applied to temporal reasoning. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming: 6th International Symposium (PLILP'94)*, pages 230–243. Springer, Berlin, Heidelberg, 1994.

Frü96. Thom W. Frühwirth. Temporal annotated constraint logic programming. *J. Symb. Comput.*, 22(5/6):555–583, 1996.

MP93. Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.

NA06. Vitor Nogueira and Salvador Abreu. Temporal contextual logic programming. In Francisco J. López Fraguas, editor, *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP'06)*, Madrid, Spain, November 2006. Electronic Notes in Theoretical Computer Science.

OM94. Mehmet A. Orgun and Wanli Ma. An overview of temporal and modal logic programming. In *ICTL '94: Proceedings of the First International Conference on Temporal Logic*, pages 445–479, London, UK, 1994. Springer-Verlag.

RF00. Alessandra Raffaetà and Thom Frühwirth. *Labelled deduction*, chapter Semantics for temporal annotated constraint logic programming, pages 215–243. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Sub94. V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Trans. Database Syst.*, 19(2):291–331, 1994.

A Operational Semantics

In this section we assume the following notation: C, C' for contexts, u for unit, $\theta, \sigma, \varphi, \epsilon$ for substitutions, α, β, γ for temporal annotations and $\emptyset, G$ for non-annotated goals.

We also assume a prior computation of the least upper bound for the units `th` annotations. This procedure is rather straightforward and can be describe as:

if $A\ th\ I$ and $B\ th\ J$ are in a unit u, such that I and J overlap, then remove those facts from u and insert $A\ th(I \sqcup J)$. This procedure stops when there are no more facts in that conditions. Moreover, the termination is guaranteed because at each step we decrease the size of a finite set, the set of th annotated facts.

Null goal

$$\frac{}{C\alpha \vdash \emptyset[\epsilon]} \tag{1}$$

The null goal is derivable in any temporal annotated context, with the *empty* substitution ϵ as result.

Conjunction of goals

$$\frac{C\alpha \vdash G_1[\theta] \qquad C\alpha\theta \vdash G_2\theta[\sigma]}{C\alpha \vdash G_1, G_2[\theta\sigma\lceil vars(G_1, G_2)]} \tag{2}$$

To derive the conjunction derive one conjunct first, and then the other in the same context with the given substitutions [5].
Since C may contain variables in unit designators or temporal terms that may be bound by the substitution θ obtained from the derivation of G_1, we have that θ must also be applied to $C\alpha$ in order to obtain the updated context in which to derive $G_2\theta$.

Context inquiry

$$\frac{}{C\alpha \vdash :> C'\beta[\theta]} \begin{cases} \theta = \text{mgu}(C, C') \\ \beta \sqsubseteq \alpha \end{cases} \tag{3}$$

In order to make the context switch operation (4) useful, there needs to be an operation which fetches the context. This rule recovers the current context C as a term and unifies it with term C', so that it may be used elsewhere in the program. Moreover, the annotation β must be less (or equal) informative than the annotation α ($\beta \sqsubseteq \alpha$).

Context switch

$$\frac{C'\beta \vdash G[\theta]}{C\alpha \vdash C'\beta\ :< G[\theta]} \tag{4}$$

The purpose of this rule is to allow execution of a goal in an arbitrary temporal annotated context, independently of the current annotated context. This rule causes goal G to be executed in context $C'\beta$.

Reduction

[5] The notation $\delta\lceil V$ stands for the restriction of the substitution δ to the variables in V.

$$\frac{(uC\alpha)\ \theta\sigma \vdash B\theta\sigma[\varphi]}{uC\ \alpha \vdash G[\theta\sigma\varphi\lceil vars(G)]} \begin{cases} H \leftarrow B \in u \\ \theta = \text{mgu}(G, H) \\ (u\theta\sigma)\ \beta \in u \\ \alpha \sqsubseteq \beta \end{cases} \tag{5}$$

This rule expresses the influence of temporal reasoning on context search. In an informal way we can say that when a goal (G) has a definition ($H \leftarrow B \in u$ and $\theta = \text{mgu}(G, H)$) in the topmost unit ($u$) of the annotated context ($uC\alpha$), and such unit satisfies the temporal conditions, to derive the goal we must call the body of the matching clause, after unification [6]. The verification of the temporal conditions stands for checking if there is a unit temporal annotation ($(u\theta\sigma)\beta \in u$) that is "more informative" than the annotation of the context ($\alpha \sqsubseteq \beta$), i.e. if $(u\theta\sigma)\ \alpha$ is true.

Context traversal:

$$\frac{C\alpha \vdash G[\theta]}{uC\alpha \vdash G[\theta]}\ \{\, pred(G) \notin \overline{u} \tag{6}$$

When none of the previous rules applies and the predicate of `G` isn't defined in the predicates of `u` ($\overline{u}$), remove the top element of the context, i.e. resolve goal G in the supercontext.

Application of the rules It is almost direct to verify that the inference rules are mutually exclusive, leading to the fact that given a derivation tuple $C\alpha \vdash G[\theta]$ only one rule can be applied.

[6] Although this rule might seem complex, that has to do essentially with the abundance of unification's ($\theta\sigma\varphi$)

Proposal of Visual Generalized Rule Programming Model for Prolog*

Grzegorz J. Nalepa[1] and Igor Wojnicki[1]

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
gjn@agh.edu.pl, wojnicki@agh.edu.pl

Abstract. The rule-based programming paradigm is omnipresent in a number of engineering domains. However, there are some fundamental semantical differences between it, and classic programming approaches. No generic solution for using rules to model business logic in classic software has been provided so far. In this paper a new approach for Generalized Rule-based Programming (GREP) is given. It is based on a use of advanced rule representation called XTT, which includes an extended attribute-based language, a non-monotonic inference strategy, with an explicit inference control on the rule level. The paper shows, how some typical programming constructions, as well as classic programs can be modelled in this approach. The paper also presents possibilities of an efficient integration of this technique with existing software systems.

1 Introduction

The rule-based programming paradigm is omnipresent in a number of engineering domains such as control and reactive systems, diagnosis and decision support. Recently, there has been a lot of effort to use *rules* to model business logic in classic software. However, there are some fundamental semantical differences between it, and classic procedural, or object-oriented programming approaches. This is why no generic modelling solution has been provided so far. The motivation of this paper is to investigate the possibility of modelling some typical programming structures with the rule-based programming with use of on extended forward-chaining rule-based system.

In this paper a new approach for generalized rule-based programming is given. The *Generalized Rule Programming* (GREP) is based on the use of an advanced rule representation, which includes an extended attribute-based language [1], a non-monotonic inference strategy, with an explicit inference control at the rule level. The paper shows, how some typical programming constructions (such as loops), as well as classic programs (such as factorial) can be modelled in this approach. The paper presents possibilities of efficient integration of this technique with existing software systems in different ways.

* The paper is supported by the Hekate Project funded from 2007–2008 resources for science as a research project.

In Sect. 2 some basics of rule-based programming are given, and in Sect. 3 some fundamental differences between software and knowledge engineering are identified. Then, in Sect. 4 the extended model for rule-based systems is considered. The applications of this model are discussed in Sect. 5. This model could be integrated in the classic software system in several ways. The research presented in this paper is work-in-progress. Directions for the future research as well as concluding remarks are given in Sect. 6.

2 Concepts of the Rule-Based Programming

Rule-Based Systems (RBS) constitute a powerful AI tool [2] for specification of knowledge in design and implementation of systems in the domains such as system monitoring and diagnosis, intelligent control, and decision support. For the state-of-the-art in RBS see [3,4,1].

In order to design and implement a RBS in a efficient way, the knowledge representation method chosen should support the designer introducing a scalable *visual representation*. As the number of rules exceeds even relatively very low quantities, it is hard to keep the rule-base consistent, complete, and correct. These problems are related to knowledge-base verification, validation, and testing. To meet security requirements a *formal analysis end verification* of RBS should be carried out [5]. This analysis usually takes place after the design. However, there are design and implementation methods, such as the XTT, that allow for on-line verification during the design and gradual refinement of the system.

Supporting rulebase modelling remains an essential aspect of the design process. The simplest approach consists in writing rules in the low-level RBS language, such as one of *Jess* (`www.jessrules.com`). More sophisticated are based on the use of some classic visual rule representations. This is a case of *LPA VisiRule*, (`www.lpa.co.uk`) which uses decision trees. Approaches such as XTT aim at developing new visual language for *visual rule modelling*.

3 Knowledge in Software Engineering

Rule-based systems (RBS) constitute today one of the most important classes of the so-called Knowledge Based Systems (KBS). RBS found wide range of industrial applications is some „classic AI domains" such as decision support, system diagnosis, or intelligent control. However, due to some fundamental differences between knowledge (KE) and software engineering (SE) the technology did not find applications in the mainstream software engineering.

What is important about the KE process, is the fact that it should *capture* the expert knowledge and *represent* it in a way that is suitable for processing (this is the task for a knowledge engineer). The level at which KE should operate is often referred to as *the knowledge level* [6]. In case of KBS there is no single universal engineering approach, or universal modelling method (such as UML in software engineering).

However, software engineering (SE) is a domain where a number of mature and well-proved design methods exist. What makes the SE process different from knowledge engineering is the fact that systems analysts try to *model* the *structure* of the real-world information system in the structure of computer software system. So the structure of the software corresponds, to some extent, to the structure of the real-world system.

The fundamental differences between the KE and SE approaches include: declarative vs. procedural point-of-view, semantic gaps present in the SE process, between the requirements, the design, and the implementation, and the application of a gradual abstraction as the main approach to the design. The solution introduced in this paper aims at integrating a classic KE methodology of RBS with SE. It is hoped, that the model considered here, *Generalized Rule Programming* (GREP), could serve as an effective bridge between SE and KE.

4 Extended Rule Programming Model

The approach considered in this paper is based on an extended rule-based model. The model uses the *XTT* knowledge method with certain modifications. The XTT method was aimed at forward chaining rule-based systems. In order to be applied to the general programming, it is extended in several aspects.

4.1 XTT – EXtended Tabular Trees

The *XTT* (*EXtended Tabular Trees*) knowledge representation [7], has been proposed in order to solve some common design, analysis and implementation problems present in RBS. In this method three important representation levels has been addressed: *visual* – the model is represented by a hierarchical structure of linked extended decision tables, *logical* – tables correspond to sequences of extended decision rules, *implementation* – rules are processed using a Prolog representation.

On the visual level the model is composed of extended decision tables. A single table is presented in Fig. 1. The table represents a set of rules, having the same attributes. A rule can be read as follows:
$(A11 \in a11) \wedge \ldots (A1n \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1)$

It includes two main extensions compared to the classic RBS: 1) non-atomic attribute values, used both in conditions and decisions, 2) non-monotonic reasoning support, with dynamic assert/retract operations in decision part. Every table row corresponds to a decision rule. Rows are interpreted from the top row to the bottom one. Tables can be linked in a graph-like structure. A link is followed when rule (row) is fired.

On the logical level a table corresponds to a number of rules, processed in a sequence. If a rule is fired and it has a link, the inference engine processes the rule in another table. The rule is based on an *attributive language* [1]. It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee h$ where p is a literal in SAL (set attributive logic, see [1]) in a form $A_i(o) \in t$ where $o \in O$ is a

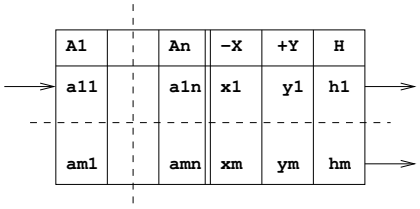


Fig. 1. A single XTT table.

object referenced in the system, and $A_i \in A$ is a selected attribute of this object (property), $t \subseteq D_i$ is a subset of attribute domain A_i. Rules are interpreted using a unified knowledge and fact base, that can be dynamically modified during the inference process using Prolog-like assert/retract operators in rule decision part. Rules are implemented using Prolog-based representation, using terms, which is a flexible solution (see [8]). However, it requires a dedicated meta-interpreter [9].

This model has been successfully used to model classic rule-based expert systems. For the needs of general programming described in this paper, some important modifications are proposed.

4.2 Extending XTT into GREP

Considering using XTT for general applications, there have been several extensions proposed regarding the base XTT model. These are: *Grouped Attributes*, *Attribute-Attribute Comparison*, *Link Labeling*, *Not-Defined Operator*, *Scope Operator*, *Multiple Rule Firing*. Applying these extensions constitute *GREP*. Additionally there are some examples given in Sect. 5 regarding the proposed extensions.

Grouped Attributes provide means for putting together some number of attributes to express relationships among them and their values. As a result a complex data structure, called a *group*, is created which is similar to constructs present in programming languages (i.e. C structures). A group is expressed as:

$$Group(Attribute1, Attribute2, \ldots, AttributeN)$$

Attributes within a group can be referenced by their name:

$$Group.Attribute1$$

or position within the group:

$$Group/1$$

An application of Grouped Attributes could be expressing spatial coordinates:

$$Position(X, Y)$$

where *Position* is the group name, X and Y are attribute names.

The *Attribute-Attribute Comparison* concept introduces a powerful mechanism to the existing XTT comparison model. In addition to comparing an attribute value against a constant (*Attribute-Value Comparison*) it allows for comparing an attribute value against another attribute value. The *Attribute-Value Comparison* can be expressed as a condition:

```
if (Attribute OPERATOR Value) then ...
```

where `OPERATOR` is a comparison operator i.e. equal, greater then, less than etc., while *Attribute-Attribute Comparison* is expressed as a condition:

```
if (Attribute1 OPERATOR Attribute2) then ...
```

where `OPERATOR` is a comparison operator or a function, in a general case:

```
if (OPERATOR(Attribute1,...,AttributeN)) then ...
```

The operators and functions used here are predefined.

The *Link Labeling* concept allows to reuse certain XTTs which is similar to subroutines in procedural programming languages. Such a reused XTT can be executed in several contexts. There are incoming and outgoing links. Links might be labeled (see Fig.2). In such a case, if the control comes from a labeled link it has to be directed through an outgoing link with the same label. There can be multiple labeled links for a single rule. If an outgoing link is not labeled it means that if a corresponding rule is fired the link will be followed regardless of the incoming link label. Such a link (or links) might be used to provide a control for exception-like situations.

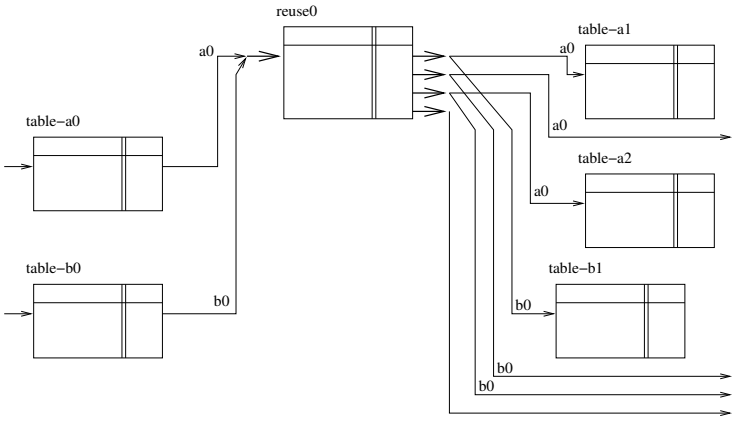


Fig. 2. Reusing XTTs with Link Labeling.

In the example given in Fig. 2 an outgoing link labeled with $a0$ is followed if the corresponding rule (row) of $reuse0$ is fired and the incoming link is labeled with $a0$. This happens if the control comes from XTT labeled as $table\text{-}a0$. An

outgoing link labeled with $b0$ is followed if the corresponding rule of $reuse0$ is fired and the incoming link is labeled with $b0$; the control comes from *table-b*0.

The proposed *Not-Defined* (N/D) operator checks if a value for a given attribute has been defined. It has a broad application regarding modelling basic programming structures, i.e. to make a certain rule fired if the XTT is executed for the first time (see Sect. 5.2).

The graphical *Scope Operator* provides a basis for modularized knowledge base design. It allows for treating a set of XTTs as a certain *Black Box* with well defined input and output (incoming and outgoing links), see Fig. 3. *Scope Operators* can be nested. In such a way a hierarchy of abstraction levels of the system being designed is provided, making modelling of conceptually complex systems easier. The scope area is denoted with a dashed line. Outside the given scope only conditional attributes for the incoming links and the conclusion attributes for the outgoing links are visible. In the given example (Fig. 3) attributes A, B, C are input, while H, I are outputs. Any value changes regarding attributes: E, F, and G are not visible outside the scope area, which consists of *table-b0* and *table=b1* XTTs; no changes regarding values of E, F, and G are visible to *table-a0*, *table-a1* or any other XTT outside the scope.

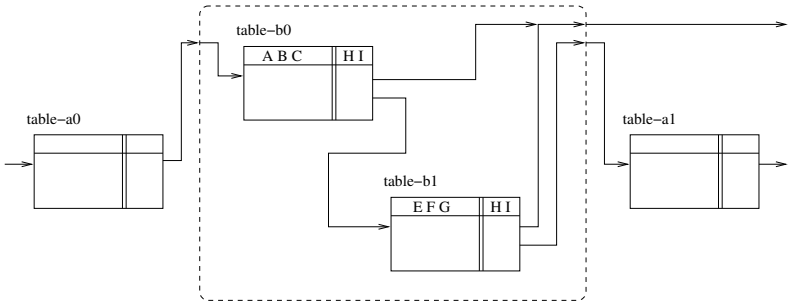


Fig. 3. Introducing Graphical Scope.

Since multiple values for a single attribute are already allowed, it is worth pointing out that the new inference engine being developed treats them in a more uniform and comprehensive way. If a rule is fired and the conclusion or assert/retract use a multi-value attribute such a conclusion is executed as many times as there are values of the attribute. It is called *Multiple Rule Firing*. This behavior allows to perform aggregation or set based operations easily. Some examples are given in Sec. 5.

5 Applications of GREP

This section presents some typical programming constructs developed using the XTT model. It turned out that extending XTT with the modifications described in Sect. 4.2 allows for applying XTT in other domains than rule-based systems making it a convenient programming tool.

5.1 Modelling Basic Programming Structures

Two main constructs considered here are: a conditional statement, and a loop.

Programming a conditional with rules is both simple and straightforward, since a rule is by definition a conditional statement. In Fig. 4 a single table system is presented. The first row of the table represents the main conditional statement. It is fired if C equals some value v, the result is setting F to $h1$, then the control is passed to other XTT following the outgoing link. The next row implements the `else` statement if the condition is not met ($C \neq v$) then F is set to $h2$ and the control follows the outgoing link. The `F` attribute is the decisive one.

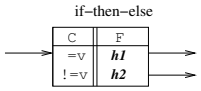


Fig. 4. A conditional statement.

