26TH WORKSHOP ON LOGIC PROGRAMMING

– WLP 2012 –

BONN, SEPTEMBER 24-25, 2012

PROCEEDINGS

Andreas Behrend            Dietmar Seipel            Michael Hanus
Francois Bry             Rainer Manthey (eds.)
Foreword

This volume consists of the contributions presented at the 26th Workshop on Logic Programming (WLP 2012), which was held at Bonn-Aachen International Center for Information Technology (B-IT), Bonn, Germany, from September 24 to 25, 2012. The series of workshops on (constraint) logic programming brings together researchers interested in logic programming, constraint programming, and related areas like databases and artificial intelligence. Previous workshops have been held in Austria, Germany, Switzerland, and Egypt, serving as the annual meeting of the Society of Logic Programming (GLP, Gesellschaft für Logische Programmierung e.V.).

This event received a total of 14 submissions from authors of 6 countries (Austria, Germany, Italy, Egypt, Spain and Sweden). Each submission was assigned to three members of the PC for reviewing and 11 submissions were accepted for presentation. Besides technical contributions, the program includes also system descriptions and application papers.

Finally, we would like to thank all the authors who have submitted papers, all colleagues who have presented invited talks, and all members of the program committee and external referees for reviewing the submissions and for their contributions to the success of the workshop.

Bonn, September 20, 2012

Andreas Behrend
WLP 2012 Conference Organization

Program Chair:
Andreas Behrend, University of Bonn, Germany

Program Committee
Slim Abdennadher – German University of Cairo
Andreas Behrend – University of Bonn
Christoph Beierle – University of Hagen
Francois Bry – University of Munich (LMU)
Juergen Dix – Technical University of Clausthal
Wolfgang Faber - University of Calabria
Michael Fink – Technical University of Vienna
Michael Hanus – University of Kiel
Petra Hofstedt – Technical University of Cottbus
Steffen Hölldobler – Technical University of Dresden
Tomi Janhunen – University of Aalto
Ulrich John - SIR Dr. John UG
Gabriele Kern-Isberner - Technical University of Dortmund
Alessandra Mileo – University of Galway (DERI)
Torsten Schaub - University of Potsdam
Jan Sefranek - Comenius University
Dietmar Seipel - University of Würzburg
Hans Tompits - Technical University of Vienna
Janis Voigtländer - University of Bonn
Marina De Vos - University of Bath

Local Organization
Ali Polat
Fabian Schmidt
Sahar Vahdati

Additional Reviewers
Stephan Frank
Michael Köster
Alessandra Mileo
Sean Policarpio
Torsten Schaub
Federico Schlesinger
Table of Contents

Invited Keynote
The Computational Essence of Sorting Algorithms ................................................................. 1
   Ralf Hinze

Research Track Session 1: Constraints and Logic Programming
Soft Constraint Logic Programming for Electric Vehicle Travel Optimization ....................... 3
   Giacoma Monreale, Ugo Montanari, and Nicklas Hoch
A Variant of Earley Deduction With Partial Evaluation ....................................................... 21
   Heike Stephan and Stefan Brass
Causes for Checking the Integrity of the Evolution of Databases ........................................... 33
   Hendrik Decker

Invited Keynote
What is next after 25 years of SWI-Prolog? ........................................................................ 47
   Jan Wielemaker

Research Track Session 2: Application of Logic Programming
An Application of Update Propagation on Logic Programs Representing Java Source Code .......... 49
   Richard Tantius, Daniel Speicher, and Andreas Behrend
Knowledge Engineering for Business Rules in PROLOG ....................................................... 61
   Ludwig Ostermayer and Dietmar Seipel
Clone Removal in Java Programs as a Process of Stepwise Unification ................................... 75
   Daniel Speicher and Andri Bremm

System Description
A Logic Programming Approach for the Reconstruction of Integration Networks ................. 81
   Daniel Ritter
CHR meets Google Docs ........................................................................................................... 97
   Slim Abdennadher and Mahitab Sharawy
Declarative Web Programming with PROLOG and XUL .................................................... 113
   Christian Schneiker and Dietmar Seipel
Research Track Session 3: Languages and Semantics

Extending Object-Oriented Languages by Declarative Specifications of Complex Objects Using Answer-Set Programming................................................................. 129

Johannes Oetsch, Jörg Pührer and Hans Tompits

A Denotational Semantics for Weak Encapsulation in Curry ......................................................... 139

Jan Christiansen, Michael Hanus, Fabian Reck, and Daniel Seidel
The Computational Essence of Sorting Algorithms

Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road
Oxford, OX1 3QD, England
ralf.hinze@cs.ox.ac.uk,
http://www.cs.ox.ac.uk/ralf.hinze/

Abstract. Sorting algorithms are an intrinsic part of functional programming folklore as they exemplify algorithm design using folds and unfolds. This has given rise to an informal notion of duality among sorting algorithms: insertion sorts are dual to selection sorts. Using bialgebras and distributive laws, we formalise this notion within a categorical setting. We use types as a guiding force in exposing the recursive structure of bubble, insertion, selection, quick, tree, and heap sorts. Moreover, we show how to distill the computational essence of these algorithms down to one-step operations that are expressed as natural transformations.

This is joint work with Daniel W.H. James, Thomas Harper, Nicolas Wu, José Pedro Magalhães.
Soft Constraint Logic Programming for Electric Vehicle Travel Optimization *

Giacoma Valentina Monreale\(^1\), Ugo Montanari\(^1\) and Nicklas Hoch\(^2\)

\(^1\) Dipartimento di Informatica, Università di Pisa, Italy
vale@di.unipi.it, ugo@di.unipi.it
\(^2\) Volkswagen AG, Corporate Research Group Wolfsburg, Germany
nicklas.hoch@volkswagen.de

Abstract. Soft Constraint Logic Programming is a natural and flexible declarative programming formalism, which allows to model and solve real-life problems involving constraints of different types. In this paper, after providing a slightly more general and elegant presentation of the framework, we show how we can apply it to the e-mobility problem of coordinating electric vehicles in order to overcome both energetic and temporal constraints and so to reduce their running cost. In particular, we focus on the journey optimization sub-problem, considering sequences of trips from a user’s appointment to another one. Solutions provide the best alternatives in terms of time and energy consumption, including route sequences and possible charging events.

1 Introduction

Classical constraint satisfaction problems (CSPs) \([12]\) represent an expressive and natural formalism useful to specify different types of real-life problems. A CSP can be described as a set of variables associated with a domain of values, and a set of constraints. A constraint is a limitation of the possible combinations of the values of some variables. So, solving a CSP consists in finding an assignment of values to all its variables guaranteeing that all constraints are satisfied.

Despite their applicability, the main limit suffered by CSPs is the ability of just stating if an assignment of certain values to the variables is allowed or not. This is indeed not enough to model scenarios where the knowledge is not either entirely available or not crisp. In these cases constraints are preferences and, when the problem is overconstrained, one would like to find a solution that is not so bad, i.e., the best solution according to the levels of preferences. For this reason, in \([2, 3]\), the soft CSP framework has been proposed. It extends classical constraints by adding to the usual notion of CSP the concept of a structure representing the levels of satisfiability or the costs of a constraint. Such a structure is represented by a semiring, that is, a set with two operations: one (usually denoted by \(+\)) is used to generate an ordering over the levels, while

\* Research partially supported by the EU FP7-ICT IP ASCEns.
the other one (denoted by ×) is used to define how two levels can be combined and which level is the result of such a combination.

Constraint logic programming (CLP) [10] extends logic programming (LP) by embedding constraints in it: term equalities is replaced with constraints and the basic operation of LP languages, the unification, is replaced by constraint handling in a constraint system. It therefore inherits the declarative approach of LP, according to which the programmer specifies what to compute while disregarding how to compute it, by also offering efficient constraint-solving algorithms.

However, only classical constraints can be handled in the CLP framework. So, in [4], it has been extended to also handle soft constraints. This has led to a high-level and flexible declarative programming formalism, called Soft CLP (SCLP), allowing to easily model and solve real-life problems involving constraints of different types. Roughly speaking, SCLP programs are logic programs where constraints are represented by predicates which are defined by clauses whose body is a value of the semiring modelling the levels of satisfiability or the costs of the constraints. The flexibility of the approach is due to the fact that the same framework can be used to handle different kinds of soft constraints by simply choosing different semirings. It can indeed be used to handle fuzzy, probabilistic, prioritized and optimization problems, as well as classical constraints.

In this paper, before presenting an application of the SCLP framework to the e-mobility, we provide a slightly more general and elegant presentation of the SCSP framework based on the notion of named semiring, as briefly presented in [7]. In particular, soft constraints are elements of the named semiring which, besides the additive and multiplicative operations useful to combine the constraints, is equipped by a permutation algebra and a hiding operator allowing an explicit handling of names. The permutation algebra indeed allows to characterize the support of each constraint, that is, the subset of variables on which the constraint really depends, while the hiding operator allows us to remove variables from the support of a constraint. Then, the SCLP framework can be seen as the general case of the SCSP one.

We propose the SCLP framework as a high-level specification formalism useful to naturally model and also solve some e-mobility optimization problems. With e-mobility indeed new constraints must be considered: electric vehicles (EVs) have a limited range and they take long time to charge. So, it is needed to guarantee that throughout the user’s itinerary the EV never underruns a limit energy level and that the user always arrives in time to all his appointments.

We consider the EV travel optimization problem described in [9]. In it the user has a set of appointments and he makes a series of decisions about the sequences of trips from an appointment to another one, for example which route to take, if and where to charge the EV and so on. The aim is to find the optimal combination of travel choices which minimizes the users cost criteria. In [9], the authors propose a hierarchical presentation of the mobility framework, which they exploit to decompose the optimization problem in sub-problems, and in particular, they study the journey level one. Here, beside it, we consider the optimization problem of the lower level, i.e., the one of the trip level. This last
problem consists in finding the best trips in terms of travel time and energy consumption, while the former one consists in finding the optimal journey, that is, the optimal sequence of coupled trips, again in terms of the same criteria, guaranteeing that the user reaches each appointment in time and that the state of charge (SoC) of the EV never falls below a given threshold.

The trip level problem substantially coincides with the multicriteria version of the shortest path problem modelled in [5] as an SCLP program. So, starting from a slightly different specification of this problem, we propose an SCLP program modelling the journey problem. In order to also actually execute both the SCLP programs, we propose CIAO Prolog [6], a system supporting constraint logic programming. We therefore explicitly implement the soft framework, by defining two predicates, the plus and the times ones, which respectively model the additive and the multiplicative operations of the semiring.

This work arises from the research activity in the European project ASCENS (Autonomic Service Component ENSembles) aiming at studying formal models, languages, and programming tools, for the modelling and the development of autonomous, self-aware adaptive systems. E-mobility is right one of the case studies of the project and our work represents a first answer to the need of a high-level, declarative, executable specification language and of a powerful and flexible programming environment where e-mobility problems can be easily and naturally modelled and solved. We indeed show as the SCLP framework is able to satisfy all these requirements and it can thus be used as a support for rapid prototyping and exploratory programming for this kind of problems.

The paper is organized as follows. Section 2 introduces the SCSP framework as an instantiation of the named semiring framework. Section 3 briefly recalls the SCLP language and then Section 4 shows how the trip and journey optimization problems can be modelled and solved through (S)CLP programs. Finally, Section 5 concludes the paper by illustrating some open venues for further works.

2 Soft Constraints by means of Named Semirings

This section presents the soft CSP framework based on semiring [2, 3] as an instantiation of the more general framework based on named semiring [7].

The notion of named semiring is based on the ones of c-semiring (c stands for constraint) and permutation algebra.

2.1 C-Semiring

Definition 1 (c-semiring). A c-semiring is a tuple $\langle A, +, \times, 0, 1 \rangle$ such that:

- $A$ is a set and $0, 1 \in A$;
- $+: A \times A \rightarrow A$ is a commutative, associative and idempotent operation, such that $0$ is its unit element and $1$ is its absorbing element;
- $\times: A \times A \rightarrow A$ is a commutative and associative operation, such that it distributes over $+$, $1$ is its unit element, and $0$ is its absorbing element.
Thanks to the idempotence of the + operator, the relation $\langle A, \leq \rangle$, defined as $a \leq b$ if $a + b = b$, is a partial order. Intuitively, $a \leq b$ means that $b$ is better than $a$ or that $a$ implies $b$.

It is possible to prove that: (i) the two operations $+$ and $\times$ are monotone on $\leq$; (ii) 0 is its minimum and 1 its maximum; (iii) $\langle A, \leq \rangle$ is a complete lattice and $+$ is its least upper bound. Finally, if $\times$ is idempotent then: (iv) $+$ distribute over $\times$; (v) $\langle A, \leq \rangle$ is a distributive lattice, and (vi) $\times$ is its greatest lower bound.

### 2.2 Permutation Algebra

Here we briefly recall the notion of permutation algebra. We refer the reader to [8] for a detailed introduction.

In the following, we fix a chosen infinite, countable, totally ordered set $\mathcal{N}$ of names, which we denote by $x, y, z, \ldots$.

**Definition 2 (Permutations).** A name substitution is a function $\sigma : \mathcal{N} \to \mathcal{N}$, while a permutation $\rho$ is a bijective name substitution. The set of all such permutations on $\mathcal{N}$ is denoted by $\mathcal{P}(\mathcal{N})$.

**Definition 3 (Kernel).** Let $\rho \in \mathcal{P}(\mathcal{N})$ be a permutation on $\mathcal{N}$. The kernel of $\rho$ is the set of the names that are changed by the permutation, formally, $K(\rho) = \{ x \in \mathcal{N} | \rho(x) \neq x \}$.

A permutation $\rho$ is finite if its kernel is finite.

From now on we consider only finite permutations.

In the following, we introduce the notion of permutation algebra. It consists of a pair composed of a carrier set and a description of how the elements of the carrier set are transformed by permutations.

**Definition 4 (Permutation Algebras).** The permutation signature $\Sigma_p$ on $\mathcal{N}$ is defined as the set of unary operators $\{ \hat{\rho} | \rho \in \mathcal{P}(\mathcal{N}) \}$ plus the two axioms $\hat{id}(a) = a$ and $\hat{\rho_1}(\hat{\rho_2}(a)) = \hat{\rho_1\rho_2}(a)$.

A permutation algebra $\mathcal{A} = (\mathcal{A}, \{ \hat{\rho}_A \})$ consists of a carrier set $\mathcal{A}$ and the set of the interpreted operations $\hat{\rho}_A$.

**Definition 5 (Support).** Let $\mathcal{A}$ be a permutation algebra and $a$ an element of its carrier set. The support of $a$, $\text{supp}(a)$, is the smallest set of names such that, given a permutation $\rho$, if $\rho(x) = x$ for all $x \in \text{supp}(a)$, then $\hat{\rho}_A(a) = a$.

Intuitively, $\text{supp}(a)$ represents the free names of $a$: indeed the permutations which do not modify them are not influent on $a$.

**Definition 6 (Finitely supported algebra).** A permutation algebra $\mathcal{A}$ is finitely-supported if each element of its carrier has finite support.

---

3 Actually, in order to prove this result, we must assume that the sum of an infinite number of elements exists.
2.3 Named c-Semiring.

A named semiring [7] is a c-semiring plus a finitely-supported permutation algebra $\mathcal{A}$ and a hiding operator $(\nu x.)$. The permutation algebra allows characterizing the finite set of free names of each element $c$ of the named semiring (represented by the support of $c$), while $(\nu x.)$ applied to $c$ makes the name $x$ local in $c$.

Definition 7 (Fusion). A (name) fusion is a total equivalence relation on $\mathcal{N}$ with only finitely many non-singular equivalence classes. We denote by $x = y$ the fusion with a unique non-singular equivalence class consisting of $x$ and $y$.

Definition 8 (Named c-semiring). A named c-semiring $C = \langle C, +, \times, \nu x., \{\hat{\rho}_C\}, 0, 1 \rangle$ is a tuple where:

- $x = y \in C$ for $x, y \in \mathcal{N}$;
- $\langle C, +, \times, 0, 1 \rangle$ is a c-semiring;
- $\langle C, \{\hat{\rho}_C\} \rangle$ is a finite-support permutation algebra;
- $\nu x. : C \rightarrow C$, for each name $x$, is a unary operation;
- for all $c, d \in C$ and for all $\rho$ the following axioms hold:

(FUSE) $x = y \times c$ iff $x = y \times [y/x]c$

(HIDE) $\nu x.1 = 1$ $\nu x.\nu y.c = \nu y.\nu x.c$ $\nu x. (c \times d) = c \times \nu x.d$ if $x \notin \text{supp}(c)$

$\nu x. (c + d) = c + \nu x.d$ if $x \notin \text{supp}(c)$ $\nu x.c = \nu y.([y/x]c)$ if $y \notin \text{supp}(c)$

(PERM) $\hat{\rho}_C 0 = 0$ $\hat{\rho}_C 1 = 1$ $\hat{\rho}_C (c \times d) = \hat{\rho}_C c \times \hat{\rho}_C d$ $\hat{\rho}_C (c + d) = \hat{\rho}_C c + \hat{\rho}_C d$

$\hat{\rho}_C (\nu x.c) = \nu x. (\hat{\rho}_C c)$ if $x \notin K(\rho)$

In the (FUSE) axiom, $[y/x]c$ denotes $c$ where $y$ is replaced by $x$.

2.4 The Named SCSP Framework

As briefly shown in [7], named c-semirings can be suitably instantiated to model SCSPs.

Definition 9 (Constraints). Let $S = \langle A, +, \times, 0, 1 \rangle$ be a c-semiring, $\mathcal{N}$ a set of totally ordered names, and $D$ a finite domain of interpretation for $\mathcal{N}$. A soft constraint is a function $(\mathcal{N} \rightarrow D) \rightarrow A$, which associates a value of $A$ to each assignment $\eta : \mathcal{N} \rightarrow D$ of the names.

We define $C$ as the set of all soft constraints over $\mathcal{N}$, $D$ and $A$.

Definition 10. Let $S = \langle A, +, \times, 0, 1 \rangle$ be a c-semiring, $\mathcal{N}$ a set of totally ordered names, and $D$ a finite domain of interpretation for $\mathcal{N}$. Moreover, let $C$ be the set of all soft constraints over $\mathcal{N}$, $D$ and $A$. We define the $C_{\text{SCSP}}$ as the named c-semiring $(C, +', \otimes, \nu x., \{\hat{\rho}_C\}, 0', 1')$, where fusions $x = y$ are defined as $(x = y)\eta = 1$ if $\eta(x) = \eta(y)$ and $(x = y)\eta = 0$ otherwise;

$(c_1 + c_2)\eta = c_1\eta + c_2\eta; (c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta; (\nu x.c)\eta = \sum_{d \in D}(\eta[d/x]); (\hat{\rho}_C c)\eta = c\eta'$ with $\eta'(x) = \eta(\rho(x)); 0'\eta = 0$ and $1'\eta = 1'$ for all $\eta$. 

7
The assignment \( \eta[d/x] \) is defined as usual: \( \eta[d/x](y) = d \) if \( x = y \) and \( \eta[d/x](y) = \eta(y) \) otherwise.

Note that, by definition, each constraint \( c \in C \) involves all the names in \( \mathcal{N} \), but it really depends on the assignment of the names in \( \text{supp}(c) \), and intuitively, restricting means eliminating a name from the support of the constraint, by choosing the best value for it.

**Definition 11 (Soft CSP).** A soft constraint satisfaction problem (SCSP) is a pair \( \langle C, Y \rangle \), where \( C \subseteq C \) is a set of constraints and \( Y \subseteq \mathcal{N} \) is a set of names.

Intuitively, \( Y \) represents the set of interface names of the constraint set \( C \).

**Definition 12 (Solution).** Let \( P = \langle C, Y \rangle \) be an SCSP. The solution of \( P \) is the constraint \( \text{Sol}(P) = (\nu x_1) \ldots (\nu x_n)(\otimes C) \), with \( \{x_1, \ldots, x_n\} = \bigcup_{c_i \in C} \text{supp}(c_i) \setminus Y \).

**Definition 13 (Best level).** Let \( P = \langle C, Y \rangle \) be an SCSP. Then, the best level of consistency of \( P \) is defined as the constraint \( \text{blevel}(P) = (\nu x_1) \ldots (\nu x_n)(\otimes C) \), where \( \{x_1, \ldots, x_n\} = \bigcup_{c_i \in C} \text{supp}(c_i) \).

Varying the semiring \( S \), on which the named semiring \( \mathcal{C}_{\text{SCSP}} \) is based, several kinds of problems can be represented: we consider the semiring \( \mathcal{C}_{\text{CSP}} = \langle \{\text{true, false}\}, \lor, \land, \text{false, true} \rangle \) for classical CSPs; \( \mathcal{C}_{\text{F CSP}} = \langle \{x | x \in [0, 1]\}, \max, \min, 0, 1 \rangle \) for fuzzy CSPs; and \( \mathcal{C}_{\text{W CSP}} = \langle \bigcup \{+\infty, \min, +, +\infty, 0\} \rangle \) for optimization CSPs.

**Example 1.** Let \( P \) be the SCSP with three names \( \mathcal{N} = \{x, y, z\} \), which can take values in \( D = \{\text{red, blue, green}\} \). We assume that only \( x \) and \( y \) are of interest, while the constraints model the fact that all names must take different values.

We consider the semiring \( \mathcal{C}_{\text{W CSP}} \), introduced above, and we define three constraints, one for each pair of names, \( c_{xy}, c_{yz} \) and \( c_{zz} \), associating the worst semiring value to the assignments which give the same color to both the names of interest for the constraint. In particular, we associate \( +\infty \) to all the assignments giving the same color to both names, if at most one name takes the color red then we associate 1, otherwise we associate 2.

In Horn logic this problem can be expressed as the predicate \( P(x, y) \) below:

\[
P(x, y) : - Q(x, y), Q(y, z), Q(z, x)
\]

\[
Q(v, w) : - \text{if } v = w \text{ then } +\infty \text{ else if either } v \text{ or } w \text{ are red then } 1 \text{ else } 2.
\]

The predicate \( Q(v, w) \) represents the constraint shown on the left of Fig. 1, where the names \( v \) and \( w \) represents the only names of the support. Note that in this representation we only show the assignment for the names of the support of the constraint. Therefore, each table entry actually represents different entries, one for each possible color of the names which are not in the support. The constraint representing the problem is then represented by the combination of three constraints. Indeed, \( Q(x, y), Q(y, z) \) and \( Q(z, x) \) respectively represent the three constraints \( c_{xy}, c_{yz} \) and \( c_{zz} \), obtained by applying to the constraint represented by \( Q(v, w) \) the permutations mapping the names of the support \( v \) and \( w \) to the
names of interest of the three constraints. As said above, the relevant names of the problem are just \( x \) and \( y \), thus the solution of the problem is obtained by combining the three constraints and restricting the scope of the name \( z \). It can be expressed as \( (\nu z)(c_{xy} \otimes c_{yz} \otimes c_{zx}) \). The resulting constraint, represented by the predicate \( P(x, y) \), is shown on the right of Fig. 1. Also in this case, we show the assignment of just the two names of the support. Each table entry is the minimum value among the ones of the solutions providing a different color for \( z \). In this case, the best level of solution is 4, the minimum over all the entries.

### 3 Soft Constraint Logic Programming

This section briefly introduces the *soft constraint logic programming* (SCLP). For a more detailed and complete introduction, we refer the reader to [4].

The SCLP framework extends the classical constraint logic programming to also handle SCSPs. We can say that an SCLP program over a certain \( c \)-semiring \( S \) is just a CLP program where constraints are defined over \( S \). In the following, we fix a semiring \( S = \langle A, +, \times, 0, 1 \rangle \).

An SCLP program is hence a set of clauses composed of a head and a body, plus a goal. The head of a clause is simply an atom, while the body can be either a collection of atoms, or a \( c \)-semiring value, or a special symbol \( 2 \) denoting that it is empty. In this two last cases clauses are called facts and define predicates representing constraints. When the body is empty, we interpret it as 1, the best element of the semiring. Atoms are \( n \)-ary predicate symbols followed by a tuple of \( n \) terms, which can be either a constant or a variable or an \( n \)-ary function symbol followed by \( n \) terms. Ground terms are terms without variables, and finally, a goal is a collection of atoms.

**Example 2.** As an example, consider the simple SCLP program on the left of Fig. 2, previously proposed in [4]. We consider the semiring \( S_{KCSP} \) and the domain \( D = \{a, b, c\} \). The program is composed of six clauses. The last two are facts and the semiring values 2 and 3, associated respectively with the atoms.
t(a) and r(a), mean that they respectively cost 2 and 3 units. The set $\mathcal{N}$ of the semiring contains all possible costs, and the operations $\min$ and $+$ allows us to minimize the sum of the costs. We consider as goal the atom :-s(a): later on we will show its semantics.

Three equivalent semantics for the SCLP languages have been defined in [4]: the model-theoretic, the fix-point, and the operational one. These semantics are conservative extensions of the corresponding ones for logic programming: this means that by choosing the e-semiring $S_{CSP}$ we get exactly the LP semantics.

Actually, we can see the SCLP framework as the general case of the SCSP one. As shown in Example 1, we can indeed express an SCSP program as a set of predicates representing the constraints together with a unique clause which represents the problem and combines all constraints. In the case of the SCLP, the main difference is that the depth of clause nesting is unlimited and the possible values $D$ of variables are elements of the Herbrand universe. So, the meaning of a predicate $P(x,y)$ assigns a semiring value to all evaluations of variables $x$ and $y$ to the Herbrand domain. This is exactly what for example the fix-point semantics of the SCLP language does.

In order to present the fix-point semantics, we need to introduce the notion of interpretation and the $T_P$ operator, mapping interpretations into interpretations.

**Definition 14 (Interpretation).** An interpretation $I$ consists of a domain $D$, representing the Herbrand universe, together with a function which takes a predicate and an instantiation of its arguments (that is, a ground atom), and returns an element of the semiring: $I : \bigcup_n (P_n \rightarrow (D^n \rightarrow A))$, where $P_n$ represents the set of $n$-ary predicates and $A$ is the set of the values of the semiring.

Since interpretations are functions from ground atoms to semiring values, we consider programs composed of clauses where the head and the body contain only ground atoms. So, for example, in the SCLP program of Example 2, the clause $s(X) :- p(X,Y)$ is replaced with all its instantiations. In particular, for each $d \in D$, we have three clauses $s(d) :- p(d,a)$, $s(d) :- p(d,b)$, and $s(d) :- p(d,c)$.

**Definition 15 ($T_P$ Operator).** Let $P$ be an SCLP program and $\mathcal{I}_P$ the set of all its interpretations. Moreover let $I$ be an interpretation and $GA$ a ground...
atom, such that $P$ contains $k$ clauses defining the predicate in $GA$ and the clause $i$ is of the shape $GA : -B_1, \ldots, B_n$. Then, we define the operator $T_P : IS_P \rightarrow IS_P$ as $T_P(I)(GA) = \sum_{i=1}^{k}(\prod_{j=1}^{n} I(B_j))$. Whenever $B_j$ is a semiring value, its meaning $I(B_j)$ is fixed in any interpretation $I$ and it is the semiring value itself.

**Definition 16 (Partial Order of Interpretations).** Let $P$ be a program and $IS_P$ the set of all its interpretations. We define the structure $\langle IS_P, \preceq \rangle$, where $\forall I_1, I_2 \in IS_P, I_1 \preceq I_2$ if $I_1(GA) \leq I_2(GA)$ for any ground atom $GA$, where $\leq$ is the order induced by the semiring.

Note that $\langle IS_P, \preceq \rangle$ is a complete partial order and its glb coincides with the glb operation in the lattice $A$ (extended to interpretations). Since the function $T_P$ is monotone and continuous over this complete partial order, then $T_P$ has a least fix-point $lf p(T_P) = \text{glb}(\{I | T_P(I) \preceq I\})$ and it can be obtained by computing $T_P \uparrow \omega$, i.e., by applying $T_P$ to the bottom of the partial order of interpretations, and then repeatedly applying it until a fix-point.

**Example 3.** Consider again the SCLP program in Fig. 2. In the definition of the $T_P$ operator we have to consider the additive and multiplicative operations of the semiring $S_{KCSP}$, that is, the $\text{min}$ and $+$ operations. As in [4], we start the computing of the semantics from the bottom of the partial order of interpretations, $I_0$, which maps each semiring element into itself and each ground atom into $+\infty$. The table on the right of Fig. 2 shows the value associated by the interpretations to the most interesting ground atoms. The interpretation $I_4$ represents the fix-point of $T_P$. As an example, we show the computation for one of the most interesting case, the ground atom $s(a)$, which also corresponds to our goal. We said that the clause $s(X) : - p(X, Y)$ is considered equivalent to all its instantiations. Therefore, $I_4(s(a)) = \text{min}\{I_3(p(a, a)), I_3(p(a, b)), I_3(p(a, c))\} = \text{min}\{+\infty, 2, 3\} = 2$.

4 The Electric Vehicle Travel Optimization Problem

This section presents the EV travel optimization problem, introduced in [9], and shows how it can be naturally modelled and solved in the SCLP framework.

**General description of the problem.** A user has a set of appointments, each of them is in a location and has a starting time and a duration. The user makes a series of decisions regarding the sequences of trips from an appointment to another one. For example, he decides which route he wants to follow, where to park and if and how to charge the EV at the appointment location.

All possible combinations of travel choices form the choice set. A travel choice is optimal if it minimizes the users cost criteria.

In particular, finding a single optimal trip consists in finding the best trips in terms of travel time and energy consumption. Finding instead the optimal journey, that is, the optimal sequence of coupled trips, consists in finding the best journeys not only in terms of travel time and energy consumption, but
also in terms of other important criteria for the user, such as the charging cost, the number of charging events, etc. However, in finding the solution it needs to guarantee that the user reaches each appointment in time and that the state of charge (SoC) of the vehicle never falls below a predefined threshold.

In [9], the authors propose a hierarchical presentation of the e-mobility framework, which they exploit to decompose the optimization problem in sub-optimization problems. In particular, they identify four levels of mobility: the component level, whose main tasks are the inter- and intra-component coordination; the trip level, whose main task is the time and energy optimal routing; the journey level, which handles sequences of trips together with charging and parking strategies; and the mobility level, which handles mobility services, such as, car and ride service.

Each level represents a different optimization problem and the results of the lower level will be inputs of the higher level. However, since, in general, the best solution of the lower level could be not optimal for the higher one, the results of the lower level could contain several solutions and not only the best one.

In the following, we consider only the trip level and the journey level optimization problems. In particular, we first present their formalization and then we show how we can model them in the SCLP framework.

**Formalizations of the trip and journey level optimization problems.**

The trip level optimization problem substantially coincides with the multi-criteria shortest path problem. The road network is indeed represented by a directed graph $G := (N, E)$, where each arc $e \in E$ from a node $p$ to a node $q$ has associated a label $\langle c_T, c_E \rangle$, that is, a pair whose elements represent the costs, respectively in terms of time and energy consumption, of the arc from $p$ to $q$.

So, given the road network $G$, such as the one on the left of Fig. 3, a source node $n_s$ and a destination node $n_d$, the problem consists in finding all the best paths between $n_s$ and $n_d$ in terms of time and energy consumption. Note that, since the costs of the arcs are elements of a partially ordered set, the solution can contain several paths, that is, all paths which are not dominated by others, but which have different incomparable costs. For example, if we want to know the best paths from $p$ to $t$ in the graph of Fig. 3, the solution will contain both the paths $\{p, t\}$ with cost $\langle 3, 9 \rangle$ and $\{p, q, t\}$ with cost $\langle 4, 8 \rangle$. The former is indeed better in terms of time while the latter in terms of energy consumption.

As far as the journey level optimization problem is concerned, we use the formalization presented in [9]. Actually, we consider a simpler version of it, which avoids to consider car parks and the time that the users would take to go from either the car park or the charging station to the location of the appointment. Moreover, we consider only the time and energy consumption as cost criteria to be minimized. All these simplifications allows a slender and more readable presentation of the SCLP program modelling the problem.

Let $A = \{A_1, \ldots, A_n\}$ be the set of the user’s appointments. In order to describe the problem, we use different time variables. All of them have the shape $i_Y^Z$, where $i$ denotes the appointment, $Y \in \{D, A\}$ ($D$ stands for drive and $A$ for appointment), and $Z \in \{S, E\}$ ($S$ stands for start and $E$ for end).
Each appointments is defined by a location $L_i$, a starting time $i t_A^S$, an end time $i t_A^E$ and therefore a duration $i d_A$. In order to go from an appointment $A_i$ to the next one $A_{i+1}$, the user leaves with an EV from the location $L_i$ at time $i t_A^P$ and drive to location $L_{i+1}$. The user travels along the route alternative $i R_D$ (computed by the trip level problem), which consumes energy and hence reduces the SoC. Obviously, the chosen route must allow the user to arrive to destination with the SoC of his EV. We assume that the SoC always decreases during driving and increases during charging events. The user arrives at $i t_A^E$ and the appointment starts at $i t_A^S$. The user must arrive in time to the appointment, so it is required that $i t_A^E \leq i t_A^S$. During the appointment, it is also possible to schedule a charging event if the SoC of the EV is not enough to continue the journey. We assume to have a set of charging stations. Each of them is simply defined by its name $CSname$, the number of available charging spots $SpotsNum$, and the location $L$ where it is.

Therefore, given the road network $G$, a set of appointments, as the ones described in the table in the middle of Fig. 3, and a set of charging stations, as the ones in the rightmost table of Fig. 3, the problem consists in finding all the best journeys through all the appointment locations in terms of time and energy consumption. As for the travel optimization problem, also here the solution can contain several journeys, that is, all the non-dominated ones.

**SCLP programs for the optimization problems.** In the following, we show how the SCLP framework can be used as a linguistic support and a high-level and flexible programming environment where naturally modeling and solving the two optimization problems presented above.

As far as the trip level optimization problem, we propose a slightly different version of the model proposed in [5] for the multi-criteria shortest path problem. So, as there, we consider an SCLP program over the $c$-semiring denoted $P^H(S)$ which, given a source node $n_s$ and a target node $n_d$, allows us to obtain the set of the costs of all non-dominated paths from $n_s$ to $n_d$. 

---

4 This is the typical behaviour of EVs, however, as explained in [9], in particular cases it might also increase during driving and decreases during charging.
The semiring $P^H(S)$ is obtained starting from a semiring $S = \langle A, +, \times, 0, 1 \rangle$, which in our case is the one modelling the costs associated to each edge, i.e., $S = \langle N^2, \text{min}', +', (\infty, \infty), (0, 0) \rangle$, where $\text{min}'$ and $+'$ are the $\text{min}$ and $+$ operations extended to pairs. Indeed, in general we want to minimize the sum of each cost, but, since we want to obtain all the non-dominated paths, we consider $P^H(S)$.

Given a semiring $S$, we define $P^H(S) = \langle P^H(A), \cup, \times^*, \emptyset, A \rangle$, where $P^H(A)$ is the Hoare Power Domain of $A$, that is, $P^H(A) = \{ S \subseteq A | x \in S, y \leq_S x \implies y \in S \}$. These sets are isomorphic to those containing just the non-dominated values, thus, in the following, we will use this more compact and efficient representation, where each element of $P^H(A)$ will represent the costs of all non-dominated paths from a node to another one. The top element of the semiring is the set $A$ (its compact form is $\{1\}$, which in our example is $\{(0,0)\}$); the bottom element is the empty set; $\cup$ is the formal union that takes two sets and gives their union; $\times^*$ takes two sets and produces another one obtained by multiplying (using the multiplicative operation of the original semiring, in our case $+'$) each element of the first set with each element of the second one.

Note that, in the partial order induced by the additive operation of this semiring, $a \leq_{pr}(S) b$ intuitively means that for each element of $a$, there exists an element of $b$ which dominates it (in the partial order of the original semiring).

Following [5], in order to also really execute the SCLP program, we model the problem with a program in CIAO Prolog [6], a system supporting CLP, by explicitly implementing the soft framework. The program is shown in Fig. 4.

Here we consider the road network presented in Fig. 3, so we have a set of clauses modelling it. In particular, we have a set of facts modelling all the edges.
? paths(p,t,10,BestPaths).

BestPaths = [[p,t],3,9],[[p,q,t],2+2,4+4]] ?.
no ?-

Fig. 5. The output for the query paths(p,t,10,BestPaths).

of the graph. Each fact has the shape edge(n_s,n_d,[c_T,c_E]), where n_s represents the source node, n_d represents the destination node and the pair [c_T,c_E] represents the costs of the edge in term of time and energy. Note that, differently from what would happen in the pure SCLP framework, these facts (representing constraints) have the cost in the head of the clauses and not in the body. This is needed for implementing the soft framework, and in particular the two operations of the semiring.

Moreover, there are two clauses path describing the structure of paths: the upper one models the base case, where a path is simply an edge, while the lower one represents the recursive case, where a path is an edge plus another path. The head of the path clauses has the following shape path(n_s,n_d,L_N,L_V,[c_T,c_E],Lim), where n_s and n_d are respectively the source and destination nodes, L_N is the list needed to remember, at the end, all the visited nodes of the path in the ordering of the visit, L_V is the list of the already visited nodes needed to avoid infinite recursion where there are graph loops, [c_T,c_E] is used to remember the cost of the path in terms of time and energy, and finally, Lim represents the maximum amount of energy that the EV can consume. It is used to retrieve only the paths with a total cost in terms of energy equal to or less than the passed value.

The times and plus clauses are useful to model the soft framework. In particular, the first clause is useful to model the multiplicative operation of the semiring allowing us to compose the global costs of the edges together, time with time and energy with energy. The plus predicate instead mimics the additive operation and it is useful to find the best, i.e. non-dominated, paths among all the possible solutions. The plus predicate is indeed used in the body of the paths clause, which collects all the paths from a given source node to a given destination node and returns the best solutions chosen with the help of the plus predicate. So, if we want to know the best paths, in the graph of Fig. 3, from p to t with a total cost in terms of energy consumption less than or equal to 10, we have to perform the CIAO query paths(p,t,10,BestPaths), where the BestPaths variable will be instantiated with the list containing all the non-dominated paths. In particular, for each of them, the list will contain the sequence of the nodes in the path and the total cost of the path in terms of time and energy. The output of the CIAO program for this query is shown in Fig. 5.

Now, by using the SCLP program modelling the travel optimization problem, we can also show the one modelling the journey level problem. Also in this case, as before, we consider the $P^H(S)$ semiring and we propose a CIAO program, where we also model the soft framework. The CIAO program modelling the journey optimization problem is presented in Fig. 6.
We have a set of facts modelling the user’s appointments and the charging stations. In particular, for each appointment $A_i$, there is a clause \( \text{appointment}(L_i, t_{A_i}^S, id_A^i) \), while for each charging station we have a clause \( \text{chargingStation}(\text{CName}, \text{SpotsNum}, L) \).

Moreover, there are four \textit{journey} clauses describing the structure of journeys. The upper two represent the base case, while the other two represent the recursive case. The first clause models the case where a journey is simply a path with a cost in terms of energy less than or equal to the SoC of the EV. The second clause models the case where the SoC of the EV is not enough to do any path and so a charging event, incrementing the energy level, must be scheduled. The third \textit{journey} clause represents the case where a journey is a path with a cost in terms of energy less than or equal to the SoC of the EV, plus another journey. Finally, the last clause models the recursive case where a charging event is needed. In all cases we check that the paths allow the user to arrive in time.

The head of the journey clauses has the shape \( \text{journey}(L_L, L_P, L_{ChEv}, [C_T, C_E], \text{SoC}) \), where $L_L$ is the list of the locations of the appointments, $L_P$ is the list needed to remember, at the end, all the paths of the journey in the correct ordering, $L_{ChEv}$ is the list needed to

\[
\begin{align*}
\text{journey}(X, Y, [P], [T, E], \text{SoC}) :&= \\
\text{appointment}(X, T_x, D_x), \\
\text{appointment}(Y, T_y, D_y), \\
\text{timeSum}(T_x, D_x, T, Arr_T), \\
\text{Arr_T} < T_y.
\end{align*}
\]

\[
\begin{align*}
\text{journey}(X, Y, [P], [[X, ID]], [T, E], \text{SoC}) :&= \\
\text{appointment}(X, T_x, D_x), \\
\text{appointment}(Y, T_y, D_y), \\
\text{chargingStation}(\text{ID}, \text{Spots}, X), \text{Spots} > 0, \\
\text{newSoC}(\text{SoC}, D_x), \text{NewSoC}, \\
\text{timeSum}(T_x, D_x, T_t, Arr_T), \\
\text{Arr_T} < T_y.
\end{align*}
\]

\[
\begin{align*}
\text{journey}(X, Y, [P], [X], [T, E], \text{SoC}) :&= \\
\text{appointment}(X, T_x, D_x), \\
\text{appointment}(Y, T_y, D_y), \\
\text{timeSum}(T_x, D_x, T, Arr_T), \\
\text{Arr_T} < T_y.
\end{align*}
\]

\[
\begin{align*}
\text{journey}(X, Y, [P], [[X, ID], [X]], [T, E], \text{SoC}) :&= \\
\text{appointment}(X, T_x, D_x), \\
\text{appointment}(Y, T_y, D_y), \\
\text{chargingStation}(\text{ID}, \text{Spots}, X), \text{Spots} > 0, \\
\text{newSoC}(\text{SoC}, D_x), \text{NewSoC}, \\
\text{timeSum}(T_x, D_x, T, Arr_T), \\
\text{Arr_T} < T_y.
\end{align*}
\]

?- journeys([p,r,t],10,BestJourneys).
BestJourneys =  
  [[[p,r],[r,s,t]],2+(2+1),7+(3+1),[[r,csr1]]],
   [[[p,r],[r,q,t]],2+(1+2),7+(1+4),[[r,csr1]]],
   [[[p,q,r],[r,s,t]],2+1+(3+1),4+1+(3+1),[]],
   [[[p,q,r],[r,q,t]],2+1+(1+2),4+1+(1+4),[]]
].

Fig. 7. The output for the query journeys([p,r,t], 10, BestJourneys).

remember all the charging events needed to complete the journey, \([C_T, C_E]\)
represents the cost of the journey in terms of time and energy, and finally, \(SoC\)
represents the current energy level of the EV.

To make the program as readable as possible, we omit the predicates newSoC
and timeSum, useful to respectively compute the new energy level of the EV
after a charging event and the arriving time of the user to an appointment.

The plus clauses are useful to model the soft framework and they are very
similar to the ones of the trip level problem. The only difference is that here we
have to consider the charging events. Moreover, note that we reuse the times
predicate defined in the CIAO program in Fig. 4.

The journeys clause collects all the journeys through a set of locations (the
ones of the user’s appointments) and returns the best solutions chosen with the
help of the plus predicate. So, if we want to know the best journeys, in the graph
of Fig. 3, through the locations where the user has the appointments, with an
EV having an energy level equal to 10, we have to perform the CIAO Prolog
query journeys([p, r, t], 10, BestJourneys), where \(p, r, t\) are the locations of the
appointments and the BestJourneys variable will be instantiated with the list
containing all the non-dominated journeys. In particular, for each of them, the
list will contain the sequence of the paths of the journey, the total cost of the
journey in terms of time and energy, and the list of the charging events, each of
them described by the name of the charging station and its location. The output
of the CIAO program for this query is shown in Fig. 7.

5 Conclusion

In this paper we proposed the SCLP framework as a high-level declarative, ex-
cutable specification notation to model in a natural way some aspects of the
e-mobility optimization problem [9], consisting in coordinating electric vehicles
in order to overcome both energetic and temporal constraints. In particular, we
considered the trip and journey optimization sub-problems, consisting in finding
respectively the energy- and time-optimal route from one destination to
another one, and the optimal sequence of coupled trips, in terms of the same
criteria, guaranteeing that the user reaches each appointment in time. For both
the optimization problems, we provided an SCLP program in CIAO Prolog, by
explicitly implementing the soft framework, that is, the additive and the mul-
типlicative operations of the chosen semiring. The former is a slight variant of the
CIAO program proposed in [5, Section 4.4] to specify the multicriteria version
of the shortest path problem. With respect to the program proposed there, here
we implemented a different semiring, (also proposed in [5]), i.e., the one based
on the Hoare Power Domain operator, which allowed us to obtain only the best
(i.e. non-dominated) routes in terms of time and energy consumption. We thus
provided an implementation of the two operations of this semiring, by defining
two predicates modelling them. The SCLP program modelling the journey opti-
mization problem then uses the trip optimization problem results as inputs. It
is also based on the same semiring that, in this case, allowed us to find the best
journeys in terms of the two cost criteria.

As said above, the soft framework is explicitly implemented into each CIAO
program: there is for example a different plus predicate in each optimization
program we have proposed. However, it would be interesting to study a general
way to embed the soft framework in Ciao Prolog. Trivially, one could provide a
library offering a more general implementation of the operations of the semiring
of each type of problem. Most interestingly, one could instead think to provide
a meta-level implementing more efficiently the soft framework.

Differently from our solution, which allows us to obtain the set of all the opti-
mal journeys, in the mathematical model proposed in [9] a form of approxima-
tion is introduced, by considering an aggregated cost function to be optimized. Their
goal is indeed to minimize this cost function, which considers different cost crite-
rinia: besides the travel time and the consumed energy, they also take into account
the charging cost, the number of charging events, etc. In modelling the problem,
here we introduced a simplification by considering just the two main cost criteria,
that allows us a slender presentation of the work. However, it is obvious that the
SCLP programs can be easily modified to also take into account several other
cost criteria. On the other side, we preferred not to introduce any approximation
of the solution, by instead returning all optimal journeys considered equivalently
feasible. However, since the use of partially ordered structures, as in our case, can
in general lead to a potentially exponential number of undominated solutions,
sometimes it becomes crucial to keep the number of configurations as low as
possible through some form of approximation allowing us to adopt a total order.
In this case, the right solution could be to adopt a function that composes all
the criteria in a single one and then to choose the best tuple of costs according
to the total ordering that the function induces [5, Section 6.1].

As said above, our aim is mainly to propose the SCLP framework as an ex-
pressive and natural specification language to model optimization problems. We
indeed think that not only problems representing an extension of the one treated
here can be modelled by adapting the solution we presented easily enough, but
that in general our approach can be followed to model hierarchical optimization
problems. All the SCLP programs we proposed are effective only when data of
small size are considered. We are indeed conscious that the proposed encodings
cannot be used to really solve the problem on practical cases, but on the other
side, we think that CIAO represents a powerful system programming environment allowing us not only to write declarative specifications but also to reason about them.

It is therefore clear that here we do not take care of the performances of the proposed programs and that our aim is not to compare the performance with existing algorithms solving these problems. We indeed leave as future work the study of how to improve the performance of our programs. In [5, Section 8], the authors show some possible solutions that could be used towards this end, such as tabling and branch-and-bound techniques (implementable for example in ECLiPSe [1]). We however would also like to study how our programs can take advantage of the use of dynamic programming techniques based, for example, on the perfect relaxation algorithm for CSPs [11].

Finally, from a theoretical point of view, as future work, we plan to propose a more general framework based on named semiring, allowing us to give a unifying presentation of the SCSP and SCLP frameworks, providing an explicit handling of the names.

References

A Variant of Earley Deduction With Partial Evaluation

Heike Stephan and Stefan Brass

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

Abstract. We present an algorithm for query evaluation given a logic program consisting of function-free Datalog rules. It is based on Earley Deduction [4, 6] and uses a partial evaluation similar to the one we developed for our SLDMagic method [1]. With this, finite automata modeling the evaluation of given queries are generated. In certain cases, the new method is more efficient than SLDMagic and the standard Magic Set method since it can process several deduction steps as one.

1 Introduction

The goal of deductive database systems is to offer integrated systems which permit to do programming and database tasks in a single, declarative language. This would improve the current situation in which several languages are mixed, e.g. Java and SQL. While SQL is declarative and has successfully shown the advantages of declarative languages, the language used for application programming is usually non-declarative.

Whereas in earlier times deductive database research was concentrated only on recursive query evaluation, now new applications, e.g. for the semantic web, are in the focus. But even so mundane tasks as the generation of web pages must be considered if deductive databases should be used for real world application programming. In [3] we made a proposal for declarative output and also investigated how sorting can be integrated in Datalog, which is obviously important for output and also for database queries.

However, even the classical task of bottom-up query evaluation deserves more research in order to improve efficiency [2]. The new method presented in this paper loops through a sequence of states (sets of rules being processed). From one state to the next, the successor state is determined by a single database fact and the preceding rule set. At compilation time, when the database state is not yet known, partial evaluation of the program is done by using facts with abstract values. The method is based on Earley Deduction [4, 6] which exploits the similarity of context free grammar rules and rules of logic programs. The partial evaluation is similar to the one we developed for our SLDMagic method [1]. It makes the algorithm very competitive because program analysis and abstract execution can already be done at compilation time.
The algorithm is also interesting because it especially fits applications in which input must be parsed (after all, the Earley algorithm is a parsing algorithm). An input text, e.g. "abc", can be represented by Datalog facts as follows:

input(1, a, 2).
input(2, b, 3).
input(3, c, 4).
eof(4).

This is very similar to the standard difference list technique for definite clause grammars, but since Datalog has no lists we use positions in the input.

First we give some basic definitions (Section 2), then we describe the basic deduction method (Section 3) and finally present the partial evaluation (Section 4).

2 Basic Definitions

Definition 1 (Rule). A rule is a formula of the form \( A \leftarrow B_1 \land \cdots \land B_n \) where \( A \) and \( B_i, i = 1, \ldots, n \), are positive literals, i.e. atomic formulas \( p(t_1, \ldots, t_m) \) with a predicate \( p \) of arity \( m \) and terms \( t_j, j = 1, \ldots, m \). Terms are variables or constants. In the above rule, \( A \) is called the head and \( B_1 \land \cdots \land B_n \) is called the body. A rule with empty body (i.e., \( n = 0 \)) and without variables is called a fact.

In the context of deductive databases, the range restriction condition ensures that no derived fact contains variables. In the remaining paper, this condition is assumed to be satisfied for every rule.

Definition 2 (Range Restriction). A rule is range restricted iff every variable that appears in the head appears also in the body.

Definition 3 (EDB- and IDB-Predicates, Program, and Database). Predicates are partitioned into EDB ("extensional database") predicates defined by facts and IDB ("intensional database") predicates defined by rules. A logic program is a finite set of rules with an IDB predicate in the head and at least one body literal. A database is a finite set of facts with EDB predicate.

The requirement that the body of a program rule is non-empty simplifies later definitions but is no restriction: One can use a special EDB predicate true without arguments in the rule body.

Definition 4 (Answer Predicate, Goal Rule, and Query). We assume that an IDB predicate answer is distinguished as "main" predicate. It must not appear in the body of a program rule. A rule with the predicate answer in the head is called goal rule and represents a query to the logic program.

The goal of query evaluation is to determine the answer-facts which are derivable from program and database together.
3 Deduction Method

The deduction method uses sequences of states to compute facts of the answer relation.

Definition 5 (Rule Normalization). Let \( VAR \) be the set of variables of a rule \( R \), and let this set be ordered by the occurrence of its elements in \( R \): \( VAR = \{ V_i | i \in \{0, \ldots, |VAR|−1\} \} \) where \( V_j < V_k \) iff the first occurrence of \( V_j \) is before the first occurrence of \( V_k \) and \( j < k \) iff \( V_j < V_k \) (\( j,k \in \{0, \ldots, |VAR|−1\} \)). Let further \( \mathcal{X} = \{ X_i | i \in \mathbb{N} \cup \{0\} \} \) be an ordered set of variables: \( X_j < X_k \) iff \( j < k \) (\( j,k \in \mathbb{N} \cup \{0\} \)). Then rule \( R \) is normalized by substituting every \( V_i \in VAR \) by \( X_i \in \mathcal{X} \) (\( i \in \{0, \ldots, |VAR|−1\} \)).

Definition 6 (State). A state is a set of normalized rules.

Definition 7 (Selection Function). A selection function chooses for every rule \( A ← B_1, \ldots, B_n \) with \( n \geq 1 \) an index \( i \in \{1, \ldots, n\} \) (i.e. a body literal).

In every rule of a state one body literal is selected; for simplicity of presentation it is assumed that this is the leftmost body literal. However, we note that in the database context the selection function is an important optimization parameter. So a real implementation will use a selection function that tries to make use of input constants and possibly existing indexes or other database access structures.

From the rules in a state new rules are derived by two basic derivation steps that are already described in [4], a “downward” instantiation and an “upward” reduction step. In a way, this can be viewed as splitting up the SLD-resolution step, which avoids deriving rules with arbitrary length. A derived rule is first normalized before added to a state.

If the selected literal of a rule in the state unifies with the head of a program rule, an instance of the program rule is derived by renaming all variables in the program rule and applying the most general unifier of the selected literal and the program rule head to the program rule. Thus, instantiation corresponds to calling an IDB predicate as in Prolog’s four port box model. Several instances can be derived from the same rule.

Definition 8 (Instance, Instantiation). Let \( R = A ← B_1, \ldots, B_n \) be a rule of the program and \( K ← L_1, \ldots, L_m \), \( m > 0 \), a rule in the state with selected literal \( L_1 \). Let \( R' \) be the rule resulting from \( R \) by a renaming of variables so that no variable in \( R' \) occurs in a rule in the state, i.e. there is a substitution \( \theta \) so that \( R' = R \theta = A' ← B'_1, \ldots, B'_n \). A rule \( R'' \) is an instance of \( R \) iff \( L_1 \) and \( A' \) are unifiable with most general unifier \( \sigma \) and \( R'' = R' \sigma \).

A reduction is performed with a fact, either of the database or a derived one. If there is a derived rule with a selected literal that unifies with the fact, this rule is reduced by the fact and a new rule, the reduct, is created by applying the most general unifier and removing the selected literal. When the last body literal is removed by reduction, an IDB fact results. Thus, reduction is a special case of resolution with a fact. Again, one fact can be used for several reductions.
Definition 9 (Reduct, Reduction). Given a derived rule $R = A \leftarrow B_1 \land B_2 \land \ldots \land B_n$ with selected literal $B_1$ and a fact $F$ in the database or in the state, the rule $R'$ is a reduct of $R$ iff $B_1$ and $F$ are unifiable with most general unifier $\sigma$ such that $R' = (A \leftarrow B_2 \land \ldots \land B_n) \sigma$. The corresponding derivation step is called reduction, $F$ reduces $R$ to $R'$.

Definition 10 (Initial State). The initial state consists of the goal rule and all rules that can be iteratively derived by instantiation.

At a state transition, exactly one EDB fact is used to compute the successor state.

Definition 11 (Dependency-Relation of Rules). A rule $R$ depends directly on a rule $R'$ iff the selected literal in $R$ is unifiable with the head literal of $R'$.

A rule $R$ depends on a rule $R'$ with respect to a state $S$ iff there are rules $R_1, \ldots, R_n \in S$ such that $R_1 = R$, each $R_i, i = 1, \ldots, n - 1$, depends directly on $R_{i+1}$, and $R_n$ depends directly on $R'$ (note that $R'$ does not have to be contained in $S$).

Definition 12 (Successor State). Let a program $P$, a database $D$, a state $S$, and a fact $F \in D$ be given. The successor state $S'$ is constructed as follows:

1. First, $S'$ is initialized with all rules that result from reduction applied to rules in $S$ with fact $F$. If the result is empty, there is no successor state.
2. If $S'$ now contains IDB facts, reduction is applied repeatedly to rules in $S$ with facts in $S'$ and the results are inserted into $S'$ until nothing changes.
3. Then instantiation is applied iteratively to each rule $R \in S'$ with a selected IDB-literal. All instances are added to the successor state.
4. Finally, rules $R \in S$ that depend (with respect to $S$) on a rule with at least one body literal in $S'$ are copied to $S'$. The copied rules are those that still have a chance of being reduced by an IDB fact.

Definition 13 (State Sequence). States $S_1, \ldots, S_n$ form a state sequence iff every $S_{i+1}$ is the successor state for $S_i$ and a fact $F_i$ of the database, $i = 1, \ldots, n - 1$.

Definition 14 (Computed Answers). A fact $\text{answer}(c_1, \ldots, c_m)$ is computed if there is a state sequence $S_1, \ldots, S_n$ such that $S_1$ is the initial state and $\text{answer}(c_1, \ldots, c_m) \in S_n$.

There can only be finitely many different states for a given program $P$ and database $D$ for the following reasons:

- The state contains only predicates and constants occurring in the finite set $P \cup D$.
- No derived rule can become longer than the longest program rule.
- A state does not contain two rules that differ only in the names of their variables.
However, the state sequence could be cyclic, so one must check whether a newly constructed state is indeed new. Of course, optimizations are possible and subject of our further research.

**Example 1.** Let the left recursive version of the standard transitive closure program be given:

1. \( \text{path}(X_0, X_1) \leftarrow \text{edge}(X_0, X_1) \).
2. \( \text{path}(X_0, X_1) \leftarrow \text{path}(X_0, X_2) \land \text{edge}(X_2, X_1) \).

Let the database be

3. \( \text{edge}(1, 2) \).
4. \( \text{edge}(2, 3) \).

Now let the following goal rule be given:

5. \( \text{answer}(X_0) \leftarrow \text{path}(1, X_0) \).

The initial state \( S_0 \) consists of the goal rule plus rules added by instantiation:

6. \( \text{answer}(X_0) \leftarrow \text{path}(1, X_0) \). //goal [5]
7. \( \text{path}(1, X_0) \leftarrow \text{edge}(1, X_0) \). //inst. of [1] because of [6]
8. \( \text{path}(1, X_0) \leftarrow \text{path}(1, X_1) \land \text{edge}(X_1, X_0) \). //inst. of [2] because of [6]

Rule [8] also calls for instantiation but that gives again [7] and [8].

Now there is only one database fact, \( \text{edge}(1, 2) \), that leads to a successor state, and by reducing with this fact we reach state \( S_1 \):

9. \( \text{path}(1, 2) \). //Reduction of [7] with [3]
11. \( \text{path}(1, X_0) \leftarrow \text{edge}(2, X_0) \). //Reduction of [8] with [9]
12. \( \text{answer}(X_0) \leftarrow \text{path}(1, X_0) \). //Copy of [6] because of [11]
13. \( \text{path}(1, X_0) \leftarrow \text{path}(1, X_1) \land \text{edge}(X_1, X_0) \). //Copy of [8] because of [11]

Again, reduction with only one database fact, \( \text{edge}(2, 3) \), is possible and gives the state \( S_2 \):

15. \( \text{answer}(3) \). //Reduction of [12] with [14]
16. \( \text{path}(1, X_0) \leftarrow \text{edge}(3, X_0) \). //Reduction of [13] with [14]
17. \( \text{answer}(X_0) \leftarrow \text{path}(1, X_0) \). //Copy of [12] because of [16]
18. \( \text{path}(1, X_0) \leftarrow \text{path}(1, X_1) \land \text{edge}(X_1, X_0) \). //Copy of [13] because of [16]

No more reductions with database facts can be applied to rules in \( S_2 \).

**Theorem 1 (Correctness).** Let a program \( P \) and a database \( D \) be given. Every computed answer is indeed a logical consequence of \( P \cup D \).

**Proof.** This is easy: Each step (reduction and instantiation) is a logical consequence of \( P \cup D \) and the previously computed rules.
Theorem 2 (Completeness). For every ground substitution \( \theta \) such that \( \text{answer}(X_1, \ldots, X_q) \theta \) is a logical consequence of the program and the database, \( \text{answer}(X_1, \ldots, X_q) \theta \) is computed.

The completeness theorem is a corollary of the following lemma, if \( S_0 \) is the initial state and the rule considered is the goal rule.

Lemma 1. Let a program \( P \) and a database \( D \) be given. If a state \( S_0 \) contains a rule \( R = A \leftarrow B_1 \land \cdots \land B_n \) and there is a ground substitution \( \theta \) such that each \( B_i \theta \) is a logical consequence of \( P \cup D \), then there is a state sequence \( S_0, S_1, \ldots, S_m \) such that \( A \theta \) is contained in \( S_m \). Furthermore, any rule \( R' \in S_0 \) that depends on \( R \) is contained in every state \( S_1, \ldots, S_{m-1} \).

Proof. Since the \( B_i \theta \) are logical consequences of \( P \cup D \), they are contained in the least fixpoint of \( T_{P \cup D} \), and because there are no function symbols, this is reached after a finite number of iterations. The proof is by induction on the maximum (over \( i, i = 1, \ldots, n \)) of the number of steps needed to derive \( B_i \theta \) with the \( T_{P \cup D} \)-operator.

If this is 1, all \( B_i \theta \) are contained in \( D \). For proving the first step of induction, there is a second induction on \( n \) (the number of body literals in rule \( R \)). If this is 1, then \( R = A \leftarrow B_1 \), \( S_m = S_i \) is the direct successor of \( S_0 \) and contains \( A \theta \). \( S_0 \) contains all rules depending on \( R \). Now assume that the theorem is proven for rules \( R \) with \( n \) body literals \( B_i \), all of them with EDB predicate. Assume further that \( S_0 \) contains \( A \leftarrow B_1 \land B_2 \land \cdots \land B_{n+1} \). If \( B_1 \) is selected, there exists a direct successor state of \( S_0 \) with the fact \( B_1 \theta_1 \) that contains \( (A \leftarrow B_2 \land \cdots \land B_{n+1}) \theta_1 \) and all rules in \( S_0 \) depending on this rule, where \( \theta_1 \) is \( \theta \) restricted to the variables occurring in \( B_1 \). From the hypothesis of the second induction the theorem follows, and \( m = n + 1 \).