A loop can be easily programmed, using the dynamic fact base modification feature. In Fig. 5 a simple system implementing the *for*-like loop is presented. In the XTT table the initial execution, as well as the subsequent ones are programmed. The I attribute serves as the counter. In the body of the loop the value of the decision attribute Y is modified depending on the value of the conditional attribute X. The loop ends when the counter attribute value is greater than the value z. This example could be easily generalized into the *while* loop. Using the non-atomic attribute values (an attribute can have a *set* of values) the *foreach* loop could also be constructed.

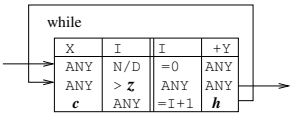


Fig. 5. A loop statement.

5.2 Modelling Simple Programming Cases

A set of rules to calculate a factorial is showed in Fig. 6. An argument is given as the attribute X. The calculated result is given as Y. The firs XTT ($cactorial0$) calculates the result if $X = 1$ or $X = 0$, otherwise control is passed to $factorial1$ which implements the iterative algorithm using the S attribute as the counter.

Since an attribute can be assigned more than a single value (i.e. using the assert feature), certain operations can be performed on such a set (it is similar to aggregation operations regarding Relational Databases). An example of sum function is given in Fig. 7. It adds up all values assigned to X and stores the

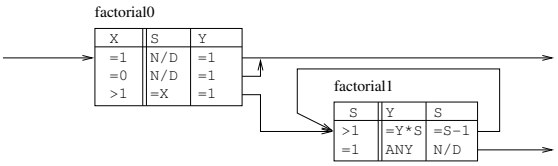


Fig. 6. Calculating Factorial of X, result stored as Y.

result as a value of Sum attribute. The logic behind is as follows. If Sum is not defined then make it 0 and loop back. Than, the second rule is fired, since Sum is already set to 0. The conclusion is executed as many times as values are assigned to X. If Sum has a value set by other XTTs prior to the one which calculates the sum, the result is increased by this value.

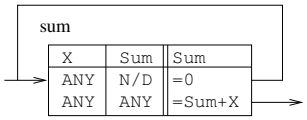


Fig. 7. Adding up all values of X, result stored as Sum.

There is also an alternative implementation given in Fig. 8. The difference, comparing with Fig. 7, is that there is an incoming link pointing at the second rule, not the first one. Such an approach utilizes the partial rule execution feature. It means that only the second (and subsequent, if present) rule is investigated. This implementation adds up all values of X regardless if Sum is set in previous XTTs.

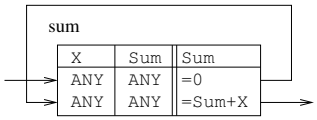


Fig. 8. Adding up all values of X, result stored as Sum, an alternative implementation.

Assigning a set of values to an attribute based on values of another attribute is given in Fig. 9. The given XTT populates Y with all values assigned to X. It uses the XTT assert feature.

Using XTT, even a complex task such as browsing a tree can be implemented easily. A set of XTTs finding successors of a certain node in a tree is given in Fig. 10. It is assumed that the tree is expressed as a group of attributes $t(P, N)$, where N is a node name, and P is a parent node name. The XTTs find all successors of a node whose name is given as a value of attribute P (it is allowed to specify multiple values here). A set of successors is calculated as values of F. The first XTT computes immediate child nodes of the given one. If there are some child nodes, control is passed to the XTT labeled *tree2*. It finds child nodes

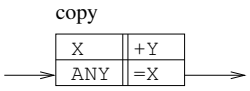


Fig. 9. Copying all values of X into Y.

of the children computed by *tree1* and loops over to find children's children until no more child nodes can be found. The result is stored as values of F.

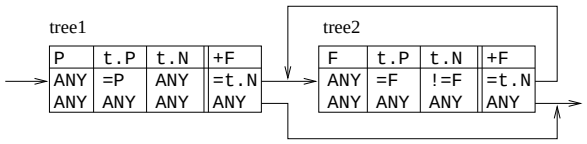


Fig. 10. Finding successors of a node P in a tree, results stored as F.

Up to now GREP has been discussed only on the conceptual level, using the visual representation. However, it has to be accompanied by some kind of runtime environment. The main approach considered here is the one of the classic XTT. It is based on using a high level Prolog representation of GREP. Prolog semantics includes all of the concepts present in GREP. Prolog has the advantages of flexible symbolic representation, as well as advanced meta-programming facilities [9]. The GREP-in-Prolog solution is based on the XTT implementation in Prolog, presented in [8]. In this case a term-based representation is used, with an advanced meta interpreter engine provided.

6 Evaluation and Future Work

In the paper the results of the research in the field of knowledge and software engineering are presented. The research aims at the unification of knowledge engineering methods with software engineering. The paper presents a new approach for Generalized Rule-based Programming called GREP. It is based on the use of an advanced rule representation called XTT, which includes an extended attribute-based language, a non-monotonic inference strategy, with explicit inference control on the rule level.

The original contribution of the paper consists in the extension of the XTT rule-based systems knowledge representation method, into GREP, a more general programming solution; as well as the demonstration how some typical programming constructions (such as loops), as well as classic programs (such as factorial, tree search) can be modelled in this approach. The expressiveness and completeness of this solution has been already investigated with respect to a number of programming problems which were showed. However, GREP lacks features needed to replace traditional programming languages at the current

stage. The problem areas include: general data structures support, limited actions/operations in the decision and precondition parts, and input/output operations, including environment communication.

Future work will be focused on the GREP extensions, especially *hybrid operators* and use cases. *Hybrid operators*, defined in a similar way as Prolog predicates, could offer extended processing capabilities for attributes, especially grouped ones, serving for generative, restrictive or triggering operations. These are to generate attribute values, restrict admissible attribute values, or trigger arbitrary code execution, respectively. Functionality of such operators would be defined in a backward chaining manner to provide query-response characteristics.

While proposing extensions to GREP, one should keep in mind, that in order to be both coherent and efficient, GREP cannot make an extensive use of some of the facilities of other languages, even Prolog. Even though Prolog semantics is quite close to the one of GREP there are some important differences – related to the different inference (forward in GREP, backward in Prolog) and some programming features such as recursion and unification. On the other hand, GREP offers a clear *visual* representation that supports a high level logic design. The number of failed visual logic programming attempts show, that the powerful semantics of Prolog cannot be easily modelled [10]. So from this perspective, GREP expressiveness is, and should remain weaker than that of Prolog.

In its current stage GREP can successfully model a number of programming constructs and approaches. This proves GREP can be applied as a general purpose programming environment. However, the research should be considered an experimental one, and definitely a work in progress.

References

1. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
2. Negnevitsky, M.: Artificial Intelligence. A Guide to Intelligent Systems. Addison-Wesley, Harlow, England; London; New York (2002) ISBN 0-201-71159-1.
3. Liebowitz, J., ed.: The Handbook of Applied Expert Systems. CRC Press, Boca Raton (1998)
4. Jackson, P.: Introduction to Expert Systems. 3rd edn. Addison–Wesley (1999) ISBN 0-201-87686-8.
5. Vermesan, A., Coenen, F., eds.: Validation and Verification of Knowledge Based Systems. Theory, Tools and Practice. Kluwer Academic Publisher, Boston (1999)
6. Newell, A.: The knowledge level. Artificial Intelligence **18** (1982) 87–127
7. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science **31** (2005) 89–95
8. Nalepa, G.J., Ligęza, A.: Prolog-based analysis of tabular rule-based systems with the xtt approach. In Sutcliffe, G.C.J., Goebel, R.G., eds.: FLAIRS 2006: proceedings of the 19th international Florida Artificial Intelligence Research Society conference, AAAI Press (2006) 426–431
9. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall (1996)
10. Fisher, J.R., Tran, L.: A visual logic. In: SAC. (1996) 17–21

Prolog Hybrid Operators in the Generalized Rule Programming Model*

Igor Wojnicki[1] and Grzegorz J. Nalepa[1]

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
gjn@agh.edu.pl, wojnicki@agh.edu.pl

Abstract. This paper describes the so-called Hybrid Operators in Prolog – a concept which extends the Generalized Rule Based Programming Model (GREP). This extension allows a GREP-based application to communicate with the environment by providing input/output operations, user interaction, and process synchronization. Furthermore, it allows for the integration of such an application with contemporary software technologies including Prolog based code. The proposed Hybrid Operators extend GREP as a knowledge-based software development concept.

1 Introduction

Rule-Based Systems (RBS) constitute a powerful and well-known AI tool [1] for specification of knowledge. They are used in design and implementation of systems in the domains such as system monitoring and diagnosis, intelligent control, and decision support (see [2,3,4]). From a point of view of classical knowledge engineering (KE) a rule-based expert system consists of a knowledge base and an inference engine. The KE process aims at designing and evaluating the knowledge base, and implementing the inference engine. In order to design and implement a RBS in a efficient way, the chosen knowledge representation method should support the designer introducing a scalable *visual representation*.

Supporting rulebase modelling remains an essential aspect of the design process. The simplest approach consists in writing rules in the low-level RBS language, such as one of *Jess* (`www.jessrules.com`). More sophisticated are based on the use of some classic visual rule representations. This is a case of *LPA VisiRule*, (`www.lpa.co.uk`) which uses decision trees. Approaches such as XTT aim at developing new visual language for *visual rule modelling*.

When it comes to practical implementation of RBS, a number of options exist. These include expert systems shells such as *CLIPS*, or Java-based *Jess*; and programming languages. In the classic AI approach *Prolog* becomes the language of choice, thanks to its logic-based knowledge representation and processing. The important factor is, that Prolog semantics is very close to that of RBS.

* The paper is supported by the Hekate Project funded from 2007–2008 resources for science as a research project.

RBS are found in a wide range of industrial applications in some „classic AI domains", e.g. decision support, system diagnosis, or intelligent control. However, due to some fundamental differences between knowledge and software engineering, the technology did not find applications in the mainstream software engineering.

2 Generalized Rule Programming with XTT

The *XTT* (*EXtended Tabular Trees*) knowledge representation [5], has been proposed in order to solve some common design, analysis and implementation problems present in RBS. In this method three important representation levels has been addressed: *visual* – the model is represented by a hierarchical structure of linked extended decision tables, *logical* – tables correspond to sequences of extended decision rules, *implementation* – rules are processed using a Prolog representation.

On the visual level the model is composed of extended decision tables. A single table is presented in Fig. 1. The table represents a set of rules, having the same attributes. A rule can be read as follows:
$(A11 \in a11) \wedge \ldots (A1n \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1)$

It includes two main extensions compared to classic RBS: 1) non-atomic attribute values, used both in conditions and decisions, 2) non-monotonic reasoning support, with dynamic assert/retract operations in decision part. Every table row corresponds to a decision rule. Rows are interpreted from top the row to the bottom one. Tables can be linked in a graph-like structure. A link is followed when a rule (row) is fired.

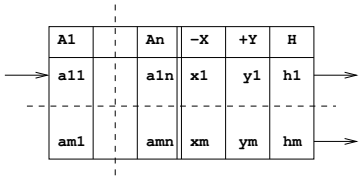


Fig. 1. A single XTT table.

On the logical level a table corresponds to a number of rules, processed in a sequence. If a rule is fired and it has a link, the inference engine processes the rule in another table. The rule is based on an *attributive language* [4]. It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee h$ where p is a literal in SAL (set attributive logic, see [4]) in a form $A_i(o) \in t$ where $o \in O$ is a object referenced in the system, and $A_i \in A$ is a selected attribute (property) of this object, $t \subseteq D_i$ is a subset of attribute domain A_i. Rules are interpreted using a unified knowledge and fact base, that can be dynamically modified during the inference process using Prolog-like assert/retract operators in rule decision.

Rules are implemented using Prolog-based representation, using terms, which is a flexible solution (see [6]). However, it requires a dedicated meta-interpreter [7].

This approach has been successfully used to model classic rule-based expert systems. Considering using XTT for general applications in the field of Software Engineering, there have been several extensions proposed regarding the base XTT model. The *Generalized Rule Programming model* or *GREP* uses the *XTT* knowledge method with certain modifications. The XTT method was aimed at forward chaining rule-based systems (RBS). In order to be applied to general programming, it is modified in several aspects. The modifications considered in GREP are *Grouped Attributes*, *Attribute-Attribute Comparison*, *Link Labeling*, *Not-Defined Operator*, *Scope Operator*, *Multiple Rule Firing*. Applying these extensions constitute *GREP* [8]. All of these extensions have been described in [8]. Here, only some of them, needed for further demonstration of hybrid operators will be shortly discussed.

Grouped Attributes provide means for putting together a given number of attributes to express relationships among them and their values. As a result a complex data structure, called a *group*, is created which is similar to constructs present in programming languages (i.e. C structures). A group is expressed as: $Group(Attribute1, Attribute2, \ldots, AttributeN)$. Attributes within a group can be referenced by their name: $Group.Attribute1$ or position within the group: $Group/1$. An application of Grouped Attributes could be expressing spatial coordinates: $Position(X, Y)$ where *Position* is the group name, X and Y are attribute names.

The *Attribute-Attribute Comparison* concept introduces a powerful mechanism to the existing XTT comparison model. In addition to comparing an attribute value against a constant (*Attribute-Value Comparison*) it allows for comparing an attribute value against another attribute value. The *Attribute-Value Comparison* can be expressed as a condition:

```
if (Attribute OPERATOR Value) then ...
```

where `OPERATOR` is a comparison operator i.e. equal, greater then, less than etc., while *Attribute-Attribute Comparison* is expressed as a condition:

```
if (Attribute1 OPERATOR Attribute2) then ...
```

where `OPERATOR` is a comparison operator or a function in a general case:

```
if (OPERATOR(Attribute1,...,AttributeN)) then ...
```

The proposed *Not-Defined* (N/D) operator checks if a value for a given attribute has been defined. It has a broad application regarding modelling basic programming structures, i.e. to make a certain rule fired if the XTT is executed for the first time.

3 Motivation for the GREP Extension

The concept of the *Generalized Rule-based Programming* (GREP) presented in [8] provides a coherent rule programming solution for general software design and

implementation. However, at the current stage GREP still lacks features needed to replace traditional programming languages. The most important problem is with limited actions/operations in the decision and precondition parts, and input/output operations, including communication with environment.

Actions and operations in the decision and condition parts are limited to using assignment and comparison operators (assert/retract actions are considered assignment operations), only simple predefined operators are allowed.

Interaction with the environment of GREP based application is provided by means of attributes. Such an interface while being sufficient for expert systems becomes insufficient for general purpose software. Particularly there is no basis for executing arbitrary code (external processes, calling a function or method) upon firing a rule. Such an execution would provide an ability to trigger operations outside the GREP application (i.e. running actuators regarding control systems). The only outcome, possible now in GREP, is setting an attribute to a given value.

Similarly conditions are not allowed to spawn any additional inference process, i.e. in order to get specific values from arbitrary code (external process, function or method; reading sensor states for control systems).

There is also an assumption that setting appropriate attribute triggers some predefined operation, however such operations are not defined by XTTs and they have to be provided by other means. Similarly comparing an attribute with some value might trigger certain operations which lead to value generation for the triggered attribute.

So it can be concluded, that in order to serve as an efficient implementation tool, GREP in its current form should be extended even more. This extension consist in the introduction of *hybrid operators*. Such operators offer an extended processing capabilities for attributes and their values. The *hybrid operators* can serve as generative, restrictive or triggering operations. These are to generate attribute values, restrict admissible attribute values, or trigger arbitrary code execution, respectively.

4 Hybrid Operators

GREP assumes that there is only one, single attribute in a column to be modified by rules or there is a single *Grouped Attribute* (GA), since the operators are capable of modifying only a single attribute values. Applying GREP to some cases (see Section 6) indicated a need for more complex operators working on multiple attributes at the same time. To introduce such operators certain changes to the current GREP are necessary. These changes constitute XTTv2 which will be subsequently referred to as XTT in this paper.

The following extensions, which provide *Hybrid Operators* functionality, are proposed: *Defined Operators* (DOT), and *Attribute Sets* (ASET).

A *Defined Operator* is an operator of the following form:

$$Operator(Attribute1, ..., AttributeN)$$

Its functionality is defined by other means and it is not covered by XTT. An operator can be implemented in almost any declarative programming language such as Prolog, procedural or object oriented as: C/C++, Java, or it can even correspond to database queries written in SQL. A Hybrid Operator is an interface between XTT and other programming languages, the targeted languages are: Prolog, Java and C. This is where the name *Hybrid* depicts that such an operator extends XTT with other programming languages and paradigms.

In general, a rule with a DOT is a regular production rule as any other XTT based one:

```
IF (Operator(Attribute1,...,AttributeN)) THEN ...
```

Since, a DOT is not limited to modify only a single attribute value, it should be indicated which attribute values are to be modified. This indication is provided by ASETs.

An ASET is a list of attributes whose values are subject to modifications in a given column. The ASET concept is similar to this of *Grouped Attributes* (GA) to some extent. The difference is that a GA defines explicitly a relationship among a number of attributes while an ASET provides means for modifying value of more than one attribute at a time. Attributes within an ASET do not have to be semantically interrelated.

An XTT table with ASET and DOT is presented in Fig. 2. There are two ASETs: attributes A, B, C and X, Y. In addition to the ASETs there are two regular attributes D and Z.

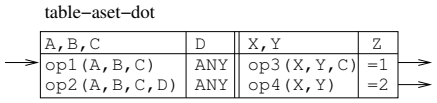


Fig. 2. Hybrid Operators: Attribute Sets (ASET) and Defined Operators (DOT)

The first column, identified by an ASET: A, B, C, indicates that these three attributes are subject to modification in this column. Similarly a column identified by X, Y indicates that X and Y are subject to modifications in this column. Following the above rules, the operator $op2()$ modifies only A, B, C while D is accessed by it, but it is not allowed to change D values. Similarly operator $op3()$ is allowed to change values of X and Y only.

Depending on where a DOT is placed, either in the condition or conclusion part, values provided by it have different scope. These rules apply to any operators, not only DOTs. If an operator is placed in the condition part, all value modifications are visible in the current XTT table only. If an operator is placed in the conclusion part, all value modifications are applied globally (within the XTT scope, see [8]), and they are visible in other XTT tables.

There are four modes of operation a DOT can be used in: *Restrictive Mode*, *Generative Mode*, *Restrictive-Generative Mode*, and *Setting Mode*. The *Restric-*

tive Mode narrows number of attribute values and it is indicated as `-`. The *Generative Mode* adds values. It is indicated as `+`. The *Restrictive-Generative Mode* adds or retracts values; indicated as `+-`. Finally, the *Setting Mode* sets attribute values, all previous values are discarded (attributes without `+` or `-` are by default in the *Setting Mode*).