Now assume that the theorem is proven for all cases where the body literals \( B_i \theta \) are derivable after at most \( k \) steps of the \( T_{P \cup D} \)-operator. This means that \( A \theta \) can be derived after \( k + 1 \) steps and computed with a state sequence \( S_0, \ldots, S_m \) of length \( m + 1 \).

For the induction step, suppose that all \( B_i \theta \) are derivable after at most \( k + 1 \) steps of the \( T_{P \cup D} \)-operator. Again, to prove the theorem there is an induction on the number of \( B_i \) in \( R \). If this is 1, then \( R = A \leftarrow B_1 \). We consider only the case that \( B_1 \) is an IDB literal (the other case is already shown above). In this case, an instantiation is performed, so \( S_0 \) contains a rule \( B \leftarrow C_1 \land \cdots \land C_l \) where \( B \) unifies with \( B_1 \). This rule is either the new instance or a rule already present in the state and equal to the new instance. From the inductive hypothesis follows that \( B \theta_1 \) (where \( \theta_1 \) is \( \theta \) restricted to the variables occurring in \( B \)) can be computed with a state sequence \( S_0, \ldots, S_m \), and that \( S_{m-1} \) contains the rule \( R = A \leftarrow B_1 \) which depends on \( B \). Since \( B_1 \) is the only body literal, \( \theta_1 = \theta \).

Thus, in state \( S_m \) a reduction with \( B \theta \) and \( A \leftarrow B_1 \) in \( S_m-1 \) can be performed so that \( S_m \) also contains \( A \theta \).

If \( S_0 \) contains a rule \( A \leftarrow B_1 \land B_2 \land \cdots \land B_n \) where \( B_1 \) is selected and an IDB literal, again it also contains a rule \( B \leftarrow C_1 \land \cdots \land C_l \). A state sequence \( S_0, \ldots, S_{m'} \) can be computed where \( S_{m'} \) contains \( B \theta_1 \), \( S_{m'-1} \) contains \( A \leftarrow B_1 \land B_2 \land \cdots \land B_n \).
and, after a reduction with the fact $B \theta_1$, $S_{m'}$ contains $A \leftarrow B_2 \land \cdots \land B_n \theta_1$ as well as all rules in $S_{m'-1}$ depending on this rule. Finally, from both inductive hypotheses the theorem follows.

\[\square\]

4 Partial Evaluation

Especially in database context, the facts of the extensional database might not be known before execution time, and as the aim is to compile a program beforehand, an abstraction from actual data values must be developed. For this purpose, abstract values taken from an infinite set of symbolic constant values, $V$, are used instead of the data values that are known only at execution time. Constants in program rules and in the query are not substituted, so no symbolic value may occur in a program rule or in the query. Via a partial evaluation an automaton can be constructed that models the process of query evaluation.

We only need to redefine a state transition. For a given state, create a set with all selected EDB literals in the state that are not equal to each other. Two literals are considered equal if they differ only in the names of their variables. For every literal in this set there is a state transition assigned to it which is labeled with the literal. Choose a literal and substitute its variables with new symbolic values that have not yet been used elsewhere. The resulting “symbolic fact” represents all facts that could be obtained from a query to the corresponding EDB relation at execution time. Thus, a state transition can be viewed as a data retrieving interface. Now reduce all possible rules in the state with the symbolic fact and add the derived rules to a new state. If a state contains a fact of the answer relation, it is a final state. In the same way as an EDB literal with symbolic values can be viewed as a representative of a set of facts, a state with symbolic values can be viewed as a representative of a set of states that depends on the actual data values.

Example 2. Consider again the transitive closure program with the same goal rule. The database is irrelevant now, only name and arity of EDB predicates are needed. Let the set $V$ of symbolic values be \{c_0, c_1, \ldots\}. For the initial state $S_0$ there are no differences.

\[\begin{align*}
[6] \text{answer}(X_0) & \leftarrow \text{path}(1, X_0). \\
[7] \text{path}(1, X_0) & \leftarrow \text{edge}(1, X_0). \\
[8] \text{path}(1, X_0) & \leftarrow \text{path}(1, X_1) \land \text{edge}(X_1, X_0).
\end{align*}\]

In rule [7], $\text{edge}(1, X_0)$ is selected, so $X_0$ is substituted by the symbolic value $c_0$. A state transition labeled with $\text{edge}(1, X_0)$ is created, and a transition with the symbolic fact $\text{edge}(1, c_0)$ gives $S_1$:

\[\begin{align*}
[9] \text{path}(1, c_0). & \quad //\text{Reduction of } [7] \text{ with fact } [6] \\
[11] \text{path}(1, X_0) & \leftarrow \text{edge}(c_0, X_0). & \quad //\text{Reduction of } [8] \text{ with } [9] \\
[12] \text{answer}(X_0) & \leftarrow \text{path}(1, X_0). & \quad //\text{Copy of } [6] \text{ because of } [11] \\
[13] \text{path}(1, X_0) & \leftarrow \text{path}(1, X_1) \land \text{edge}(X_1, X_0). & \quad //\text{Copy of } [8] \text{ because of } [11]
\end{align*}\]
From $S_1$ the successor state $S_2$ is reached by a transition with the symbolic fact $\text{edge}(c_0, c_1)$, labeled with the literal $\text{edge}(c_0, X_0)$.

14 path(1, c_1).          //Reduction of [11] with fact
15 answer(c_1).            //Reduction of [12] with [14]
16 path(1, X_0) ← edge(c_1, X_0). //Reduction of [13] with [14]
17 answer(X_0) ← path(1, X_0).  //Copy of [12] because of [16]
18 path(1, X_0) ← path(1, X_1) ∧ edge(X_1, X_0). //Copy of [13] because of [16]

All following states resemble $S_1$ but have symbolic constants $c_2, c_3, \ldots$ instead of $c_1$. □

For programs without recursive rules, the above state construction algorithm works well. In the other cases, as in our example, there will be infinitely many states since there are infinitely many symbolic values to be used at state transitions. Nevertheless, a part of these cases can be handled by trying to find finite many equivalence classes of states. It may be noticed that states are generated which have a similar structure but different symbolic values. Similar states generate again similar states because the same derivation steps are applied to similar sets of rules. Therefore they can be combined in one equivalence class of states.

**Definition 15 (Equivalent States).** Let two states $S_1$ and $S_2$ be given. Let further SYMB$_1$ be the set of symbolic values occurring in $S_1$ and SYMB$_2$ the set of symbolic values occurring in $S_2$. $S_1$ is equivalent to $S_2$ iff a bijective mapping map from SYMB$_1$ to SYMB$_2$ exists so that the state $S_1'$ obtained by replacing every symbolic value $v$ in $S_1$ by map($v$) is equal to $S_2$.

The construction of an automaton with partial evaluation is straightforward. The states of this automaton represent equivalence classes of those states that are constructed during the derivation process. Consequently, when a state is constructed for which an equivalent state already exists, these states are fused to one state in the automaton.

**Example 3.** Consider again the transitive closure program with the goal rule and with symbolic values (Examples 1 and 2). It is visible that $S_1$ and $S_2$ are equivalent: if in $S_2$ the symbolic value $c_1$ is mapped to $c_0$, both rule sets are equal. With this, the following state transition function results:

\[
\begin{align*}
\delta(S_0, \text{edge}(1, X_0)) &= S_1 \\
\delta(S_1, \text{edge}(c_0, X_0)) &= S_1
\end{align*}
\]

□

In certain cases it happens that arbitrarily many different rules with the same structure of constants and variables but different values are accumulated in one state. In these cases the process of partial evaluation and automata construction does not terminate. A part of these cases results from tail recursive program rules. The problem with a tail recursive rule is that, starting from the last literal of the rule, arbitrarily long instantiation chains are created which have to be
kept in the state for reduction. These cases can be dealt with by introducing an additional derivation step and performing a resolution step instead of an instantiation when processing the last literal of a rule.

**Definition 16 (Extension of Deduction Method).** The algorithm described in Definition 12 is extended as follows:

1. A reduction with a fact \( F \) the predicate of which is an EDB predicate is not affected and performed as described above.
2. Reductions with IDB facts are applied repeatedly, but only to rules \( R = A \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n \) where \( n > 1 \).
3. Instantiations are only applied to rules \( R = A \leftarrow B_1 \wedge \cdots \wedge B_n \) with selected literal \( B_1 \) where the predicate of \( B_1 \) is an IDB predicate and \( n > 1 \). Otherwise, if \( n = 1 \), new rules are derived by applying a resolution step to \( A \leftarrow B_1 \) and program rules where the head literal unifies with \( B_1 \). The derivation step is therefore called last literal resolution.
4. If a rule \( R = A \leftarrow B_1 \wedge \cdots \wedge B_n \) depends on a rule \( R' \) in the successor state it is only copied to the successor state if \( n > 1 \).

The initial state consists of the goal rule and all rules that can be iteratively derived by instantiation and last literal resolution.

**Example 4.** Consider the tail recursive version of the transitive closure program:

\[
\begin{align*}
[1] & \text{path}(X_0, X_1) \leftarrow \text{edge}(X_0, X_1). \\
[2] & \text{path}(X_0, X_1) \leftarrow \text{edge}(X_0, X_2) \wedge \text{path}(X_2, X_1).
\end{align*}
\]

With the goal rule

\[
[3] \text{answer}(X_0) \leftarrow \text{path}(1, X_0).
\]

the initial state is:

\[
\begin{align*}
[4] & \text{answer}(X_0) \leftarrow \text{path}(1, X_0). & \text{//goal rule} \\
[5] & \text{answer}(X_0) \leftarrow \text{edge}(1, X_0). & \text{//last literal resolution of [4]} \\
[6] & \text{answer}(X_0) \leftarrow \text{edge}(1, X_1) \wedge \text{path}(X_1, X_0). & \text{//last literal resolution of [4]}
\end{align*}
\]

A transition with the symbolic fact \( \text{edge}(1, c_0) \) gives \( S_1 \):

\[
\begin{align*}
[8] & \text{answer}(X_0) \leftarrow \text{path}(c_0, X_0). & \text{//reduction of [6] with fact} \\
[9] & \text{answer}(X_0) \leftarrow \text{edge}(c_0, X_0). & \text{//last literal resolution of [8]} \\
[10] & \text{answer}(X_0) \leftarrow \text{edge}(c_0, X_1) \wedge \text{path}(X_1, X_0). & \text{//last literal resolution of [8]}
\end{align*}
\]

The following states are equivalent to this state. The transition function is the same as for the left recursive program:

\[
\begin{align*}
\delta(S_0, \text{edge}(1, X_0)) &= S_1 \\
\delta(S_1, \text{edge}(c_0, X_0)) &= S_1
\end{align*}
\]

\[
\square
\]
In order to achieve termination, the other cases of infinitely growing states have to be excluded for the time being. For this the notion of the schema of a rule is introduced, and the set of valid states is restricted to those states that do not contain two rules with the same schema. Porter uses a similar definition for a schema in [5].

**Definition 17 (Schema of a Rule).** Let a normalized rule \( R \) be given, and let \( \text{CONST}_V \) be the set of those constants in \( R \) that do not occur in the program, which means they are symbolic values of the set \( V \). Let \( \text{CONST}_V \) be ordered by the occurrence of its elements in \( R \): \( \text{CONST}_V = \{ c_i \mid i = 0, \ldots, |\text{CONST}_V| - 1 \} \) where \( c_j < c_k \) iff the first occurrence of \( c_j \) is before the first occurrence of \( c_k \) and \( j < k \) iff \( j, k \in \{0, \ldots, |\text{CONST}_V| - 1\} \). Let further \( B = \{ b_i \mid i \in \mathbb{N} \cup \{0\} \} \) be an ordered set of constants: \( b_j < b_k \) iff \( j < k \) \((j,k \in \{\mathbb{N} \cup \{0\}\})\). The schema of \( R \) is the rule obtained by replacing every \( c_i \in \text{CONST}_V \) by \( b_i \in B \) \((i \in \{0, \ldots, |\text{CONST}_V| - 1\})\).

**Example 5.** Consider state \( S_1 \) of the tail recursive program in Example 4. The schemata of the rules in this state are as follows:

\[
\begin{align*}
[7] & \text{answer}(b_0). \\
[8] & \text{answer}(X_0) \leftarrow \text{path}(b_0, X_0). \\
[9] & \text{answer}(X_0) \leftarrow \text{edge}(b_0, X_0). \\
[10] & \text{answer}(X_0) \leftarrow \text{edge}(b_0, X_1) \land \text{path}(X_1, X_0).
\end{align*}
\]

\( \square \)

**Definition 18 (Valid State).** Let \( S \) be a state and \( \text{SCH} \) be the set of schemata of the rules in \( S \). The state \( S \) is valid iff \( |\text{SCH}| = |S| \).

For a given program that meets all requirements mentioned above there are only finitely many possibilities to create valid states, so partial evaluation is guaranteed to terminate.

An implementation of the automaton will use states where the symbolic values are replaced by assignable variables that hold the actual data values. Explicit constants can be included in the target code and no derivations have to be performed so that the runtime states are very compact and the main task at state transitions should be accessing and selecting the data. Different results for the \textit{answer} predicate can be obtained by backtracking or by concurrent processing of alternative transitions.

## 5 Conclusion

We have presented an algorithm for efficient query evaluation and preprocessing of function-free logic programs based on Earley Deduction. The algorithm can process non-recursive, left- and tail-recursive rules and has been proven to be correct, complete, and terminating. While Earley Deduction can in principle be used for arbitrary logic programs, still the basic algorithm presented here is already an improvement to it because it processes several derivations for one
fact in one step. There is a special optimization potential when it can be proven that only one fact is applicable in a state and we do not have to check whether there is a cycle in the state sequence. Subjects of our future research include further optimizations for special applications and an efficient implementation of the generated automaton.

Further material, including a demo program (written in SWI Prolog) showing the state sequences for a given program and query, is available at http://dbs.informatik.uni-halle.de/Earley.

References

Causes for Checking the Integrity of the Evolution of Databases

Hendrik Decker

Instituto Tecnológico de Informática, UPV
Ciudad Politécnica de la Innovación, 46022 Valencia, Spain
Tel.: +34 96 3877 244, Fax: +34 96 3877 239, email: hendrik@iti.upv.es

Abstract. Inconsistency in large database systems is commonplace and therefore must be controlled in order to not get out of hand. Consistency in database systems is encoded by integrity constraints. Inconsistency thus corresponds to constraint violations. Database system services need to function in spite of extant integrity violations, but inconsistency should not increase beyond control in the course of the evolution of such systems. Evolution is effected by updates that may involve insertions and deletions of relational facts as well as schema updates. We show how to determine the causes of violations. Knowledge about such causes can be used to control inconsistency: an increase of integrity violations by updates can be prevented, while tolerating extant inconsistencies, even if the database schema is altered, and even if the schema is unsatisfiable.

1 Introduction

Whenever a database or its schema is updated, its integrity should be checked. In case the update would cause the violation of an integrity constraint, it should either be rejected, or a warning should be issued such that the user or the application may modify the update, so that consistency will be preserved.

However, integrity is not always checked, be it out of neglect, or for trading off consistency for performance, or due to the incompatibility of legacy data with constraints, or for any other reason. Hence, mechanisms for tolerating extant inconsistencies are needed, while an increase of integrity violations by updates should be avoided. Unfortunately, conventional integrity checking methods are not meant to be applied in the presence of inconsistency. They all require that integrity remains satisfied in the updated state.

Example 1. Consider the constraints \( I = \leftarrow \text{married}(x, y), \text{male}(x), \text{male}(y) \) and \( I' = \leftarrow \text{married}(x, y), \text{female}(x), \text{female}(y) \) in a database system \( D \) of a civil registry office. They deny cosexual marriages. Yet, suppose \( D \) contains the fact \( \text{married}(\text{jim}, \text{pat}) \), where \( \text{jim} \) and \( \text{pat} \) are both male, perhaps due to a gender reversal after nuptials. Clearly, that amounts to a violation of \( I \). Thus, conventional integrity checking methods are no longer applicable for updates of \( D \),
since they require that \( D \) does not violate integrity before checking any update for integrity preservation. Obviously, that is unnecessarily restrictive, in general. For instance, consider the update \( U = \text{insert married}(joe, susan) \), which should be acceptable, because it is independent of the extant violation of \( I \) and does not cause any new violation.

Similarly, also schema updates, such as, e.g., inserting the denial constraint \( I'' = \leftarrow \text{married}(x, y), \text{married}(x, z), x \neq z \), which forbids bigamy, is traditionally not considered to be efficiently checkable in the presence of extant constraint violations, even if \( I'' \) is not violated by any tuple in \( D \).

In [14, 15], we have shown that, contrary to common belief, most, though not all integrity checking methods are inconsistency-tolerant, i.e., the total satisfaction requirement can simply be waived without incurring any penalty. In other words, there is no problem with checking updates such as \( U \) in Example 1 in the presence of inconsistency, provided the used method is inconsistency-tolerant.

More precisely, we show in [14, 15] that each inconsistency-tolerant method guarantees the preservation of the satisfaction of all ‘cases’, i.e., instances of constraints that have been satisfied in the old state, even if other instances are violated. Hence, such methods make sure that inconsistency never increases across checked updates of base facts and view definitions, while tolerating extant inconsistencies.

In [6], we have shown that case-based inconsistency tolerance is possible also for schema updates. There, we have also seen that integrity violation can be controlled by inconsistency-tolerant methods if new or modified constraints are checked only against current and future updates, but not against legacy data. Moreover, we have shown in [6] that inconsistency tolerance can guarantee a reliable handling of hard and soft constraints through schema updates.

In [8], we have shown that, for definite databases and constraints, essentially the same guarantees of integrity preservation can be made if, instead of cases of constraint violations, the increase of causes of violations across updates of base facts and view definitions is checked. Causes as defined in [8] are minimal sets of database clauses, the presence or absence of which is responsible for integrity violations. As opposed to cases, causes also provide a basis for computing answers that have integrity in spite of extant violations, as shown in [10].

In [9], we have shown that cause-based inconsistency tolerance is also possible for updates of relational databases that involve schema updates, and that all advantages obtained by case-based inconsistency-tolerant methods for schema updates also are achieved by cause-based methods. However, the minimality condition for causes as defined in [8, 9] does not scale up to non-monotonic negation. For instance, \( \emptyset \) but not \( \neg q \) would be a cause for the answer no to the query \( \leftarrow p \) in \( D = \{ p \leftarrow q \} \).

In this paper, we are going to see that the generalization of causes in [10] to justify the integrity of positive and negative answers in definite and non-definite databases works as well also for controlling the evolution of integrity, i.e., for updates including schema alterations. Moreover, the generalized definition in [10] is formalized in a much simpler and less circumstantial manner in this paper.
2 Preliminaries

Our terminology and notation largely adhere to *datalog* [1, 16]. In particular, for each database $D$, the well-known completion [4] of $D$ is denoted by $\text{comp}(D)$, and $\models$ denotes logical consequence (more precisely, truth in all Herbrand models). We assume a universal underlying language $\mathcal{L}$, of which $\mathcal{L}^c$ be the set of constant terms in $\mathcal{L}$ and $\mathcal{H}_\mathcal{L}$ the Herbrand base of $\mathcal{L}$.

In 2.1, we recapitulate some database fundamentals. In 2.2, we characterize the updates that we are going to deal with. In 2.3, we abstractly define the concept of integrity checking as implemented by conventional integrity checking methods. Later, in 3.3, that definition will be generalized in terms of inconsistency tolerance, based on a concept of causes, to be introduced in Section 3.

2.1 Basics

Each database schema (in short, schema) consists of a set $T$ of table definitions, a set $V$ of view definitions and a set $IC$ of integrity constraints, a.k.a. integrity theory. Each table in $T$ uniquely corresponds to a base predicate with some arity. Each view in $V$ corresponds to a view predicate, defined by a set of clauses of the form $A \leftarrow B$ where $A$ is an atom and $B$ a conjunction of literals with predicates that recur on base predicates. Each constraint in $IC$ is a first-order predicate logic sentence. Unless explicitly stated otherwise, each constraint be represented in the form of a denial, i.e., a clause without head whose body is a conjunction of literals.

A database is a pair $D=(S,E)$ where $S$ is a schema and $E$ is an extension of the table definitions in $S$, i.e., a set of ground base predicate facts. As usual, we assume that, for each schema $S=(T, V, IC)$ and each database $D=(S,E)$, $V \cup E$ determines a unique minimal Herbrand model, sometimes called the standard model of $D$.

For a database $D$, an integrity theory $IC$ and $I \in IC$, let $D(IC) = \text{sat}$ (resp., $D(I) = \text{sat}$) denote that $IC$ (resp., $I$) is satisfied in $D$, and $D(IC) = \text{vio}$ (resp., $D(I) = \text{vio}$) that it is violated. ‘Consistency’ and ‘inconsistency’ are synonymous to ‘satisfied’ and, resp., ‘violated’ integrity.

From now on, let $D$ always denote a database, $IC$ the integrity theory of the schema of $D$, $I$ a constraint in $IC$, $E$ the extension of $D$, and $U$ an update.

We may use ‘;’ as a delimiter between elements of sets, for avoiding confusion with the use of ‘,’ which also symbolizes conjunction of literals in the body of clauses.

2.2 Updates

Updates map databases to databases. For a database $D$ with view definitions $V$ and an update $U$, let $D^U$ denote the database into which $D$ is mapped by $U$, $V^U$ the views of $D^U$ and $IC^U$ the integrity theory of $D^U$. $D$ is also called the old database and $D^U$ the new database. If $IC^U = IC$, $U$ is called a conventional update. Let $U_v$ denote the maximal subset of $U$ that is a conventional update. Moreover, if $V^U = V$ and $IC^U = IC$, $U$ is called an extensional update.
2.3 Integrity Checking

From now on, let $\mathcal{M}$ always denote an integrity checking method (in short, method). Each $\mathcal{M}$ can be formalized as a function that maps pairs $(D, U)$ to $\{\text{ok}, \text{ko}\}$, where $\text{ok}$ means that $\mathcal{M}$ sanctions $U$ and $\text{ko}$ that it does not. The computation of $\mathcal{M}(D, U)$ usually involves less access to $E$, and thus is more efficient than a brute-force check, i.e., a plain evaluation of all constraints against the entire database, without any simplification.

For simplicity, we only consider methods $\mathcal{M}$ and classes of pairs $(D, U)$ such that the computation of $\mathcal{M}(D, U)$ terminates. That can always be achieved by a timeout mechanism with output $\text{ko}$.

Below, Definition 1 captures the conventional concept of soundness and completeness of integrity checking methods for updates including schema updates.

**Definition 1. (Conventional Integrity Checking)**

$\mathcal{M}$ is called sound or, resp., complete if, for each $(D, U)$ such that $D(\text{IC}) = \text{sat}$, the implication (1) or, resp., (2) holds.

\[ \mathcal{M}(D, U) = \text{ok} \implies D^U(\text{IC}^U) = \text{sat} \quad (1) \]

\[ D^U(\text{IC}^U) = \text{sat} \implies \mathcal{M}(D, U) = \text{ok} \quad (2) \]

Both (1) and (2) relate the output $\text{ok}$ of $\mathcal{M}$ to integrity satisfaction. We omit symmetric relationships of $\text{ko}$ and integrity violation, since soundness and, resp., completeness for $\text{ko}$ and violation is equivalent to (2) and, resp., (1).

**Example 2.**

Let $\text{IC} = \{I, I'\}$ and $U$ be as in Example 1, and suppose that $D(\text{IC}) = \text{sat}$. Most methods $\mathcal{M}$ evaluate the simplified constraints $\leftarrow \text{male}(\text{joe})$, $\text{male}(\text{susan})$ and $\leftarrow \text{female}(\text{joe})$, $\text{female}(\text{susan})$, and output $\mathcal{M}(D, U) = \text{ok}$ if $\text{joe}$ is registered as male and $\text{susan}$ as female (and, curiously, also if $\text{joe}$ would be registered as female and $\text{susan}$ as male). Otherwise, $\mathcal{M}(D, U) = \text{ko}$.

3 Causes

In this section, we first revisit the definitions of causes for integrity violation in definite databases [8] and for constraints with negation in relational databases [9]. Then, we extend them to non-definite databases and constraints by simplifying a previous generalization in [10].

Informally speaking, causes are minimal explanations of why an answer is given or why a constraint is violated. In 3.1, we formalize this idea in a homogeneous form for definite databases, queries and constraints, as well as for constraints with negation in relational databases. In 3.2, we generalize the preceding definitions for ‘normal’ deductive databases, queries and constraints with non-monotonic negation in the body of clauses, as well as for negative answers. In 3.3, we outline the use of causes for inconsistency-tolerant integrity checking, which is the cornerstone for an inconsistency-tolerant control of the evolution of databases, and particularly of schema updating, as addressed in Section 4.
3.1 Simple Causes

For the definition below, recall that the violation of a constraint of the form $\leftarrow B$ corresponds to a non-empty answer to the query $\leftarrow B$.

**Definition 2.** (Causes for Definite Databases, Queries and Constraints)
Let $V$ be the view definitions and $E$ the extension of a database $D$, $V^*$ the set of all ground instances of $V$, $B$ a conjunction of atoms and $\theta$ an answer to $\leftarrow B$ in $D$. A subset $C$ of $V^* \cup E$ is called a *cause* of $\theta$, and also a *cause* of $\forall(B\theta)$ in $D$, if $C \models \forall(B\theta)$ and there is no proper subset $C'$ of $C$ with that property. If $\leftarrow B$ is a constraint, we also call $C$ a *cause of the violation* of $\leftarrow B$ in $D$.

**Example 3.**
Let $D$ be as in Example 1. Clearly, \{married(jim, pat), male(jim), male(pat)\} is a cause of the violation of $I$ in $D$.

Causes according to Definition 2 can be computed straightforwardly, and essentially come for free, by tracing SLD refutations of denials [10]. However, due to the non-monotonicity of database negation, both the definition and the computation of causes become much more involved for causes of negative answers, and for causes of answers and constraint violations in databases with clauses that may contain negative literals in their body.

Small first steps into that direction had been taken in Section 6 of [8] and [9], for negative answers as well as queries and constraints with negative literals, but only for flat relational databases. A variant of the corresponding definition which likens it more to Definition 2 is reproduced below.

**Definition 3.** (Causes for Negation in Relational Databases)
Let $D$ be a relational database, $B$ a conjunction of literals and $\theta$ an answer to $\leftarrow B$ in $D$. A set $C$ of ground literals such that $\text{comp}(D) \models C$ is called a *cause* of $\theta$, and also a *cause* of $\forall(B\theta)$ in $D$, if $C \models \forall(B\theta)$ and there is no proper subset $C'$ of $C$ with that property. If $\leftarrow B$ is a constraint, we also call $C$ a *cause of the violation* of $\leftarrow B$ in $D$.

**Example 4.** Let $I^\sim = \leftarrow \text{married}(x,y), \text{male}(x), \neg \text{female}(y)$ and $D$ as in Example 1. Clearly, a cause of the violation of $I^\sim$ in $D$ is \{married(jim, pat), male(jim), \neg female(pat)\}.

In fact, the definition in [8] is somewhat more general than Definition 3, in that $\leftarrow B$ is replaced by an arbitrary first-order sentence. Yet, for queries, the definition remains fairly bland, since each answer in a flat relational database essentially explains itself. Thus, the computation of causes in relational databases is even simpler than the computation of causes in the definite case. However, Definition 3 is as interesting as Definition 2 for explaining the violation of constraints, since the facts that cause violations can be used for explanations [11] and repairs [12].

37
3.2 General Causes

Due to the non-monotonicity of database negation, the definitions in 3.1 unfortunately do not scale up to explain negative answers, nor to explain answers and constraint violations in normal databases, queries and constraints, i.e., negative literals may occur in the body of clauses. An extension of the definition of causes in [8] in order to cope with database negation had first been proposed in [10]. Below, we present a less complicated but equivalent version of that extension.

First, we recall that \( \text{comp}(D) \) essentially consists of the if-and-only-if completions (in short, completions) of all predicates in \( \mathcal{L} \). For a predicate \( p \) in \( \mathcal{L} \), let \( p_D \) denote the completion of \( p \) in \( D \).

**Definition 4.** Let \( D \) be a database, \( p \) a predicate in \( \mathcal{L} \), \( n \) the arity of \( p \), \( x_1, \ldots, x_n \) the \( \forall \)-quantified variables in \( p_D \) and \( \theta \) a substitution of \( x_1, \ldots, x_n \). For \( A = p(x_1, \ldots, x_n)\theta \), the completion of \( A \) in \( D \) is obtained by applying \( \theta \) to \( p_D \) and is denoted by \( A_D \). Further, let \( \text{comp}(D) = \{ A_D | A \in \mathcal{H}_L \} \), and \( \text{iff}(D) \) and \( \text{only-if}(D) \) be obtained by replacing \( \leftrightarrow \) in each \( A_D \in \text{comp}(D) \) by \( \leftarrow \) and, resp., \( \rightarrow \). Finally, let \( \text{iff}(D) = \text{iff}(D) \cup \text{only-if}(D) \). The usual equality axioms of \( \text{comp}(D) \) that interpret \( = \) as identity be associated by default also to \( \text{iff}(D) \).

Clearly, \( \text{iff}(D) \) is equivalent to the set of all ground instances of clauses in \( D \). Moreover, \( \text{comp}(D) \), \( \text{comp}(D) \) and \( \text{iff}(D) \) clearly have the same logical consequences. However, the characterization of causes in Definition 5 below by minimal subsets of \( \text{iff}(D) \) is more precise than it could be if subsets of \( \text{comp}(D) \) were used instead.

**Definition 5.** (Causes for Normal Databases, Queries and Constraints)

Let \( D \) be a normal database, \( B \) a conjunction of literals and \( \theta \) an answer to \( \leftarrow B \) in \( D \). A subset \( C \) of \( \text{iff}(D) \) is called a cause of \( \theta \), and also a cause of \( \forall(B\theta) \) in \( D \), if \( C \models \forall(B\theta) \) and there is no proper subset \( C' \) of \( C \) with that property. If \( \leftarrow B \) is a constraint, we also call \( C \) a cause of the violation of \( \leftarrow B \) in \( D \).

For easy reading, we may represent elements of \( \text{only-if}(D) \) in a simplified form, in subsequent examples. Simplifications are obtained by replacing ground equations with their truth values and by common equivalence-preserving rewritings for the composition of subformulas with true or false. Also, we may identify \( D \) with \( (V, E) \), while representing constraints or integrity theories explicitly, and omitting \( T \).