An example with the above modes indicated is given in Fig. 3. An ASET in the first column $(+A, -B, C)$ indicates that A can have some new values asserted, B retracted and C set. An ASET in the third column $(X, + - Y)$ indicates that X has a new set values, while some Y values are retracted and some asserted.

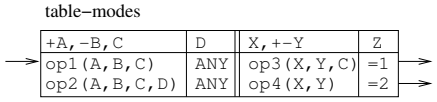


Fig. 3. Hybrid Operators: Modes

Hybrid Operators may be used in different ways. Especially three use cases, called schemas are considered, these are: *Input*, *Output*, and *Model*. These schemas depict interaction between XTT based application and its environment, i.e. external procedures (written in C, Java, Prolog etc.) accessible through DOTs. This interaction is provided on an per attribute basis. The schemas are currently not indicated in XTT tables, such an indication is subject to further research.

The *Input Schema* means that an operator reads data from environment and passes it as attribute values. The *Output Schema* is similar: an operator sends values of a given attribute to the environment. Pure interaction with XTT attributes (no input/output operations) is denoted as the *Model Schema*.

There are several restrictions regarding these schemas. The Input schema can be used in condition part, while the Output schema in conclusion part only. Model schemas can be used both in condition or conclusion parts.

Regardless of the schema: Model, Input, or Output, any operation with a DOT involved can be: *blocking* or *non-blocking*. A blocking operation means that the DOT blocks upon accessing an attribute i.e. waiting for a value to be read (Input Schema), write (Output Schema), or set (Model Schema). Such operations may be also *non-blocking*, depending on the functionality needed.

The schemas can provide semaphore-like synchronization based on attributes or event triggered inference i.e. an event unblocking a certain DOT, which spawns a chain of rules to be processed in turn.

There is a drawback regarding Hybrid Operators. It regards validation of an XTT based application. While an XTT model can be formally validated, DOTs cannot be, since they might be created with a non-declarative language (i.e.: C, Java). Some partial validation is doable if all possible DOT inputs and outputs are known in the design stage. It is a subject of further research.

5 GREP Model Integration with HOP

The XTT model itself is capable of an advanced rule-based processing. However, interactions with the environment were not clearly defined. The Hybrid Operators, introduced here, fill up this gap. An XTT based logic can be integrated with other components written in Prolog, C or Java. This integration is considered on an architectural level. It follows the Mode-View-Controller (MVC) pattern [9]. In this case the XTT, together with Prolog based Hybrid Operators, is used to build the application logic: the *model*, where-as other parts of the application are built with some classic procedural or object-oriented languages such C or Java.

The application logic interfaces with object-oriented or procedural components accessible through Hybrid Operators. These components provide means for interaction with an environment which is user interface and general input-output operations. These components also make it possible to extend the model with arbitrary object-oriented code. There are several scenarios possible regarding interactions between the model and the environment. In general they can be subdivided into two categories providing *view* and *controller* functionalities which are output and input respectively.

An input takes place upon checking conditions required to fire a rule. A condition may require input operations. A state of such a condition is determined by data from the environment. Such data could be user input, file contents, a state of an object, a result from a function etc. It is worth pointing out that the input operation could be blocking or non-blocking providing a basis for synchronization with environment. The input schema act as a *controller* regarding MVC.

The output schema takes place if a conclusion regards an output operation. In such a case, the operation regards general output (i.e. through user interface), spawning a method or function, setting a variable etc. The conclusion also carries its state, which is true or false, depending on whether the output operation succeeded or failed respectively. If the conclusion fails, the rule fails as well. The output schema acts as the *view* regarding MVC.

There are several components to integrate to provide a working system. They are presented in Fig. 4. The application's logic is given in a declarative way as the Knowledge Base using XTT. Interfaces with other systems (including Human-Computer Interaction) are provided in classical sequential manner through Hybrid Operators. There is a bridging module between the Knowledge Base and the sequential Code (C/Java language code): the *Sequential Language INterface* (SLIN). It allows communication in both directions. The Knowledge Base can be extended, new facts or rules added by a stimulus coming through SLIN from the View/Interface. There are two types of information passed this way: events generated by the HeaRT (HEkate RunTime) and knowledge generated by the Code (through Hybrid Operators).

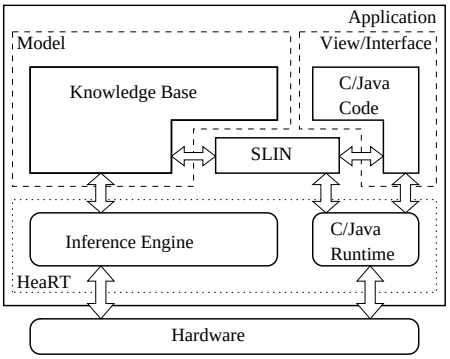


Fig. 4. System Design

6 HOP Applications Examples

Let us consider two generic programming problems examples: factorial calculation and tree browsing.

To calculate factorial of X and store it as a value of attribute Y an XTT tables given in Fig. 5 are needed. It is an iterative approach. Such a calculation can be presented in a simpler form with use of a Hybrid Operator, which provides a factorial function in other programming language, more suitable for this purpose. The XTT table which calculates a factorial of X and stores the result in Y is presented in Fig. 6. In addition to the given XTT table, the $fact()$ operator has to be provided. It can be expressed in Prolog, based on the recursive algorithm, see Fig. 7. Such an approach provides cleaner design at the XTT level.

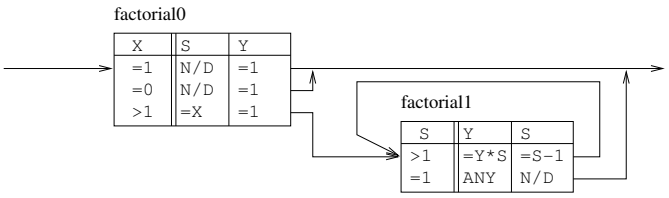


Fig. 5. XTT: factorial function

Problems like TSP or browsing a tree structure have well known and pretty straightforward solutions in Prolog. Assuming that there is a tree structure denoted as: $tree(Id, Par)$, where $tree.Id$ is a node identifier and $tree.Par$ is a parent node identifier.

To find all predecessors of a given node, an XTT table is given in Fig. 8. The XTT searches the tree represented by a grouped attribute $tree/2$. Results will be stored in a grouped attribute $out/2$ as $out(Id, Pre)$, where $out.Id$ is a node

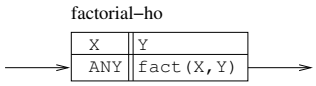


Fig. 6. XTT: factorial function, using a Hybrid Operator

```
fact(0,0).
fact(1,1).
fact(A,B):- A > 1, T is A - 1, fact(T,Z), B is Z * A.
```

Fig. 7. XTT: factorial Hybrid Operator in Prolog

identifier and $out.Pre$ is its predecessor node. The search process is narrowed to find predecessors of a node with $out.Id = 2$ only (see attribute $out.Id$ in the condition part of the XTT). It is provided by a hybrid operator $pred()$.

The hybrid operator $pred()$ is implemented in Prolog (see Fig. 9). It consists of two clauses which implement the recursive tree browsing algorithm. The first argument of $tree/3$ predicate is to pass the tree structure to browse. Since $tree/2$ in XTT is a grouped attribute it is perceived by Prolog as a structure.

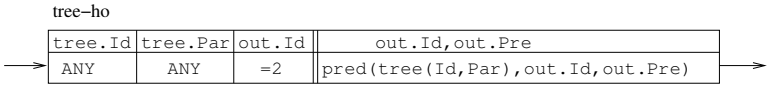


Fig. 8. XTT: Browsing a tree

```
pred(Tree,Id,Parent):-Tree, Tree=..[_,Id,Parent].
pred(Tree,Id,Parent):-Tree, Tree=..[Pred,Id,X],
                      OtherTree=..[Pred,_,_],
                      pred(OtherTree,X,Parent).
```

Fig. 9. Browsing a tree, a Hybrid Operator in Prolog

The hybrid operator $pred()$ defined in Fig. 9 works in both directions. It is suitable both for finding predecessors and successors in a tree. The direction depends on which attribute is bound. In the example (Fig. 8) the bound one is $out.Id$. Bounding $out.Pre$ changes the direction, results in search for all successors of the given as $out.Pre$ node in the tree. The tree is provided through the first argument of the hybrid operator. The operator can be used by different XTT, and works on different tree structures.

7 Concluding Remarks

The *Generalized Rule Programming* concept, extended with Hybrid Operators in Prolog becomes a powerful concept for software design. It strongly supports MVC approach: the model is provided by XTT based application extended with Prolog based Hybrid Operators; both View and Controller functionality (or in other words: communication with the environment, including input/output operations) are provided also through Hybrid Operators. These operators can be implemented in Prolog or (in general case) other procedural or object-oriented language such as Java or C.

Hybrid Operators extend forward-chaining functionality of XTT with arbitrary inference, including Prolog based backward-chaining. They also provide the basis for an integration with existing software components written in other procedural or object-oriented languages. The examples showed in this paper indicate that GREP and the Hybrid Operators can be successfully applied as a Software Engineering approach. It is worth pointing out that this approach is purely knowledge-based. It constitutes Knowledge based Software Engineering.

The results presented herein should be considered a work in progress. Future work will be focused on formulating a complete Prolog representation of GREP extended with HOPs, as well as use cases.

References

1. Negnevitsky, M.: Artificial Intelligence. A Guide to Intelligent Systems. Addison-Wesley, Harlow, England; London; New York (2002) ISBN 0-201-71159-1.
2. Liebowitz, J., ed.: The Handbook of Applied Expert Systems. CRC Press, Boca Raton (1998)
3. Jackson, P.: Introduction to Expert Systems. 3rd edn. Addison–Wesley (1999) ISBN 0-201-87686-8.
4. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
5. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science **31** (2005) 89–95
6. Nalepa, G.J., Ligęza, A.: Prolog-based analysis of tabular rule-based systems with the xtt approach. In Sutcliffe, G.C.J., Goebel, R.G., eds.: FLAIRS 2006: proceedings of the 19th international Florida Artificial Intelligence Research Society conference, AAAI Press (2006) 426–431
7. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall (1996)
8. Nalepa, G.J., Wojnicki, I.: Visual software modelling with extended rule-based model. In: ENASE 2007: International Working Conference on Evaluation of Novel Approaches to Software Engineering. (2007)
9. Burbeck, S.: Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). Technical report, Department of Computer Science, University of Illinois, Urbana-Champaign (1992)

The Kiel Curry System KiCS *

Bernd Braßel and Frank Huch
CAU Kiel, Germany
{bbr,fhu}@informatik.uni-kiel.de

Abstract. This paper presents the Kiel Curry System (KiCS) for the lazy functional logic language Curry. Its main features beyond other Curry implementations are: flexible search control by means of search trees, referentially transparent encapsulation and sharing across non-determinism.

1 Introduction

The lazy functional logic programming language Curry [9, 1] combines the functional and the logical programming paradigms as seamlessly as possible. However, existing implementations of Curry, like PAKCS [8], the Münster Curry Compiler (MCC) [10] and the Sloth system [5], all contain seams which cannot easily be fixed within these existing systems.

To obtain a really seamless implementation, we developed a completely new implementation idea and translation scheme for Curry. The basic idea of the implementation is handling non-determinism (and free variables) by means of tree data structures for the internal representation of the search space. In all existing systems this search space is directly traversed by means of a special search strategy (usually depth-first search). This makes the integration of important features like user-defined search strategies, encapsulation and sharing across non-determinism almost impossible for these systems.

Since the logical features of Curry are now treated like data structures, the remaining part is more closely related to lazy functional programming (e.g. in Haskell [11]). As a consequence, we decided to use Haskell as target language in our new compiler in contrast to Prolog, as in PAKCS and Sloth, or C in MCC. The advantage of using a lazy functional target language is that most parts of the source language can be left unchanged (expecially the deterministic ones) and the efficient machinery behind the Haskell compiler, e.g. the Glasgow Haskell Compiler (ghc), can be reused without large programming effort.

KiCS provides an interactive user interface in which arbitrary expressions over the program can be evaluated and a compiler for generating a binary executable for a `main` function of a program, similar to other existing systems. The next sections present the new key features of the Curry implementation KiCS, which allow seamless programming within Curry.

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

2 Representing Search

To represent search in Haskell, our compiler employs the concept proposed in [3]. There each non-deterministic computation yields a data structure representing the actual search space in form of a tree. The definition of this representation is independent of the search strategy employed and is captured by the following algebraic data type:

```
data SearchTree a = Fail | Value a | Or [SearchTree a]
```

Thus, a non-deterministic computation yields either the successful computation of a completely evaluated term v (i.e., a term without defined functions) represented by `Value` v, an unsuccessful computation (`Fail`), or a branching to several subcomputations represented by `Or` $[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are search trees representing the subcomputations.

The important point is that this structure is provided lazily, i.e., search trees are only evaluated to head normal form. By means of pattern matching on the search tree, a programmer can explore the structure and demand the evaluation of subtrees. Hence, it is possible to define arbitrary search strategies on the structure of search trees. Depth-first search can be defined as follows:

```
depthFirst :: SearchTree a -> [a]
depthFirst (Val v) = [v]
depthFirst Fail    = []
depthFirst (Or ts) = concatMap depthFirst ts
```

Evaluating the search tree lazily, this function evaluates the list of all values in a lazy manner, too. As an example for the definition of another search strategies, breadth-first search can be defined as follows:

```
breadthFirst :: SearchTree a -> [a]
breadthFirst st = unfoldOrs [st]
  where
    partition (Value x) y = let (vs,ors) = y in (x:vs,ors)
    partition (Or xs)   y = let (vs,ors) = y in (vs,xs++ors)
    partition Fail      y = y

    unfoldOrs [] = []
    unfoldOrs (x:xs) =
      let (vals,ors) = foldr partition ([],[]) (x:xs)
       in vals ++ unfoldOrs ors
```

This search strategy is fair with respect to the `SearchTree` data structure. However, if some computation runs infinitely without generating nodes in the `SearchTree`, then other solutions within the `SearchTree` will not be found.

Our concept also allows to formulate a *fair search*, which is also tolerant with respect to non-terminating, non-branching computations. Fair search is realized by employing the multi-threading capabilities of current Haskell implementations. Therefore the definition of fair search is primitive from the point of the

Curry system.

```
fairSearch :: SearchTree a -> IO [a]
fairSearch external
```

Search trees are obtained by *encapsulated search*. In [3] it is shown in many examples and considerations that the interactions between encapsulation and laziness is very complicated and prone to many problems. [3] also contains a wishlist for future implementations of encapsulation. KiCS is the first implementation to fully meet this wish list. Concretely, there are two methods to obtain search trees:

1. The *current* (top-level) state of search can only be accessed via an io-action `getSearchTree :: a -> IO (SearchTree a)`.
2. A conceptual copy of a given term can be obtained by the primitive operation `searchTree :: a -> SearchTree a`. This copy does not share any of the non-deterministic choices of the main branch of computation, although all deterministic computations are shared, i.e., only computed once, cf. the next subsection.

This two concepts of encapsulation avoid all known conflicts with the other features of functional logic languages.

3 Sharing across Non-Determinism

Another key feature for a seamless integration of lazy functional and logic programming is sharing across non-determinism, as the following example shows:

Example 1 (Sharing across Non-Determinism). We consider parser combinators which can elegantly make use of the non-determinism of functional logic languages to implement the different rules of a grammar. A simple set of parser combinators can be defined as follows:

```
type Parser a = String -> (String,a)

success :: a -> Parser a
success r cs = (cs,r)

symb :: Char -> Parser Char
symb c (c':cs) | c==c' = (cs,c)

(<*>)  :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2) str = case p1 str of
                   (str1,f) -> case p2 str1 of
                                 (str2,x) -> (str2,f x)

(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> p = success f <*> p
```

```
parse :: Parser a -> String -> a
parse p str = case p str of
                ("",r) -> r
```

As an example for a non-deterministic parser, we construct a parser for the inherently ambiguous, context-free language of palindromes without marked center $L = \{w\overleftarrow{w} \mid w \in \{a, b\}\}$. We restrict to this small alphabet for simplicity of the example. If parsing is possible the parser returns the word w and fails otherwise:

```
pal :: P String
pal = ((\ c str _ -> c:str) <$> (symb 'a' <*> p1 <*> symb 'a'))
    ? ((\ c str _ -> c:str) <$> (symb 'b' <*> p1 <*> symb 'b'))
    ? success ""
```

where `? :: a -> a -> a` is the built-in operator for branching, defined as

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

In all Curry implementations the parser `p1` analyses a `String` of length 100 within milliseconds. We call this time t_{parse}.

Unfortunately, this program does not scale well with respect to the time it takes to compute the elements of the list to be parsed. Let's assume that the time to compute an element within the list $[e_1, \ldots, e_{100}]$ is $t \gg t_{parse}$ and constructing the list $[e_1, \ldots, e_{100}]$ takes time $100 \cdot t$. Then one would expect the total time to compute `parse pal` $[e_1, \ldots, e_{100}]$ is $100 \cdot t + t_{parse}$. But measurements for the Curry implementation PAKCS ([8]) show that e.g., for $t = 0.131s$ it takes more than $5000 \cdot t$ to generate a solution for a palindrome $[e_1, \ldots, e_{100}]$ and $9910 \cdot t$ to perform the whole search for all solutions. We obtain similar results for all the other existing implementations of lazy functional languages.

The reason is that all these systems do not provide sharing across non-determinism. For each non-deterministic branch in the parser the elements of the remaining list are computed again and again. Only values which are evaluated before non-determinism occurs are shared over the non-deterministic branches. This behavior is not only a matter of leaking implementations within these systems. There also exists no formal definition for sharing across non-determinism yet.

The consequence of this example is that programmers have to avoid using non-determinism, if a function might later be applied to expensive computations. I.e., non-determinism has to be avoided completely. Laziness even complicates this problem: many evaluations are suspended until their value is demanded. A programmer cannot know which values are already computed and, hence, may be used within a non-deterministic computation. Thus, sadly, when looking for possibilities to improve efficiency, the programmer in a lazy functional logic language is well advised to first and foremost try to eliminate the logic features

he might have employed, which does not sound like a seamless integration of functional and logic programming.

KiCS provides sharing across non-determinism which becomes possible by handling non-determinism by means of a tree data structure. There is only one global heap in which all non-deterministic computations are performed. If the evaluation of a data structure in one non-deterministic branch is determinisitic it is automatically shared to all other branches.

4 Narrowing instead of Residuation

Implementations of functional and functional logic languages usually provide numbers as an external data type and reuse the default implementation of the underlying language (e.g. C) for the implementation of operations like `(+)`, `(-)`, `(<=)`, `(==)`. This provides a very efficient implementation of pure computations within the high-level language.