**Example 5.**

a) Let \( D = \{ p(x) \leftarrow q(x), r(x); q(1); q(2); r(2); s(1); s(2) \} \). The only cause of the violation of \( \leftarrow s(x), \sim p(x) \) in \( D \) is \( \{ s(1); p(1) \rightarrow q(1) \land r(1); \sim r(1) \} \).

b) Let \( D = \{ p \leftarrow q(1, x); q(2, y) \leftarrow r(y); r(1) \} \). The only cause of the violation of \( \leftarrow \sim p \) in \( D \) is \( \{ p \rightarrow \exists x q(1, x) \} \cup \{ \sim q(1, i) | i \in \mathcal{L}^c \} \).

c) Let \( D = \{ p \leftarrow q(x, x); q(x, y) \leftarrow r(x,y); r(1); s(2) \} \). Each cause of \( \sim p \) in \( D \) contains \( \{ p \rightarrow \exists x q(x, x) \} \cup \{ q(i, i) \rightarrow r(i) \land s(i) | i \in \mathcal{L}^c \} \cup \{ \sim r(2); \sim s(1) \} \) and, for each \( j > 2 \) in \( \mathcal{L}^c \), either \( \sim r(j) \) or \( \sim s(j) \), and nothing else.
d) Let \( D = \{ p \leftarrow q; p \leftarrow \neg q \} \), \( D' = \{ p \leftarrow q; p \leftarrow \neg q; q \} \) and \( I = \leftarrow p \). Clearly, \( D \) is a cause of the violation of \( I \) in \( D \) and in \( D' \). Another cause of \( p \) in \( D \) is \( \{ p \leftarrow \neg q; \neg q \} \). Another cause of \( p \) in \( D' \) is \( \{ p \leftarrow q; q \} \).

e) Let \( D = \{ p \leftarrow \neg q; q \leftarrow r; r \leftarrow s; r \leftarrow \neg t \} \). The two causes of the violation of \( \leftarrow p \) in \( D \) are \( \{ p \leftarrow \neg q; q \rightarrow \neg r; r \leftarrow \neg s; \neg s \} \) and \( \{ p \leftarrow \neg q; q \rightarrow \neg r; r \leftarrow \neg t; \neg t \} \).

f) Let \( D = \{ p(x) \leftarrow r(x); r(1) \} \) and \( I = \exists x(r(x) \land \neg p(x)) \). A denial form of \( I \) is \( \leftarrow \text{vio} \), where \( \text{vio} \) is defined by \( \{ \text{vio} \leftarrow \neg q; q \leftarrow r(x), \neg p(x) \} \), where \( q \) is a fresh 0-ary predicate. Thus, the causes of the violation of \( I \) in \( D \) are the causes of \( \text{vio} \) in \( D' = D \cup \{ \text{vio} \leftarrow \neg q; q \leftarrow r(x), \neg p(x) \} \). Thus, for each \( K \subseteq \mathcal{L} \) such that \( 1 \in K \), \( \{ \text{vio} \leftarrow \neg q \} \cup \{ p(i) \leftarrow r(i) \mid i \in K \} \cup \{ q \rightarrow \exists x(r(x) \land \neg p(x)) \} \cup \{ \neg r(i) \mid i \notin K \} \) is a cause of \( \text{vio} \) in \( D' \).

### 3.3 Cause-based inconsistency tolerance

By common sense, an update \( U \) should be rejected only if \( U \) would cause the violation of a constraint, independent of the presence or absence of violations that are independent of \( U \). However, conventional methods require that all constraints be satisfied before any update could be checked for integrity preservation.

**Example 6.** As in Example 1, let us assume a database \( D \) with constraints for preventing bigamy, married minors and other unlawful relationships. Yet, \( D \) may contain entries of persons married twice, underage spouses, etc. Such integrity violations may be due to omissions (e.g., an unrecorded divorce), neglect (e.g., integrity checking switched off for schema evolution) or other irregularities (e.g., a changed marriage legislation). Although new marriages that satisfy all constraints can be entered without problems into \( D \), such updates have traditionally been considered to be not checkable by conventional methods if \( D \) contains facts that violate integrity.

As opposed to conventional approaches, cause-based integrity checking, as defined below, is inconsistency-tolerant, since it may sanction updates that do not increase the amount of causes of integrity violation, no matter how high the amount of inconsistency is before the update is executed. As we are going to see in Example 11, Definition 6 entails that even each unsatisfiable schema is tolerable, for updates that do not violate any constraint.

**Definition 6.** (Cause-based Integrity Checking)

a) Let \( cv(D, I) = \{ C \mid C \) is a cause of violation of \( I \) in \( D \} \) denote the set of all causes of the violation of \( I \) in \( D \), modulo renamings of variables.

b) A method \( M \) is called sound and, resp., complete for cause-based inconsistency-tolerant integrity checking if, for each tuple \( (D, U) \), implication 3 or, resp., 4 holds.

\[
M(D, U) = \text{ok} \Rightarrow cv(D^U, I) \subseteq cv(D, I) \quad \text{for each } I \in IC \tag{3}
\]

\[
\text{if } cv(D^U, I) \subseteq cv(D, I) \text{ for each } I \in IC \Rightarrow M(D, U) = \text{ok} \tag{4}
\]
Definition 6 is going to be illustrated by examples 7 and, later, 8–12. Examples 7 and 12 feature extensional updates. Examples 8–10 are schema updates involving changes of tables, view definitions or integrity constraints. Example 12 also features the method in [17], which is not inconsistency-tolerant.

**Example 7.** For a table \( p \), \( I = \leftarrow p(x, y), p(x, z), y \neq z \) is a primary key constraint on the first column of \( p \). Updates such as \( U = \text{insert} \ p(a, b) \) can be accepted by each cause-based inconsistency-tolerant method \( M \), unless another \( p \)-tuple with the same key, e.g., \( p(a, c) \), is in the database. In that case, \( \{p(a, b), p(a, c)\} \) would be a new cause that violates \( I \), and \( M \) would have to reject \( U \). Otherwise, \( M \) can accept \( U \), independent of any extant violation of \( I \), e.g., by stored facts \( p(b, b) \) and \( p(b, c) \). Then, \( \{p(b, b), p(b, c)\} \) is a cause of the violation of \( I \), but no new cause of the violation of \( I \) is introduced by \( U \). Since \( U \) does not increase the amount of causes of inconsistency, \( U \) can be accepted.

In [15], *case-based* inconsistency-tolerant integrity checking is studied. Case-based methods make sure that the amount of violated cases of constraint does not increase across updates. In [14, 15], we have shown that many, though not all methods in the literature are sound for case-based inconsistency tolerance. An analogous result for cause-based methods is

**Theorem 1.** Each method shown to be sound for case-based integrity checking in [14, 15] also is sound for cause-based inconsistency-tolerant integrity checking of conventional updates.

**Proof.** Theorem 1 follows from the more general result that each case-based method whatsoever is cause-based for conventional updates and updates of view definitions. That has been shown for definite databases and denials in [8], but that proof generalizes straightforwardly to normal databases and arbitrary integrity constraints.

In the remainder of this paper, we are going to show that the results for checking arbitrary schema updates with case-based methods in [6] continue to hold also for cause-based methods.

### 4 Inconsistency-tolerant Schema Updating

In 4.1 we are going to show that cause-based methods that are inconsistency-tolerant for checking conventional updates in normal databases with normal constraints can also be used for schema updates involving changes in the integrity theory. We illustrate that result for table alterations in 4.2, for updates of integrity theories in 4.3, and for updates of view definitions in 4.4. In 4.3, we also propose an inconsistency-tolerant checking policy that makes schema updates more efficient. In 4.4, we also show that cause-based integrity checking even tolerates unsatisfiable schema definitions. In 4.5, we show that cause-based inconsistency tolerance is applicable to control the preservation of integrity for safety-critical schema updates in evolving databases.
4.1 Cause-based Checking of Schema Updates

Most conventional methods for integrity checking in deductive databases are conceived for updates that may not only involve conventional updates but also insertions or alterations of integrity constraints. (No checking is needed for the deletion of constraints since that may never cause any integrity violation.) However, to check new or altered constraints cannot take advantage of the incrementality of checking updates of base facts or view definitions. Rather, new or altered constraints usually are evaluated brute-force against the updated state, in order to check whether they are violated or not.

The following result states that methods that are cause-based for conventional updates continue to be inconsistency-tolerant according to Definition 6 for arbitrary schema updates, involving table alterations, updates of view definitions and integrity theories.

**Theorem 2.**
Each method \( \mathcal{M} \) that is sound for cause-based integrity checking of conventional updates also is sound for cause-based integrity checking of schema updates.

**Proof.** For updates that involve only base facts and view definitions, the result follows from the generalization of the result in [8], as mentioned in the proof of Theorem 1. For updates involving changes in the integrity theory, the result can be shown by verifying (3) and (4) of Definition 6 by induction on the number of constraints inserted by \( U \), assuming that each \( I \in I^U \setminus IC \) is evaluated brute-force in \( D^U \).

Theorem 2 addresses the soundness of cause-based methods for schema updates. With regard to scaling up the completeness of cause-based methods from conventional updates to schema updates including modifications of the integrity theory, a somewhat surprising result is going to be presented in 4.3.

In the following subsections, we illustrate Theorem 2 by examples of checking various kinds of schema updates with cause-based methods: table alterations in 4.2, modifications of integrity theories in 4.3, alterations of view definitions and updates of databases with an unsatisfiable schema in 4.4, and updates by safety-critical applications in 4.5.

4.2 Inconsistency-tolerant table alterations

Example 8 is going to illustrate that the amount of causes of violation never increases whenever any table alteration is checked by an inconsistency-tolerant method. It may even decrease, since some causes of violation may disappear by altering tables. Thus, cause-based inconsistency-tolerant methods support inconsistency-tolerant partial repairs [12], i.e., updates that reduce the amount of violated causes, while surviving violations are tolerated.
Example 8. Let $I = \langle q(x, x, y) \rangle$ constrain $q$ to be void of tuples the first and second arguments of which coincide. Let $q(a, a, a), q(a, a, b)$ be the only facts in $D$ that match $q(x, x, y)$. Clearly, each of them is a cause of the violation of $I$. Hence, conventional integrity checking refuses to handle the update $U$ which alters the definition of $q$ by swapping the second and third columns. As opposed to that, each complete cause-based inconsistency-tolerant method $M$ will admit $U$ and output $ok$ if $D$ contains no fact matching $q(x, y, x)$ such that $x \neq a$, and $ko$ otherwise (e.g., if $q(b, a, b) \in D$). Thus, $U$ does not only contain, but even diminish $cv(D, I)$.

4.3 Inconsistency-tolerant integrity updates

As seen in 4.1, inconsistency tolerance of each cause-based method $M$ for arbitrary schema updates $U$ in databases $D$ can be achieved by computing $M(D, U_c)$ and evaluating each inserted or modified constraint brute-force against $D^U$. (For deleted constraints, no checking is needed. For example, to delete $I$ and $I'$ in Example 1 clearly cannot cause any integrity violation.)

Example 9. Assume $U$ and $IC = \{ I \}$ as in Example 7, and let $U'$ consist of $U$ and the insertion of the constraint $I' = \langle p(x, x) \rangle$. Further, let us assume that $U$ does not cause any violation of $I$. Thus, a brute-force evaluation of $I'$ against $D^U$ will result in $M(D, U) = ok$ if $D(I') = sat$, and $M(D, U) = ko$ if not, e.g., if $(c, c)$ is a row in $p$.

Yet, brute-force evaluation is inefficient, inflexible and possibly unfeasible. For instance, if, in Example 9, $(c, c)$ is a row in $p$, then the cause $p(c, c)$ of the violation of $\langle p(x, x) \rangle$ in $D^U$ is not tolerated if $I'$ is checked brute-force. Of course, that is sound, since $U'$ would increase the amount of violations.

However, in a database $D = ((T, V, IC), E)$ that is operational while undergoing schema updates, it may be impossible to delay operations until inserted constraints are evaluated entirely against possibly huge volumes of legacy data. Then, it should be advantageous to check an arbitrary schema update $U$ by computing $M(((T, V, IC^U), E), U_c)$. That is, inserted constraints are not evaluated brute-force, but are just checked against the update. Thus, the current and all future updates are prevented from introducing new causes of violation, at the expense of having to tolerate possible violations of inserted constraints by legacy data. If needed, they can be dealt with at less busy times, e.g., during night runs.

The following theorem reveals another advantage of using this policy: it may achieve completeness, while brute-force checking of inserted constraints cannot.

**Theorem 3.**
No method $M$ that checks inserted constraints brute-force is complete for cause-based integrity checking.

*Proof.* If, in Example 9, no fact matching $p(a, y)$ is in $D$ and $p(c, c)$ is the only fact that violates $I'$, then, for each $M$ that evaluates $I'$ brute-force, $M(D, U) = ko$ holds. However, condition (4) in Definition 6 warrants the output $ok$, since $U_c = U$ clearly does not cause any violation of $IC^U$. Hence, $M$ is not complete.
Example 10. (Example 9, continued).
If $M$ checks $U'$ by computing $M(((T,V,IC^{U'}),E), U'_c)$, it will output \textit{ok}, since $U'_c = U$ and $I'$ is not relevant for $U$. Thus, causes such as $p(c,c)$ of the violation of $I'$ in $D$ are tolerated by $M$.

Clearly, checking inserted constraints only with regard to $U_c$ is much more efficient than evaluating them against the whole database. By the way, the latter is advocated in the SQL99 standard (cf. [19]), while the former lacks standardization, but is common practice.

4.4 View Modification and Unsatisfiability

Cause-based methods guarantee that all cases satisfied in the old state will remain satisfied in the new state, and that the set of causes of constraint violations does not grow larger. In fact, that continues to hold even if the given schema is unsatisfiable. Thus, inconsistency-tolerant methods are also unsatisfiability-tolerant. Hence, also the standard premise that the schema be satisfiable can be waived. We illustrate that by the following example where the modification of a view causes the unsatisfiability of the schema.

Example 11. Let \{p(x,y) ← q(x,y), q(y,z); q(x,y) ← r(x,y); q(x,y) ← s(x,y)\} be the view definitions in $D$, and $IC = \{← p(x,x); \exists x,y r(x,y)\}$. Clearly, the schema is satisfiable. Let $(a,a), (a,b)$ be all tuples in $s$. Thus, $← p(a,a)$ is a violated case. There are no more violated ground cases of $← p(x,x)$ if and only if neither $r(b,a)$ nor any fact matching $r(x,x)$ is in $D$. Moreover, $\exists x,y r(x,y)$ is a violated case if and only if the extension of $r$ is empty. Now, let $U$ be the insertion of $q(x,y) ← r(y,x)$. Clearly, $U$ makes the schema unsatisfiable. However, each inconsistency-tolerant method $M$ can be soundly applied to check $U$ for preserving satisfied cases. It is easy to see that $M(D,U) = \text{ko}$ if and only if there is any tuple of form $(A,B)$ in $r$ such that $(B,A)$ is not in $r$ and $(A,B)$ matches neither $(b,a)\) nor $(x,x)$. Otherwise, all cases of $← p(x,x)$ that are violated in $D^U$ are already violated in $D$, hence $M(D,U) = \text{ok}$.

Although, in Example 11, the updated schema is unsatisfiable, it makes sense to accept further updates of $D^U$ that do not violate any satisfied case. Such updates may even lower the number of violated cases, e.g., $U' = \text{delete} \ s(a,a)$.

4.5 Updating safety-critical integrity theories

As already seen, inconsistency tolerance is desirable. Yet, the need of safety-critical applications to have a set of ‘hard’ integrity constraints that are totally satisfied at all times should not be lightheartedly compromised. Hence, methods are called for that can guarantee total satisfaction of all hard constraints for dynamic schema maintenance. Fortunately, each inconsistency-tolerant method is capable of providing such a service. To see this, we first define the following reliability property, then infer Theorem 4 from it, and then interpret the definition and the result in terms of a dynamic maintenance of safety-critical applications.
Definition 7. \( M \) is called **reliable for hard constraints** if, for each pair \((D, U)\) and each subset \( ICh \) of \( IC \) such that each \( I \) in \( ICh \) is a **hard** constraint, i.e., \( I \) must always be satisfied, i.e., \( D(I) = \text{sat} \), the following implication holds.

\[
M(D, U) = \text{ok} \implies D^U(ICh) = \text{sat}.
\]  

(5)

Theorem 4. Each cause-based method is reliable for hard constraints.

Proof. Let \( M \) be a cause-based method, and \( ICh \subset IC \). Then, (5) follows by applying Definition 6, property (3) and Definition 7.

By definition, each reliable method for hard constraints can maintain the total satisfaction of \( ICh \) across updates, even if \( IC \setminus ICh \) is violated. However, for methods that are not inconsistency-tolerant, e.g., the one in [17], (5) may not hold, i.e., they may not be reliable. Example 12 shows that.

Example 12. Let \( IC = \{←q(a), r(x,x), ←q(b), r(x,x)\}, ICh = \{←q(b), r(x,x)\} \), and \( D(ICh) = \text{sat} \). Further, let \( q(a) \) and \( r(b, b) \) be the only facts in \( D \) that cause a violation of \( ←q(a), r(x,x) \) in \( D \), and \( U = \text{insert} q(b) \). To check \( U \), the method \( M_G \) in [17] drops \( q(b) \) in \( ←q(b), r(x,x) \), since \( U \) makes it true, thus obtaining the simplification \( ←r(x,x) \). Since \( ←q(a), r(x,x) \) is not relevant for \( U \), the assumption of \( M_G \) that \( D(ICh) = \text{sat} \) wrongly entails that \( D^U(←q(a), r(x,x)) = \text{sat} \). Now, assume that \( D \) is distributed, \( q \) is locally accessible and \( r \) is remote. Then, \( M_G \) infers that also \( D^U(←r(x,x)) = \text{sat} \), since \( q(a) \) is true in \( D^U \). Hence, it unreliably outputs \( \text{ok} \), although \( D^U(ICh) = \text{vio} \). As opposed to that, \( M(D, U) = \text{ko} \) holds for each cause-based inconsistency-tolerant method \( M \).

5 Related work

Inconsistency tolerance is a subject of increasing importance [3]. Also inconsistency-tolerant integrity checking has received considerable attention recently [15]. The work in [15] is not based on causes, but on cases, i.e., instances of constraints that are relevant for given updates. In [8], it is shown that each case-based method is inconsistency-tolerant wrt causes, for updates of base facts and view definitions, but that the converse does not hold in general.

In general, causes are a smarter basis for inconsistency-tolerant integrity checking than cases, as argued in [7, 8]. Among others, the concept of completeness is less problematic for causes than for cases. As outlined in [8, 10], another advantage of causes over cases is that causes also provide a basis for computing answers that have integrity in inconsistent databases. The latter are similar to, though different from consistent query answers [2]. The relationship of the latter to cases is discussed in [10, 15], and their relationship to causes in [8]. Moreover, causes offer advantages over cases with regard to partial repairs [12], and an automation of integrity checking for concurrent transactions [13].

Case-based inconsistency-tolerant schema updates are the theme of [6]. This paper upgrades the latter, by having causes take the place of cases, and by generalizing from definite to normal databases and constraints.
The concept of causes as proposed in [18] is much more complicated and involved than ours. The one in [18] is meant for explaining answers to human agents, but not, as in this paper, to programmed agents, which may cater for integrity checking, repairing query answering with integrity, and more.

The work in [5] is related to this paper by the topic of integrity updates. Like this paper, it is application-oriented, and additionally focuses also on issues such as database migration and web information systems, which we do not address, due to space limitation. However, inconsistency tolerance is not an issue in [5].

6 Conclusion

We have outlined how to extend conventional approaches to database schema update management. We have proposed a cause-based approach that can deal with arbitrary schema updates, including changes of the integrity theory. We have shown that such updates can be dealt with efficiently and reliably, without compromising hard integrity requirements for safety-critical applications.

Interestingly, the advantages of our cause-based approach could not be obtained by employing any integrity checking method that is not inconsistency-tolerant. Fortunately, however, the usual requirements of the total satisfaction of each database state and the satisfiability of the database schema can simply be waived, for most methods, without incurring any penalty.

Ongoing work is concerned with applying a general, measure-based approach to integrity checking of conventional updates also to schema updates, and with the inconsistency-tolerant preservation of integrity across concurrent schema updates in distributed and replicated databases.

References

What is next after 25 years of SWI-Prolog?

Jan Wielemaker

FEW, Computer Science
University of Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam
The Netherlands
J.Wielemaker@vu.nl

Abstract. This year, SWI-Prolog turned 25. That is worth a celebration. More importantly, it is a good moment to look backward as well as forward. The landscape around programming and programming infrastructure including programming languages has changed dramatically in the past 25 years. This provides opportunities. The Web of Data – or the Semantic Web – provides a data and a knowledge representation framework in which logic plays an important role. Few IT solutions are monolithic these days. (Web-)service middleware, which is often rule-based, is an obvious place where logic programming can be (and is being) deployed. At the same time where, technically, being a small language becomes less of a problem, there are major bottlenecks, such as fragmented and low-traffic discussion forums, as well as low visibility on the Internet in the form of libraries and program fragments that can act as a starting point to solve one’s problem. Finding an answer to these problems is, in my view, crucial.
Towards an Application of Update Propagation on Logic Programs Representing Java Source Code

Richard Tantius, Daniel Speicher, Andreas Behrend

University of Bonn, Computer Science III,
Römerstraße 164, 53117 Bonn, Germany
{tantius, dsp, behrend}@cs.uni-bonn.de

Abstract. Logic programs are now used as a representation of object-oriented source code in academic prototypes for about a decade. This representation allows a clear and concise implementation of analyses of the object-oriented source code. The full potential of this approach is far from being explored. In this paper, we report about an application of the well-established theory of update propagation within logic programs. Given the representation of the object-oriented code as facts in a logic program, a change to the code corresponds to an update of these facts. We demonstrate how update propagation provides a generic way to generate incremental versions of such analyses.

Keywords: logic program, update propagation, application, logic meta programming, refactoring impact

1 Introduction

In this paper, we show how update propagation can be employed for efficiently computing-derived information within the domain of logic fact-bases representing object-oriented source code.

1.1 Logic Meta Programming

Logic Meta Programming approaches build a detailed representation of a program (typically) written in another programming language in a logic fact-base [14], [8], [10]. This representation allows analysing the original program by means of a logic program built on those facts. This approach has been used to detect code locations that need design improvement [13]. It provides a suitable basis for different static analyses from the implementation of code quality metrics to the implementation of type constraints. Recently, we have been arguing that logic meta programming should be used to integrate the knowledge about good design structures and suspicious design structures, creating a database of code quality knowledge, which can be evolved over time [13].
1.2 Update Propagation

Update propagation (UP) is an established database research topic, which has been studied over the last 40 years mainly in the context of integrity checking and materialized views maintenance, e.g., [4], [2], [9], [7], [11], [12]. Therefore, UP is known from the SQL and Datalog world. UP makes a contribution to efficiently compute implicit changes of derived relations resulting from explicitly performed updates of extensional facts of a logic fact-base. SQL view specifications and Datalog rules, as well as Prolog rules are related forms of deductive rules. This supports the idea to adapt UP to the Prolog world, and also use sets of deltas, together with specialized update statements to incrementally maintain derived predicates (which can be a software analysis, implemented in Prolog). The original predicates are required only once for materializing their initial answers, the specialized delta versions are used in update statements afterwards for continuously updating the materialized results. Assuming that a great portion of the materialized content of the original logic rules remains unchanged, the application of such update statements may considerably enhance the efficiency of computing the state of such relations after an update of the fact-base.

1.3 Refactoring Impact Prediction

UP enables computing the results of an analysis after an update of the fact-base without actually executing the change. Such an update of the fact-base can be induced for example by applying a structural improvement like a refactoring [6] to the source code. This allows to efficiently execute a "what-if"-style analysis. The approach enables us to evaluate a potential refactoring of Java programs, by computing quality attributes like software metrics before the refactoring and simulate via UP how the metric result changes due to the refactoring.

1.4 Automation

Figure 1 introduces our implementation which automates several aspects of the refactoring impact prediction via UP. The picture shows the different components of the system. Those components may consist of several Prolog modules. The application covers the derivation of a suitable abstract model, on which we build the software analysis we intend to employ. We use the Logic Meta Programming approach JTransformer presented in [8], to derive such a model. The implementation also handles the refactoring simulation on the level of that abstract model, metric computation and UP rule generation. We have used SWI-Prolog (in the version 6.0.2\(^1\)) for the implementation. Beside the metric definitions that had to be implemented in a strictly declarative syntax, so that we can apply UP, we used the full feature set of the SWI-Prolog environment for the other parts of the implementation.

\(^1\) The Project homepage of SWI-Prolog: http://www.swi-prolog.org/ (accessed 17.08.12).
2 Model and Analysis

Software analyses like software metrics are a frequently-studied approach to detect lack of quality and are also capable of making improvements of quality measurable. Logic Meta Programming Approaches provide the capability to represent software systems as logic programs. A refactoring in this context, therefore, can be understood as a transformation imposing changes on an extensional fact-base. We discuss the structural cohesion metric Lack of Cohesion in Methods LCOM1 [5], as an example of such a software analysis. Cohesion can be defined as the degree of how closely module components are related to each other. A unified framework for structural metrics was presented by Briand in [3], who created a common model for existing metrics. The model unified the syntactical representation and operational semantic of those metrics. In order to provide the information about the source code the metric relies on, we also present a simplified abstract model as basis for LCOM1. This meta model will be directly derived from the Logic Meta Programming fact-base.

2.1 Abstract Cohesion Model

The Logic Meta Programming approach [8], which we use to derive our abstract cohesion model, is based on Prolog. For this reason, we represent the relevant information for the LCOM1 metric as Prolog facts. We will also present the LCOM1 metric itself as a logic program in the following. To sufficiently describe the information relevant for cohesion metric, we need to take the following information into account: Which class contains a certain method or field? Which methods are called and which fields are accessed by a method? The presented model is based on the cohesion model as presented in Briand [3] and was adapted to Prolog. We consider the following Prolog predicates:

\[
\begin{align*}
&c(N). & \text{class} \\
&cm(C, M). & \text{class contains method} \\
&cf(C, F). & \text{class contains field} \\
&m(N, F). & \text{method accesses field} \\
&mm(M, N). & \text{method invokes method}
\end{align*}
\]
The related Prolog facts are ground versions of the predicates from above and the variables C,M,F,... are bound to unique identifiers for the corresponding elements. In the next subsection, we demonstrate how we extract this model from the JTransformer fact-base. In the following section, we build the LCOM1 metric on top of those facts as a logic program.

2.2 Model Fact Derivation

We derive the abstract cohesion model directly from the fact-base created by the JTransformer Logic Meta Programming approach. JTransformer builds a so-called Abstract Syntax Tree (AST) that already is a full model representation in Prolog of the Java language. The JTransformer facts are called Program Element Facts PEF, they are the starting point for creating the cohesion model.

Figure 1 gives an overview of the various components of our Implementation. The architecture consists of three layers. In the first layer, we have the metric rules and the facts created by JTransformer. The layer in the middle is the meta programming layer, in which we process the rules and facts. Here we compile the UP rules. In the layer at the bottom, we have the subsystems which actually contain the executable code. Each subsystem may consist of several Prolog modules.

Based on generator predicates, the Model Facts Converter component creates cohesion model facts (from the JTransformer facts) and asserts them to a prolog module (cohesion_model). Additionally, we perform checks, if an element should be included at all. In the case of classes, we do not consider the three following class types. Interface classes do not provide method calls and attribute references. Cohesion cannot be examined here. Classes from external dependencies are supposed to be examined elsewhere. Anonymous classes are not supposed to be analysed standalone.

The implementation of the class generator predicates is as follows:

\[
\text{generate}(\text{FactsModule}, \text{c}, [\text{ClassId}]) :-
\text{\% Here we only use the class id}
\text{\quad classT(ClassId, _, _, _),}
\text{\quad source_class(ClassId).}
\]

For the free variable FactsModule, we use the module cohesion_model mentioned above as a default value. The implementation of the class analysis considers different JTransformer facts to determine the class type:

\[
\text{source_class(ClassId) :-}
\text{\% JTransformer facts}
\text{\quad not(externT(ClassId)),}
\text{\quad not(interfaceT(ClassId)),}
\text{\quad \% See below}
\text{\quad not(anonymous_class__(ClassId)).}
\]
anonymous_class__(Class):-
classT(Class, _, ClassName, _),
string_concat('ANONYMOUS$', _, ClassName).

On top of the provided model facts, we create our software analyses, for example
the cohesion metric presented before. The deductive **Metric rule definition** is
the starting point (also shown in Figure 1). We define the deductive part of the
metric in a Prolog module:

```prolog
:- module(metricName_deductive_ruleset, []).
  rule1(A, B, ...) :- ...
```

For example:

```prolog
:- module(lcom1_deductive_ruleset, []).
...
```

### 2.3 Declarative Metric Implementation

The **LCOM1** metric definition we present in the following is based on the defi-
nition given by Briand in [3].

**Query and Mapping** The computation of structural metrics can be divided
into two steps. First, a query step collects the elements or relations that are
relevant for the metric. Second, a mapping maps the result of the query to a
number. Separating these steps has the benefit that we can discuss both steps on
their own. A query result may be evaluated with different mappings. A mapping
may be applied to the result of different queries. The separation of query and
mapping has the advantage, that we can apply the update propagation approach
to the deductive rules defining the query.

**Query** **LCOM1** counts the number of method pairs within a class that do not
access even one common field. We split this definition into two predicates. Each
predicate will be first described in natural language, then as a logic program.

*M and N in C are connected, if M accesses a field F, that belongs to the class C,
and N accesses that field F as well.*

```prolog
cp(C, M, N) :-
  mf(M,F), cf(C, F), mf(N,F).
```

*M and N in C are a pair of methods lacking cohesion, if M is a method in C,
N is as well a method in C and M and N are not a connected pair in C:*

```prolog
lp(C, M, N) :-
  cm(C, M), cm(C, N), not(cp(C, M,N)).
```

53
Mapping To complete the LCOM1 computation rule, we need to perform some additional steps after the deductive part.

\[
\text{lcom1}(C, R):= \\
\text{findall}([M, N], (\text{cp}(C, M, N), \text{not}(M=N)), E), \\
\text{length}(E, T), \\
R \text{ is } T/2.
\]

3 Refactoring as Cohesion Model Update

We model a refactoring as an update on our cohesion model. Performing the refactoring only on the abstract level of the cohesion model, helps to concentrate the computation only on aspects relevant for the metric computation. Because we use update propagation in the following, we do not directly perform model updates, rather every update will generate a so called delta fact, which depicts the actual change and will be discussed in detail in Section 4. In the following, we briefly discuss the refactorings we use and describe their effects on the level of the model. Both refactorings assume in our setting, that the moved element will be extracted into a new class, creating the following delta fact: \text{add}_c(#\text{newClassId})

Since we exclude constructor methods from our model and do not consider modificators as public, private and protected, the following refactorings require no special preconditions to be applied.