However, in the context of functional logic languages, this approach results in a major drawback: numbers cannot be guessed by means of narrowing. As a consequence, the semantic framework has to be extended, e.g., by *residuation* [6, 7]. The idea is that all (externally defined) functions on numbers suspend on free variables as long as their value is unbound. These residuated functions have to be combined with a generator which specifies all possible values of a free variable. As an example, we consider Pythagorean triples (a, b, c) with $a^2 + b^2 = c^2$. This problem can be implemented in the existing Curry implementations by means of the test and generate pattern [2]:

```
pyt | a*a+b*b=:=c*c &
      c =:= member [1..] & b =:= member [1..c] &
      a =:= member [1..c]
    = (a,b,c)
  where a,b,c free

member (y:ys)  = y ? member ys
```

First, the search tests whether a combination of a, b and c is a Pythagorean triple. Since in Curry natural numbers cannot be guessed this test is suspended by means of residuation for external functions `(+)` and `(*)`. Now the programmer has to add good generator functions which efficiently enumerate the search space.

If, like common in Curry implementations, a depth first search strategy is used, it is especially important to restrict the search space for a and b to a finite domain. In practical applications, finding good generator functions is often much more complicated than in this simple example.

Moreover, residuation sacrifices completeness and detailed knowledge of internals is required to understand why the following very similar definition produces a run-time error (suspension):

```
pyt' | a*a + b*b =:= c*c = generate a b c where a,b,c free

generate a b c | c =:= member [1..] & b =:= member [1..c] &
                 a =:= member [1..c]
  = (a,b,c)
```

On the other hand, the most beautiful standard examples for the expressive power of narrowing are defined for *Peano numbers*:

```
data Peano = O | S Peano

add O     m = m
add (S n) m = S (add m n)

mult O     _ = O
mult (S n) m = add m (mult m n)
```

These functions cannot only be used for their intended purpose. We can also invert them to define new operations. For instance, the subtraction function can be defined by means of `add`:

```
sub n m | add r m =:= n = r
  where r free
```

Note that we obtain a partial function, since there is no Peano number representation for negative numbers. By means of narrowing, a solution for the constraint `add r m =:= n` generates a binding for `r`. This binding is the result of `sub`.

For a solution of the Pythagorean triples, it is much easier to use Peano numbers instead of predefined integers. We can easily define a solution to this problem as follows:

```
pyt | (a `mult` a) `add` (b `mult` b) =:= (c `mult` c) = (a,b,c)
  where a,b,c free
```

It is not necessary to define any generator functions at all. In combination with breadth first search, all solutions are generated.

Furthermore, if the result c of the Pythagorean equation is already known (e.g., $c = 20$) and you are only interested in computing a and b, then it is even possible to compute this information with depth first search. The given equation already restricts the search space to a finite domain.

```
pyt | c =:= intToPeano 20 &
      (a `mult` a) `add` (b `mult` b) =:= (c `mult` c) = (a,b,c)
  where a,b,c free
```

We obtain the solutions: $(a, b) \in \{(0, 20), (12, 16), (16, 12), (20, 0)\}$.

Unfortunately, using Peano numbers is not appropriate for practical applications, as a simple computation using Peano numbers in PAKCS shows: computing the square of 1000 already takes 7 seconds. As a consequence, the developers of Curry [9] proposed the external type `Int` in combination with residuating, external functions for numbers.

In KiCS numbers are implemented as binary encodings of natural numbers

```
data Nat = IHi | O Nat | I Nat
```

which in a second step are extend with an algebraic sign and a zero to represent arbitrary integers.

```
data Int = Pos Nat | Zero | Neg Nat
```

To avoid redundant representations of numbers by leading zeros, the type `Nat` encodes only positive numbers. The leading one of the binary encoding (its most significant bit), terminates the `Nat` value. Applying the constructor `O` to a `Nat` duplicates is while `I` duplicates and afterwards increments it. Similarly, in pattern matching even numbers can be matched by `O` and odd numbers by `I`.

The implementation of the standard operations for numbers is presented in [4]. To get an impression how the implementation works, we present the definition of the addition for natural numbers here:

`add :: Nat -> Nat -> Nat`	
`IHi ‘add‘ m = succ m`	$1 + m = m + 1$
`O n ‘add‘ IHi = I n`	$2n + 1 = 2n + 1$
`O n ‘add‘ O m = O (n ‘add‘ m)`	$2n + 2m = 2 \cdot (n + m)$
`O n ‘add‘ I m = I (n ‘add‘ m)`	$2n + (2m + 1) = 2 \cdot (n + m) + 1$
`I n ‘add‘ IHi = O (succ n)`	$(2n + 1) + 1 = 2 \cdot (n + 1)$
`I n ‘add‘ O m = I (n ‘add‘ m)`	$(2n + 1) + 2m = 2 \cdot (n + m) + 1$
`I n ‘add‘ I m = O (succ n ‘add‘ y)`	$(2n + 1) + (2m + 1) = 2(n + 1 + m)$

where `succ` is the successor function for `Nat`. Similarly, all standard definitions for numbers (`Nat`s as well as `Int`s) can be implemented. In contrast to other Curry implementation, KiCS uses this `Int` definition and the corresponding operations (`(+)`, `(*)`, `(-)`,...) for arbitrary computations on numbers.

Using these numbers within KiCS, many search problems can be expressed as elegantly as using Peano numbers. For instance, all solutions for the Pythagorean triples problem can in KiCS (in combination with breadth first search) be computed with the following expression:

```
let a,b,c free in a*a + b*b =:= c*c &> (a,b,c)
```

For a fixed `c`, e.g. previously bound by `c =:= 20`, all solutions are also computed by depth first search.

The same implementation is used for characters in KiCS, which allows to guess small strings as well. To avoid the overhead related to encoding characters as binary numbers, internally a standard representation as Haskell character is used in the case when characters are not guessed by narrowing.

5 Performance

Although we did not invest much time in optimizing our compiler, performance is promising, as the following benchmarks show:

– naive: Naive reverse of a list with 10000 elements.
– fib: Fibonacci of 25.
– XML: Read and parse an XML document of almost 2MB.

	PAKCS	MCC	KiCS
naive:	22.50 s	4.24 s	3.89 s
fib:	3.33 s	0.06 s	0.30 s
XML:	-	3.55 s	8.75 s

Loading the large XML file is not possible in PAKCS. The system crashes, since memory does not suffice.

When comparing run-times for fib, one has to consider that KiCS uses algebraic representation of numbers. Anyhow, we are faster than PAKCS, but much slower than the MCC. Here MCC is able to optimize a lot, but numbers cannot be guessed in MCC.

A comparison for non-deterministic computations is difficult. For several examples, KiCS is much slower than PAKCS and MCC, which results from the additional overhead to construct the search trees data structures. On the other hand, in many examples, KiCS is much faster than PAKCS and MCC, because of sharing across non-determinism. We think that constructing fair examples here is almost impossible and hence we omit such examples here.

There have been many interesting real world applications written in Curry and used with the PAKCS system. We have already ported many of the external functions provided in PAKCS and, thus, these applications can also be compiled with KiCS or at lest will be compilable in the near future.

The implementation of KiCS just started and there should be a lot of room for optimizations, which we want to investigate for future work. However, we think our approach is promising and it is the first platform for implementing new features for lazy functional-logic languages, like sharing across non-determinism and encapsulated search.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.
3. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
4. B. Braßel, F. Huch, and S. Fischer. Declaring numbers. In *Proceedings of the 16th International Workshop on Funcional and (Constraint) Logic Programming (WFLP'07)*, Paris (France), July 2007.
5. Emilio Jess Gallego and Julio Mario. An overview of the sloth2005 curry system. In Michael Hanus, editor, *First Workshop on Curry and Functional Logic Programming*. ACM Press, September 2005.

6. M. Hanus. On the completeness of residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 192–206. MIT Press, 1992.
7. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
8. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2006.
9. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
10. W. Lux and H. Kuchen. An efficient abstract machine for curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer, 1999.
11. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

Narrowing for Non-Determinism with Call-Time Choice Semantics*

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. In a recent work we have proposed *let*-rewriting, a simple one-step relation close to ordinary term rewriting but able, via local bindings, to express sharing of computed values. In this way, *let*-rewriting reflects the call-time choice semantics for non-determinism adopted by modern functional logic languages, where programs are rewrite systems possibly non-confluent and non-terminating. Equivalence with *CRWL*, a well known semantic framework for functional logic programming, was also proved. In this paper we extend that work providing a notion of *let*-narrowing which is adequate for call-time choice as proved by a lifting lemma for *let*-rewriting similar to Hullot's lifting lemma for ordinary rewriting and narrowing.

1 Introduction

Programs in functional-logic languages (see [7] for a recent survey) are constructor based term rewriting systems possibly non-confluent and non-terminating, as happens in the following example:

$$\begin{array}{ll} coin \to 0 & repeat(X) \ \to \ X : repeat(X) \\ coin \to 1 & heads(X : Y : Ys) \ \to \ (X, Y) \end{array}$$

Here *coin* is a 0-ary non-deterministic function that can be evaluated to 0 or 1, and *repeat* introduces non-termination on the system. The presence of non-determinism enforces to make a decision about *call-time* (also called *singular)* or *run-time choice* (or *plural*) semantics [10, 15]. Consider for instance the expression $heads(repeat(coin))$:

- run-time choice gives $(0,0), (0,1), (1,0)$ and $(1,1)$ as possible results. Rewriting can be used for it as in the following derivation:

$$\begin{array}{l} heads(repeat(coin)) \to heads(coin : coin : \ldots) \to \\ (coin, coin) \to (0, coin) \to (0, 1) \end{array}$$

- under call-time choice we obtain only the values $(0,0)$ and $(1,1)$ (*coin* is evaluated only once and this value must be *shared*).

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03) and Promesas-CAM (S-0505/TIC/0407).

Modern functional-logic languages like Curry [8] or Toy [12] adopt call-time choice, as it seems more appropriate in practice. Although the classical theory of TRS (term rewriting systems) is the basis of some influential papers in the field, specially those related to *needed narrowing* [2], it cannot really serve as technical foundation if call-time choice is taken into account, because ordinary rewriting corresponds to run-time choice. This was a main motivation for the *CRWL*[1] framework [5, 6], that is considered an adequate semantic foundation (see [7]) for the paradigm.

From an intuitive point of view there must be a strong connection between *CRWL* and classical rewriting, but this has not received much attention in the past. Recently, in [11] we have started to investigate such a connection, by means of *let-rewriting*, that enhances ordinary rewriting with explicit *let*-bindings to express sharing, in a similar way to what was done in [13] for the λ-calculus. A discussion of the reasons for having proposed *let*-rewriting instead of using other existing formalisms like term graph-rewriting [14, 3] or specific operational semantics for FLP [1] can be found in [11].

In this paper we extend *let*-rewriting to a notion of *let-narrowing*. Our main result will be a *lifting lemma* for *let*-narrowing in the style of Hullot's one for classical narrowing [9].

We do not pretend that *let*-narrowing as will be presented here can replace advantageously existing versions of narrowing like needed narrowing [2] or natural narrowing [4], which are well established as appropriate operational procedures for functional logic programs. *Let*-narrowing is more a complementary proposal: needed or natural narrowing express refined strategies with desirable optimality properties to be preserved in practice, but they need to be patched in implementations in order to achieve sharing (otherwise they are unsound for call-time choice). *Let*-rewriting and *let*-narrowing intend to be the theoretical basis of that patch, so that sharing needs not be left anymore out of the scope of technical works dealing with rewriting-based operational aspects of functional logic languages. Things are then well prepared for recasting in the future the needed or natural strategies to the framework of *let*-rewriting and narrowing.

The rest of the paper is organized as follows. Section 2 contains a short presentation of *let*-rewriting. Section 3 tackles our main goal, extending *let*-rewriting to *let*-narrowing and proving its soundness and completeness. Finally, Section 4 contains some conclusions and future lines of research. Omitted proofs can be found at `gpd.sip.ucm.es/fraguas/papers/longWLP07.pdf`.

2 Preliminaries

2.1 Constructor-based term rewrite systems

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function)

[1] *CRWL* stands for *C*onstructor-based *Re**W*riting *L*ogic.

symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$.

We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

The set $Subst$ of *substitutions* consists of mappings $\theta : \mathcal{V} \longrightarrow Exp$, which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $ran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. Given $W \subseteq \mathcal{V}$ we write by $\theta|_W$ the restriction of θ to W, and $\theta|_{\setminus D}$ is a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. In most cases we will use c-substitutions $\theta \in CSubst$, verifying that $X\theta \in CTerm$ for all $X \in dom(\theta)$. We say that e *subsumes* e', and write $e \preceq e'$, if $e\theta = e'$ for some θ; we write $\theta \preceq \theta'$ if $X\theta \preceq X\theta'$ for all variables X and $\theta \preceq \theta'[W]$ if $X\theta \preceq X\theta'$ for all $X \in W$.

A *constructor-based term rewriting system* $\mathcal{P}$ (*CTRS*, also called *program* along this paper) is a set of c-rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. Given a program $\mathcal{P}$, its associated rewrite relation $\to_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\theta] \to_{\mathcal{P}} \mathcal{C}[r\theta]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\theta \in Subst$. Notice that in the definition of $\to_{\mathcal{P}}$ it is allowed for θ to instantiate extra variables to any expression. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$.

2.2 Rewriting with local bindings

In [11] we have proposed *let-rewriting* as an alternative rewriting relation for CTRS that uses *let*-bindings to get an explicit formulation of sharing, i.e., call-time choice semantics. Although the primary goal for this relation was to establish a closer relationship between classical rewriting and the *CRWL*-framework of [6], *let*-rewriting is interesting in its own as a simple one-step reduction mechanism for call-time choice. This relation manipulates *let-expressions*, defined as: $LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$, where $X \in \mathcal{V}$, $h \in CS \cup FS$, and $\overline{e}_1, \ldots, \overline{e}_n \in LExp$. The notation $let\ \overline{X = a}\ in\ e$ abbreviates

$let\ X_1 = a_1\ in\ \ldots\ in\ let\ X_n = a_n\ in\ e$. The notion of context is also extended to the new syntax: $\mathcal{C} ::= [] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots,\mathcal{C},\ldots)$.

Free and *bound* (or *produced*) variables of $e \in LExp$ are defined as:

$$FV(X) = \{X\};\ FV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} FV(e_i);$$
$$FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\});$$
$$BV(X) = \emptyset;\ BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i);$$
$$BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\}$$

We assume a variable convention according to which the same variable symbol does not occur free and bound within an expression. Moreover, to keep simple the management of substitutions, we assume that whenever θ is applied to an expression $e \in LExp$, the necessary renamings are done in e to ensure that $BV(e) \cap (dom(\theta) \cup ran(\theta)) = \emptyset$. With all these conditions the rules defining application of substitutions are simple while avoiding variable capture:

$$X\theta = \theta(X); \quad h(\overline{e})\theta = h(\overline{e\theta}); \quad (let\ X = e_1\ in\ e_2)\theta\ =\ (let\ X = e_1\theta\ in\ e_2\theta)$$

The *let*-rewriting relation $\rightarrow_l$ is shown in Figure 1. The rule **(Fapp)** performs a rewriting step in a proper sense, using a rule of the program. Note that only c-substitutions are allowed, to avoid copying of unevaluated expressions which would destroy sharing and call-time choice. **(Contx)** allows to select any subexpression as a redex for the derivation. The rest of the rules are syntactic manipulations of *let*-expressions. In particular **(LetIn)** transforms standard expressions by introducing a *let*-binding to express sharing. On the other hand, **(Bind)** removes a *let*-construction for a variable when its binding expression has been evaluated. **(Elim)** allows to remove a binding when the variable does not appear in the body of the construction, which means that the corresponding value is not needed for evaluation. This rule is needed because the expected normal forms are c-terms not containing *let*s. **(Flat)** is needed for flattening nested *let*s, otherwise some reductions could become wrongly blocked or forced to diverge (see [11]). Figure 2 contains a *let*-rewriting derivation for the expression *heads(repeat(coin))* using the program example of Sect. 1.

3 *Let*-narrowing

It is well known that in functional logic computations there are situations where rewriting is not enough, and must be lifted to some kind of *narrowing*, because the expression being reduced contains variables for which different bindings might produce different evaluation results. This might happen either because variables are already present in the initial expression to reduce, or due to the presence of extra variables in the program rules. In the latter case *let*-rewriting certainly works, but not in an effective way, since the parameter passing substitution 'magically' guesses the right values for those extra variables.

The standard definition of *narrowing* as a lifting of rewriting in ordinary TRS says (adapted to the notation of contexts): $\mathcal{C}[f(\overline{t})] \leadsto_\theta \mathcal{C}\theta[r\theta]$, if θ is a mgu

(Contx)	$\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $e \to_l e'$, $\mathcal{C} \in Cntxt$
(LetIn)	$h(\ldots, e, \ldots) \to_l let\ X = e\ in\ h(\ldots, X, \ldots)$ if $h \in CS \cup FS$, e takes one of the forms $e \equiv f(\overline{e'})$ with $f \in FS^n$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable
(Flat)	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ assuming that Y does not appear free in e_3
(Bind)	$let\ X = t\ in\ e \to_l e[X/t]$, if $t \in CTerm$
(Elim)	$let\ X = e_1\ in\ e_2 \to_l e_2$, if X does not appear free in e_2
(Fapp)	$f(t_1\theta, \ldots, t_n\theta) \to_l e\theta$, if $f(t_1, \ldots, t_n) \to e \in \mathcal{P}$, $\theta \in CSubst$

Fig. 1. Rules of *let*-rewriting

$heads(repeat(coin)) \to_l$	(**LetIn**)
$let\ X = repeat(coin)\ in\ heads(X) \to_l$	(**LetIn**)
$let\ X = (let\ Y = coin\ in\ repeat(Y))\ in\ heads(X) \to_l$	(**Flat**)
$let\ Y = coin\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	(**Fapp**)
$let\ Y = 0\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	(**Bind**)
$let\ X = repeat(0)\ in\ heads(X) \to_l$	(**Fapp**)
$let\ X = 0 : repeat(0)\ in\ heads(X) \to_l$	(**LetIn**)
$let\ X = (let\ Z = repeat(0)\ in\ 0 : Z)\ in\ heads(X) \to_l$	(**Flat**)
$let\ Z = repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**Fapp**)
$let\ Z = 0 : repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**LetIn**, **Flat**)
$let\ U = repeat(0)\ in\ let\ Z = 0 : U\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**Bind**), **2**
$let\ U = repeat(0)\ in\ heads(0 : 0 : U) \to_l$	(**Fapp**)
$let\ U = repeat(0)\ in\ (0, 0) \to_l$	(**Elim**)
$(0, 0)$	

Fig. 2. A let-rewriting derivation

of $f(\overline{t})$ and $f(\overline{s})$, where $f(\overline{s}) \to r$ is a fresh variant of a rule of the TRS. We note that frequently the narrowing step is not decorated with the whole unifier θ, but with its projection over the variables in the narrowed expression. The condition that the binding substitution θ is a mgu can be relaxed to accomplish with certain narrowing strategies like needed narrowing [2], which use unifiers but not necessarily most general ones.

This definition of narrowing cannot be directly translated as it is to the case of *let*-rewriting, for two important reasons. The first is not new: because of call-time choice, binding substitutions must be c-substitutions, as already happened in *let*-rewriting. The second is that produced variables (those introduced by **(LetIn)** and bound in a *let* construction) should not be narrowed, because their role is to express intermediate values that are evaluated at most once and shared, according to call-time choice. Therefore the value of produced variables should be

better obtained by evaluation of their binding expressions, and not by bindings coming from narrowing steps. Furthermore, to narrow on produced variables destroys the structure of *let*-expressions.

The following example illustrates some of the points above.

Example. Consider the following program over natural numbers (represented with constructors 0 and s):

$$\begin{array}{ll} 0+Y \to Y & even(X) \to if\ (Y{+}Y == X)\ then\ true \\ s(X)+Y \to s(X+Y) & if\ true\ then\ Y \to Y \\ 0 == 0 \to true & s(X) == s(Y) \to X == Y \\ 0 == s(Y) \to false & s(X) == 0 \to false \\ coin \to 0 & coin \to s(0) \end{array}$$

Notice that the rule for *even* has an extra variable Y. With this program, the evaluation of *even(coin)* by *let*-rewriting could start as follows:

$$\begin{array}{l} even(coin) \to_l let\ X = coin\ in\ even(X) \\ \to_l let\ X = coin\ in\ if\ Y + Y == X\ then\ true \\ \to_l^* let\ X = coin\ in\ let\ U = Y + Y\ in\ let\ V = (U == X)\ in\ if\ V\ then\ true \\ \to_l^* let\ U = Y + Y\ in\ let\ V = (U == 0)\ in\ if\ V\ then\ true \end{array}$$

Now, all function applications involve variables and therefore narrowing is required to continue the evaluation. But notice that if we perform classical narrowing in (for instance) *if V then true*, then the binding $\{V/true\}$ will be created thus obtaining *let U=Y+Y in let true=(U==0) in if true then true* which is not a legal expression of $LExp$ (because of the binding *let true =(U==0)*). Something similar would happen if narrowing is done in $U == 0$. What is harmless is to perform narrowing in $Y + Y$, giving the binding $\{Y/0\}$ and the (local to this expression) result 0. Put in its context, we obtain now:

$$\begin{array}{l} let\ U = 0\ in\ let\ V = (U == 0)\ in\ if\ V\ then\ true \\ \to_l let\ V = (0 == 0)\ in\ if\ V\ then\ true \\ \to_l let\ V = true\ in\ if\ V\ then\ true \\ \to_l if\ true\ then\ true \to_l true \end{array}$$

This example shows that *let*-narrowing *must protect produced variables* against bindings. To express this we could add to the narrowing relation a parameter containing the set of protected variables. Instead of that, we have found more convenient to consider a distinguished set $PVar \subset \mathcal{V}$ of *produced variables* $X_p, Y_p, \ldots$, to be used according to the following criteria: variables bound in a *let* expression must be of $PVar$ (therefore *let* expressions have the form *let* X_p*=e in e'*); the parameter passing c-substitution θ in the rule **(Fapp)** of *let*-rewriting replaces extra variables in the rule by c-terms not having variables of $PVar$; and rewriting (or narrowing) sequences start with initial expressions e not having free occurrences of produced variables (i.e., $FV(e) \cap PVar = \emptyset$). Furthermore we will need the following notion:

Definition 1 (Admissible substitutions). *A substitution θ is called* admissible *iff $\theta \in CSubst$ and $(dom(\theta) \cup ran(\theta)) \cap PVar = \emptyset$.*

The one-step *let*-narrowing relation $e \leadsto^{l}_{\theta} e'$ (assuming a given program $\mathcal{P}$) is defined in Fig. 3. The rules *Elim, Bind, Flat, LetIn* of *let*-rewriting are kept untouched except for the decoration with the empty substitution ϵ. The important rules are **(Contx)** and **(Narr)**. In **(Narr)**, $\theta \in CSubst$ ensures that call-time choice is respected; notice also that produced variables are non bound in the narrowing step (by *(ii)*), and that bindings for extra variables and for variables in the expression being narrowed cannot contain produced variables (by *(iii)*). Notice, however, that if θ is chosen to be a mgu (which is always possible) then the condition *(iii)* is always fulfilled. Notice also that not the whole θ is recorded in the narrowing step, but only its projection over the relevant variables, which guarantees that the annotated substitution is an admissible one. In the case of **(Contx)** notice that θ is either ϵ or is obtained by **(Narr)** applied to the inner e. By the conditions imposed over unifiers in **(Narr)**, θ does not bound any produced variable, including those in *let* expressions surrounding e, which guarantees that any piece of the form $let\ X_p = r\ in\ \ldots$ occurring in $\mathcal{C}$ becomes $let\ X_p = r\theta\ in\ \ldots$ in $\mathcal{C}\theta$ after the narrowing step.

(Contx) $\mathcal{C}[e] \leadsto^{l}_{\theta} \mathcal{C}\theta[e']$ if $e \leadsto^{l}_{\theta} e'$, $\mathcal{C} \in Cntxt$
(Narr) $f(\overline{t}) \leadsto^{l}_{\theta|_{FV(f(\overline{t}))}} r\theta$, for any fresh variant $(f(\overline{p}) \to r) \in \mathcal{P}$ and $\theta \in CSubst$ such that:
i) $f(\overline{t})\theta \equiv f(\overline{p})\theta$.
ii) $dom(\theta) \cap PVar = \emptyset$.
iii) $ran(\theta|_{\setminus FV(f(\overline{p}))}) \cap PVar = \emptyset$.
(X) $e \leadsto^{l}_{\epsilon} e'$ if $e \to_{l} e'$ using $\boldsymbol{X} \in \{Elim, Bind, Flat, LetIn\}$.

Fig. 3. Rules of *let*-narrowing

The one-step relation $\leadsto^{l}_{\theta}$ is extended in the natural way to the multiple-steps narrowing relation $\leadsto^{l^*}$, which is defined as the least reflexive relation verifying:

$$e \leadsto^{l}_{\theta_1} e_1 \leadsto^{l}_{\theta_2} \ldots e_n \leadsto^{l}_{\theta_n} e' \ \Rightarrow\ e \leadsto^{l^*}_{\theta_1 \ldots \theta_n} e'$$

We write $e \leadsto^{l^n}_{\theta} e'$ for a n-steps narrowing sequence.

3.1 Soundness and completeness of *let*-narrowing

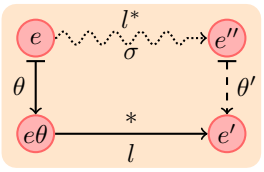
In this section we show the adequacy of *let*-narrowing wrt *let*-rewriting. We assume a fixed program $\mathcal{P}$. We start by proving *soundness*, as stated in the following result:

Theorem 1 (Soundness of *let*-narrowing). *For any $e, e' \in LExp$, $e \leadsto^{l^*}_{\theta} e'$ implies $e\theta \to^{*}_{l} e'$.*

Completeness is, as usual, more complicated to prove. The key result is the following generalization to the framework of *let*-rewriting of Hullot's *lifting lemma* [9] for classical rewriting and narrowing, stating that any rewrite

sequence for a particular instance of an expression can be generalized by a narrowing derivation.

Lemma 1 (Lifting lemma for *let*-rewriting). *Let $e, e' \in LExp$ such that $e\theta \to_l^* e'$ for an admissible θ, and let $\mathcal{W}$ be a set of variables with $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$. Then there exist a* let*-narrowing derivation $e \leadsto_\sigma^{l^*} e''$ and an admissible θ' such that $e''\theta' = e'$ and $\sigma\theta' = \theta[\mathcal{W}]$. Besides, the* let*-narrowing derivation can be chosen to use mgu's at each* **(Narr)** *step. Graphically:*



As an immediate corollary we obtain the following completeness result of *let*-narrowing for finished *let*-rewriting derivations:

Theorem 2 (Completeness of *let*-narrowing).
Let $e \in LExp, t \in CTerm$ and θ an admissible c-substitution. If $e\theta \to_l^* t$, then there exist a let*-narrowing derivation $e \leadsto_\sigma^{l^*} t'$ and an admissible θ' such that $t'\theta' = t$ and $\sigma\theta' = \theta[FV(e)]$.*

3.2 *Let*-narrowing versus narrowing for deterministic systems

The relationship between *let*-rewriting ($\to_l$) and ordinary rewriting ($\to$) is examined in [11], where $\to_l$ is proved to be sound wrt $\to$, and complete for the class of *deterministic* programs, a notion close but not equivalent to confluence (see [11] for the technical definition).

The class of deterministic programs is conjectured in [11] to be wider that that of confluent programs, and it certainly contains all inductively sequential programs (without extra variables). The following result holds (see [11]):

Theorem 3. *Let $\mathcal{P}$ be any program, $e \in Exp, t \in CTerm$. Then:*

(a) *$e \to_l^* t$ implies $e \to^* t$.*
(b) *If in addition $\mathcal{P}$ is deterministic, then the reverse implication holds.*

Joining this with the results of the previous section we can easily establish some relationships between *let*-narrowing and ordinary rewriting/narrowing, as follows (we assume here that all involved substitutions are admissible):

Theorem 4. *Let $\mathcal{P}$ be any program, $e \in Exp, \theta \in CSubst, t \in CTerm$. Then:*

(a) *$e \leadsto_\theta^{l^*} t$ implies $e\theta \to^* t$.*
(b) *If in addition $\mathcal{P}$ is deterministic, then:*
  (b_1) *If $e\theta \to^* t$, there exist $t' \in CTerm$, $\sigma, \theta' \in CSubst$ such that $e \leadsto_\sigma^{l^*} t'$, $t'\theta' = t$ and $\sigma\theta' = \theta[var(e)]$ (and therefore $t' \preceq t, \sigma \preceq \theta[var(e)]$).*
  (b_2) *If $e \leadsto_\theta^* t$, the same conclusion of (b_1) holds.*

Part *(a)* expresses soundness of $\leadsto^l$ wrt rewriting, and part *(b)* is a completeness result for $\leadsto^l$ wrt rewriting/narrowing, for the class of deterministic programs.

4 Conclusions

Our aim in this work was to progress in the effort of filling an existing gap in the functional logic programming field, where up to recently there was a lack of a simple and abstract enough one-step reduction mechanism close enough to ordinary rewriting but at the same time respecting non-strict and call-time choice semantics for possibly non-confluent and non-terminating constructor-based rewrite systems (possibly with extra variables), and trying to avoid the complexity of graph rewriting [14]. These requirements were not met by two well established branches in the foundations of the field: one is the *CRWL* approach, very adequate from the point of view of semantics but operationally based on somehow complicated narrowing calculi [6, 16] too far from the usual notion of term rewriting. The other approach focuses on operational aspects in the form of efficient narrowing strategies like needed narrowing [2] or natural narrowing [4], based on the classical theory of rewriting, sharing with it the major drawback that rewriting is an *unsound* operation for call-time choice semantics of functional logic programs. There have been other attempts of coping operationally with call-time choice [1], but relying in too low-level syntax and operational rules.

In a recent work [11] we established a technical bridge between both approaches (*CRWL*/classical rewriting) by proposing a notion of rewriting with sharing by means of local *let* bindings, in a similar way to what has been done for sharing and lambda-calculus in [13]. Most importantly, we prove there strong equivalence results between *CRWL* and *let*-rewriting.

Here we continue that work by contributing a notion of *let*-narrowing (narrowing for sharing) which we prove sound and complete with respect to *let*-rewriting. We think that *let*-narrowing is the simplest proposed notion of narrowing that is close to the usual notions of TRS and at the same time is proved adequate for call-time choice semantics. The main technical insight for let-narrowing has been the need of protecting produced (locally bound) variables against narrowing over them. We have also proved soundness of *let*-narrowing wrt ordinary rewriting and completeness for the wide class of deterministic programs, thus giving a technical support to the intuitive fact that combining sharing with narrowing does not create new answers when compared to classical narrowing, and at the same time does not lose answers in case of deterministic systems. As far as we know these results are new in the narrowing literature.

The natural extension of our work will be to add strategies to *let*-rewriting and *let*-narrowing, an issue that has been left out of this paper but is needed as foundation of effective implementations. But we think that the clear script we have followed so far (first presenting a notion of rewriting with respect to which we have been able to prove correctness and completeness of a a subsequent notion of narrowing, to which add strategies in future work) is an advantage rather than a lack of our approach.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 268–279. ACM Press, 1994.
3. R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.
4. S. Escobar, J. Meseguer, and P. Thati. Natural narrowing for general term rewriting systems. In *RTA*, pages 279–293, 2005.
5. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.
6. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
7. M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.
8. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
9. J. Hullot. Canonical forms and unification. In *Proc. 5th Conference on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.
10. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
11. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming, ACM Press*, pages 197–208, 2007.
12. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
13. J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.
14. D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.
15. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.
16. R. d. Vado-Vírseda. A demand-driven narrowing calculus with overlapping definitional trees. In *Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'03)*, pages 213–227. ACM Press, 2003.

Java type unification with wildcards

Martin Plümicke

University of Cooperative Education Stuttgart/Horb
Florianstraße 15, D–72160 Horb
m.pluemicke@ba-horb.de

Abstract. With the introduction of Java 5.0 the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<? extends Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.
In this paper we present a type unification algorithm for Java 5.0 type terms. The algorithm unifies type terms, which are in subtype relationship. For this we define Java 5.0 type terms and its subtyping relation, formally.
As Java 5.0 allows wildcards as instances of generic types, the subtyping ordering contains infinite chains. We show that the type unification is still finitary. We give a type unification algorithm, which calculates the finite set of general unifiers.

1 Introduction

With the introduction of Java 5.0 [1] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. For example the term

`Vector<? extends Vector<AbstractList<Integer>>>`

is a correct type in Java 5.0.
Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not.
This has caused us to develop a Java 5.0 type inference system which assists the programmer by calculating types automatically. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principal types.
Following the ideas of [2], we reduce the Java 5.0 *type inference problem* to a Java 5.0 *type unification problem*. The Java 5.0 *type unification problem* is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that

$\sigma(\theta_1) \leq^* \sigma(\theta_2)$, where $\leq^*$ is the Java 5.0 subtyping relation.

The type system of Java 5.0 is very similar to the type system of polymorphically order-sorted types, which is considered for the logical languages [3–6] and for the functional object–oriented language OBJ–P [7]. But in all approaches the type unification problem was not solved completely.

In [8] we have done a first step solving the type unification problem. We restricted the set of type terms, by disallowing wildcards, and presented a type unification algorithm for this approach. This algorithm led to the Java 5.0 type inference system presented in [9].
In this paper we extend our algorithm to type terms with wildcards. This means that we solve the original type unification problem and give a complete type unification algorithm. We will show, that the type unification problem is not still unitary, but finitary.
The paper is organized as follows. In the second section we formally describe the Java 5.0 type system including its inheritance hierarchy. In the third section we give an overview of the type unification problem. Then, we present the type unification algorithm and give an example. Finally, we close with a summary and an outlook.

2 Subtyping in Java 5.0

The Java 5.0 types are given as type terms over a type signature TS of class/interface names and a set of bounded type variables BTV. While the type signature describes the arities of the the class/interface names, a bound of a type variable restricts the allowed instantiated types to subtypes of the bound. For a type variable `a` bounded by the type ty we will write $\mathtt{a}|_{ty}$.

Example 1. Let the following Java 5.0 program be given:

```
class A<a> implements I<a> { ...}
class B<a> extends A<a> { ...}
class C<a extends I<b>,b> { ...}
interface I<a> { ...}
interface J<a> { ...}
class D<a extends B<a> & J<b>, b> { ...}
```

Then, the corresponding type signature TS is given as: $TS^{(\mathtt{a}|_{\mathtt{Object}})} = \{\,\mathtt{A}, \mathtt{B}, \mathtt{I}, \mathtt{J}\,\}$, $TS^{(\mathtt{a}|_{\mathtt{I<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\,\mathtt{C}\,\}$, and $TS^{(\mathtt{a}|_{\mathtt{B<a>\&J<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\,\mathtt{D}\,\}$.
As $\mathtt{A}, \mathtt{I} \in TS^{(\mathtt{a}|_{\mathtt{Object}})}$ and `Integer` is a subtype of `Object`, the terms `A<Integer>` and `I<Integer>` are Java 5.0 types. As `I<Integer>` is a subtype of itself and `A<Integer>` is also a subtype of `I<Integer>`, the terms `C<I<Integer>, Integer>` and `C<A<Integer>, Integer>` are also type terms. But as `J<Integer>` is no subtype of `I<Integer>` the term `C<J<Integer>, Integer>` is no Java 5.0 type.

For the definition of the inheritance hierarchy, the concept of Java 5.0 *simple types* and the concept of *capture conversion* is needed. For the definition of Java 5.0 simple types we refer to [1], Section 4.5, where Java 5.0 parameterized types are defined. We call them, in this paper, Java 5.0 *simple types* in contrast to the function types of methods. Java 5.0 simple types consists of the Java 5.0 types as in Example 1 presented and wildcard types like `Vector<? extends Integer>`. For the definition of the *capture conversion* we refer to [1] §5.1.10. The *capture*

conversion transforms types with wildcard type arguments to equivalent types, where the wildcards are replaced by implicit type variables. The capture conversion of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ is denoted by $CC(\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,)$.

The inheritance hierarchy consists of two different relations: The "extends relation" (denoted by $<$) is explicitely defined in Java 5.0 programs by the *extends*, and the *implements* declarations, respectively. The "subtyping relation" (cp. [1], Section 4.10) is built as the reflexive, transitive, and instantiating closure of the extends relation.
In the following we will use ${}_{?}\theta$ as an abbreviation for the type term "`? extends` θ" and ${}^{?}\theta$ as an abbreviation for the type term "`? super` θ".