3.1 Move Method, Move Field

The move method refactoring moves a method from one class to another. Though in a real world refactoring, we would need to adjust the code in several ways, so that it remains functioning and compileable we add two simple delta facts to our model:

\[
\text{add}_c(C, M) \\
\text{del}_c(C, M)
\]

Similar to the move method refactoring the move field refactoring moves a field from one class to another, the resulting delta facts are as follows:

\[
\text{add}_c(C, F) \\
\text{del}_c(C, F)
\]

4 Update Propagation

In this section, we show how to apply update propagation in Prolog. Because of the different evaluation mechanisms for SQL views (set-oriented, bottom-up) and Prolog rules (instance-oriented, top-down), however, the transformation techniques from UP could not be applied directly. Instead, the specific properties of Prolog rules have to be taken into account in order to achieve a complete and sound update propagation based on delta predicates.
Fig. 2. The derived update propagation and indirect and direct transition rules for the LCOM1 definition.

4.1 Rule Transformation in Prolog

The task of UP is to systematically compute the set of all induced changes, starting from the physical changes of base data. Technically, this is a set of delta facts for any affected predicate which may be stored in corresponding delta relations. For each predicate symbol \( p \), we will use a pair of delta predicates \(<\text{add}_p, \text{del}_p>\) representing the insertions and deletions induced on \( p \) by an update. The initial set of delta facts represents the so-called \( UP \) seeds.

In the following, we briefly review a transformation-based approach to \( UP \) where the Prolog rules and the \( UP \) seeds are employed to derive propagation rules for computing delta relations. A propagation rule refers to at least one delta predicate in its body in order to provide a focus on the underlying changes when computing induced updates. For showing the effectiveness of an induced update, however, references to the state of a predicate before and after the base update has been performed are necessary.

For each predicate \( p \) we use \( \text{old}_p \) to refer to its old state before the changes given in the delta sets have been applied (technically the rule behind \( \text{old}_p \) is the unmodified version of \( p \)). We use \( \text{new}_p \) to refer to the new state of \( p \). These state relations are never completely computed, but are queried with bindings from the delta sets in the propagation rule body, and thus act as a test of effectiveness. An induced insertion or induced deletion can be simply represented by the difference between the two consecutive database states. We consider the following Prolog rule:

\[
\text{add}_{\text{lp}}(C, M, N) \leftarrow \text{cm}(C, M), \text{cm}(C, N), \text{not}(\text{cp}(C, M, N)), \text{not}(\text{lp}(C, M, N)).
\]
p(X) :- q(Y), r(Z), not s(C).

The difference rules may look as follows:

add_p(X) :- add_q(Y), new_r(Z), not(new_m(C)), not(old_p(X)).
add_p(X) :- new_q(Y), add_r(Z), not(new_m(C)), not(old_p(X)).
add_p(X) :- new_q(Y), new_r(Z), del_m(C), not(old_p(X)).

The propagation rules basically perform a comparison of the old and new versions of the predicates. While providing a focus on insertions into q and r, all necessary combinations of delta and state predicates are considered. Because of the negative referenced predicate s, an additional rule has to be considered, which covers new derivations for p due to a deletion from s.

All propagation rules also contain the additional effectiveness test not old_p(X), to check for the effectiveness of the induced insertions in case of alternative derivations of the same fact p in the old state. As an optimization we can drop the test, in case there are no alternative derivations, or if the set of insertions can be overestimated. To avoid the full determination of state predicates, we should move the delta predicates as far left as possible in the rule body. This way the bindings provided by the delta facts can be used for restricting the evaluation of state predicates. This leads to the following rules:

add_p(X) :- add_q(Y), new_r(Z), not(new_m(C)).
add_p(X) :- new_q(Y), add_r(Z), not(new_m(C)).
add_p(X) :- del_m(C), new_q(Y), new_r(Z).

For simulating the new predicate state from a given update and the old state, so called transition rules [12] can be used. The transition rules of a derived predicate infer its new state from the new states of the underlying predicates. Thus, for a rule A :- L1, . . . , Ln a transition rule of the form new_A :- new_L1, . . . , new_Ln is considered. In contrast, for every extensional predicate A so-called incremental transition rules are used:

new_A :- old_A, not(del_A).
new_A :- add_A.

which explicitly refer to the computed changes to A.

For a rule A :- L1, . . . , Ln we may also use the direct transition rules if there are no mutual dependencies between the predicate and the predicates in the body. In any case, the indirect transition rules of the form

new_A :- new_L1, . . . , new_Ln

are used in the effectiveness test of negative propagation rules.

As an example, consider the following Prolog program

p(X) :- q(X,Y), r(Y), not(s(Y)).
q(1,2). r(3). s(4).
q(2,3). r(4). s(5).
q(3,4). r(5). s(6).
and the insertion $r(2)$ into relation $r$. The following propagation and transition rules

$$p(X) :- \text{add}_r(Y), \text{new}_q(X,Y), \text{not(new}_s(Y)).$$
$$\text{new}_q(X,Y) :- q(X,Y), \text{not(del}_q(X,Y)).$$
$$\text{new}_s(X,Y) :- \text{add}_q(X,Y).$$
$$\text{new}_s(X,Y) :- s(X,Y), \text{not(del}_s(X,Y)).$$
$$\text{new}_s(X,Y) :- \text{add}_s(X,Y).$$

were derived using the scheme described above. These rules allow for efficiently computing the induced insertion $p(1)$ (represented by the fact $\text{add}_p(1)$) by avoiding any redundant recomputations.

5 Propagation of Cohesion Model Updates

In this section, we apply UP as presented in Section 4 to our cohesion model and the metric rules from Section 2.

Before we employ UP to simulate the model state and metric results after the refactoring, we first need to generate delta facts as described in Section 3. At definition time of the metric rules, we may also derive the propagation and transition rules for UP. Figure 2 shows the result of the rule derivation process. We can see that for each body literal of a rule that appears in the metric definition, we create a positive and a negative propagation rule, so that for the cohesive pair rule $cp$ we obtain six propagation rules in total. We replace negated delta literals simply by their opposite versions, for example: $\text{not(del}_cp) \Leftrightarrow \text{add}_cp$. For both rules $lp$ and $cp$, we also see two derived transition rules $\text{nwi}_lp$ and $\text{nwi}_cp$, simulating the versions of those predicates in the new state, which operate on the propagation rules.

5.1 Rule Generation

The UP rule generator component consists of two subcomponents, which can be seen in the middle of Figure 1. The Rule Analyser component collects meta information about the metric rules from the deductive rule modules and the UP Rules Generator, which creates the UP rules, based on the collected meta information. SWI-Prolog provides various meta predicates to examine loaded programs. It is important that UP ensures that the augmented rule set (which includes UP rules as shown in Figure 2 and the original rules like those for $\text{LCOM1}$) generated for a set of logic rules which is guaranteed to terminate, still keeps this property. This was shown for the language set of Datalog [1], [7], which only allows straight forward declarative rules, in comparison to Prolog. We also need to assure this in Prolog, it would be unfavourable if the UP rules got stuck in infinite loops. Prolog allows a broad variety of syntax constructs. For the metric definition, therefore, we only allow the model predicates and predicates defined in the template module itself and those from the cohesion model. We also allow a narrow list of built-in predicates, namely $=/2$, member/2.
and the negation predicate \texttt{not/1}. We also do not allow any complex terms in the head of the rules, like: \texttt{cp(a(A),B,[H,T]) :- ...}. Though this is a sharp restriction of Prolog, we were able to describe several structural cohesion metrics, as long as they contained a deductive part.

\textbf{Rule Analyser} First, the \textbf{Rule Analyser} determines all predicates defined in the template module which contains the \texttt{LCOM1} metric rules we presented before. For \texttt{LCOM1} those are \texttt{cp} and \texttt{lp}. The analyser collects various meta information, which we need to build the UP rules. An overview of the collected meta information:

\begin{verbatim}
head_of_rule(headId, groundHead, [name, arity])
body_predicate_of_rule(headId, bodyId, positionInBody, groundPredicate)
rule_variables(headId, bodyId, headBodyPrefix, positionInBody, groundPredicate)
predicate_dependencies_transitive_closure( [name, arity], dependencies)
\end{verbatim}

We need to determine the free variables in the head and the body of the rules. We also need to check, if the rules contain self references. This is relevant to determine the positioning of the delta terms (\texttt{del_} and \texttt{add_} in the body of propagation rules). Second, in order to create the indirect transition rules \texttt{nwi_P} of a predicate \texttt{P}, we need to analyse, if there are transitive mutual dependencies between \texttt{P} and its body predicates \texttt{L_i} (where \(0 \leq i \leq \#\text{number of body predicates of } \texttt{P}\)). The result will determine if we rather use \texttt{nwi_L_i} or \texttt{nwd_L_i} in \texttt{nwi_P}; this is important to ensure that the rules still terminate. We therefore create a predicate dependency graph.

\textbf{UP Rules Generator} Based on the information collected by the rule analyser, the \textbf{UP Rules Generator} creates the UP rules. As mentioned before, we create negative and positive Propagation Rules (for the metric), the direct Transition Rules (metric and model facts) and indirect Transition Rules (metric only). The rules created are asserted to a module. After completing the generation process, we also compile all rules to static predicates by using the Prolog \texttt{compile_predicates} predicate. The rule creation is based on string concatenation, therefore we convert all predicates to atoms, and prepend the necessary augmentations to each predicate.

6 \ Conclusions and Future Work

Earlier research had shown that the representation of object-oriented source code as a fact-base in a logic program allows for clear and concise implementation of static analyses of the object-oriented code. We explored the potential of this
approach further by applying the well-established theory of update propagation to it. Update propagation gives us a generic way to transform any analysis represented as a (sufficiently well-formed) logic program into an incremental version. Given an actual or hypothetical small change to the fact-base, this incremental version of the analysis provides an efficient way to calculate the new result of the analysis after the (hypothetical) change.

We implemented a transformation, generating the propagation and transition rules based on the original rules. Besides the applicability of update propagation to our setting, we already conducted several experiments to explore the performance benefits. In our experiments, update propagation was significantly faster than actually transforming our model and re-evaluating the original definition. As a future work, we intend a detailed study of the performance benefits.

We focused on the precalculation of the refactoring impact on metrics as this is in line with our research interest. Nevertheless, there is no reason to limit update propagation for logic meta programming to metrics or not even to refactoring. In the context of refactoring, update propagation could for example be used to verify that certain constraints will still hold true after the refactoring. This would be in line with the original motivation of update propagation.

A precise definition of the pairs of abstract refactorings on the model and the corresponding refactorings on the source code, could ease and clarify the nature of the induced updates. Model updates. More sophisticated refactorings also need complex preconditions, before they may be applied legally. We should be able to check those conditions on the level of source model.

References

1. A. Behrend. Soft Stratification for Transformation-Based Approaches to Deductive Databases, University of Bonn: PhD thesis, 2004


Knowledge Engineering for Business Rules in PROLOG

Ludwig Ostermayer, Dietmar Seipel
University of Würzburg, Department of Computer Science
Am Hubland, D–97074 Würzburg, Germany
{ludwig.ostermayer,dietmar.seipel}@uni-wuerzburg.de

Abstract. To bridge the gap between business analysts and developers, most companies nowadays rely on business process management (BPM). When controlling progressively more complex business processes, the connection between business rules and BPM gains a tremendous importance. Currently, the most popular BPM tools are JBPX, an extensible workflow engine written in pure JAVA, and DROOLS, an expert system framework, that uses rules for knowledge representation. It is possible to combine JBPX and DROOLS. However, there are still flaws with the verification and the update of knowledge bases, and in particular with building more complex knowledge structures.
In this paper, we want to propose an alternative knowledge engineering approach for building, updating and testing complex, structured sets of business rules using SWI PROLOG, and we illustrate it by an example in the field of e-commerce.


1 Introduction

Business rules are statements that define or restrict certain aspects of a business process [6]. They model business structures, and they control the behavior of business processes. Business rules can be applied to persons, processes, business behavior and computer systems in companies. While business rules can occur purely informally, the careful, unambiguous, and consistent formulation of the rules is particularly important. Thus, cost-intensive misconceptions can be avoided, the communication can be improved, the legal regulations are obeyed, and the customer satisfaction is improved.
Due to the increasing complexity of system development, a communication problem arises between developers and business analysts. Both need a common language for the problem domain. In fact, the business analysts should have the possibility to create process logic specifications on a formal, abstract level, which can then be processed by the developers (software engineers).
Currently the most popular tool for business process management [1] is JBPX, an extensible workflow engine written purely in JAVA, and the most popular tool for business rules is DROOLS, an expert system framework, that uses rules for knowledge representation. It is possible to combine JBPX and DROOLS. However, there are still flaws with the verification and the update of knowledge bases, and in particular with building more complex knowledge structures.
**Knowledge engineering** for business rules has some specialties coming from the fact that the business analyst and the customer should be able to understand and refine – some of – the declarative specification. In this paper, we propose using logic programming in SWI PROLOG for implementing business rules. For illustration, we use a simple example out of a large e-commerce system, which has originally been developed in JAVA, and which we are reimplementing currently using logic programming. Especially the implementation of the business logic was overly complicated, irreproducible, and difficult to maintain; changing a rule in the original system was a very difficult task. The first step was breaking down the original program logic into clear, formal and understandable rules. As a result, we present the aforementioned example in Section 3.

There already exist **program development tools** for PROLOG— or DATALOG—based systems, cf., e.g., [12, 18]. We have also developed concepts for analyzing rule bases and components for integrated development environments: [14] investigates the analysis and refactoring of logic programs, [2] presents anomaly tests for rules and ontologies in the semantic web rules language SWRL. We would like to transfer these concepts to business rules and extend them to a more general framework based on DROOLS, PROLOG, JAVA, etc. For exchanging data between PROLOG and standard business rule systems, there exist XML–based formats.

Recently, domain specific languages (DSL) [9] have proven to be useful for supporting knowledge engineering with business rules. Techniques from logic programming, such as meta–programming, have been used for a long time to support abstraction. Building DSLs with logic programming can be much simpler than the standard approach, where compilers have to be developed. We think that the syntax and the semantics of PROLOG are well–suited for developing an internal DSL. We hope to achieve this goal in an ongoing project (using, e.g., infix notation with specially defined operators), and we want to additionally support this by a graphical user interface.

The rest of this paper is organized as follows: In Section 2, we report some related work on business rules systems and business process modeling. In Section 3, we illustrate our approach using an example from e-commerce. In Section 4, we describe a DATALOG–like, stratified bottom–up evaluation of business rules; we have developed a visualization by proof trees, whose generation can be configured flexibly by the developer. Section 5 summarizes important concepts of knowledge engineering with business rules – some are already included in our approach, others are ideas for future work.

## 2 Business Process Modelling

Detailed knowledge about the business logic is essential for a company. This may sound obvious, but most IT systems have been developed over a long period of time, and parts of the program that reflect business processes are hidden behind thousands of lines of code.

Refactoring and updating business logic or even understanding how the processes are modeled may be a challenge. This challenge could be solved by business process modeling with JBPM and by business rules systems such as DROOLS.
2.1 JAVA Business Process Modelling (JBPM)

JAVA business process modeling (JBPM) is based on BPMN 2.0, which is an OMG standard [1] for modeling business processes; it uses the JBoss process definition language JPDL to define process definitions in files. JPDL is a graphic-oriented programming language. The modeling uses nodes, transitions, and actions. Each node has a type defining its runtime behavior. During the flow of a process, the nodes trigger commands that are executed as the nodes are reached. The flow of a process is directed by transitions. Actions perform the logic, when a transition event or a node is reached in a process.

The JPDL process engine runs through a directed process graph. It usually consists of nodes, transitions, one start state, and one end state. The process graph represents the process definitions in a clear manner. Thus, traceability is always given.

In our project we paid special attention to decision nodes. When a decision node is reached in a process, the action is that a request – in an XML format – for a decision is sent to a server. The server is implemented in SWI PROLOG, and it uses the HTTP protocol [19]. Thus, we can easily communicate between SWI PROLOG and JBPM and transfer the business logic to PROLOG. In parallel, we also investigate JAVA-based frameworks for evaluating business rules. Moreover, frameworks blending logic programming with JAVA as well as Eclipse Plugins for logic programming might be helpful [12, 18].

2.2 The Business Rules System DROOLS

The JAVA-based expert system DROOLS has been developed by the JBOSS community. Its core module Expert provides the rule engine. For efficiently matching rules and facts, an enhanced implementation of the Rete algorithm [8] is used, the object-oriented Rete algorithm. Knowledge representation in DROOLS uses rules and decision tables. The special module Flow provides an extensible workflow engine.

However, as a non-standardized solution, DROOLS comes with some flaws and deficiencies [11]. The structure of the knowledge representation is still flat. In Version 5, new nodes and timers have been added, but the problem of a non-existing rule hierarchy persists. Since Version 5.4.0, the conversion of the DROOLS rule language to XML is no longer supported [10]. Furthermore, DROOLS has no tools for fully supporting the knowledge engineering process; in particular, there is no automated identification of rule anomalies, such as redundant or contradictory rules. Finally, the strong language and platform dependency makes a broader applicability difficult.

3 Case Study: Taxes in International E-Commerce

Electronic commerce (E-Commerce) has experienced an accelerated growth in recent years. Figure 1 shows the relative number of individuals in the EU-27 ordering goods or services over the internet in the 12 months prior to the survey in 2010. For keeping the accountancy transparent and traceable (24 hours a day, 7 days a week), an automated approach is indispensable. Rule-based systems can support the bookkeeping by automated decision making.
The Delivery Threshold. In the European Union, the sales tax for shipments to foreign countries has to be paid in the country where the shipment starts (home country). However, the sales tax has to be paid to the country of dispatch [4], as soon as the accumulated shipments abroad exceed a certain net merchandise value per year. We call this limit the delivery threshold; it is, e.g., 100000 € for shipments from Germany to France.

Assume that we have a credit transfer on our banking account that results from a shipment to a foreign country. The legal text determining the accountancy for paying the sales tax can be simplified and translated into two rules:

1. **When** the delivery threshold has been exceeded last year or previously this year, **then** the sales tax has to be paid in the country of dispatch. For this years calculation, we include a preliminary profit of the currently processed invoice. To determine this preliminary profit, the sales tax of the home country has to be used.

2. **When** the first rule does not fire, **then** the sales tax has to be paid in the home country.

It is a difficult task to arrange rules clearly and precisely. In the underlying law texts, four rules are used with redundant conditions in the rule body. Formulating rules without ambiguity and redundancy is very essential for knowledge representation.

In the following, we will first describe the XML data formats for the input and the output, and then we will illustrate the PROLOG implementation of our computations. Internally, the program consists of basic predicates that reflect the circumstances of the business case, and a business rule part with the program logic. At any time, only a single invoice is processed.

### 3.1 Input and Output in XML

In our simplified example, only the sales amount, the kind of taxation (e.g., food or non-food) and the country of dispatch are of interest. In reality, for every combination of taxation and country, there exists an account with a unique account number to guarantee consistent traceability.
For an already paid invoice, we have to compute the resulting bookings for a company’s internal accountancy. We assume that the invoice is in XML format with various invoice positions that are classified by line, tax type and total amount, e.g.:

```xml
<invoice country="France" year="2012">
  <position line="1" type="food" total="211.00"/>
  <position line="2" type="nonfood" total="119.60"/>
</invoice>
```

The resulting bookings will be represented in an enriched XML format. Before we describe how the program works, we want to show the resulting bookings in XML. All amounts were rounded to a full cent amount:

```xml
<booking country="France" year="2012">
  <position line="1" type="food" total="211.00">
    <profit country="France" amount="200.00"/>
    <taxes country="France" amount="11.00"/>
  </position>
  <position line="2" type="nonfood" total="119.60">
    <profit country="France" amount="100.00"/>
    <taxes country="France" amount="19.60"/>
  </position>
</booking>
```

Using the above mentioned tax rules, we split the total amount of an invoice position into two parts: profit and taxes. Each of them specifies the country and the corresponding amount of money.

### 3.2 Basic Predicates

There are two groups of basic predicates: the first one describes the currently processed invoice and the current status for last year and this year. Note that every invoice is represented by its own fact base. The second group provides the configuration of the system. We show in our example only the necessary part of the configuration.

**Predicates for Representing the Invoices.** The facts for `annual_sales_so_far/4` provide the business volume for a country in a given year. Only these facts change after the processing of an invoice, since the business volume for the current year must be updated. The facts for `invoice/2` and `position/3` are extracted from the XML document for the invoice. In our example, Germany is the home country.

```prolog
% annual_sales_so_far(Destination, Year, Total)
annual_sales_so_far('France', 2011, 92300).
annual_sales_so_far('France', 2012, 99800).
```
Predicates for the Configuration. The facts for `tax/3` provide the information about the taxes for a given country, i.e., the tax rates for the different tax types. The facts for `delivery_threshold/2` specify the delivery threshold for the different countries; for our current invoice, only France is of interest.

3.3 Business Rules

The rule for `annual_sales/3` aggregates the business volume for the current year including the recent invoice using the predicate `ddbase_aggregate/3` from the DiSLOG Developers' Kit (DDK) [17]. We have to add the business volume given by `annual_sales_so_far/3` to the net value of the current invoice, which sums up the net values of its positions. For obtaining these net values, the tax rate of the home country is used, which is given by `tax/3`.

```prolog
annual_sales(Destination, Year, Total) :-
    ddbase_aggregate( [Destination, Year, sum(Value)],
        { annual_sales_so_far(Destination, Year, Value)
        ; invoice_position(Destination, Year, Value) },
        Tuples ),
    member([Destination, Year, Total], Tuples).

invoice_position(Destination, Year, Net_Value) :-
    invoice(Destination, Year),
    position(_, Type, Total),
    home_country(Home_Country),
    tax(Home_Country, Type, T),
    Net_Value is Total/(1+T).
```
The predicate `tax_country/1` implements the two business rules for the delivery threshold: it determines the country where the taxes have to be paid.

```prolog
tax_country(Tax_Country) :-
    invoice(Destination, Year),
    delivery_threshold(Destination, Threshold),
    ( ( Y is Year - 1 ; Y is Year ),
        annual_sales(Destination, Y, Total),
        Total > Threshold ->
            Tax_Country = Destination
    ; home_country(Home_Country),
        Tax_Country = Home_Country )
```

Knowing the tax country, the rule for `booking_position/5` calculates the correct amount for every position and the modes `profit` and `taxes`. The derived head atoms can easily be transformed to an XML document of the form shown above.

```prolog
booking_position(Line, Type, Mode, Country, Amount) :-
    invoice(Destination, _),
    position(Line, Type, Value),
    tax_country(Tax_Country),
    tax(Tax_Country, Type, T),
    ( Mode = profit, Country = Destination,
        Amount is Value/(1+T)
    ; Mode = taxes, Country = Tax_Country,
        Amount is Value*T/(1+T) )
```

The business rules are evaluated bottom-up in a stratified manner [5]. The predicate `invoice_position/3` has to be evaluated before `annual_sales/3`; this precedence is due to the meta-predicate `ddbase_aggregate/3` for aggregation. Another well-known meta-predicate that requires stratification in deductive databases is default negation `not/1`. In principle, the other rules could be evaluated in a simultaneous bottom-up iteration – in our system, however, they are evaluated subsequently.

Some of the used predicates are deterministic for a given input. The all-results-inference-capability is especially essential for the two non-deterministic predicates `invoice_position/3` and `booking_position/5`. In JAVA-based systems such as DROOLS, it is much more complicated to perform such computations – this turned out when we compared the PROLOG implementation of the business rules with an alternative implementation in DROOLS.

### 3.4 Calculation of Profits and Taxes – An Example

In the following, we want to process the given invoice with the program bookings step by step. First, the business volume in `annual_sales/3` is determined. We summarize the derived facts below. As one can see, the delivery threshold for 2011 was not exceeded, but for 2012 it was. Apparently, the company’s profit has increased from 2011 to 2012.
Since the annual sales to the destination country France – including the currently processed invoice – exceed the delivery threshold in 2012, the tax has to be paid in France:

\[
\text{annual_sales('France', 2011, 92300).} \\
\text{annual_sales('France', 2012, 100098).} \\
\]

Finally, \text{booking_position/5} calculates the resulting booking positions for the modes \text{profit} and \text{tax}:

\[
\begin{align*}
\text{booking_position(1, food, profit, 'France', 200.00).} \\
\text{booking_position(1, food, taxes, 'France', 11.00).} \\
\text{booking_position(2, nonfood, profit, 'France', 100.00).} \\
\text{booking_position(2, nonfood, taxes, 'France', 19.60).}
\end{align*}
\]

The derivation of the first fact for \text{booking_position/5} is visualized in Figure 2. For a better overview, we use abbreviated predicate names in the proof trees.

![Fig. 2: Visualization of Proof Trees](image)

If we assume a different value for \text{annual_sales_so_far/3} in 2012, then the annual sales to the destination country France – including the currently processed invoice – do no longer exceed the delivery threshold in 2012:
\$ annual_sales_so_far(Destination, Year, Total)
annual_sales_so_far('France', 2011, 92300).
annual_sales_so_far('France', 2012, 80000).

Thus, the tax has to be paid in the home country Germany:

tax_country('Germany').

We derive the following facts for booking_position/5:

| booking_position(1, food, profit, 'France', 197.20). |
| booking_position(1, food, taxes, 'Germany', 13.80). |
| booking_position(2, nonfood, profit, 'France', 100.50). |
| booking_position(2, nonfood, taxes, 'Germany', 19.10). |

The derivation of the first fact for booking_position/5 is visualized in Figure 3. The \{If \rightarrow Then; Else\} statement has been replaced by pure PROLOG with conjunction and disjunction.
4 Bottom-Up Evaluation with Proof Trees

We have extended the DDK package DATALOG* [16] such that it can generate proof trees [5]. The proof trees are obtained based on an (automatic) program transformation. Compared to standard DATALOG, the extension DATALOG* allows for a larger set of connectives (including conjunction and disjunction), for function symbols, and for stratified PROLOG meta-predicates (including aggregation and default negation) in rule bodies. DATALOG* programs are evaluated bottom-up – just like standard DATALOG programs.

The rule nodes in the proof trees – see Figures 2 and 3 – are labeled by numbers, and they are shown as blue boxes. The nodes for the derived atoms are shown as red circles, that are labeled with the atoms. Basic predicates from the configuration or calls to PROLOG for built-in predicates (such as is/2 and >/2) are given by nodes in the form of an orange circle.

The visualization of the proof trees can be configured by the developer. Firstly, it is possible to group body atoms in the proof trees for better readability; these atoms are then joined by an and node, that is depicted as a white rhombus. Secondly, trivial body atoms can be excluded to reduce the complexity of the visualization: e.g., the atoms for invoice/2 and delivery_threshold/2 are excluded, since they are part of the input, and they are not modified during the computation. Thirdly, the included atoms can be simplified suitably: e.g., the predicate symbol annual_sales/4 is abbreviated to sales/4, tax_country/1 is abbreviated to country/1, and booking_position/5 is abbreviated to booking/5.

A proof tree is encoded in a term structure that resembles XML. We process this XML using the query, transformation and update language FQUERY of the DDK. In principle, however, the proof trees can be analyzed and further processed using standard XML tools as well. The term structures do not lead to any termination problems even for recursive rule sets, since – during the bottom-up evaluation – we do not allow that an atom is used in its own derivation. This can be tested by investigating the proof tree. Obviously, the construction of the proof trees costs some extra runtime. But for realistic examples of business rules, this is by far no problem. The main issue is to assist the business analyst during the knowledge engineering phase by a suitable, not too verbose visualization of the derivations.

5 Knowledge Engineering for Business Rules

In practice, it is a long, iterative process to model the application domain in cooperation with the customer (domain expert), and to develop the business rules. There are many communication problems and misunderstandings. The example, that we have shown above, is simplified; we have shown only a small portion of the business rules, and we have omitted details, such as the account numbers which depend on the different countries, types (food or nonfood), and modes (profit or tax). However, even for the small example, it took a few sessions with the customer to completely understand and suitably model the problem.

It is crucial that the business rules are formal and to some extent understandable for the customer as well – who usually has very little experience with PROLOG. Thus, we
need an integrated development environment, that supports – among others – domain
specific languages, visualization, and anomaly tests.

Business rules – in connection with domain specific languages (DSL) – can bridge
the gap between the customer and the developer, since – normally – the customer cannot
understand the – often procedural – code written by the developer. A business analyst
helps to communicate based on a declarative specification with business rules and DSLs.

5.1 Knowledge Engineering in PROLOG

In a domain specific language, the developers can implement the business rules, and the
customer can still – largely – understand this formalization. The formalization should
be executable – like in declarative programming. Using PROLOG’s infix operators, it
should be possible to iteratively develop a domain specific language. Using PROLOG’s
 parsing techniques (DCGs), this language can be improved even further. At any time of
the development phase, the developer and the customer can discuss about the DSL spec-
ification. In future work, we will investigate the use of DSLs for declarative knowledge
engineering.

Usually, business rules have to be evaluated in a bottom–up, forward chaining man-
ner. During the development phase, a visualization of the program execution is essential.
Using DATALOG, we can already evaluate the PROLOG specification bottom–up, and
during this evaluation we can generate proof trees.

We could also investigate the declarative specification by looking for design anoma-
lies. In our example, we could manually simplify an over–complicated set of four initial
rules for determining the tax country to a single and much more readable rule. In the future, we would like to – partially – automatize such simplifications. We might also
detect certain inconsistencies or redundancies in the set of business rules.

The standard tools for business rules, such as DROOLS, do not support the develop-
ment phase well. However, it is conceivable, that we develop the business rule specifi-
cation in PROLOG and later compile it to DROOLS. Up to Version 5.3, there exists an XML
format for DROOLS rules, that could be used for the data exchange; the DSL feature of
DROOLS itself could also be used. Moreover, technologies of compiling PROLOG code
to JAVA, that are known in the PROLOG community, might be used.

5.2 Knowledge Engineering in DROOLS

In PROLOG, the list of all solutions to a given query can be simply computed using the
 meta–predicate findall/3. In DROOLS, such a computation is only possible using
a combination of JAVA and a query language like SQL (e.g., embedded SQL program-
ing). This might be very complicated and unnecessarily blow up the implementation
in DROOLS.

Although JAVA and DROOLS are connected tightly, there are some flaws. Moreover,
DROOLS does not even work together properly with all JAVA concepts. For example,
when we need to use a class in the condition block of a business rule, then that class
must be instantiated in the DROOLS session, even if the class has just static attributes
and static methods, such as, e.g., a constant class. Only when we use such a class in an
action block, then we need not instantiate it in the DROOLS session.
When we looked at rules from DROOLS, we noticed that the rule format is not well-suited for analyses. For our investigations, we transformed the original XML into an enriched XML using Extensible Stylesheet Language Transformations (XSLT). Our aim was not only to investigate rules in DROOLS this way, but to export business rules from PROLOG to DROOLS. However, since Version 5.4.0, the conversion from DROOLS to XML is no longer supported, which complicates our undertaking.

5.3 Domain Specific Languages

Domain specific languages (DSL) [9] have recently become popular in knowledge engineering for business rules. Unlike general-purpose programming languages (such as JAVA and C), a DSL is usually a subset of a programming language or a specification language for a special problem domain; we call this an internal DSL. External DSLs are completely new languages with a separate syntax and semantics (such as, e.g., the query language SQL for relational databases).