Definition 1 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\, BTV\,)$). *Let TS be a type signature of a given Java 5.0 program and $<$ the corresponding extends relation. The* subtyping relation $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

- *if $\theta < \theta'$ then $\theta \leq^* \theta'$.*
- *if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\,\theta_1\,) \leq^* \sigma_2(\,\theta_2\,)$ for all substitutions σ_1, $\sigma_2 : BTV \rightarrow \mathsf{SType}_{TS}(\, BTV\,)$, where for each type variable a of θ_2 holds $\sigma_1(\,a\,) = \sigma_2(\,a\,)$ (soundness condition).*
- *$a \leq^* \theta_i$ for $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ and $1 \leqslant i \leqslant n$*
- *It holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ if for each θ_i and θ'_i, respectively, one of the following conditions is valid:*
  - *$\theta_i = {}_{?}\overline{\theta}_i$, $\theta'_i = {}_{?}\overline{\theta}'_i$ and $\overline{\theta}_i \leq^* \overline{\theta}'_i$.*
  - *$\theta_i = {}^{?}\overline{\theta}_i$, $\theta'_i = {}^{?}\overline{\theta}'_i$ and $\overline{\theta}'_i \leq^* \overline{\theta}_i$.*
  - *$\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\, BTV\,)$ and $\theta_i = \theta'_i$*
  - *$\theta'_i = {}_{?}\theta_i$*
  - *$\theta'_i = {}^{?}\theta_i$ (cp. [1] §4.5.1.1 type argument containment)*
- *Let $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ be the capture conversions of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ and $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ then holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.*

It is surprising that the condition for σ_1 and σ_2 in the second item is not $\sigma_1(\,a\,) \leq^* \sigma_2(\,a\,)$, but $\sigma_1(\,a\,) = \sigma_2(\,a\,)$. This is necessary in order to get a sound type system. This property is the reason for the introduction of wildcards in Java 5.0 (cp. [1], §5.1.10).
The next example illustrates the subtyping definition.

Example 2. Let the Java 5.0 program from Example 1 be given again. Then the following relationships hold:

- `A<a>` $\leq^*$ `I<a>`, as `A<a>` $<$ `I<a>`
- `A<Integer>` $\leq^*$ `I<Integer>`, where $\sigma_1 = [a \mapsto$ `Integer`$] = \sigma_2$
- `A<Integer>` $\leq^*$ `I<? extends Object>`, as `Integer` $\leq^*$ `Object`
- `A<Object>` $\leq^*$ `I<? super Integer>`, as `Integer` $\leq^*$ `Object`

There are elements of the extends relation, where the sub-terms of a type term are not variables. As elements like this must be handled especially during the unification (*adapt* rules, Fig. 1), we declare a further ordering on the set of type terms which we call the *finite closure* of the extends relation.

Definition 2 (Finite closure of $<$). *The* finite closure $\mathbf{FC}(<)$ *is the reflexive and transitive closure of pairs in the subtyping relation* $\leq^*$ *with* $C(a_1 \ldots a_n) \leq^* D(\theta_1, \ldots, \theta_m)$*, where the* a_i *are type variables and the* θ_i *are type terms.*
If a set of bounded type variables BTV is given, the finite closure $\mathbf{FC}(<)$ *is extended to* $\mathbf{FC}(<, BTV)$*, by* $a|_\theta \leq^* a|_\theta$ *for* $a|_\theta \in BTV$*.*

Lemma 1. *The* finite closure $\mathbf{FC}(<)$ *is finite.*

Now we give a further example to illustrate the definition of the subtyping relation and the finite closure.

Example 3. Let the following Java 5.0 program be given.

```
abstract class AbstractList<a> implements List<a> {..}
class Vector<a> extends AbstractList<a> {..}
class Matrix<a> extends Vector<Vector<a>> {..}
```

Following the soundness condition of the Java 5.0 type system we get

$$\texttt{Vector<Vector<a>>} \not\leq^* \texttt{Vector<List<a>>},$$

but

$$\texttt{Vector<Vector<a>>} \leq^* \texttt{Vector<? extends List<a>>}.$$

The finite closure $\mathbf{FC}(<)$ is given as the reflexive and transitive closure of

$$\begin{aligned}\{\ &\texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}\\ &\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>} \leq^* \texttt{AbstractList<Vector<a>>}\\ &\leq^* \texttt{List<Vector<a>>}\ \}.\end{aligned}$$

3 Type Unification

In this section we consider the type unification problem of Java 5.0 type terms. The type unification problem is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

The algorithm solving the type unification problem is an important basis of the Java 5.0 type inference algorithm.

3.1 Overview

First we give an overview of approaches considering the type unification problem. on polymorphically order-sorted types. In [3] the type unification problem is mentioned as an open problem. For the logical language TEL in [3] an incomplete type inference algorithm is given. The incompleteness is caused by the fact, that subtype relationships of polymorphic types which have different arities (e.g. `List(a)` $<$ `myLi(a,b)`) are allowed. This leads to the property that there are infinite chains in the type term ordering. Nevertheless, the type unification fails

in some cases without infinite chains, although there is a unifier. Let for example $\texttt{nat} < \texttt{int}$, and the set of unequations $\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{int} \lessdot a \,\}$ be given, then $\{\, \texttt{a} \mapsto \texttt{nat} \,\}$ is determined, such that $\{\, \texttt{int} \lessdot \texttt{nat} \,\}$ fails, although $\{\, \texttt{a} \mapsto \texttt{int} \,\}$ is a unifier. For $\{\, \texttt{int} \lessdot \texttt{a}, \texttt{nat} \lessdot a \,\}$ the algorithm determines the correct unifier $\{\, \texttt{a} \mapsto \texttt{int} \,\}$.
In the typed logic programs of [5] subtype relationships of polymorphic types are allowed only between type constructors of the same arity. In this approach a *most general type unifier (mgtu)* is defined as an upper bound of different principal type unifiers. In general there are no upper bounds of two given type terms in the type term ordering, which means that there are in general no mgtu in the sense of [5]. For example for $\texttt{nat} < \texttt{int}$, $\texttt{neg} < \texttt{int}$, and the set of unequations $\{\, \texttt{nat} \lessdot \texttt{a},\ \texttt{neg} \lessdot \texttt{a} \,\}$ the mgtu $\{\, \texttt{a} \mapsto \texttt{int} \,\}$ is determined. If the type term ordering is extended by $\texttt{int} < \texttt{index}$ and $\texttt{int} < \texttt{expr}$, then there are three unifiers $\{\, \texttt{a} \mapsto \texttt{int} \,\}$, $\{\, \texttt{a} \mapsto \texttt{index} \,\}$, and $\{\, \texttt{a} \mapsto \texttt{expr} \,\}$, but none of them is a mgtu in the sense of [5].
The type system of PROTOS-L [6] was derived from TEL by disallowing any explicite subtype relationships between polymorphic type constructors.
In [6] a complete type unification algorithm is given, which can be extended to the type system of [5]. They solved the type unification problem for type term orderings following the restrictions of PROTOS-L respectively the restrictions of [5]. Additionally, the result of this paper is, that the type unification problem is not unitary, but finitary. This means in general that there is more than one general type unifier. For the above example the algorithm determines where $\texttt{nat} < \texttt{int}$, $\texttt{neg} < \texttt{int}$, $\texttt{int} < \texttt{index}$, and $\texttt{int} < \texttt{expr}$ and the set of unequations $\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a} \,\}$ is given, the three general unifiers $\{\, \texttt{a} \mapsto \texttt{int} \,\}$, $\{\, \texttt{a} \mapsto \texttt{index} \,\}$, and $\{\, \texttt{a} \mapsto \texttt{expr} \,\}$.
Finally, in [8] we disallowed wildcards, which means that there is no subtyping in the arguments of the type term (soundness condition, Def. 1), but subtype relationship of type constructors with different arities is allowed. For type term orderings following this restriction we presented a type unification algorithm and proved that the type unification problem is also finitary. For example for $\texttt{myLi<b, a>} < \texttt{List<a>}$ and the set of unequations $\{\, \texttt{myLi<Integer, a>} \lessdot \texttt{List<Boolean>} \,\}$ the general unifier $\{\, \texttt{a} \mapsto \texttt{Boolean} \,\}$ is determined. For $\{\, \texttt{myLi<Integer, Integer>} \lessdot \texttt{List<Number>} \,\}$ the algorithm fails, as `Integer` is indeed a subtype of `Number`, but subtyping in the arguments is prohibited.

The type systems of TEL, respectively the other logical languages, and of Java 5.0 are very similar. The Java 5.0 type system has the same properties considering type unification, if it is restricted to simple types without parameter bounds (but including the wildcard constructions). The only difference is, that in TEL the number of arguments of a supertype type can be greater, whereas in Java 5.0 the number of arguments of a subtype can be greater. This means that infinite chains have a lower bound in TEL and an upper bound in Java 5.0. Let us consider the following example: In TEL for $\texttt{List(a)} \leq \texttt{myLi(a,b)}$ holds:

$$\texttt{List(a)} \leq \texttt{myLi(a,List(a))} \leq \texttt{myLi(a,myLi(a,List(a)))} \leq \ldots$$

In contrast in Java 5.0 for $\texttt{myLi<b, a>} < \texttt{List<a>}$ holds:

$\ldots \leq^*$ `myLi<?myLi<?List<a>, a>, a>` $\leq^*$ `myLi<?List<a>, a>` $\leq^*$ `List<a>`

The open type unification problem of [3] is caused by these infinite chains. We will now present a solution for the open problem.

Our type unification algorithm bases on the algorithm by A. Martelli and U. Montanari [10] solving the original untyped unification problem. The main difference is, that in the original unification an unifier σ is demanded, such that $\sigma(\theta_1) = \sigma(\theta_2)$, whereas in our case an unifier σ is demanded, such that $\sigma(\theta_1) \leq^* \sigma(\theta_2)$. This means, that in the original case a result of the algorithm $a = \theta$ leads to one unifier $[a \mapsto \theta]$. In contrast to that, a result $a \leq^* \theta$ leads to a set of unifiers $\{ [a \mapsto \overline{\theta}] \mid \overline{\theta} \leq^* \theta \}$ in our algorithm.

3.2 Type unification algorithm

In the following we denote $\theta \lessdot \theta'$ for two type terms, which should be type unified. During the unification algorithm $\lessdot$ is replaced by $\lessdot_?$ and $\doteq$, respectively. $\theta \lessdot_? \theta'$ means that the two sub-terms θ and θ' of type terms should be unified, such that $\sigma(\theta)$ is a subtype of $\sigma(\theta')$. $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

In the following, the type unification algorithm is shown.

(adapt)
$$\frac{Eq \cup \{ D\langle\theta_1, \ldots, \theta_n\rangle \lessdot D'\langle\theta'_1, \ldots, \theta'_m\rangle \}}{Eq \cup \{ D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot D'\langle\theta'_1, \ldots, \theta'_m\rangle \}}$$

where there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

- $(D\langle a_1, \ldots, a_n\rangle \leq^* D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle) \in \mathbf{FC}(<)$

(adaptExt)
$$\frac{Eq \cup \{ D\langle\theta_1, \ldots, \theta_n\rangle \lessdot_? {}_?D'\langle\theta'_1, \ldots, \theta'_m\rangle \}}{Eq \cup \{ D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot_? {}_?D'\langle\theta'_1, \ldots, \theta'_m\rangle \}}$$

where $D \in \Theta^{(n)}$ or $D = {}_?\overline{X}$ with $\overline{X} \in \Theta^{(n)}$ and
there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

- ${}_?D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle) \in \mathbf{grArg}(D\langle a_1, \ldots, a_n\rangle)$

(adaptSup)
$$\frac{Eq \cup \{ D'\langle\theta'_1, \ldots, \theta'_m\rangle \lessdot_? {}^?D\langle\theta_1, \ldots, \theta_n\rangle \}}{Eq \cup \{ D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot_? {}_?D'\langle\theta'_1, \ldots, \theta'_m\rangle \}}$$

where $D' \in \Theta^{(n)}$ or $D' = {}^?\overline{X}$ with $\overline{X} \in \Theta^{(n)}$ and
there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

- ${}^?D\langle a_1, \ldots, a_n\rangle \in \mathbf{grArg}(D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle)$

Fig. 1. Java 5.0 type unification adapt rules

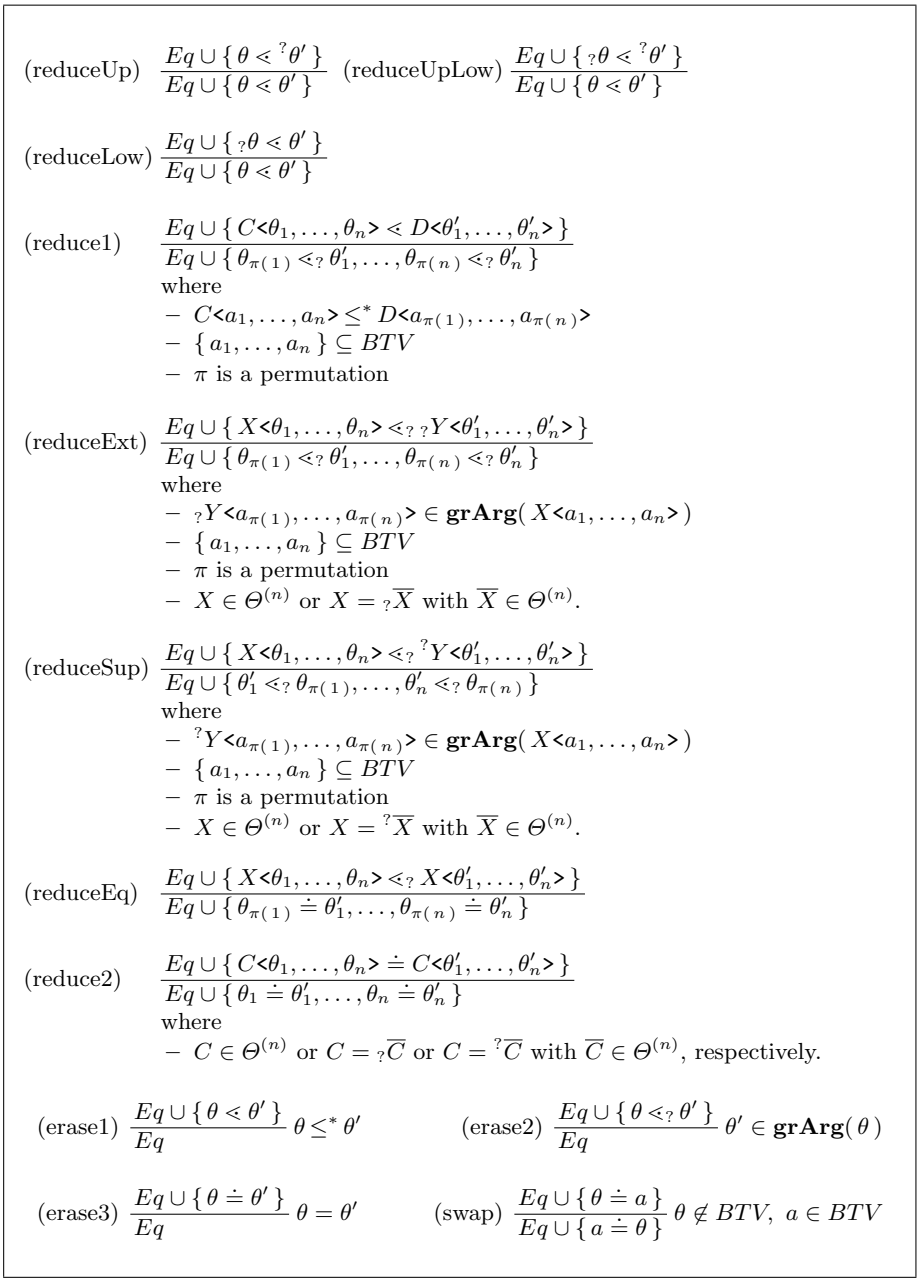


Fig. 2. Java 5.0 type unification rules with wildcards

The type unification algorithm is given as follows

Input: Set of equations $Eq = \{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$
Precondition: $\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\, BTV \,)$ for $1 \leqslant i \leqslant n$.
Output: Set of all general type unifiers $Uni = \{\, \sigma_1, \ldots, \sigma_m \,\}$
Postcondition: For all $1 \leqslant i \leqslant n$ and for all $1 \leqslant j \leqslant m$ holds $(\sigma_j(\, \theta_i \,) \leq^* \sigma_j(\, \theta'_i \,))$.

The algorithm itself is given in seven steps:

1. Repeated application of the *adapt* rules (fig. 1), the *reduce* rules, the *erase* rules and the *swap* rule (fig. 2) to all elements of Eq. The end configuration of Eq is reached if for each element no rule is applicable.
2. $Eq'_1 =$ *Subset of pairs, where both type terms are type variables*
3. $Eq'_2 = Eq \backslash Eq'_1$
4. Eq'_{set}

$$
\begin{aligned}
= \{\, Eq'_1 \,\} \times (\; & \bigotimes_{(a \lessdot \theta') \in Eq'_2} \{\, ([a \doteq \theta] \cup \sigma) \mid (\overline{\theta} \leq^* C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}) \in \mathbf{FC}(\, < \,), \\
& \qquad \overline{\theta}' \in \{\, C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \mid \theta'_i \in \mathbf{grArg}(\, \theta_i \,), 1 \leqslant i \leqslant n \,\} \\
& \qquad \sigma \in \mathsf{Unify}(\, \overline{\theta}', \theta' \,), \\
& \qquad \theta \in \mathbf{smaller}(\, \sigma(\, \overline{\theta} \,) \,) \,\}) \\
\times (\; & \bigotimes_{(a \lessdot_? {}_?\theta') \in Eq'_2} \{\, ([a \doteq \theta] \cup \sigma) \mid (\overline{\theta} \leq^* C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}) \in \mathbf{FC}(\, < \,), \\
& \qquad \overline{\theta}' \in \{\, C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \mid \theta'_i \in \mathbf{grArg}(\, \theta_i \,), 1 \leqslant i \leqslant n \,\} \\
& \qquad \sigma \in \mathsf{Unify}(\, \overline{\theta}', \theta' \,), \\
& \qquad \theta \in \mathbf{smArg}(\, \sigma(\, {}_?\overline{\theta} \,) \,) \,\}) \\
\times (\; & \bigotimes_{(a \lessdot_? {}^?\theta') \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{smArg}(\, {}^?\theta' \,) \,\}) \\
\times (\; & \bigotimes_{(a \lessdot_? \theta') \in Eq'_2} \{\, [a \doteq \theta'] \,\}) \\
\times (\; & \bigotimes_{(\theta \lessdot a) \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{greater}(\, \theta \,) \,\}) \\
\times (\; & \bigotimes_{({}_?\theta \lessdot_? a) \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\, {}_?\theta \,) \,\}) \\
\times (\; & \bigotimes_{({}^?\theta \lessdot_? a) \in Eq'_2} \{\, ([a \doteq {}^?\theta'] \cup \sigma) \mid (\overline{\theta} \leq^* C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}) \in \mathbf{FC}(\, < \,), \\
& \qquad \overline{\theta}' \in \{\, C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \mid \theta'_i \in \mathbf{grArg}(\, \theta_i \,), 1 \leqslant i \leqslant n \,\} \\
& \qquad \sigma \in \mathsf{Unify}(\, \overline{\theta}', \theta \,), \\
& \qquad \theta' \in \mathbf{smaller}(\, \sigma(\, \overline{\theta} \,) \,) \,\}) \\
\times (\; & \bigotimes_{(\theta \lessdot_? a) \in Eq'_2} \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\, \theta \,) \,\}) \\
\times \{\, & [a \doteq \theta \mid (a \doteq \theta) \in Eq'_2] \,\}
\end{aligned}
$$

5. Application of the following *subst* rule

$$(\text{subst}) \; \frac{Eq' \cup \{\, a \doteq \theta \,\}}{Eq'[a \mapsto \theta] \cup \{\, a \doteq \theta \,\}} \; a \text{ occurs in } Eq' \text{ but not in } \theta$$

for each $a \doteq \theta$ in each element of $Eq' \in Eq'_{set}$.