DSLs are increasingly used in business process modeling, since the standard approaches, that are often based on JAVA, are too difficult to understand and cannot be modified or extended flexibly enough. The idea is to assist business analysts already during the customer contact with developing a formal specification of the business rules based on a DSL. Subsequently, this initial formal specification can be corrected and refined by the developers. Ideally, it should already be executable, but it could also be reimplemented later in a standard programming language. The abstract, declarative DSL representation can, however, be understood and validated by the business analyst at any time during the development phase.

Unfortunately, developing suitable DSLs turned out to be difficult in practical projects, and it brings a non-negligible additional effort at the beginning of the software project. Frequently, projects fail, since the DSL was not developed carefully enough.

We think that logic programming technology – especially using PROLOG – could be beneficial here by simplifying the development of a DSL. In some sense, DATALOG∗ can also be considered as an internal DSL that is embedded in PROLOG. In fact, many DATALOG∗ programs can even equivalently be evaluated using PROLOG’s top-down evaluation.

6 Conclusions

In this paper, we have presented a PROLOG–based approach for the bottom-up evaluation of business rules, that supports the visualization of the derivation processes using suitable, customized proof trees.

Knowledge engineering with declarative concepts turned out to be very beneficial for business rules.

Integrated development environments for PROLOG generally provide helpful tools for debugging and tracing. We are planning to additionally use techniques for anomaly analysis, which we have developed before.

In the future, we will extend our knowledge engineering approach by developing suitable domain specific languages based on PROLOG technology.
References

Clone Removal in Java Programs
as a Process of Stepwise Unification

Daniel Speicher and Andri Bremm
University of Bonn, Computer Science III,
Römerstraße 164, 53117 Bonn, Germany
dsp@cs.uni-bonn.de, bremm@uni-bonn.de

Abstract. Cloned code is one of the most important obstacles against consistent software maintenance and evolution. Although today’s clone detection tools find a variety of clones, they do not offer any advice how to remove such clones. We explain the problems involved in finding a sequence of changes for clone removal and suggest to view this problem as a process of stepwise unification of the clone instances. Consequently the problem can be solved by backtracking over the possible unification steps.

Keywords: unification, backtracking, clone analysis, refactoring, program dependence graph, lambda expression

1 Introduction

In the last decades, practicable and reliable code clone detection tools [9] have been developed. These tools are able to find different kinds of clones, which are all important in some situations. Developers, who are interested in the removal of clones by refactoring [2], want to know whether and how a clone can be eliminated. A refactoring is a change to the source code that alters (typically improves) the design, but does not change the observable behavior of the software. We present an approach that gives precise refactoring suggestions depending on the set of available refactorings.

Fowler et al. provided in [2] a comprehensive catalog of such refactorings. A developer may use for example the rename method refactoring to change the name of a method to a more expressive one. A tool that offers this refactoring has to make sure that the name of the method is not only changed in the declaration of the method, but as well at every method invocation. As the same name might be used for different methods in different scopes, a textual “search and replace” is not guaranteed to keep the observable behavior intact. The tool needs to know the Abstract Syntax Tree of the code including resolved bindings to methods. JTransformer [12] provides this information for Java programs as facts on which logic programs can reason.

Identical clones can be removed by extract method, extract class, pull up method refactorings together with the appropriate adaptations at the
Fig. 1. Two cloned methods. The methods differ in the name of two variables (adults vs. children, isAdult vs. isChild), one extra statement (line 3), and a non-unifiable expression (line 5 vs. line 14).

call side. If the clones have some differences, we suggest to start with the Program Dependence Graphs [3] (PDG) for the clone instances, to identify the statements that are equal or unifiable (i.e. can be made equal through refactorings) and finally rearrange the control flow based on the PDG so that all non-unifiable statements are separated. The example in the Figures 1 to 3 illustrates our approach.

2 Program Dependence Graphs

We start with two potential clone candidates. These may have been found with one of the existing clone detection tools like Simian [14] or Scorpio [6, 13]. For each of the clone candidates we build the PDG.

Such a PDG consists of one node for every statement and of two kind of edges representing control and data dependencies. There is a control dependency from a control statement to all directly enclosed statements. In our example the for-loop in line 4 controls the execution of line 5 and 6 while the execution of line 7 in turn is controlled by line 6. There is a data dependency from a statement $s_1$ to a statement $s_2$, if $s_1$ writes a variable that is read in $s_2$ and there is at least one possible execution on which $s_1$ is the last statement writing this variable before reaching $s_2$. In our example there is a data dependency from statement 2 to statement 9 as the for-loop might not be executed.

In extension to the established definition our data dependencies take as well method invocations into account. If a method returns a value without performing side effects we consider the method invocation only as a read access to the object. If the method does have side effects, we consider the invocation as a write and read access to the object. The PDG in Figure 2 gives an example of
Fig. 2. A PDG of the method adults in Figure 1. The invocation of add in line 7 is interpreted as a write on adults leading to the data dependency from 7 to 9. The invocation of getAge in line 5 is interpreted as a read on p, so that there is no data dependency from 5 to 7. The extra statement in line 3 has no data dependencies, so that it can freely be moved below node 1. The data dependencies of the non-unifiable statement in line 5 forbid reordering but allow extracting into a lambda expression.

a data dependency resulting from this approach. This is still a heuristic way to transfer the concept of a data dependency to the object-oriented setting. Deeper analysis could label the data dependencies with a more precise characterization of the state that is changed by the method. In addition alias analysis could find additional hidden dependencies as a change to one object via one variable is a change to the object behind its aliases.

Once we have built the PDGs of the clone candidates, they are compared and nodes for equal or unifiable statements are mapped to each other. Whether two statements are unifiable depends on the refactorings that are considered.

3 Statement Unification

The rename refactoring allows us to consistently change the names of local variables and parameters or even fields and methods. Therefore we take at least the rename refactoring into account. Our example in Figure 1 and 3 illustrates this. If we consider further refactorings more statements become unifiable although some at the price of complexer parameter lists.

Differences in literals can be removed with the introduce parameter refactoring. The generalize types refactoring [7] allows to find differences in type declarations that are more specific than required by the usage of the declared object. If a type generalization is not possible e.g. because different specific return types are required by the callers of a method, the refactoring introduce type parameter can help. Finally method signatures that differ just in the order of parameters can be unified with the reorder parameters refactoring.

In our example the difference between line 5 and line 14 can not be removed and the statement in line 3 has no counterpart in the second method. These differences require changes to the control flow.
4 Control Flow Unification

The PDG contains only as much information about the control flow as is relevant for the state of the variables at each line of the method. Therefore a node can freely be reordered (directly) below the node that controls it as long as the order of nodes with data dependencies is preserved.\footnote{Tsantalis and Chatzigeorgiou in [8] correctly emphasize that there is one additional criterion to be considered. Although a write access to a variable following a read access can not influence the value of the variable, and therefore is not represented by a data dependency, moving the write access upwards may create a data dependency and change the behavior. Adding these so called "anti-dependencies" to the PDG an preserving their order solves the problem. Our example does not show anti-dependencies.} This allows us to separate non-unifiable statement from unifiable statements, as it is the case with node 3 in our example.

Another possibility to “remove” non-unifiable statements is to use the EXTRACT METHOD refactoring. A group of contiguous statements is extractable if the corresponding nodes have to other statements only outgoing data dependencies for at most one variable and no outgoing control dependency [8]. In the PDG in Figure 2 the nodes 5, 6, and 7 together as well as the node 5 on its own is extractable.

The EXTRACT METHOD refactoring is especially helpful if the clones are in classes that are siblings in the class hierarchy. If in this case all differences can be extracted the remaining method can be PULLED UP to a common ancestor of the siblings. This sequence of refactorings is called FORM TEMPLATE METHOD and is explained in detail in [2].

If the classes containing the clones are unrelated the STRATEGY design pattern in combination with TEMPLATE METHOD may be used [1]. But, if there is only one or two differences and these differences are small, these pattern do not pull their weight and the introduction of lambda expressions is the method of choice.\footnote{Lambda expressions are essential for every functional language and have been available for some object-oriented languages as well. Finally they will be introduced to Java in the next version. The planned syntax for lambda expression in Java is explained in JSR 335 [4].} The preconditions for the EXTRACT LAMBDA EXPRESSION refactoring are the same as for EXTRACT METHOD. Our example illustrates the use of lambda expressions to extract the difference between line 5 and 14.

5 Related Work and Conclusion

CloneDifferentiator [10] analyses and visualises the differences between the PDG of clones. The refactorings EXTRACT METHOD, INTRODUCE PARAMETER and the use of Generics are suggested. ARIES [5] calculates metrics to decide whether a refactoring is appropriate. For example EXTRACT METHOD is only recommended when the fragment refers to only a few variables outside the fragment.
Fig. 3. The methods after the clone refactoring. The differing variables were renamed. The extra statement in the first method has been separated from the contiguous block of unifiable statements. The non-unifiable expressions have been extracted into lambda expressions. Lambda expressions consist of a parameter list surrounded by round brackets. The statements of the lambda expression follow after the arrow. If there is only one statement it is possible to omit the return keyword.

We described a process that derives for related clones one (or more) ways to remove the clones, by applying a series of refactorings. The parameters of the refactorings can be precisely (although not necessarily uniquely) derived from the context so that a tool can present precise refactoring suggestions to the developer. Elements of the presented approach such as the generation of the PDG are implemented as part of Cultivate [11].

The approach to search for a sequence of refactoring steps by exploring the different possibilities to unify statements and control flow naturally arises from the problem. As we start with clone candidates found by existing tools, the amount of data to be processed is limited: We know already which methods to compare and do not have to compare all possible method pairs. In addition typically only a few statements in the methods are unifiable, so that the graph matching is not as expensive as in the general case.

References


11. "Cultivate": http://sewiki.iai.uni-bonn.de/cultivate

12. "JTransformer": http://sewiki.iai.uni-bonn.de/jtransformer


A Logic Programming Approach to Integration Network Inference

Daniel Ritter
SAP AG, Technology Development – Process and Network Integration, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
daniel.ritter@sap.com

Abstract. The discovery, representation and reconstruction of (technical) integration networks from Network Mining (NM) raw data is a difficult problem for enterprises. This is due to large and complex IT landscapes within and across enterprise boundaries, heterogeneous technology stacks, and fragmented data. To remain competitive, visibility into the enterprise and partner IT networks on different, interrelated abstraction levels is desirable.

We present an approach to represent and reconstruct the integration networks from NM raw data using logic programming based on first-order logic. The raw data expressed as integration network model is represented as facts, on which rules are applied to reconstruct the network. We have built a system that is used to apply this approach to real-world enterprise landscapes and we report on our experience with this system.

Keywords: Datalog, Integration Networks, Knowledge Representation, Logic Programming, Network Inference, Network Mining

1 Introduction

Enterprises are highly connected to partners and even competitors as part of value chains consisting of business processes. The business document exchange is actually implemented by complex, underlying networks of application and middleware systems, called integration networks. To remain competitive enterprises have to adapt their business processes in a timely and flexible manner, which requires visibility and control over the integration network. However, currently information is locked into systems of an enterprise. To overcome this situation, a new discipline, called Network Mining (NM), strives to discover and extract raw data hidden within heterogeneous systems in complex enterprise landscapes [20].

The raw data implicitly contains information about the integration network, i.e. middleware and application. From that, our system reconstructs integration networks. For the system user, the resulting linked real-world data describing the "as-is" network can then be captured in e.g. network-centric BPMN models [19].

A generalized view of such a network is shown in Fig. 1. When looking at an enterprise landscape, the systems within the integration network can be classified into different categories based on the integration content and the role they
play. The classification provides insight into the capabilities and complexity of
the network and allows to manage business processes, contextualized visualization
and operation on the network. These categories span from applications
with embedded integration or even mediation capabilities, like proxies, enterprise
services, composite applications or applications with service adaptation (Categories I+II),
over standalone Enterprise Service Bus (ESB) or middleware instances with flexible pipeline
processing, e.g. mapping, routing and connectivity for legacy systems (Category III+IV), to Business to Business (B2B) gateways
for cross-enterprise document exchange (Categories V+VI) and system management
solutions, which allow to operate these systems, their software and lifecycle
(Category VII).

In this paper we present an approach to model and reconstruct integration
networks from discovered raw data using logic programming, more precisely
standard Datalog with recursion and stratified negation. We describe how information
in form of NM raw data can be represented independent of their original
domain in a Network Integration Model (NIM) and how user facts can be added.
We have chosen Datalog to represent this model, which we use to develop Dat-
alog programs (i.e. a finite set of Datalog rules) that express the network. That
means identifying entity equivalences, computing edges and semantic references
as well as dealing with user input. We validated our approach on simulated in-
tegration network data and report our experience with the network inference
Datalog system in real-world enterprise networks as well as possible extensions.

In Section 2 we describe the problem domain and state on design principles
and decisions in Section 3. Section 4 defines the NIM and Section 5 introduces
the inference algorithm. Section 6 shows experimental results and states on ex-

Fig. 1. Sample (technical) Integration Network showing logical systems as participants
with embedded integration capabilities and standalone middlewares as well as B2B
gateways

In this paper we present an approach to model and reconstruct integration
networks from discovered raw data using logic programming, more precisely
standard Datalog with recursion and stratified negation. We describe how information
in form of NM raw data can be represented independent of their original
domain in a Network Integration Model (NIM) and how user facts can be added.
We have chosen Datalog to represent this model, which we use to develop Dat-
alog programs (i.e. a finite set of Datalog rules) that express the network. That
means identifying entity equivalences, computing edges and semantic references
as well as dealing with user input. We validated our approach on simulated in-
tegration network data and report our experience with the network inference
Datalog system in real-world enterprise networks as well as possible extensions.

In Section 2 we describe the problem domain and state on design principles
and decisions in Section 3. Section 4 defines the NIM and Section 5 introduces
the inference algorithm. Section 6 shows experimental results and states on ex-
periences. Section 7 concludes with related work, before we draw conclusions and outline future research in Section 8.

2 Motivation

Our premise is that relevant data for computing integration networks is hidden in enterprise system landscapes. However, for that it has to be discovered by NM from mostly disjoint domains in different formats with different meaning [20]. The integration networks derived from the discovered information consist of nodes and edges on different abstraction levels.

The basic entities of the integration network are logical systems (e.g. tenants, applications, integration middleware) and message flows, which are either direct connectivity or mediated communication/ integration. The actual information about these entities as well as their semantics are discovered by Network Mining (NM) systems [20]. However, the discovered raw data is domain-specific and needs to be translated into a domain independent model for network inference, while preserving its semantics. The definition of a Network Integration Model (NIM) is the basis for applying network inference algorithms. Since the raw data comes from disjoint domains, in different formats with different semantics, inference algorithms have to deal with possibly duplicate, fragmented, uncertain or incorrect information while computing the network. Fig. 2 schematically shows some of these challenges. For instance, entity equivalences have to be identified and handled. Direct and transitive edges have to be calculated and semantic relations between nodes have to be inferred. Fig. 2(a) shows systems SX 1 and SX 2 discovered from domain X exchanging messages over middleware system MWX 1, and systems SY 1 and SY 2 discovered from domain Y exchanging messages over middleware system MWY 1. Here, SX 2 and SY 2 denote the same system, as well as MWX 1 and SY 1 are equivalent. Based on the inferred equivalences, the nodes are partitioned as equivalence relations Eq, i.e. Eq(MWX 1 , SY 1 ) and Eq(SX 2 , SY 2 ), and the edges are computed accordingly (see Fig. 2(b)). Systems or applications run on physical hosts, e.g. H 1 from discovery domain Host. The relationships between systems and hosts are not considered as edges but semantic references within the network. Hosts build the bridge to the related domain of system management networks, which are addressed by [18, 11]. A new host CS 1 is added to the network as user knowledge on which SY 1 runs. When merging systems MWX 1 and SY 1 the semantic relation is preserved.

3 Design Principles and Decisions

The major design decisions taken were about finding a representation for an integration model and a language to express inference algorithms. We needed to select (1) an approach, which does not require to modify the system when changing the inference programs or the integration model, (2) a well-understood
representation for information suitable for the inference approach, and (3) a sufficiently powerful inference technique, simple enough to be used by our customers and partners to define their own inference programs.

The necessity of (1) is derived from developing the inference programs in the early prototypes. The domain of the data and the scope of inference evolved - and it will continue to do so as more data sources are integrated and inference is refined. Hence the lifecycle of the data model and of the inference programs needs to be decoupled from that of the system. Since system landscapes and business networks for large enterprises are very complex and many implementations need customer-specific modifications or extensions both (2) and (3) are required. As the relational model is a foundation for most business applications and is thus well-understood by customers, it is a natural choice for (2). Consequently, we initially considered SQL and its imperative extensions to express inference programs. However, as network analysis and inference are expressed more naturally using recursive rules we moved towards logic programming languages like Prolog or Datalog, choosing Datalog for its simpler semantics.

4 The Integration Network (Inference) Model

The model for representing integration networks as virtual ”as-is” enterprise landscape covers a representative intersection of entities from the enterprise integration middleware space [15]. Although this domain has many aspects, which are even differently treated in different system implementations, we identified a common, core meta-model, which we call Network Integration Model (NIM). The basic NIM entities relevant for the inference are introduced subsequently, while more entities might be explained later where necessary.
The base premise for defining an integration meta-model is to represent the actual physical hosts in the enterprise landscape as first class entities and then find the interfaces provided or called by them during message-based communication. Since most of the communication actually addresses logical entities like applications or tenants, called systems, running on the physical hosts, a System is considered a node of the network. That means, systems represent (business) application and integration logic. For the communication with other systems via messages the MessageFlow represents edges in the network. Technically, messages are exchanged over interfaces, Interface, and channels, containing e.g. service bindings and operations, which we represent as IncomingConfiguration and OutgoingConfiguration. The inbound and outbound configurations are considered separate entities, since they carry important information about the message flows, thus helping to reconstruct the network's edges. This notion can also be found in a common graph traversal algebra to set custom processors or actions when entering or leaving a node [22]. Fig. 3 shows the basic NIM entities and their relations.

5 The Network Inference Approach

The algorithm for computing integration networks consists of multiple steps, which have been identified for a parallel analysis allowing it to scale across large datasets of NM raw data. Since the information is represented in the NIM, the inference mechanism is independent of the specific integration and system domains. As discussed in Section 2, unique systems and hosts are identified by equivalence algorithms and semantic links between hosts and systems are computed (step 1). Based on that, incoming and outgoing configurations are identified (step 2) and then used to reconstruct message flows through building separate call graphs (steps 3,4) which are merged afterwards (step 5). Then message flows are linked with application and integration content (step 6) and
user knowledge is integrated. With user knowledge, the quality of the inference mechanism can be improved and information complemented or enriched. Within the inference programs, all user knowledge literals end with the "user" postfix, while discovered knowledge ends with "disc" (i.e. edb relation).

To formalize the network reconstruction, a logic programming approach is used, in which the algorithms are described by Datalog rules and the discovered raw data is a set of Datalog facts according to NIM. The different processes of adding newly discovered information and removing outdated is continuous. For that, each piece of discovered information is annotated with a timestamp. However, instead of removing outdated information that is referenced by higher layer information models as in [19], it is kept and marked outdated until it is not referenced anymore.

**Step 1: Identify unique hosts and systems** To identify hosts and systems uniquely through building equivalence classes, the single instances have to be identified. While hosts can be identified by e.g. host name, IP-address, the systems have no universally applicable identification scheme, thus they are usually identified using context dependent identifiers. For instance, the set of host identifiers can be an IP-address, the DNS name, and a host name. This information mainly comes from different, disjoint instances of system management software, mostly from IT service management [18] and virtualization systems [11]. All identifiers are contained in the equivalence class and any reference to one of them identifies the host. While these equivalence classes are not stable over time, it is quite likely that at least one of the elements of an equivalence class does not change if another one changes, thus making the identification more robust. That way, identity can be maintained over long periods of time in the presence of constant but gradual change. The raw facts from NM are host_disc(host_id, URI) and system_disc(sys_id, URI), which relate a host_id or sys_id to an addressable URI. Relations like same_host_disc(host_id1, host_id2) and same_sys_disc(sys_id1, sys_id2) connect two host or system identifiers, e.g. which refer to the same physical host or logical system. The semantic relation runs_on_disc(sys_id, host_id) connects a system to the host that it runs on. For simplicity, homogenous clusters of machines are also considered as one host.

**Listing 1.1.** Host equivalence exploiting information about system landscape

```prolog
same_sys(?sys_id1, ?sys_id2) :-
  same_disc(?sys_id1, ?sys_id2).

same_sys(?sys_id1, ?sys_id2) :-
  same_sys(?sys_id1, ?sys_id3),
  same_sys(?sys_id3, ?sys_id2).

same_host(?host_id1, ?host_id2) :-
  runs_on_disc(?sys_id1, ?host_id1),
  runs_on_disc(?sys_id2, ?host_id2),
  same_sys(?sys_id1, ?sys_id2).
```

86
Based on that, rules for e.g. `same_sys` and `same_host` are used to infer equivalence classes that allow to write rules that exploit the information about system landscapes. For instance, more than one system can run on one physical host, but one system cannot run on more than one host, Listing 1.1.

**Step 2: Determine Incoming and Outgoing Calls** In current middleware route configurations, the senders of incoming calls to the system can be registered but are mostly unknown. On the other hand, components like the file adapter and the Apple Push Notification Service (APNS) always contain the sender system [15]. However, for outgoing calls from the sender system, e.g. via HTTP, SOAP, receiver or outgoing call configurations are needed to initiate the message flow to the receiver. This results in an outgoing and incoming call graph depicted in Fig. 4(a). The `incoming_disc(sys_id, URI)` and `outgoing_disc(sys_id, URI)` facts relate a `sys_id` to a `URI` of an incoming configuration or an outgoing configuration for the identified system.

![Outgoing/ incoming configuration call graph](image1)

![Call graph extension](image2)

*Fig. 4. Outgoing and incoming configuration call graphs*

**Step 3: Determine Message Flows based on Outgoing Calls** Since outgoing calls are made to a particular endpoint, the corresponding call configurations contain an identifier for the receiving host or system. These identifiers can then be matched against the identifiers that were determined in step 1. If no identifiers are available, these call configurations are processed in step 4. To relate outgoing call configurations to receiver systems `recv_disc(URI, sys_id)` relates a `URI` to an outgoing configuration to a `sys_id` that identifies a receiving system or similarly `recv_host_disc(URI, host_id)` for hosts.
Listing 1.2. Message flow from outgoing configuration

```prolog
msg_flow(?sys_id_snd, ?sys_id_recv) :-
online_disc(?sys_id_snd, ?RCONF),
recv_disc(?RCONF, ?sys_id_recv).
```

Listing 1.3. Message flow for host configurations

```prolog
msg_flow_host(?host_id_send, ?host_id_recv) :-
rules_on_disc(?sys_id_snd, ?host_id_send),
online_disc(?sys_id_snd, ?RCONF),
recv_host_disc(?RCONF, ?host_id_recv).
```

Then message_flow(sys_id_snd, sys_id_recv) rules determine the message flows between systems (Listing 1.2) and message_flow_host(host_id_send, host_id_recv) between hosts (Listing 1.3). That results into a an extension of the call graph shown in Fig. 4(b), in which $S1$ represents a system connected to other systems via incoming and outgoing configurations.

Step 4: Determine Message Flows based on Incoming Calls Similar to the previous step, incoming call configurations are identified. For that, send_disc(URI, sys_id) facts are related via URI to incoming configurations. Again, this results in an extension of the call graph.

Step 5: Merge Call Graphs for a System So far unique hosts and systems are identified and message flows are determined for a single system. Now, the identified incoming and outgoing call configurations from different systems are matched. This is done by matching compatible protocols, message types, etc. After new message flows are identified, the call graph is extended by the merged information (see Fig. 5). In case some incoming or outgoing call configurations do not match to already identified call configurations, they are kept in the model as "unlinked" configurations for matching new configurations.

Step 6: Link Message Flows to Application and Integration Content The outgoing and incoming call configurations with hosts and systems result in a view of the network. However, these message flows only conclude communication between hosts and systems. The outgoing and incoming call configurations also have a link to application and integration content deployed and running on the systems. This content refers to the particular process or integration steps that trigger outgoing calls or receive incoming calls. In other words, process models [1] and middleware routes [15], i.e. integration flow (IFlow) or integration process, give insight into the details within systems and hosts and could be used to correlate operational data to trace messages through middleware systems.

Listing 1.4. Identifying IFlows

```prolog
iflow(?sys_id_snd, ?sys_id_recv, ?sys_id_mw, ?URI) :-
msg_flow_disc(?sys_id_snd, ?sys_id_mw, ?URI),
```
Fig. 5. Call graph extended by merged information from different systems

For instance, the IFlow \textit{iflow}(sendsys\_id, recsys\_id, mwsys\_id, URI) relates senders to receivers through a middleware system, which can be calculated e.g. through the rule in Listing 1.4.

6 Results and Experiences

For the evaluation of our approach, we used our Datalog system, which is a basic Datalog implementation in Java/OSGi based on [24], that allows to evaluate recursive rules and supports basic data types, comparisons and expressions in Datalog rules. The raw data comes from our Network Mining prototype, which discovers information in our testbed and transforms it to NIM Datalog facts. The testbed consists of two middleware systems, i.e. \textit{HXP} and \textit{H73}, of different releases for mediated communication, and have embedded IDoc and WebService capabilities for direct communication and a System Landscape Directory (SLD).
for system management information. This setup contains real-world conditions which we found in our customer landscapes, e.g. cross-middleware inference, combination of embedded and mediated communication, fragmented information registered in different domains.

The results of the experiments are shown, e.g. for systems and message flows in HXP in Table 1, 2 and for H73 in Table 3, 4. The tables show two aspects of the system, namely the discovery and the inference quality. For the inference, the entries for systems and message flows as well as top-level connections are important. The discovery is mainly depicted by attribute entries for the network entities and show minor gaps in the discovery process, e.g. in the category "Correct System Attributes" (see Table 1). For the HXP-PI system, 12 nodes and

<table>
<thead>
<tr>
<th>Category</th>
<th>Absolute Value</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found expected Systems</td>
<td>12</td>
<td>100%</td>
</tr>
<tr>
<td>Correct System Attributes</td>
<td>35</td>
<td>64%</td>
</tr>
<tr>
<td>System Attributes with Limitation</td>
<td>20</td>
<td>36%</td>
</tr>
</tbody>
</table>

55 node attributes are expected (see Table 1). In total 13 top-level connections are expected which group 31 message flows (see Table 2). Furthermore, the top-level connection have 26 attributes, while the message flows have 372 attributes.

<table>
<thead>
<tr>
<th>Category</th>
<th>Absolute Value</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found Expected Top-Level Connection Groups</td>
<td>13</td>
<td>100%</td>
</tr>
<tr>
<td>Correct Top-Level Connection Group Attributes</td>
<td>26</td>
<td>100%</td>
</tr>
<tr>
<td>Found Expected MessageFlows</td>
<td>31</td>
<td>100%</td>
</tr>
<tr>
<td>Correct MessageFlow Attributes</td>
<td>337</td>
<td>91%</td>
</tr>
<tr>
<td>MessageFlow Attributes with Limitation</td>
<td>34</td>
<td>9%</td>
</tr>
</tbody>
</table>

For the cross middleware systems and message flow inference, in total 18 unique, logical systems were inferred from 29 partially duplicate raw facts via equivalence determination (see Table 1 and 3) and 34 message flows have to be reconstructed and grouped to 17 top-level connections using incoming and outgoing call graph merge operations. For instance, logical system HXP_105 from PI-HXP with runs-on host id xxx2474 from SLD was found in the middleware configuration and SLD system information facts and merged into an equivalence class (see Table 5). At the same time, the corresponding message flows between HXP_105 to HXP_106 were reconstructed from PI configuration (conf.) and runtime (runt.) data connected to the system equivalence sets and merged into an
top-level connection group (see Table 6). The group consists of the message flow over sender interface FlightSeatAvailQuery to system HXP.106, which checks for free seats and is followed by a message to the same system over interface BookOrderRequest in case of a positive answer to the first query. If the booking order request was successful, system HXP.106 answers over interface FlightBookOrderConfirm to confirm the request. No unexpected systems or message flows were found and the complete network structure was reconstructed correctly.

Similarly, the H73-PI system has 3 parties, i.e. B2B contexts, 6 expected nodes with 31 attributes (for Table 3), 4 top-level connections, grouping 6 flows (for Table 4), with 8 attributes on the top-level connections and 78 on the message flows.

Table 3. Inference Result on BNM Testbed (Nodes of H73-PI System)

<table>
<thead>
<tr>
<th>Category</th>
<th>Absolute Value</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found expected Parties</td>
<td>3</td>
<td>100%</td>
</tr>
<tr>
<td>Found expected System</td>
<td>6</td>
<td>100%</td>
</tr>
<tr>
<td>Correct System Attributes</td>
<td>22</td>
<td>71%</td>
</tr>
</tbody>
</table>

Table 4. Inference Result on BNM Testbed (Edges of H73-PI System)

<table>
<thead>
<tr>
<th>Category</th>
<th>Absolute Value</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found Expected Top-Level Connection Groups</td>
<td>4</td>
<td>100%</td>
</tr>
<tr>
<td>Correct Top-Level Connection Group Attributes</td>
<td>8</td>
<td>100%</td>
</tr>
<tr>
<td>Top-Level Connection Group Attributes with Limitation</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Found Expected MessageFlows</td>
<td>6</td>
<td>100%</td>
</tr>
<tr>
<td>Correct MessageFlow Attributes</td>
<td>75</td>
<td>96%</td>
</tr>
<tr>
<td>MessageFlow Attributes with Limitation</td>
<td>3</td>
<td>4%</td>
</tr>
</tbody>
</table>

The detailed inference results are only shown partially due to the mass of data discovered. Hence Table 5 shows an excerpt of the results of systems with the discovered description, the inferred host and the equivalence class denoted by “discovered system”. Similarly, an excerpt of the inferred message flows are shown in Table 6. For that, the top-level connections, i.e. grouped message flows are listed with their message flows denoted by sender and receiver and the type of discovered facts from which the data came from (as “From”). In the excerpt, all message flows themselves build an equivalence class of same flows found in runtime logs (runt.) and configuration (config.).

Due to good results in our testbed, we applied the system to real-world customer landscapes as shown in Fig. 6. This real-world validation was very successful on both counts. Firstly, it proved that the auto-discovery and inference is indeed feasible and resulted in highly reliable results. Secondly, our system would
be quite helpful in the everyday work of an integration architect, consultant or integration developer, since it gives an overview of the complete integration network which is currently not possible within the integration middleware tools. The system reduces the effort to document integration scenarios substantially, in particularly by a foreseen export of network details into PDF or office format. That helps to answer specific questions about the network, which are currently still impossible (or difficult) to achieve. For example, when combining configuration and runtime data it is possible to find connections that are not used any longer or were seldom used in a given period of time. Hence, one of the customers plan an upgrade project and with such a system a substantial migration time and effort will be saved.

7 Related Work

Our approach for integration network representation and inference is based on Datalog, which is a well-researched topic [12, 24] that had its revival recently due to good parallelization capabilities, latest through the work of Hellerstein et al. [2, 14]. Even in the enterprise analytics domain, Datalog was recently applied, mainly through work of [5–7]. However, these approaches address non-network inference domains for which they define extensions.

In terms of the meta-model for integration network, [22] represents closest known related work, in which a path algebra is defined that is used to traverse arbitrary graphs. Similarly we define nodes and edges with inbound and outbound connectors, however different in terms of meaning and usage.

<table>
<thead>
<tr>
<th>System (Name)</th>
<th>Description</th>
<th>Host</th>
<th>Discovered System</th>
</tr>
</thead>
<tbody>
<tr>
<td>HXP_105</td>
<td>Booking System</td>
<td>HXP on xxx2474 SLD as Bus. System</td>
<td>PI as Bus. System</td>
</tr>
<tr>
<td>HXP_106</td>
<td>Lufthansa</td>
<td>HXP on xxx2474 SLD as Bus. System</td>
<td>PI as Bus. System</td>
</tr>
<tr>
<td>HXP_107</td>
<td>American Airlines</td>
<td>HXP on xxx2474 SLD as Bus. System</td>
<td>PI as Bus. System</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Interflug</td>
<td>Interflug unknown</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Singapore</td>
<td>Singapore Airlines unknown</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6. Excerpt of HXP-PI message flow and top-level Inference Result

<table>
<thead>
<tr>
<th>Top-level Connect.</th>
<th>Interface</th>
<th>Sender</th>
<th>Receiver</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
For BNM systems in general, related work is conducted in the area of Process Mining (PM) initiated by [1], which sits between computational intelligence and data mining. It has similar requirements for data discovery, conformance and enhancement with respect to NM [20], but does not work with network models and inference. PM exclusively strives to derive BPM models from process logs. Hence PM complements BNM in the area of business process discovery.

Gaining insight into the network of physical and virtual nodes within enterprises is only addressed by the Host entity in NIM, since it is not primarily relevant for visualizing and operating integration networks. This domain is mainly addressed by the IT service management [18] and virtualization community [11], which could be considered when introducing physical entities to our meta-model.

The linked (web) data research shares similar approaches and methodologies, which have so far neglected linked data within enterprises and mainly focused on RDF-based approaches [9, 10]. Applications of Datalog in the area of linked data [21, 8] and semantic web [16] show that it is used in the inference domain, however not used for network inference.

8 Discussion and Future Work

In this paper we introduce a new domain for information discovery, machine learning, and network reconstruction, for which we defined a modeling and inference approach to reconstruct integration networks from NM raw data using Datalog. The network model developed specifically for the connectivity and integration domains and covers an intersection of the relevant entities, which we
derived through the analysis of several middleware systems on the market. We encoded the discovered raw data as Datalog facts to create a domain independent knowledge base and applied rule-based inference representing a multi-step network inference approach. We validated our approach on a simulated integration network and reported our experiments on applying our system to real-world enterprise networks. The evaluation shows good results with respect to the challenges like equivalence class determination, flow- and cross-middleware network reconstruction as introduced in Section 2. Although the network structure could be reconstructed very well, the discovery range should be improved to attach more integration details to the attributes of the network entity instances.

Future work will be conducted in several areas, among them the improvement of the discovery range, the inference of business process models from NM data and the correlation to integration networks as well as extensions to standard Datalog to improve the current implementation. For instance, the efficient compilation of Datalog programs to current hardware [17], distributed systems [23] or pruning with CHR [4] could guarantee more efficient Datalog processing. Since not all facts have the same certainty, we will also look into probabilistic extensions of Datalog like [25, 13], which could help to express different levels of certainty with respect to network model instances. The work conducted in [3] will be considered for time aspects, which could help to prune large, outdated networks from system landscapes with historical data.

References

CHR meets Google Docs

Slim Abdelnader and Mahitab Sherief Sharawy

German University in Cairo, Computer Science Department, New Cairo City. Main Entrance El Tagamo El Khames, Cairo, Egypt
{slim.abdelnader,mahitab.sherief}@guc.edu.eg
http://met.guc.edu.eg/

Abstract. As the need to share and store documents online increased vastly, Google Docs, a web-based office suite and data storage service, became a very popular and used application. In order to automate the reasoning and the processing on stored data, our approach is to use Constraint Handling Rules (CHR) to offer users reasoning services on their stored data. In this paper, we will introduce an interface between Google Documents and CHR. We will show how users can interact with the interface to implement some reasoning methods on spreadsheets and calendars.

Keywords: CHR, Constraint Handling Rules, Constraint reasoning, Google Docs, Google Spreadsheets, Google Calendar

1 Introduction

We are living in the age of Internet, where everything is shared on it; personal information, businesses, online documents and all other types of info and media. Online documents and calendars such as Google Docs and Google Calendar is being widely used nowadays, as it is accessible from everywhere, easier to share with others, more organized and saves storage.

Since Google Docs was created in 2005 with its rapid updates and new features, its users have been increasing rapidly; storing most of their documents online and using the Google Calendar to organize their day and manage their time. We will use the popularity of Google Docs and Calendar to reason on the data stored in them using Constraint Handling Rules (CHR) offering users useful applications (e.g. consistency check, scheduling). These applications depend of course on the type of data the users store in their documents.

CHR is a declarative programming language extension [1]. CHR is not commonly used as a programming language on its own. It is commonly used to extend a host language with constraints. Current host languages include Prolog, Java and Haskell. Prolog is the most popular host language used and it is the host language used in this project. CHR with its declarativity, ease of use and being a constraint-based language, can be used to perform useful reasonings on the data stored in Google documents offering users useful applications. The aim of this project is to combine the strength of CHR with the popularity of Google
documents using Google APIs to offer users reasoning services that can be applied on the data stored in their documents. Google APIs allow programmers to create applications that can read and write data from Google Docs. The Java version of Google APIs was used in order to combine Google Docs with CHR using the JPL library. JPL is the SWI Prolog Java library that allows users to consult and run prolog programs using Java.

Summing all the above, Google Docs and Calendars are being used by many users. CHR with its reasoning power can be used to offer these users some useful applications that can make their personal and business life easier.

This paper is organized as follows: in Section 2, we will be discussing the background information needed for our work which includes the following topics: CHR, Google Documents and Google APIs. In Section 3, we will explain the system architecture. In Section 4, we will explain the application developed. And finally, the conclusion and the future work will be discussed in Section 5.

2 Background

In this section, we present the background information needed for our work which can be divided into three main sections: Constraint Handling Rules (CHR), Google Documents and Google APIs.

2.1 Constraint Handling Rules (CHR)

Constraint Handling Rules (CHR) is a high-level language especially designed for writing constraint solvers [1]. CHR is essentially a committed-choice language consisting of multi-headed rules that transform constraints into simpler ones until they are solved. Over the last decade CHR has not only taken its place as a special-purpose language for writing constraint solvers, but has matured into a general-purpose language for computational logic and beyond. In CHR, one distinguishes between two main kinds of rules. Simplification rules replace constraints by simpler constraints while preserving logical equivalence, e.g.

\[ X \leq Y \land Y \leq X \leftrightarrow X = Y. \]  

(1)

Propagation rules add new constraints that are logically redundant but may cause further simplification, e.g.

\[ X \leq Y \land Z \leq Z \rightarrow X \leq Z. \]  

(2)

Simpagation rules are a mix of propagation and simplification where the rule’s head is divided into two parts. The first part remains in the constraint store of CHR after the rule is fired while the second part is replaced by the body e.g.

\[ X \leq Y \setminus X \leq Y \leftrightarrow true. \]  

(3)
Guards can be used in rules if the firing of these rules depends on some condition. A guard is written inside the rule and it represents a condition that has to be true for the rule to be fired. Guards are written in the host language.

\[ H \implies G | B. \]  
\[ H1 \iff G | B. \]  
\[ H1 \setminus H2 \iff G | B. \]

### 2.2 Google Docs

Google Docs is a free, Web-based word processor, spreadsheet, presentation, form, and data storage service offered by Google. It allows users to create and edit documents online while collaborating in real-time with other users. Documents, spreadsheets, presentations can be created with Google Docs, imported through the web interface, or sent via email. Documents can be saved to a user’s local computer in a variety of formats including: ODF, HTML, PDF, RTF, Text, and Microsoft Word. Documents are automatically saved to Google’s servers to prevent data loss, and a revision history is automatically kept. Documents can be tagged and archived for organizational purposes. The service is officially supported on recent versions of the Firefox, Internet Explorer, Safari and Chrome browsers running on Microsoft Windows, Apple OS X, and Linux operating systems.

### 2.3 Google APIs

The Google Data APIs allow programmers to create applications that read and write data from Google services. Currently, these include APIs for Google Apps, Google Analytics, Blogger, Google Base, Google Book Search, Google Calendar, Google Code Search, Google Earth, Google Spreadsheets, Google Notebook, and Picasa Web Albums. In this project, Google spreadsheets and calendars APIs will be used. The APIs are offered in different languages (python, Java, .Net). The Java version of the APIs was used in this project.

### 3 System Architecture

One of the main challenges we faced during the implementation was to create an environment that combines the web application with CHR and Google APIs. Since the APIs and the web application are in Java and there is a Java library to send inputs to a prolog compiler and receive the solution, Java was used to create that environment.

First, users choose their spreadsheets (data spreadsheet and constraints spreadsheet (optional)) or calendar and enter their CHR solver. After that Google APIs are used to get the data from the Google Spreadsheets or Calendar, and then the retrieved data is parsed to form the constraint store. The produced constraint
store is then passed to the Prolog compiler where the reasoning is performed. Finally, the reasoning solutions are then retrieved and the output constraint store will be posted in the user’s spreadsheet or calendar. The system architecture is depicted in figure 1.

4 Main Application

The aim of this application is to allow CHR users to define their own solvers in order to manipulate the data stored in Google Spreadsheets and Calendars.

4.1 The Connection

The connection between CHR and Google Docs was the first main issue we had to resolve. An environment that combined a web application with both CHR and Google Docs was what we were aiming for.

We used Play framework [3] to create the web application for this project. Play is an open source web application framework, written in Java, which follows the model-view-controller architectural pattern. The Play framework is easy to install and use. The most important feature in Play framework is that users can easily add external libraries to the framework; which is the main reason why this framework was used.

With the web application ready to accept any external libraries, we added all the java libraries of the Google Spreadsheets and Calendars APIs. In addition to these libraries, we added the JPL library offered by SWI prolog in order to connect to CHR.
4.2 Spreadsheets

**Input Data Representation** During implementation, the first challenge that we faced was how to represent the spreadsheet in a way that the user can easily use to write a solver for any problem. First we thought of representing each cell as a unary constraint with its column header name. For example, the following spreadsheet

![Spreadsheet example 1](image)

will be translated to:

\[ c1(2) , c1(1), c2(1), c2(3). \]

This worked when there were no direct relationships between the columns. But in the case where columns depended on each other, this representation failed to capture this dependency. Like for example in the following spreadsheet:

![Spreadsheet example 2](image)

Where there is a list of tasks where some should be done before the others. Representing the following using our initial representation will not make any sense and will not capture the relation at all. For instance, the first two rows will be represented as follows:

\[
\text{tasks(a), precedence(b), tasks(a), precedence(c), tasks(b), precedence(d), tasks(c), tasks(d), tasks(e), precedence(d).}
\]
It is quite obvious that this representation says nothing for example about which task should be done first with respect to others. So, we thought of allowing the user to not only define his/her solver but also to define the form of the constraints used in that solver. This is done through letting the user save all his constraints in a spreadsheet and the application will automatically parse it and produce the constraint store. The user is provided with a template for the constraints spreadsheet and the user has to follow it. Figure 4 depicts how a constraints spreadsheet would look like.

![Constraints Spreadsheet](Image)

Where `cname` column should contain constraint name, `argsNum` contains the total number of arguments passed to this constraint and `args1, args2, .., argsN` contain the column names from where the arguments should be passed.

![Data Spreadsheet](Image)

So with a constraints spreadsheet as the one shown in figure 4 and the data spreadsheet shown in figure 5, the initial query sent to the constraint solver would look like the following:

\[
\text{sum(1), sum(5), avg(2,3), avg(6,7), test(1,2,3,4), test(5,6,7,8)}
\]

We focused on this issue as it is crucial for the user as the solver mainly depends on how the data is represented. This means that the solver written for a certain application will be different for two different data representations. We will explain this point with an example of a solver written to calculate the sum of three columns. First, if the data is represented as one constraint with three arguments one from each column, the solver would look like the following:

\[
\text{sum(X,Y,Z) <=> M is X+Y+Z, ans(M).}
\]
While if the data is represented as several unary constraints one for each value, the solver would look like the following:

\[
\text{sum}(X), \text{sum}(Y) \leftrightarrow Z \equiv X+Y, \text{sum}(Z).
\]

It is important to note that both options are available for the user. He/she can either choose to work with the predefined data representation with unary constraints or define his/her own through the constraints spreadsheet.

**Output** The next challenge that we faced after data representation and applying CHR solver to the initial queries formed was how to represent the CHR output in the spreadsheet. Similarly to what we did in data representation, we let the user define how constraints should be posted in his/her spreadsheet and where. In other words, the user specifies which constraints from the output constraint store he/she wants to be published in the spreadsheet, how to publish it and in which column. Figure 6 depicts how the user will specify this information according to a predefined template.

![Output Spreadsheet](image)

**Fig. 6. Output Spreadsheet**

Where *cname* is the constraint name, *how* is how this constraint will be represented and *where* is the column name where the output will be posted. Other fields are required if the user chose the "extract many" option. The *argsNum* is the number of arguments of the constraints and the *argsi* fields defines where each argument will be posted in the spreadsheet.

The user has three representations of constraints to choose from and write in the how column:

- **Extract**: the user should use this when the specified constraint has only one argument where he/she wishes to post. For example, if the output is the sum of entries in a column and is represented by the constraint \( \text{sum}(X) \), then \( X \) will be extracted and posted where the user specified.
- **Extract many**: this option is similar to the "Extract" option but works on constraints with multiple arguments.
- **List**: this is best used when the specified constraint has one argument which is a list. For example a constraint holding the start times of some tasks represented by the constraint \( \text{tasks}([s1,s2,s3]) \) then the elements of the list will be extracted and posted one after another in the specified column.
- **As is**: posts the output constraint store as it is in the specified place.
**Practical example** In this section we will further demonstrate the interface with a practical example.

The example we will use is the Shortest Path Problem. In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph, where the sum of weights of all edges forming the path is minimized. This can be used in finding the shortest and fastest way to get from one location to another using a road map. In this case, the locations are represented by vertices and the roads are represented by the edges of the graph; the weight of the edges will be the time needed to travel the road the edge represents.

We will explain how a CHR solver for the Shortest Path Problem [4] can be used through our interface. We will show how the data will be represented in the spreadsheet, how to form the constraints spreadsheet, how the initial query will look like and finally the output of the solver. The solver we will use computes the shortest length for each path for a directed graph. For a directed graph with edges:

\[ A_1 \rightarrow B_1, A_2 \rightarrow B_2, \ldots, A_n \rightarrow B_n \]

these edges are represented as the following constraints:

\[ \text{edge}(A_1,B_1), \text{edge}(A_2,B_2), \ldots, \text{edge}(A_n,B_n) \]

the output of the solver is the shortest paths from \( A_i \) to \( B_i \) with length \( L_i \) (\( 1 \leq i \leq m \)), represented as follows:

\[ \text{path}(A_1,B_1,L_1), \ldots, \text{path}(A_m,B_m,L_m). \]

In order to work with this solver, we have to construct the data and the constraints spreadsheets. The data spreadsheet would look like the following:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>vertex1</td>
<td>vertex2</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>a</td>
</tr>
</tbody>
</table>

**Fig. 7.** Shortest Path Problem data spreadsheet

where each edge is represented with two vertices, the first is from the column vertex1 and the second is from the column vertex2. The constraints spreadsheet would only contain the "edge" constraint and is depicted in figure 8.

The initial query for these spreadsheets would be:
Fig. 8. Shortest Path Problem constraints spreadsheet

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cname</td>
<td>argsNum</td>
<td>arg1</td>
<td>arg2</td>
</tr>
<tr>
<td>2</td>
<td>edge</td>
<td>2 vertex1</td>
<td>vertex2</td>
<td></td>
</tr>
</tbody>
</table>

The output would be all the initial constraints of edges in addition to the following path constraints:

- path(a, b, 1), path(a, c, 2), path(a, d, 3),
- path(b, c, 1), path(b, d, 2), path(c, d, 1),
- path(e, a, 2), path(e, b, 3), path(e, c, 4),
- path(e, d, 5), path(e, f, 1), path(f, a, 1),
- path(f, b, 2), path(f, c, 3), path(f, d, 4)

Now the user wants to publish this output, so he/she specifies that the path constraint should be posted and chooses the "extract many" option. The output spreadsheet is specified in figure 9.

Fig. 9. Output Spreadsheet for shortest path problem

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c</td>
<td>how</td>
<td>where</td>
<td>argsNum</td>
<td>args1</td>
<td>args2</td>
<td>args3</td>
</tr>
<tr>
<td>2</td>
<td>path</td>
<td>extract many</td>
<td>-</td>
<td>3</td>
<td>v1</td>
<td>v2</td>
<td>weight</td>
</tr>
</tbody>
</table>

According to the output and the specified output spreadsheet, the data spreadsheet will look like the one depicted in figure 10.

In addition to the normal way of posting the output to the spreadsheet according to the specified output spreadsheet, the output can also be visualized using the CHR visualization tool found in [2].

4.3 Calendars

After dealing with the Google Spreadsheets, we move to Google Calendars. Google Calendar is a free online calendar that helps users create their calendars, keep track of their events and share their schedules with friends and family.

Input Data Representation In calendars, we only have one data representation of the events which is done using universal constraints so as not to lose any information.
So, for each event a constraint is composed with the following arguments: title of event, date start, start time, date end and end time. So the constraint will have five arguments in the stated order; we can also add more arguments, for example, where the event will take place. So for a calendar that looks like the one captured in figure 11, the initial representation of the calendar would look like the following:

\[
\text{calEvent(meeting,2012-07-03,11:00:00,2012-07-03,12:00:00),}
\text{calEvent(shopping,2012-07-04,18:00:00,2012-07-04,20:00:00),}
\text{calEvent(gym,2012-07-05,15:00:00,2012-07-05,16:00:00).
}\]

After further investigation we found out that this representation as easy and intuitive as it is but it is hard to use in the solvers for comparison purposes. So we thought of representing everything in form of seconds. So, the start and end times of the tasks are represented in seconds starting from 12 am which corresponds to zero seconds. Also the start and end dates of the events are represented as the number of seconds since January 1, 1970, 00:00:00 GMT. This is done to facilitate the comparison while reasoning. In this case the initial representation of the calendar would look like the following:

\[
\text{calEvent(meeting,61302088800,39600,61302088800,43200),}
\]
calEvent(shopping,61302175200,64800,61302175200,72000),
calEvent(gym,61302261600,54000,61302261600,57600).

But after running several trials, we found out that the "JPL" library that we are using to coordinate java and prolog converts 'long' values into integer ones. This means that information would be lost.

So returning to our first attempt, we thought of having the same representation but dividing the date and time components into different arguments. Hence, the initial representation of the calendar would look like the following:

calEvent(meeting,2012,07,03,11,00,00,2012,07,03,12,00,00),
calEvent(shopping,2012,07,04,18,00,00,2012,07,04,20,00,00),
calEvent(gym,2012,07,06,16,00,00,2012,07,05,17,00,00).

In addition to the main user, the user can add other users in order to consider their calendars. This is done in order to facilitate the solver for the user. So, if a meeting should be scheduled, the other users are taken into account while running the solver.

Moreover, there is an option for the user to add a certain time slot to be considered for scheduling and added to the initial query. So if the user wants to add a certain time slot to the representation of the calendar to form the initial query, then he/she should use this option; but if he/she wants the initial query to only contain the representation of the calendar then he/she should not use this option. If this option is used then the specified time slot will be represented in the same way as any event, i.e. the constraint representing this time slot will also contain an argument for every date and time component in addition to the time slot’s name. The time slot is captured by the constraint named schedule. So if the user wants to add a slot to be scheduled on the 5th of July starting at 4 pm and ending at 6:30 pm, then it will be represented as follows:

schedule(mymeeting,2012,07,05,16,00,00,2012,07,05,18,30,00)
So adding this to the representation of the calendar, the initial query will look like the following:

\[
\text{calEvent(meeting,2012,07,03,11,00,00,2012,07,03,12,00,00)}, \\
\text{calEvent(shopping,2012,07,04,18,00,00,2012,07,04,20,00,00)}, \\
\text{calEvent(gym,2012,07,06,16,00,00,2012,07,05,17,00,00)}, \\
\text{schedule(mymeeting,2012,07,05,16,00,00,2012,07,05,18,30,00)}. \\
\]

Output The problem with calendars is that we can only manipulate events i.e. we can only add, delete or edit events. Besides, it is difficult to know what the user wants to post or do with his/her calendar after applying the solver; this is because the user might just want to check for conflicts or if he/she has any events at after a certain time or so. That is why it is very hard to generalize the way the output can be posted. So in order to make it more convenient to the user, the output of the solver is displayed along with the user's calendar. So the user can check the output and change in the calendars accordingly. This is depicted in figure 12.

![Calendar Output](image)

**Fig. 12. Calendar Output**

Practical Example In this section we will introduce two examples to demonstrate what we explained above.

Let us start with a simple example that just shifts all the events by one hour. Note that the shifting will not be actually done in the calendar but only returned to the user in the calendar output webpage as the one shown in figure 12.

So the solver will be simply represented mainly using the following rule:
calEvent(X, Ys, Ms, Ds, Hs, Mins, Ss, Ye, Me, De, He, Mine, Se) \leftrightarrow 
\text{Hs2 is Hs+1, He2 is He+1,}
\text{outputEvent}(X, Ys, Ms, Ds, Hs2, Mins, Ss, Ye, Me, De, He2, Mine, Se).

This rule is for the normal case; other rules are used to handle border line cases. These cases happen when after shifting the task moves to the next day, month or year.

So for the calendar shown in figure 11 the output will look like the following:

\text{outputEvent(meeting, 2012, 07, 03, 12, 00, 00, 2012, 07, 03, 13, 00, 00),}
\text{outputEvent(shopping, 2012, 07, 04, 19, 00, 00, 2012, 07, 04, 21, 00, 00),}
\text{outputEvent(gym, 2012, 07, 05, 16, 00, 00, 2012, 07, 05, 17, 00, 00).}

The second example uses the option of adding users other than the main one and also adds a time slot to be considered for scheduling. In this example, the user wants to schedule a meeting to fit all other users calendars if possible.

The user adds only one additional user. The main user’s calendar is shown in figure 13 and the additional user’s one is shown in figure 14.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{Fig.13.png}
\caption{Main User Calendar}
\end{figure}

In this example, the main user wants to schedule a meeting on the 5th of July starting at 4 pm and ending at 6:30 pm.

According to the calendars depicted in figures 13 and 14, the initial query will look like the following:

\text{calEvent(gym, 2012, 07, 05, 16, 00, 00, 2012, 07, 05, 17, 00, 00),}
\text{calEvent(shopping, 2012, 07, 04, 18, 00, 00, 2012, 07, 04, 20, 00, 00),}
\text{calEvent(meeting, 2012, 07, 03, 11, 00, 00, 2012, 07, 03, 12, 00, 00),}
\text{calEvent(Ramadan, 2012, 07, 20, 09, 00, 00, 2012, 07, 20, 10, 00, 00),}

109
calEvent(Aerobics,2012,07,17,13,00,00,2012,07,17,14,00,00),
calEvent(Dinner,2012,07,15,19,00,00,2012,07,15,20,00,00),
schedule(mymeeting,2012,07,05,16,00,00,2012,07,05,18,30,00).

First the user checks whether there is an event that conflicts this date using the following rule:

calEvent(E,Ys,Ms,Ds,hs,Mns,Ss,Ye,Me,De,He,Mne,Se),
schedule(E2,Ys2,Ms2,Ds2,hs2,Mns2,Ss2,Ye2,Me2,De2,He2,Mne2,Se2) ==>
inbetween(Ys,Ms,Ds,hs,Mns,Ss,Ye,Me,De,He,Mne,Se, Ys2,Ms2,Ds2,hs2,Mns2,Ss2,Ye2,Me2,De2,He2,Mne2,Se2) | schedule(E2,Ys2,Ms2,Ds2,hs2,Mns2,Ss2,Ye2,Me2,De2,He2,Mne2,Se2),
calEvent2(E1,Ys1,Ms1,Ds1,hs1,Mns1,Ss1,Ye1,Me1,De1,He1,Mne1,Se1),
schedule(E2,Ys2,Ms2,Ds2,hs2,Mns2,Ss2,Ye2,Me2,De2,He2,Mne2,Se2) <= inbetween(Ys1,Ms1,Ds1,hs1,Mns1,Ss1,Ye1,Me1,De1,He1,Mne1,Se1, Ys2,Ms2,Ds2,hs2,Mns2,Ss2,Ye2,Me2,De2,He2,Mne2,Se2) | calEvent2(E1,Ys1,Ms1,Ds1,hs1,Mns1,Ss1,Ye1,Me1,De1,He1,Mne1,Se1),
increment(Ys2,Ms2,Ds2,hs2,Mns2,Ss2,Ye2,Me2,De2,He2,Mne2,Se2, Ys3,Ms3,Ds3,hs3,Mns3,Ss3,Ye3,Me3,De3,He3,Mne3,Se3),
schedule(E2,Ys3,Ms3,Ds3,hs3,Mns3,Ss3,Ye3,Me3,De3,He3,Mne3,Se3).

Fig. 14. Additional User Calendar
Where the \texttt{inbetween} constraint checks whether there is a conflict between the timing of the two events or not. The \texttt{increment} is a prolog method that shifts the time by 30 minutes.

5 Conclusion and Future Work

Constraint handling rules (CHR) is a declarative programming language extension. CHR is a programming language that can be used as a constraint solver to reason on various types of data. On the other hand, Google Docs is a well known online word processor and spreadsheet service offered by Google. Hence, CHR was used to reason on the data stored in Google Docs and Google Calendar.

An environment where CHR is combined with Google Docs using Google APIs was created in order to implement several applications based on that environment. The implemented web interface allows the users to create their own solvers and apply them on their data stored in Google Spreadsheets and Calendars. The users are also allowed to specify how the output of their solvers will be posted to Google Docs.

The created environment smoothed our way into developing any applications and reasoning services using CHR and Google docs. As for the future work, we will try to test the scalability of the tool by applying solvers to solve real world applications on very large spreadsheets and calendars.

References

4. WebCHR Accessed: June 2012. \url{http://chr.informatik.uni-ulm.de/~webchr/}. 

Declarative Web Programming with PROLOG and XUL

Christian Schneiker and Dietmar Seipel
Department of Computer Science
University of Würzburg, Am Hubland, D – 97074 Würzburg, Germany
{christian.schneiker,dietmar.seipel}@uni-wuerzburg.de

Abstract. Modern information systems are more often web-based than simple single PC desktop applications. In the last few years, developers have used common frameworks like GWT, JSF or similar to produce thin or rich client applications with the use of Java server technology for the backend part.
This paper introduces a new way of implementing thin clients with declarative web programming and PROLOG as a powerful server. The focus of the server lies in the integration of databases. GUI scaffolding on the basis of the defined data tables, database schema resolving for generic programming, and database triggered event handling make it possible to develop easy-to-read and reliable code.

1 Introduction

Before Tim Berners-Lee and fellows like Roy Fielding developed HTTP, HTML, and the first browser named WorldWideWeb, working with terminal sessions to connect to a mainframe were already a common scenario. High prices for single computers pushed the users to connect via a terminal, consisting of a monitor and a keyboard, to a single mainframe. These mainframes were high-performance, and, of course, also very high-priced, but they gave many scientists the possibility to get tuned in.

Over the years, with the reduced hardware costs, nearly every single office and also homes have become equipped with a personal computer, not to mention mobile phones and other gadgets. The need for mainframes for computing subsided, and everyone used his own desktop version of the needed programs. With the introduction of the WWW in 1991 by Tim Berners-Lee, which started with a newsgroup message in alt.hypertext, the possibilities were already there for rich or thin client applications, but the existing slow bandwidth made it impossible to fulfill the required user experience like speed, large amount of data, and high computer graphics.

In these days, internet bandwidths with 100 MBit/s are becoming common, and multi-user applications with simultaneously connected users all over the world made it necessary to look for options. Based on HTTP, HTML, CSS and JavaScript and also other technologies, many client/server frameworks have been developed in the recent past. Java-based solutions have to be mentioned, like JSF [12] and GWT [3], as well as libraries for PROLOG, like PROLOG Server Faces (PSF) [8], Type-Oriented Construction of Web User Interfaces [4] or An ER-based Framework for Declarative Web Programming [5].
The aim of this paper is to introduce a PROLOG-based approach to declaratively design thin client applications with the XML dialect XUL (XML User Interface Language) and a few predefined JavaScript functions. Nearly all of the program logic can rest on the server, which is also PROLOG-based, and therefore the whole power of PROLOG can be used. The client itself is OS independent and uses the standard look-and-feel of the operating system. Figure 1 shows a screenshot of such a client.

![Figure 1. Screenshot of a XUL4Pl based client application.](image)

The main focus of our framework is the integration of databases. With XUL4Pl it is possible to parse the involved database’s schema for GUI scaffolding and generic code production. Thus, easy-to-read and highly reliable code could be developed. Another feature are database triggered GUI updates; on a multi-user system, all clients will be updated when a single user changes the data or even the structure of the database.

The rest of the paper is organized as follows: Section 2 gives an overview of XML-based programming technologies like JSF and PsF, XUL, and SOAP, as well as FN- QUERY, a framework for efficiently processing XML data. The main components of XUL4Pl will be described in Section 3. After a short overview of the implementation of the HTTP server, database programming with connection handling, GUI scaffolding and event triggers will be described in detail. Section 4 deals with the implementation of a client/server application.

## 2 XML Based Web Programming

This section describes the technologies on which our framework XUL4Pl is based. The framework itself relies on commonly used technologies, like JavaScript and HTTP connections. These techniques can be found in a wide variety of rich and thin client/server architectures. JSF and PsF are frameworks for Java and PROLOG for programming web applications, and they are using the same functionality in some parts. XUL is an XML-
based library for graphical user interface design, and SOAP handles the communication of data between the clients and the server. Both are extensively used in our framework.

The following subsection describes the architectures on which XUL4PL relies as well as the used frameworks.