6. (a) Foreach $Eq' \in Eq'_{set}$ which has changed in the last step start again with the first step.
   (b) Build the union Eq''_{set} of all results of (a) and all $Eq' \in Eq'_{set}$ which has not changed in the last step.
7. $Uni = \{\, \sigma \mid Eq'' \in Eq''_{set}, Eq''$ is in solved form, $\sigma = \{\, a \mapsto \theta \mid (a \doteq \theta) \in Eq'' \,\} \,\}$

In the algorithm the unbounded wildcard "`?`" is denoted as the equivalent bounded wildcard "`? extends Object`".
Furthermore, there are functions **greater** and **grArg**, respectively **smaller** and **smArg**. All functions determine supertypes, respectively subtypes by pattern matching with the elements of the finite closure. The functions **greater** and **smaller** determine the supertypes respectively subtypes of simple types, while **grArg** and **smArg** determine the supertypes respectively the subtypes of sub-terms, which are allowed as arguments in type terms.
The function Unify is the ordinary unification.

Now we will explain the rules in Fig. 1 and 2.

adapt rules: The *adapt* rules adapt type term pairs, which are built by class declarations like

`class` C`<`$a_1, \ldots, a_n$`>` `extends` D`<`D_1`<`$\ldots$`>`$, \ldots,$ D_m`<`$\ldots$`>>`.

The smaller type is replaced by a type term, which has the same outermost type name as the greater type. Its sub-terms are determined by the finite closure. The instantiations are maintained.
The *adaptExt* and *adaptSup* rule are the corresponding rules to the *adapt* rule for sub-terms of type terms.

reduce rules: The rules *reduceUp*, *reduceUpLow*, and *reduceLow* correspond to the extension of the subtyping ordering on extended simple types [11] (wildcard sub-terms of type terms).
The *reduce1* rule follows from the construction of the subtyping relation $\leq^*$, where from $C(\, a_1, \ldots, a_n \,) \;<\; D(\, a_{\pi(1)}, \ldots, a_{\pi(n)} \,)$ follows $C(\, \theta_1, \ldots, \theta_n \,) \leq^* D(\, \theta'_1, \ldots, \theta'_n \,)$ if and only if $\theta'_i \in \mathbf{grArg}(\, \theta_{\pi(i)} \,)$ for $1 \leqslant i \leqslant n$.
The *reduceExt* and the *reduceSup* rules are the corresponding rules to the *reduce1* rule for sub-terms of type terms.
The *reduceEq* and the *reduce2* rule ensures, that sub-terms must be equal, if there are no wildcards (soundness condition of the Java 5.0 type system).

erase rules: The *erase* rules erase type term pairs, which are in the respective relationship.

swap rule: The *swap* rule swaps type term pairs, such that type variables are mapped to type terms, not vice versa.

Now we give an example for the type unification algorithm.

Example 4. In this example we use the standard Java 5.0 types `Number`, `Integer`, `Stack`, `Vector`, `AbstractList`, and `List`. It holds `Integer` < `Number` and `Stack<a>` < `Vector<a>` < `AbstractList<a>` < `List<a>`.

As a start configuration we use

$$\{\,(\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_{?}\texttt{Number>}),\ (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>})\,\}.$$

In the first step the reduce1 rule is applied twice:

$$\{\,\texttt{a} \lessdot_{?} {}_{?}\texttt{Number},\ \texttt{Integer} \lessdot_{?} \texttt{a}\,\}$$

With the second and the third step we receive in step four:

$$\begin{aligned}\{\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\}\,\}\end{aligned}$$

In the fifth step the rule *subst* is applied:

$$\begin{aligned}\{\ &\{\,\texttt{Integer} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \underline{\{\,{}_{?}\texttt{Number} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\}},\\ &\{\,{}_{?}\texttt{Integer} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,{}^{?}\texttt{Integer} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,{}_{?}\texttt{Number} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,{}_{?}\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}, \{\,{}^{?}\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\{\,\texttt{Integer} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,{}_{?}\texttt{Number} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\underline{\{\,{}_{?}\texttt{Integer} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}}, \{\,{}^{?}\texttt{Integer} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\},\\ &\underline{\{\,\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}}, \{\,{}_{?}\texttt{Number} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\,\},\\ &\{\,{}_{?}\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\,\}\{\,{}^{?}\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\,\}\,\}\end{aligned}$$

The underlined sets of type term pairs lead to unifiers.
Now we have to continue with the first step (step 6(a)). With the application of the *erase3* rule and step 7, we get:

$$\{\,\{\,a \mapsto {}_{?}\texttt{Number}\,\}, \{\,a \mapsto {}_{?}\texttt{Integer}\,\}, \{\,a \mapsto \texttt{Integer}\,\}\,\}.$$

The following example shows, how the algorithm works for subtype relationships with different numbers of arguments and wildcards, which causes infinite chains (cp. Section 3.1).

Example 5. Let $\texttt{myLi<b, a>} < \texttt{List<a>}$ and the start configuration $\{\,\texttt{List<x>} \lessdot \texttt{List<}{}_{?}\texttt{List<Integer>>}\,\}$ be given. In the first step the reduce 1 rule is applied: $\{\,\texttt{x} \lessdot_{?} {}_{?}\texttt{List<Integer>}\,\}$. With the second step we get the result:

$$\begin{aligned}\{\ &\{\,\texttt{x} \mapsto \texttt{List<Integer>}\,\}, \{\,\texttt{x} \mapsto {}_{?}\texttt{List<Integer>}\,\},\\ &\{\,\texttt{x} \mapsto \texttt{myLi<b, Integer>}\,\}, \{\,\texttt{x} \mapsto {}_{?}\texttt{myLi<b, Integer>}\,\}\,\}.\end{aligned}$$

All other inifite numbers of unifiers are instances of these general unifiers.

The following theorem shows, that the type unification problem is solved by the type unification algorithm.

Theorem 1. *The type unification algorithm determines exactly all general type unifiers for a given set of type term pairs. This means that the algorithm is sound and complete.*

Now we give a sketch of the proof.

Proof. We do the proof in two steps. First we prove the *soundness* and then the *completeness*.

Soundness For the proof of the soundness, we take a substitution which is a result of the algorithm. Then we show that this substitution is a general unifier of the algorithm input. We do this by proving for each transformation, that if the substitution is a general unifier of the result of the transformation then the substitution is also a general unifier of the input of the transformation.

Completeness For the completeness we prove that if there is a general unifier of a set of type term pairs then this general unifier is also determined by the algorithm. We prove this, by showing for each transformation of the algorithm, that if a substitution is the general unifier of a set of type terms before the transformation is done, then the substitution is also a general unifier of at least one set of type term pairs after the transformation.

Step One: For step one for each rule of Fig. 1 and of Fig. 2 is showed: If σ is a general unifier before the application then σ is still a general unifier after the application.

Step two and three: There is nothing to prove, as the type term pairs are only seperated in different sets.

Step four: There are different cases, which have to be considered. As the following hold, there are always at least one set of type term pairs after the application, where σ is a general unifier after the application, if it has been a general unifier before the application: For σ with $\sigma(\, a \,) \leq^* \sigma(\, \theta' \,)$, there is a $\theta \leq^* \theta'$, where $\sigma(\, a \,) = \sigma(\, \theta \,)$ respectively for σ with $\sigma(\, \theta \,) \leq^* \sigma(\, a \,)$, there is a $\theta \leq^* \theta'$, where $\sigma(\, a \,) = \sigma(\, \theta' \,)$.

Step five: As for a general unifier σ of $\{\, a \doteq \theta \,\}$ holds $\sigma(\, a[a \mapsto \theta] \,) = \sigma(\, \theta \,)$, the subst rule preserves the general unifiers.

Step six and seven: As the sets of type terms are unchanged the general unifiers are preserved.

As step four of the type unification algorithm is the only possibility, where the number of unifiers are multiplied, we can, according to lemma 1 and theorem 1, conclude as follows.

Corollary 1 (Finitary). *The type unification of Java 5.0 type terms with wildcards is finitary.*

Corollary 2 (Termination). *The type unification algorithm terminates.*

Corollary 1 means that the open problem of [3] is solved by our type unification algorithm.

4 Conclusion and Outlook

In this paper we presented an unification algorithm, which solves the type unification problem of Java 5.0 type terms with wildcards. Although the Java 5.0 subtyping ordering contains infinite chains, we showed that the type unification is finitary. This means that we solved the open problem from [3].
The Java 5.0 type unification is the base of the Java 5.0 type inference, as the usual unification is the base of type inference in functional programming languages.
We will extend our Java 5.0 type inference implementation [9] without wildcards by wildcards, which means that we have to substitute the Java 5.0 type unification algorithm without wildcards [8] by the new unification algorithm presented in this paper.

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. 3rd edn. The Java series. Addison-Wesley (2005)
2. Damas, L., Milner, R.: Principal type-schemes for functional programs. Proc. 9th Symposium on Principles of Programming Languages (1982)
3. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (May 1989)
4. Hanus, M.: Parametric order-sorted types in logic programming. Proc. TAPSOFT 1991 **LNCS**(394) (1991) 181–200
5. Hill, P.M., Topor, R.W.: A Semantics for Typed Logic Programs. In Pfenning, F., ed.: Types in Logic Programming. MIT Press (1992) 1–62
6. Beierle, C.: Type inferencing for polymorphic order-sorted logic programs. In: International Conference on Logic Programming. (1995) 765–779
7. Plümicke, M.: OBJ–P The Polymorphic Extension of OBJ–3. PhD thesis, University of Tuebingen, WSI-99-4 (1999)
8. Plümicke, M.: Type unification in Generic–Java. In Kohlhase, M., ed.: Proceedings of 18th International Workshop on Unification (UNIF'04). (July 2004)
9. Plümicke, M., Bäuerle, J.: Typeless Programming in Java 5.0. In Gitzel, R., Aleksey, M., Schader, M., Krintz, C., eds.: 4th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, Mannheim University Press (August 2006) 175–181
10. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems **4** (1982) 258–282
11. Plümicke, M.: Formalization of the Java 5.0 type system. In: 24. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Bad Honnef, Christian–Albrechts–Universität, Kiel (2. - 4. Mai 2007) (to appear).

System Demonstrations

A Solver-Independent Platform for Modeling Constrained Objects Involving Discrete and Continuous Domains

Ricardo Soto[1,2] and Laurent Granvilliers[1]

[1] LINA, CNRS, Université de Nantes, France
[2] Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso, Chile
{ricardo.soto,laurent.granvilliers}@univ-nantes.fr

Abstract. The constrained object modeling paradigm allows to represent systems as hierarchies of objects under constraints. This paper focuses on the modeling of constrained objects involving discrete and continuous domains. To this end, a new solver-independent modeling language called COMMA will be introduced. The mapping process from this language into the RealPaver solver will be explained.

1 Introduction

Complex structures as circuits, engines and molecules are in general entities composed by many pieces. These pieces have features and a number of rules or laws that define the structure assembly of the system. Modeling these structures is not quite natural using either a pure logic or a pure declarative constraint language. It seems more appropriate to adopt a compositional approach where the pieces of the system are represented by objects, the features of the pieces are represented by attributes and the rules are represented by constraints. This approach is the basis of the constrained object modeling paradigm.

Although this approach seems to be a good candidate for modeling this class of problems, it has received little attention from the Constraint Programming (CP) community [6]. For instance, currently there is no modeling language available based on this paradigm.

In this paper, we introduce the new modeling language COMMA [1], in order to extend previous work done under this paradigm [3, 10, 9, 7] by providing important features such as a richer constraint language (closer to OPL [13]), solver independence, extensibility, and the capability of modeling and solving constrained objects involving discrete and continuous domains.

The first implementation of COMMA is written in Java (20000 lines) where models can be translated to three different solvers: ECLiPSe [14], Gecode/J [2], and RealPaver [8].

This paper is organized as follows: Section 2 gives a summary of the features of COMMA. Section 3 explains the mapping process from COMMA models to RealPaver models. Conclusions and future work are given in Section 4.

2 COMMA Features

The main features of the language are described in this section, namely the core of the language, important characteristics from the user point of view, and the system architecture for parsing.

- *Objects + Constraints* COMMA is built from a combination of a constraint language with an object-oriented language. The object part is an abstraction of the Java programming style. Classes and objects can be defined using a simple notation. The constraint language provides classical constructs like basic data types, data structures, conditional statements, and iteration statements.

- *Simplicity* In COMMA we try to avoid the encoding concerns that make models complex and difficult to state and understand. For instance, there is no need for object constructors to state a class, direct variable assignment can be done into the class. Other details such as object visibility (protected, private, public) have been omitted. We believe that these details are suitable for programming (such as in Gecode [2] or ILOG SOLVER [11]), but not for modeling..

- *Hierarchy* The hierarchical class diagram is a good basis for managing large scale constraint problems especially from applications fields such as operation research, engineering and design. Classes, data files, and extension files can be gathered to build components.

- *Reuse* Abstraction, constraint encapsulation, composition, and inheritance, are main features of the language supporting reuse of existing components.

- *Extensibility* The constraint language is extensible [12]. New functions and relations can be added to existent domains, extending the syntax of the constraint language. This capability makes the architecture adaptable to further upgrades of the solvers.

- *Solver independent modeling* Solver independence is considered as very important in modern modeling languages [5, 4]. This architecture gives the possibility to plug-in new solvers and to process the same model with different solvers, which is useful to learn which solver is the best for it.

- *Compiler* The compiling system is composed by three independent compilers. One for the COMMA language, one for the data and another for the extension files. The COMMA compiler is composed of one parser per constraint domain (Integer, Real, Boolean and Objects), one parser for constraints involving more than one domain (Mixed parser) and one base parser for the rest of the language (classes, import and control statements). This approach allows one to easy maintain the parsing engine, not to recompile the base language by hand. It requires specific rules to handle ambiguity and typing problems [12].

3 COMMA Overview

Let us now show some of the main features of COMMA using an example involving discrete and continuous domains. Consider the task of installing two antennas whose coverage ranges are represented by circles. The position of antennas is supposed to be discrete. The coverage range of antennas is a variable with a bounded continuous domain. An intersection is required between circles to assure a minimum coverage range. This problem can be represented as a geometrical constraint system where the center of each circle corresponds to the correct antenna location (see Figure 1).

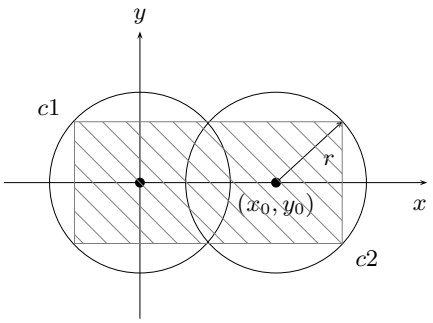


Fig. 1. Diagram of the antennas problem

Figure 2 shows the COMMA model for this problem. Two classes are defined. Class `antennas` has two constrained objects `circle` and five constraints[3] enforced over the `circle` objects. The constraints `c1.x = c2.x` and `c1.y = c2.y` allow one to control the intersection between circles. The constraint `c1.y0 = c2.y0` places the center of the circles in the same value of the `y` axis. An equal radius value for each circle is forced by `c1.r = c2.r`. The last constraint sets a distance of `1.5*radius` between the x_0 points of the circles. For the second class called `circle`, `x` and `y` represent the points of the circle with center (`x0`,`y0`). The radius is determined by the attribute `r` and the equation of the circle is represented by the given constraint. All the attributes have been restricted to specific domains to assure the coverage range given by the intersection of the antennas.

Constrained objects are translated to the solver source file by eliminating the hierarchy of the COMMA file. This process is done by expanding each constrained object declared (`c1` and `c2` in the example) adding its attributes and constraints in the flat file. The name of each attribute has a prefix corresponding to the

[3] Due to extensibility requirements of the compiler, constraints must be typed. A detailed presentation of the compiler can be found in [1].

```
class antennas {
  circle c1;
  circle c2;
  constraint intersection {
    real c1.x  = c2.x;
    real c1.y  = c2.y;
    int  c1.y0 = c2.y0;
    real c1.r  = c2.r;
    real c1.x0 + 1.5*c1.r = c2.x0;
  }
}
```

```
class circle {
  real x in [-3,3];
  real y in [-3,3];
  real r in [1.5, 2.5];
  int x0 in [0,5];
  int y0 in [0,5];
  constraint radius {
    real r^2 = (x-x0)^2 + (y-y0)^2;
  }
}
```

Fig. 2. A COMMA model for the antennas problem

concatenation of the names of objects of origin in order to avoid name redundancy. The process is done recursively. Constraints derived from the expansion of objects are added in the same way[4] (see Figure 3).

```
Variables
 real c1_x in [-3,3],
 real c1_y in [-3,3],
 real c1_r in [1.5,2.5],
 ...
 real c2_x in [-3,3],
 real c2_y in [-3,3],
 real c2_r in [1.5,2.5],
 ...
Constraints
  ...
 c1_r  = c2_r,
 c1_r^2 = (c1_x-c1_x0)^2 + (c1_y-c1_y0)^2,
 c2_r^2 = (c2_x-c2_x0)^2 + (c2_y-c2_y0)^2;
```

Fig. 3. Flat RealPaver file

4 Conclusion and Future Work

We have shown by means of a practical example how COMMA is able to model systems using a compositional approach. This platform could be improved extending the list of supported solvers. The definition of a UML profile will be useful for modeling from a graphical perspective. Finally, we plan to make the COMMA system open source for the research community.

[4] The translation process to ECLiPSe and Gecode/J can be found in [1].

References

1. *COMMA Modeling Language*. http://www.inf.ucv.cl/~rsoto/comma.
2. *Gecode System*. http://www.gecode.org.
3. A.H. Borning. The Programming Languages Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM TOPLAS*, 3(4):353–387, 1981.
4. Alan M. Frisch et al. The design of essence: A constraint language for specifying combinatorial problems. In *IJCAI*, pages 80–87, 2007.
5. R. Rafeh et al. From zinc to design model. In *PADL*, pages 215–229, 2007.
6. F. Benhamou et al. *Trends in Constraint Programming*. ISTE, 2007.
7. P. Fritzson and V. Engelson. Modelica – A Unified Object-Oriented Language for System Modeling and Simulation. In *ECOOP98*, Brussels, 1998.
8. L. Granvilliers and F. Benhamou. Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.
9. B. Jayaraman and P.Y. Tambay. Constrained Objects for Modeling Complex Structures. In *OOPSLA*, Minneapolis, USA, 2000.
10. M. Paltrinieri. On the Design of Constraint Satisfaction Problems. In *PPCP*, Orcas Island, USA, 1994.
11. J.F. Puget. A C++ implementation of CLP. In *SCIS*, Singapore, 1994.
12. R. Soto and L. Granvilliers. Dynamic parser cooperation for extending a constrained object-based modeling language. To appear in Proceedings of WLP 2007, Wuerzburg, Germany.
13. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
14. M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming, 1997.

Testing Relativised Uniform Equivalence under Answer-Set Projection in the System cc⊤*

Johannes Oetsch[1], Martina Seidl[2], Hans Tompits[1], and Stefan Woltran[1]

[1] Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,tompits}@kr.tuwien.ac.at
woltran@dbai.tuwien.ac.at
[2] Institut für Softwaretechnik, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
seidl@big.tuwien.ac.at

Abstract. The system cc⊤ is a tool for testing correspondence between logic programs under the answer-set semantics with respect to different refined notions of program correspondence. The underlying methodology of cc⊤ is to reduce a given correspondence problem to the satisfiability problem of quantified propositional logic and to employ extant solvers for the latter language as back-end inference engines. In a previous version of cc⊤, the system was designed to test correspondence between programs based on *relativised strong equivalence under answer-set projection*. Such a setting generalises the standard notion of strong equivalence by taking the alphabet of the context programs as well as the projection of the compared answer sets to a set of designated output atoms into account. This paper outlines a newly added component of cc⊤ for testing similarly parameterised correspondence problems based on *uniform equivalence*.

1 General Information

An important issue in software development is to determine whether two encodings of a given problem are equivalent, i.e., whether they yield the same result on a given problem instance. Depending on the context of problem representations, different definitions of "equivalence" are useful and desirable. The system cc⊤ [1] (short for "correspondence-checking tool") is devised as a checker for a broad range of different such comparison relations defined between *disjunctive logic programs* (DLPs) *under the answer-set semantics* [2]. In a previous version of cc⊤, the system was designed to test correspondence between logic programs based on *relativised strong equivalence under answer-set projection*. Such a setting generalises the standard notion of strong equivalence [3] by taking the alphabet of the context programs as well as the projection of the compared answer sets to a set of designated output atoms into account [4]. The latter feature

* This work was partially supported by the Austrian Science Fund (FWF) under grant P18019. The second author was also supported by the Austrian Federal Ministry of Transport, Innovation, and Technology (BMVIT) and the Austrian Research Promotion Agency (FFG) under grant FIT-IT-810806.

reflects the common use of local (hidden) variables which may be used in submodules but which are ignored in the final computation.

In this paper, we outline a newly added component of cc$\top$ for testing similarly parameterised correspondence problems but generalising *uniform equivalence* [5]—that is, we deal with a component of cc$\top$ for testing *relativised uniform equivalence under answer-set projection*. This notion, recently introduced in previous work [6], is less restrained, along with a slightly lower complexity than its strong pendant. However, in general, it is still outside a feasible means to be computed by answer-set solvers (provided that the polynomial hierarchy does not collapse). Yet, like relativised strong equivalence with projection, it can be efficiently reduced to the satisfiability problem of quantified propositional logic, an extension of classical propositional logic characterised by the condition that its sentences, generally referred to as *quantified Boolean formulas* (QBFs), are permitted to contain quantifications over atomic formulas. The architecture of cc$\top$ takes advantage of this and uses existing solvers for quantified propositional logic as back-end reasoning engines.

2 System Specifics

The equivalence notions under consideration are defined for ground disjunctive logic programs with default negation under the answer-set semantics [2]. Let $AS(P)$ be the collection of the answer sets of a program P. Two programs, P and Q, are *strongly equivalent* iff, for any program R, $AS(P \cup R) = AS(Q \cup R)$; they are *uniformly equivalent* iff, for any set F of facts, $AS(P \cup F) = AS(Q \cup F)$. While strong equivalence is relevant for program optimisation and modular programming in general [7–9], uniform equivalence is useful in the context of hierarchically structured program components, where lower-layered components provide input for higher-layered ones. In abstracting from strong and uniform equivalence, Eiter *et al.* [4] introduced the notion of a *correspondence problem* which allows to specify (i) a *context*, i.e., a class of programs used to be added to the programs under consideration, and (ii) the relation that has to hold between the answer sets of the extended programs. The concrete formal realisation of relativised uniform equivalence with projection is as follows [6]: Consider a quadruple $\Pi = (P, Q, 2^A, \odot_B)$, where P, Q are programs, A, B are sets of atoms, $\odot \in \{\subseteq, =\}$, and $\mathcal{S} \ \odot_B \ \mathcal{S}'$ stands for $\{I \cap B \mid I \in \mathcal{S}\} \odot \{J \cap B \mid J \in \mathcal{S}'\}$. Then, Π *holds* iff, for each $F \in 2^A$, $AS(P \cup F) \odot_B AS(Q \cup F)$. Furthermore, Π is called a *propositional query equivalence problem* (PQEP) if $\odot_B$ is given by $=_B$, and a *propositional query inclusion problem* (PQIP) if $\odot_B$ is given by $\subseteq_B$. Note that $(P, Q, 2^A, =_B)$ holds iff $(P, Q, 2^A, \subseteq_B)$ and $(Q, P, 2^A, \subseteq_B)$ jointly hold.

For illustration, consider the programs

$$P = \{sad \vee happy \leftarrow;\ sappy \leftarrow sad, happy;\ confused \leftarrow sappy\},$$
$$Q = \{sad \leftarrow \text{not } happy;\ happy \leftarrow \text{not } sad;\ confused \leftarrow sad, happy\},$$

which express some knowledge about the "moods" of a person, where P uses an auxiliary atom $sappy$. The programs can be seen as queries over a propositional database which consists of facts from, e.g., $\{happy, sad\}$. For the output, it would be natural to consider the common intensional atom $confused$. We thus consider $\Pi = (P, Q,$

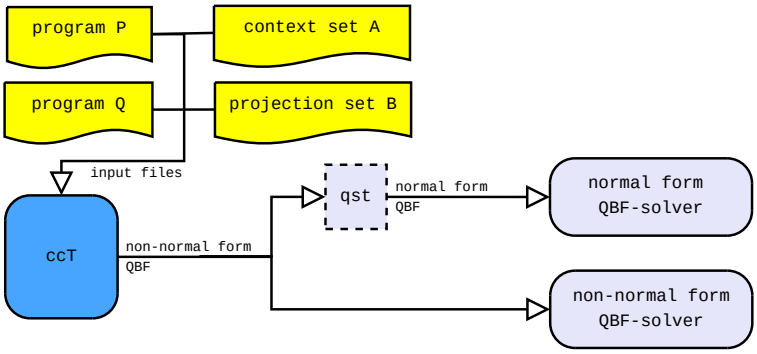


Fig. 1. Overall architecture of cc⊤.

$2^A, =_B)$ as a suitable PQEP, specifying $A = \{happy, sad\}$ and $B = \{confused\}$. It is a straightforward matter to check that Π, defined in this way, holds.

As pointed out in Section 1, the overall approach of cc⊤ is to reduce PQEPs and PQIPs to the satisfiability problem of quantified propositional logic and to use extant solvers for the latter language [10] as back-end inference engines for evaluating the resulting formulas. The reductions required for this approach are described by Oetsch *et al.* [6] but cc⊤ employs additional optimisations [11]. We note that quantified propositional logic is an extension of classical propositional logic in which sentences are permitted to contain quantifications over atomic formulas. It is standard custom to refer to the formulas of this language as *quantified Boolean formulas* (QBFs).

The overall architecture of cc⊤ is depicted in Figure 1. The system takes as input two programs, P and Q, and two sets of atoms, A and B. Command-line options select between two kinds of reductions, a direct one or an optimised one, and whether the programs are compared as a PQIP or a PQEP. Detailed invocation syntax can be requested with option `-h`. The syntax of the programs is the basic DLV syntax.[1] Since cc⊤ does not output QBFs in a specific normal form, for using solvers requiring normal-form QBFs, the additional normaliser `qst` [12] is employed. Finally, cc⊤ is developed entirely in ANSI C; hence, it is highly portable. The parser for the input data was written using LEX and YACC. Further information about cc⊤ is available at

`http://www.kr.tuwien.ac.at/research/ccT/`.

Experimental evaluations using different QBF solvers are reported in a companion paper [11].

References

1. Oetsch, J., Seidl, M., Tompits, H., Woltran, S.: ccT: A Tool for Checking Advanced Correspondence Problems in Answer-Set Programming. In: Proceedings of the 15th International Conference on Computing (CIC 2006), IEEE Computer Society Press (2006) 3–10

[1] See `http://www.dlvsystem.com/` for details about DLV.

2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
3. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic **2** (2001) 526–541
4. Eiter, T., Tompits, H., Woltran, S.: On Solution Correspondences in Answer Set Programming. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005). (2005) 97–102
5. Eiter, T., Fink, M.: Uniform Equivalence of Logic Programs under the Stable Model Semantics. In: Proceedings of the 19th International Conference on Logic Programming (ICLP 2003). Volume 2916 in Lecture Notes in Computer Science. Springer (2003) 224–238
6. Oetsch, J., Tompits, H., Woltran, S.: Facts do not Cease to Exist Because They are Ignored: Relativised Uniform Equivalence with Answer-Set Projection. In: Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI 2007), AAAI Press (2007) 458–464
7. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying Logic Programs Under Uniform and Strong Equivalence. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). Volume 2923 of Lecture Notes in Computer Science. Springer Verlag (2004) 87–99
8. Pearce, D.: Simplifying Logic Programs under Answer Set Semantics. In: Proceedings of the 20th International Conference on Logic Programming (ICLP 2004). Volume 3132 of Lecture Notes in Computer Science. Springer (2004) 210–224
9. Lin, F., Chen, Y.: Discovering Classes of Strongly Equivalent Logic Programs. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005). (2005) 516–521
10. Le Berre, D., Narizzano, M., Simon, L., Tacchella, L.A.: The Second QBF Solvers Comparative Evaluation. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004). Revised Selected Papers. Volume 3542 of Lecture Notes in Computer Science., Springer (2005) 376–392
11. Oetsch, J., Seidl, M., Tompits, H., Woltran, S.: An Extension of the System cc$\top$ for Testing Relativised Uniform Equivalence under Answer-Set Projection (2007). Submitted draft
12. Zolda, M.: Comparing Different Prenexing Strategies for Quantified Boolean Formulas (2004). Master's Thesis, Vienna University of Technology

spock: A Debugging Support Tool for Logic Programs under the Answer-Set Semantics*

Martin Gebser[1], Jörg Pührer[2], Torsten Schaub[1],
Hans Tompits[2], and Stefan Woltran[2]

[1] Institut für Informatik, Universität Potsdam,
August-Bebel-Straße 89, D-14482 Potsdam, Germany
{gebser,torsten}@cs.uni-potsdam.de
[2] Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9–11, A–1040 Vienna, Austria
{puehrer,tompits}@kr.tuwien.ac.at
woltran@dbai.tuwien.ac.at

Abstract. Logic programs under the answer-set semantics are an acknowledged tool for declarative problem solving and constitute the canonical instance of the general *answer-set programming* (ASP) paradigm. Despite its semantic elegance, ASP suffers from a lack of software-development tools. In particular, there is a need to support engineers in detecting erroneous parts of their programs. Unlike in other areas of logic programming, applying tracing techniques for debugging logic programs under the answer-set semantics seems rather unnatural since sticking to imperative solving algorithms would undermine the declarative flavor of ASP. In this paper, we outline the system spock, a debugging support tool for answer-set programs making use of ASP itself. The implemented techniques maintain the declarative nature of ASP within the debugging process and are independent from the actual computation of answer sets.

1 General Information

During the last decade, logic programs under the answer-set semantics received increasing importance as an approach for declarative problem solving. Their formal underpinnings are rooted in concepts from nonmonotonic reasoning, and they constitute the canonical instance of the general *answer-set programming* (ASP) paradigm [1] in which *models* represent solutions rather than *proofs* as in traditional logic programming. The nonmonotonicity of answer-set programs, however, is an aggravating factor for detecting sources of errors, since every rule of a program might significantly influence the resulting answer sets.

In this paper, we outline the basic features of the system spock [2] that supports developers of answer-set programs to detect and locate errors in their programs, independent of specific ASP solvers. The theoretical background of spock was introduced in previous work [3], and exploits and extends a *tagging technique* as used by Delgrande et al. [4] for compiling ordered logic programs into standard ones. In our approach, a

* This work was partially supported by the Austrian Science Fund (FWF) under project P18019.

program to debug, Π, is translated into another program, $\mathcal{T}_{\mathsf{K}}[\Pi]$, which can be seen as a kernel module providing additional information relevant for debugging Π. In particular, $\mathcal{T}_{\mathsf{K}}[\Pi]$, whose size is polynomial in the size of Π, is equipped with specialized meta-atoms, called *tags*, serving two purposes: Firstly, they allow for controlling and manipulating the formation of answer sets of Π, and secondly, tags occurring in the answer sets of the translated program reflect various properties of Π. In addition to $\mathcal{T}_{\mathsf{K}}$, spock is devised to support supplementary translations for a program to debug Π, allowing an extrapolation of non-existing answer sets in combination with explanations why an interpretation is not an answer set of Π.

2 System Specifics

Our debugging system spock implements several transformations for debugging propositional normal logic programs, involving the tags ap, bl, ok, $\overline{\mathsf{ok}}$, ko, ab_p, ab_c, and ab_l. The basic idea of tagging is to decompose the rules of an original program, separating the causal relation between the satisfaction of a rule body from the occurrence of the respective heads in an interpretation.

One task of tags is to provide information about the program, Π, to debug. E.g., there is a one-to-one correspondence between the answer sets of Π and the answer sets of $\mathcal{T}_{\mathsf{K}}[\Pi]$ such that an answer set of the translated program contains either tag $\mathsf{ap}(n_r)$ or tag $\mathsf{bl}(n_r)$, for each rule $r \in \Pi$, where n_r is a unique label for r, providing information whether r is applicable or blocked, respectively, in the related answer set of Π. For example, consider program $\Pi_{\mathsf{ex}} = \{r_1 = a \leftarrow b,\ r_2 = b \leftarrow not\ c,\ r_3 = c \leftarrow not\ a\}$, having the answer sets $\{a, b\}$ and $\{c\}$. The answer sets of the transformed program $\mathcal{T}_{\mathsf{K}}[\Pi_{\mathsf{ex}}]$ are given by $\{a, b, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{r_3})\} \cup OK$ and $\{c, \mathsf{ap}(n_{r_3}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2})\} \cup OK$, where $OK = \{\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3})\}$. From these, we get the information that, e.g., rule r_1 is applicable under answer set $\{a, b\}$ and blocked under $\{c\}$.

Apart from being used for analyzing Π and its answer sets, tagging provides a handle on the formation of answer sets of Π. By joining the translated program with rules involving, for instance, control tag $\mathsf{ko}(n_r)$, rule $r \in \Pi$ can selectively be deactivated. This feature can be utilized by more advanced debugging modules, based on our kernel transformation $\mathcal{T}_{\mathsf{K}}$, which are not restricted to analyzing actual answer sets of Π. In particular, spock features program translations for investigating why a particular interpretation I is not an answer set of a program Π. Here, three sources of errors are identified on the basis of the Lin-Zhao theorem [5]: Reasons for I not being an answer set are either related to the program, its completion, or its non-trivial loop formulas, respectively. According to this error classification, we distinguish between three corresponding *abnormality* tags, ab_p, ab_c, and ab_l, which may occur in the answer sets of the transformed program. The program-oriented abnormality tag $\mathsf{ab}_p(n_r)$ indicates that rule $r \in \Pi$ is applicable but not satisfied with respect to the considered interpretation I. The completion-oriented abnormality tag $\mathsf{ab}_c(a)$ is contained whenever atom a is in I but all rules having a as head are blocked. Finally, the presence of a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ indicates the possible existence of some loop L in Π, $a \in L$, that is unfounded with respect to I.

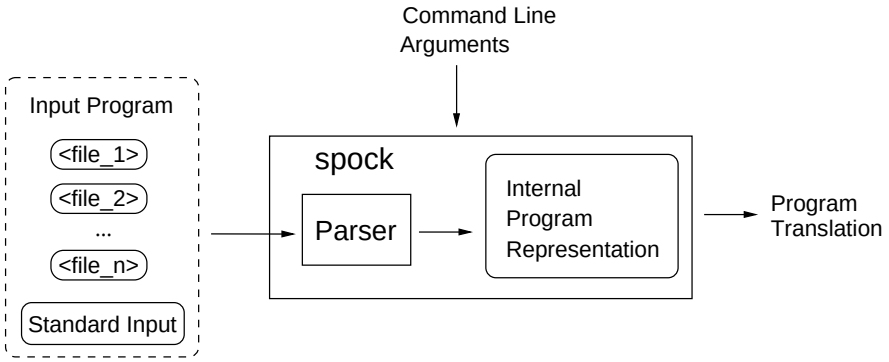


Fig. 1. Data flow of program translations.

As the number of interpretations for a program grows exponentially in the number of occurring atoms, we use standard optimization techniques of ASP to reduce the amount of debugging information, focusing on answer sets of the tagged program that involve a minimum number of abnormality tags.

The transformations to be applied are chosen by setting call parameters. Fig. 1 illustrates the typical data flow of program translations with `spock`. The tool is written in Java 5.0 and published under the GNU General Public License [6]. It can be used either with `DLV` [7] or with `Smodels` [8] (together with `Lparse`) and is available at

`http://www.kr.tuwien.ac.at/research/debug`

as a jar-package including binaries and sources.

Future work includes the integration of further aspects of the translation approach as well as the design of a graphical user interface to ease the applicability of the different features `spock` provides.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: "That is illogical captain!" – The debugging support tool spock for answer-set programs: System description. In De Vos, M., and Schaub, T., eds.: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 71–85
3. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In Baral, C., Brewka, G., and Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), number 4483 in Lecture Notes in Artificial Intelligence. Springer-Verlag (2007) 31–43
4. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3**(2) (2003) 129–187

5. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
6. Free Software Foundation Inc.: GNU General Public License - Version 2, June 1991 (1991) `http://www.gnu.org/copyleft/gpl.html`
7. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
8. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234

Author Index