2.1 Web Applications

In the last few years, with the modernisation of the network infrastructure in both intranet and internet, it is common to develop new applications based on client/server technologies. The gain in bandwidth gives the opportunity to handle large amounts of data by sending them over the net and to reduce the cost of high performance clients; the server is computing most of the program logic. While in rich client applications some of the logic still rests on the client, there are also a lot of thin clients, where nearly all of the data processing is handled by the server.

Rich client applications are a modern type of software clients which often include a unique framework for developing the client itself and also nested modules and plugins. It is common to give the user the opportunity to modify and extend the standard features. Depending on the used technology, the most used features of rich client applications are – with regard to above mentioned – OS independency, easy updateability, and the possibility of complex user interfaces; they can be used online and offline, because the whole logic is implemented on the client side and can be assisted by a server, if necessary. Data will be only transferred to the server for persistency and communication reasons. Widely used rich client applications are Eclipse and NetBeans for Java development, or Microsoft Visual Studio.

Thin client applications, on the other hand, only implement often used features of the backend logic, while most of the logic still rests on the server. Functions for GUI updates and small calculations are implemented on the client side. The advantages of thin client side applications are the huge scalability, OS independency, and low costs for the hardware for the clients. A disadvantage is of course the need for working network connections. Common thin clients are, e.g., web browsers and terminal applications.

2.2 Common Web Frameworks

Implementing client applications from scratch is far away from state-of-the-art software development. Instead, frameworks are used for, e.g., GUI design, for handling the events sent by the server or the clients, and for the connection to any common database management system. Many different frameworks are brought to the developer; especially Oracle and Sun – with their programming language Java – have focused on the integration of their technologies with common HTML and JavaScript.

This section gives a short overview of Java Server Faces (JSF), implemented by Oracle and Sun, as well as of our previously developed client/server framework Prolog Server Faces (PSF). PSF also uses PROLOG for generating web applications, but with less functionality.
**JavaServer Faces (JSF).** As a framework for server-side user interface components, Sun Microsystems and other companies initially released JavaServer Faces in 2004, using XML for implementing the view of web pages according to the MVC concept. In contrast to static HTML pages or JSP, JSF provides *stateful* web applications, *page templating* or even AJAX support and allows for developing server applications within the object-oriented programming language Java.

In JSF one can process client-generated events and alter states of components, making them event-oriented. It includes *backing beans*, which synchronize Java objects with UI components. Unlike desktop programs, web-based applications are expected to be accessed from different client types, such as desktop browsers, cell phones, and PDAs. JSF provides a flexible architecture allowing it to display components in different ways, and it also offers many validation techniques.

Since it is a server-side technology, all pages requested by the client are preprocessed by the server. Via HTTP, every single requested XML document is transformed to standard XHTML, and nested calls to Java objects formulated in an *expression language* are processed. The following example shows a JSF-XML element which is transformed to standard XHTML. The `selectOneMenu` element has an additional attribute `value` with a Java expression for setting the right value which is read from a data container, namely a Java Bean.

```xml
<h:selectOneMenu id="selectCar" value="#{carBean.currentCar}"
    <f:selectItems value="#{carBean.carList}" />
</h:selectOneMenu>
```

In this example, a list of cars is read from the Bean, and according to the values, a set of `option` elements is generated. The `selectOneMenu` element is transformed to a normal XHTML `select` element, and necessary attributes like `name` and `id` are added. The resulting valid XHTML page is transferred to the client and rendered by a browser.

```xml
<select id="selectCar">
    <option value="corolla">Corolla</option> ...
</select>
```

The framework uses standard Java classes for transforming the documents with common Java component tree operations. Even when the work with object-oriented programming languages and XML tree operations is hard to read and to debug, it makes it possible to extend the core libraries for the transformations (*core taglib*) by writing new classes and by adding them to the library.

**Prolog Server Faces (PSF).** PSF is a *stateful* and *event-driven* framework, that integrates logic programming in modern web applications. We are combining different techniques for mixing PROLOG with XHTML to develop *dynamic* web pages with the advantages of JSF for writing *condensed* XML. This XML will be expanded to XHTML with connection to XML documents and relational databases for data handling. We provide an application programming interface for combining an extended HTTP server implemented in SWI PROLOG with a huge and easy-to-extend tag library for defining web pages in a compact XML structure. For transforming XML elements, we extensively
use the XML transformation language FNTRANSFORM [10], which will be sketched in Subsection 2.5.

Like in JSF, nearly every XHTML element can be written in a compact form with additional attribute values, which read the data from complex term data structures, XML documents, or even relational databases. In our PsF framework, we have implemented the core tag library which consists of tags like HTML form, the different input element types and, of course, radio buttons and select menus.

We want to exemplify the work with PsF-XML files with the following code of a single select menu, whose data are stored in an additional XML file. The PsF-XML page contains only two elements for defining the type of the select menu as well as an element with an FNPATH expression, which handles the data for the different option types, in this case the different car models.

```xml
<h:selectOneMenu id="selectCar">
  <f:selectItems value="#{doc(cars.xml)/car-[@id, @model]}" />
</h:selectOneMenu>
```

The data can be either read from an XML document or from a PROLOG data structure. The transformation itself is handled by FNTRANSFORM, which is integrated in our framework. When a client requests a PsF-XML file, the server automatically transforms it to XHTML with one of its request handlers.

### 2.3 The XML User Interface Language

The XML User Interface Language XUL is an XML dialect for declaratively defining graphical user interfaces. It has been developed by the Mozilla Foundation for implementing platform independent GUI’s for their well-known browser Firefox and the email client Thunderbird. It is also used by a wide spectrum of companies for OS independent client/server applications like Google’s AdWords tool. The following example shows a short code snippet of a modal dialog implemented in XUL.

```xml
<dialog id="newMessageDlg" ...>
  <script type="text/javascript" src="xulfunctions.js"/>
  <dialogheader title="Messages" description="new Messages"/>
  <vbox>
    <hbox>
      <label value="Priority:"/> <menulist> ... </menulist>
      <label value="Date:"/> <datepicker type="popup"/>
      <label value="Time:"/> <timepicker/>
    </hbox>
  </vbox>
</dialog>
```

All XUL elements could be combined with JavaScript and CSS, like in normal HTML webpages. Figure 2 shows the rendered dialog. XUL frames could either be rendered by using the runtime xulrunner – this is the most common way and applications like Firefox are started like this – or by opening the XUL apps in Firefox.
2.4 Message Handling with SOAP

SOAP (Simple Object Access Protocol) is a network protocol relying on XML for passing information over a network between clients and a server.

The SOAP specification defines a messaging framework which normally consists of four parts:

- The processing model defining the rules for processing a SOAP message.
- The extensible model defining the concepts of features and modules.
- The underlying protocol binding for defining a binding to the underlying protocol like HTTP.
- The message construct defining the structure of a SOAP message.

The following example shows a SOAP message that is used for calling a function `GetStockPrice` – defined on the server side – with the parameter IBM. When the server has called the function, it can send back an answer in another SOAP message which the client can process, and based on which the client can, e.g., dynamically change the content of the GUI.

```xml
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header/>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">  
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```
2.5 **FNQUERY and FNTRANSFORM**

For the transformations in our framework, we extensively use the XML query, transformation and update language **FNQUERY** [10, 11], which is fully integrated in **SWI PROLOG**. Like in XPATH, it is possible to query complex structures with path expressions and axes. As an extension of XPATH, it is possible to select multiple branches over deeply nested structures. The sublanguage **FNTRANSFORM**, which extends XSLT, gives the user the feasibility to transform XML elements in **PROLOG**.

**FNQUERY** uses triples for representing XML documents. E.g., for the association list \( As = \{\text{color:red, model:civic}\} \) of attribute/value pairs, \( \text{cars:As:Es} \) represents an XML element with the tag \( \text{cars} \); the content \( \text{Es} \) can be a (possibly empty) list of triples.

The path language **FPATH** of **FNQUERY** is very similar to XPATH. Compound terms with the functor “/” are used for selecting subelements. The functor “@” is used for selecting the value of an attribute. E.g., the binary predicate “:=” in the call

\[ ?- M := \text{doc(cars.xml)/car}@model. \]

selects the value for the attribute \( \text{model} \) from the element \( \text{car} \) in the XML document \( \text{cars.xml} \) below and binds the result to \( M \).

\[
\begin{align*}
\text{<cars>} \\
\quad \text{<car id="corolla" model="Corolla" /}> \\
\quad \text{<car id="civic" model="Civic" /}> \\
\quad \text{<car id="city" model="City" /}> \\
\text{</cars>}
\end{align*}
\]

It is even possible to query with *multiple* location paths. The following expression selects the attributes \( \text{id} \) and \( \text{model} \) and forms pairs \( [\text{Id}, M] \) of the results:

\[ ?- \text{Pair := doc(cars.xml)/car-[@id, @model].} \]

3 **The Framework XUL4Pl for GUI Programming**

For declarative GUI programming and native **PROLOG** rule implementation, we have developed XUL4Pl as a thin client framework. It is fully integrated into our **SWI PROLOG** framework, and it can be accessed online. The framework’s HTTP server itself is implemented in **PROLOG** as well as functions for easy-to-use GUI and database handling. The user interface is based on XUL, for which we are using an extended version for communication and data exchange between client and server. We also use four JavaScript functions for calling the server and for sending and retrieving information. For defining the GUI, we use an extended version of XUL for which we have implemented features for data communication and easy-to-use GUI programming.

The following subsections describe our framework XUL4Pl in detail with respect to the implementation of the HTTP server. After this, we give an overview of the database support, database driven GUI scaffolding and updates to the user interface with database triggers. An advanced method for message handling with SOAP is described afterwards.
3.1 HTTP Server Connection

The XUL4Pl HTTP server is completely implemented in PROLOG using the HTTP support package of SWI PROLOG, implemented by Jan Wielemaker [13]. With the package, it is possible to handle data requests with GET and POST methods; even JSON data structures are possible.

In our framework, we have implemented four different handlers for processing data from the client or sending requested data, which are integrated into the DOM of the XUL-GUI. Our approach is dealing with POST data, which the server can process with the following handler:

```prolog
handle(Request) :-
    member(method(post), Request),
    post_xml_to_fn_term(Request, FN),
    format('Content-type: text/xml; charset=utf-8\n\n'),
    format('<temp xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">'),
    apply_goal_from_fn(FN),
    format('</temp>').
```

Whenever a JavaScript function for the communication with the server is called on the client side, this handler parses the given information, i.e., the predicate to call with all the parameters like values from GUI input fields or constant values. With this, it is possible to call nearly all PROLOG rules from the client. The data from the client is sent in an XML message envelope containing the data for the predicate which the server has to call.

```xml
<message>
  <goal>predicate name</goal>
  <parameter>parameter 1</parameter>
  <parameter>parameter 2</parameter>
  ...
  <parameter>parameter n</parameter>
</message>
```

The predicate `apply_goal_from_fn/1` reads the message and retrieves the information for the goal, which it has to call, together with the additional parameters. The parameters can be different values from the elements of the XUL file or complete XUL documents, which can be processed themselves. We extensively use FNQUERY to parse the information given by the message.

3.2 Database Programming

A more efficient way of defining thin clients with XUL4Pl is to connect them with a database. In this subsection, we describe how to connect our framework with a database, GUI scaffolding and the use of database triggers.
Database Statements. We have extended the list valid of XUL attributes to specify additional information needed like the database table name (db_table) and table column (db_attribute) where the data has to be stored or read:

When a GUI update is generated or the data are submitted to the server, then the XUL document structure will be parsed. E.g., to insert information into the database, for all XUL elements with the extended database attributes, we automatically call the predicate xul_item_to_insert_or_update_statement/4.

\[
\text{xul\_item\_to\_insert\_or\_update\_statements}(\text{Connection}, \text{Database}, \text{Xul}, \text{Statements}) \\
\text{:- mysql\_database\_schema\_to\_xml}(\text{Connection}, \text{Database}, \text{DB\_Schema}), \text{xul\_form\_to\_inserts}(\text{Xul}, \text{Items}), \text{maplist}(\text{fn\_item\_to\_insert\_or\_update\_statement}(\text{Connection}, \text{DB\_Schema}, \text{Database}), \text{Items}, \text{Statements}).
\]

The predicate reads the used database schema from the data dictionary and checks the document with xul_form_to_inserts/2 for the used attributes db_database and db_table and processes the data stored in the XUL file, e.g., input fields, radio buttons and drop down menues.

For XUL code above, the predicates parse the XML structure and generates SQL statements. It also automatically checks for the defined primary key and the assigned values.

\[
\text{xul\_form\_to\_inserts}(\text{Xul}, \text{Item}) \\
\text{:- D = descendant\_or\_self}, \text{findall}(\text{B}\text{;V}, ( \text{X := Xul/D::*[@db\_table=Table]}, \text{Y := X/X/D::*[@db\_attribute=Attribute]}/D::Tag, \text{Tags = [listitem, menuitem, richlistitem, radio]}),
\]

121
Afterwards, for all the processed data derived in the first step, the following predicate \texttt{fn\_item\_to\_insert\_or\_update\_statement/5} generates corresponding SQL statements. If a database row with the given value for the primary key already exists, then an update statement will be generated, otherwise an insert.

\begin{verbatim}
fn_item_to_insert_or_update_statement(Connection, DB_Schema, Database, Item, Statement) :-
  ( ( \+ fn_item_includes_primary_key(DB_Schema, Item)
  ; \+ fn_item_primary_key_is_in_database(
    Connection, DB_Schema, Database, Item) ) ->
    fn_item_to_insert_statement(
      Connection, Database, Item, Statement)
  ; fn_item_to_update_statement(
    Connection, DB_Schema, Database, Item, Statement) ).
\end{verbatim}

Reading data from the database and generating XUL elements is also possible with XUL4PL and FNQUERY.

**Database Driven GUI Scaffolding.** An elegant way of defining user interfaces with XUL4PL is to use GUI scaffolding. We have implemented many predicates to automatically generate input elements based on the underlying database. In the following example, we will explain such a predicate, namely \texttt{odbc\_attribute\_to\_fk\_menulist/4}. Its arguments are the database, the table, and the attributes to be used. Attribute and Table are given as PROLOG terms representing XML structures, so-called FN-triples, because they can also be derived automatically for what we use XML in general.

\begin{verbatim}
odbc_attribute_to_fk_menulist(
  Connection, Database, Table, Attribute, Menus) :-
  A_Name := Attribute@name,
  T_Name := Table@name,
  [A_Name, Fk_Table] := Table/foreign_key-
                   [/attribute@name, /references@table],
  mysql_use_database(Connection, Database),
\end{verbatim}
The predicate resolves the foreign keys given in Table and generates a menulist with all possible values. For these foreign key select menus, the referenced tables are read and only valid values are presented in the menu; the user cannot enter wrong data. In addition to the different foreign key values, it is also possible to include other values from the referenced table in the menu; the generated dialog is more readable.

The range of implemented predicates is huge: we have, for example, implemented the generation of trees used in applications like for file trees, complete input dialogs and wizards, as well as simple input types like listboxes, radio buttons and checkboxes.

Figure 3 shows an automatically generated XUL dialog. For the dialog, the database schema is read and drop-down menus with already assigned values corresponding to the foreign keys and input dialog are derived by our rule set.

![Automatically generated XUL dialog](image)

**Figure 3.** Automatically generated XUL dialog.

**Database Triggered GUI Updates.** Thin clients have the advantage, that they can be used in multi-user environments. When more than one user is working with such an application and data is transferred to the server, all the other apps have to recognize the change of data and have to be updated. Therefore, we use database triggers to react on such data changes.
When we insert into a database table, we call a predicate \texttt{xul\_odbc\_insert/3}, which generates the necessary insert or update statements and triggers events for the corresponding database table.

\[
xul\_odbc\_insert\text{(Connection, Database, Xul)} :- \\
\text{xul\_item\_to\_insert\_or\_update\_statements(} \\
\text{Connection, Database, Xul, Statements),} \\
\text{( foreach(Statement, Statements) do} \\
\text{ ( Statement = mysql\_insert\_tuple(} \\
\text{ Connection, EventDB:EventTable, _}) \\
\text{ ; Statement = mysql\_update\_table(} \\
\text{ Connection, EventDB:EventTable, _, _) ),} \\
\text{ call(Statement),} \\
\text{ ( xul\_trigger\_event(Connection, EventDB, EventTable),} \\
\text{ !} \\
\text{ ; writeln(user, noeventspec) ) ).}
\]

The \texttt{xul\_trigger\_event} predicate looks for registered listeners on all the acting databases. If a listener is registered with the call

\[
xul\_register\_event\_listener(+Type, +Goal, +Options)
\]

then it calls the named \texttt{Goal} with \texttt{Options}, if available. \texttt{Type} is used for consistency like it is known from SQL: e.g., \texttt{ON UPDATE CASCADE} checks if nested GUI parts should also be updated.

\[
xul\_trigger\_event\text{(Connection, Database, Table)} :- \\
\text{setof( Goal,} \\
\text{ ( xul\_event\_listener(Connection, Goal, Options),} \\
\text{ member(db(Database), Options),} \\
\text{ member(table(Table), Options) ),} \\
\text{ Goals ),} \\
\text{ ( foreach(Goal, Goals) do call(Goal) ).}
\]

### 3.3 Advanced Data Handling with SOAP

In the subsection above, we have described data handling with a short message envelope for sending and retrieving data. In modern client/server applications, it is common to use SOAP as a message format.

Therefore we have implemented a SOAP interface for the communication. The example below shows such a possible message format:

\[
<\text{soap:Envelope} \\
\text{ xmlns:soap="http://www.w3.org/2003/05/soap-envelope">} \\
<\text{soap:Header}/>
<\text{soap:Body}> \\
<\text{m:calculate\_storage\_price}
\]

124
The message lets the server call `calculate_storage_price(2012-08-01, 5)`. The server itself can now process the data, and if necessary it could answer with another SOAP message.

4 Implementation of a Client/Server Application

For a case study on how to use XUL4PL and to test robustness and effectiveness, we have implemented a client/server application for a multi-user environment. The server itself is installed on a Ubuntu Linux machine, the clients are running under Windows, Linux and Mac OS X.

![Figure 4](image)

**Figure 4.** An interface implemented with XUL. The data is derived from a MySQL database. The used OS is Mac OS Mountain Lion.

Figure 4 shows a part of a resource planning system, which we have implemented with XUL4PL. The application consists of about 40 different dialogs, tabs, and windows – nearly all of them are generated automatically. The code for data retrieval could also be reduced to a minimum with our extended version of XUL. The code for the client/server application has about 2,800 lines of XUL code, the PROLOG code could be
reduced to only 3,000 lines with database techniques like GUI scaffolding. The lines of code of the framework itself is here excluded.

We have performed a stress test, where the database stored about 80,000 rows and 20 different users worked simultaneously with the client.

5 Conclusions and Future Work

In this paper, we have introduced our framework XUL4PL for generating declarative thin client applications with a declarative GUI description language, and the combination of PROLOG as a backend-server.

With our framework, a developer has no need to implement new JavaScript functions but can focus on the definition of the PROLOG predicates for handling the application logic. The GUI can easily be defined with XML, and our extended version of XUL gives the user – in combination with our predicates for easily handling the design and behaviour – the opportunity to rapidly program user interfaces with the look-and-feel of the client’s operating system. With the combination of XUL and ODBC, storage and retrieval of information with databases are easy to use and fast to implement. GUI scaffolding and database triggered event handling are one of our main features, which help to produce highly reliable code. Messages can be sent to both client and server with a short message format as well as with SOAP.

We have tested our framework with a resource planning system, which is now used with a MySQL and PostgreSQL database and Linux as the server, the clients are currently running under Windows, Mac OS and Linux.

In the future, we will test the database connectivity with other databases like Oracle, DB2, and Microsoft SQL Server. Another feature will also be the integration of PROLOG Server Faces (PSF), such that we can easily switch between different kinds of user interfaces.

References


Extending Object-Oriented Languages by Declarative Specifications of Complex Objects using Answer-Set Programming

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. Many applications require complexly structured data objects. Developing new or adapting existing algorithmic solutions for creating such objects can be a non-trivial and costly task if the considered objects are subject to different application-specific constraints. Often, however, it is comparatively easy to declaratively describe the required objects. In this paper, we propose an approach for instantiating objects in standard object-oriented programming languages. In particular, we extend JAVA with declarative specifications in terms of answer-set programming (ASP), a well-established declarative programming paradigm from the area of logic-based artificial intelligence, from which the required objects can be automatically generated using available ASP solver technology.

1 Introduction

Imagine one has to write an algorithm for solving the following problem: Given an array of network components of three different types, where each of the components has potentially multiple cable sockets, create an undirected network graph where each node contains a component. The number of edges incident to a node is limited by the number of sockets of the respective network component. Moreover, the total number of edges must not exceed a given limit, and every component has to be transitively reachable by every other component. Also, there must not be an edge between nodes with components of the same type. The graph is represented by instances of a class Node that store references to their adjacent nodes. Also, we want to identify a node with a maximal number of edges. It will probably require some thought to come up with an algorithm that produces a respective graph structure whenever there exists one. In general, developing an algorithmic solution is sometimes a non-trivial task when data structures need to be generated that are subject to complex constraints and algorithmic off-the-shelf solutions to obtain them are unavailable. Although it might be unclear how some desired objects can be created, it is in many cases easy to describe them, i.e., to

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.
state what is needed, as suggested by the description of the above problem (used as a running example in the remainder). Although such a situation is unpleasant when one sticks to a traditional imperative programming style, this is the perfect starting point for solving the problem using declarative programming.

Our goal is to integrate declarative object specifications in object-oriented programming languages that allow for obtaining the desired instances automatically. These specifications are especially beneficial in situations where

- no algorithm is known for producing the desired objects and the development of a new procedural solution is expensive;
- objects are needed that are subject to complex constraints;
- the programmer is confronted with changing requirements on the data structures to instantiate;
- rapid prototyping is needed, i.e., when an algorithmic solution is the final goal but not feasible before most requirements have been settled.

Our concrete proposal is to combine Java with answer-set programming (ASP), a well-established paradigm for declarative problem solving from the area of logic-based artificial intelligence (AI). The idea of ASP is to declaratively specify a computational problem in terms of a logical theory (that is, a logic program) such that the solutions of the problem correspond to the models (the “answer sets”) of the theory. Following the principle of declarative methods of separating the representation from the processing of knowledge, ASP solvers are used to compute the models of the specifications. ASP has its roots in nonmonotonic reasoning and allows for expressing constraints, recursive definitions, and non-determinism in a quite natural way. ASP has been used in a wide range of applications, including semantic-web reasoning [1, 2], bioinformatics [3, 4], software testing [5], planning [6], content generation for computer games [7], configuration [8], multi-agent systems [9], cladistics [10, 11], and super optimisation [12].

Adapted to the setting of this paper, the answer-set program corresponds to a specification of the desired objects, whereas the problem solutions are the objects themselves. ASP solvers allow for exhaustively generating all, a specified number, or a random sample of objects fulfilling the specifications. Here, using methods adapted from SAT solving, it is even possible to generate random solution structures that are near uniformly distributed in the search space, which is often considered a computationally challenging task.

Also, answer-set programs may contain optimisation expressions that allow to express preferences for certain objects. In the network example, we could search for graphs that involve only a minimal number of edges. Since ASP solvers implement complete search, we can also show that no object meets some specification. In view of the constant improvement of ASP solver technology, the use of ASP for declarative programming becomes increasingly attractive.

In order to avoid programming overhead that would come with calling an ASP solver as an external application within Java code, we propose to tightly integrate Java elements into the ASP language. This way, on the one hand, Java objects, arrays, or primitive data types, that serve as parameters for the specification can be automatically translated to input for a respective solver and,
on the other hand, the returned answer sets can be automatically interpreted to build up the desired Java data structures. In particular, our proposed formalism allows Java constructors, method calls, and objects to take the role of ASP predicates and terms. By means of those, the programmer may specify objects by describing an arrangement of constructor and method calls such that the resulting objects satisfy the constraints of the application.

The interface between the procedural and the declarative parts is realised by a Java method call that takes parameters of the specification and the number of desired solutions as arguments. In return, we get a collection of solution objects that meet the specifications. A proof-of-concept tool for our proposed specification language has been implemented.

2 Answer-Set Programming in a Nutshell

ASP has been proposed as a problem solving approach in the late 1990s, building on the stable-model semantics for logic programs [13] that is genuinely declarative—in contrast to, e.g., the semantics of Prolog. ASP solvers have become increasingly efficient in recent years. In fact, the solver CLASP [14] even outperformed state-of-the-art SAT solvers at the latest SAT competitions in several categories (see http://www.satcompetition.org).

ASP comes with high-level modelling capabilities that allow for specifying problems in an easy-to-read, compact, and elaboration-tolerant way, i.e., small variations in a problem description require only small modifications of the representation.

For a comprehensive introduction to answer-set programming, including formal definitions of syntax and semantics, we refer to the well-known textbook by Baral [15]. In the following, we sketch the basic ideas of ASP.

Roughly speaking, an answer-set program is a collection of rules like

\[
a(X?, Y?) :- b(X?, Y?), \text{not } c(Y?).
\]

where \(a(X?, Y?)\), \(b(X?, Y?)\), and \(c(Y?)\) are atoms that might be true or false, and \(X?\) and \(Y?\) are schematic variables that stand for an object from a domain. Sometimes, the symbol “." is used to denote a fresh variable not appearing anywhere else. The intuition of the rule is that, for all objects \(o_1\) and \(o_2\), if \(b(o_1, o_2)\) is true and it is not known that \(c(o_2)\) is true, then \(a(o_1, o_2)\) must be true. This understanding of the negation operator \(\text{not}\) is called default negation, or negation as failure, allowing to expressing non-determinism. Consider the following program:

\[
a(o) :- \text{not } b(o). \quad b(o) :- \text{not } a(o). \quad c(o) :- .
\]

It has two answer sets, \{a(o), c(o)\} and \{b(o), c(o)\}. For the first, as atom \(b(o)\) is not known to be true, the first rule is active and derives \(a(o)\), whereas the second rule is inactive since \(a(o)\) is known to be true. Symmetrically, for answer set \{b(o), c(o)\}, the second rule is active but the first one is not. The rule “\(c(o) :- .\)” is a fact stating that \(c(o)\) is unconditionally true.
Another type of rules are constraints that do not derive anything but are used for eliminating unwanted answer sets. E.g., the constraint “:-a(o).” expresses that a(o) cannot be true. If added to the program from above it would eliminate the answer set \{a(o),c(o)\}.

We often use special atoms called cardinality constraints that allow for reasoning about sets of atoms, e.g., the cardinality constraint

\[ 2\{\text{edge}(X?,Y?):X?<Y?\}\leq 4 \]

is only true if at least two but at most four edge atoms are true for which the first argument is smaller than the second argument. It is assumed that < is a comparison relation for all objects.

### 3 Main Approach

Figure 1 illustrates the basic idea of our approach for adopting the ASP paradigm for automatic object instantiation. The programmer only provides a declarative specification and parameter values that are automatically translated to an ASP program such that the resulting answer sets are in one-to-one correspondence with all objects satisfying the specification for the given parameters. Depending on the needs of the application, one can compute all or just a predefined number of solutions. Desired objects can then be instantiated automatically from the answer sets. Intuitively, we realise such a behaviour, on the one hand, by extending the domain over which we reason in ASP to JAVA objects and data values and, on the other hand, by providing special predicates and function symbols. The latter allow for accessing, creating, and returning JAVA objects and arrays, and also for invoking constructors and object methods in the specification. In the following, these new elements are illustrated by solving the example from the introduction.
Assume network components are instances of class `Component` that has the getter-methods `getNrSock()` and `getType()`, both returning positive integers, where the domain of the latter is \{1, 2, 3\}. Nodes are represented by instances of class `Node`:

```java
package example.graph;
public class Node {
    Component c;
    List<Node> nodes = new ArrayList<Node>();
    public Node(Component c){
        this.c=c;
    }
    public void addNode(Node node){
        nodes.add(node);
    }
    ... // getters/setters
}
```

The structure of the specification of the graph is as follows.

```java
package example;
import example.graph.*;
NetworkSpec(Component[] comps, int nrCables){
    ... // ASP code
}
```

Similar to regular Java class files, a specification belongs to a package and may import classes from other packages as needed. In the following, the missing ASP code is introduced and explained step-by-step.

The first rule is a fact that consists of a cardinality constraint. It defines the search space by guessing whether there is an edge between any two different components.

```
0 {edge(C1?,C2?) : C1? != C2? : C1?comps(_): C2?comps(_)} 1.
```

The first type of expression that extends standard ASP for use with Java is of form `C?comps(I?)`, a special atom that is true if the variable `C?` is assigned the object with index `I?` in the array contained in specification parameter `comps`. Consequently, `edge` is a relation between the `Component` objects from the input array.

Next, since we deal with an undirected graph, the following rules (in pure ASP) ensure the symmetry of edges and transitively compute reachability between components.

```
edge(C1?,C2?):- edge(C2?,C1?).
reach(C1?,C2?):- edge(C1?,C2?).
reach(C1?,C2?):- reach(C1?,H?),reach(H?,C2?).
```

For ensuring that the number of edges from a component exceeds its number of sockets, the next constraint is added.
The term \( \text{C1?.getNrSock()} \) stands for the value that is returned by the method \( \text{getNrSock()} \) of the object contained in variable \( \text{C1?} \). The intuitive reading of the entire rule is that for any object \( \text{C1?} \) with an arbitrary index in parameter array \( \text{comps} \) it cannot hold that the number of edges to other objects \( \text{C2?} \) in \( \text{comps} \) is greater or equal to the number of sockets of \( \text{C1?} \) plus 1.

Next, the number of edges is restricted to the value of the integer parameter \( \text{nrCables} \).

\[
\text{new Node(\text{C1?): \text{C1?comps}_(), \text{C1?.getNrSock}() = \text{C2?.getNrSock}())}.
\]

From a logical point of view the problem is solved here. What remains is the declarative specification of how \( \text{Node} \) objects should be instantiated and configured and how to determine a return value.

For every \( \text{Component} \) object \( \text{C1?} \), we derive an atom \( \text{new Node(\text{C1?)}} \) representing a respective constructor call with the component as argument, and hence the instantiation of a \( \text{Node} \) object containing the component. Objects created this way can be referenced in other rules in a similar fashion as the elements of the \( \text{comps} \) array. In particular, an atom \( \text{N1?Node(\text{C1?)}} \) is true if \( \text{N1?} \) is the object created by constructor call \( \text{new Node(\text{C1?)}} \). The last rule states that for two nodes \( \text{N1?} \) and \( \text{N2?} \) whose components are connected by an edge, the \( \text{addNode(Node node)} \) method of \( \text{N1?} \) should be invoked with \( \text{N2?} \) as argument. This is expressed by the use of special atom \( \text{exe N1?.addNode(N2?)} \) representing a method invocation that is automatically executed after the specified objects have been created.

It is required that one of the created nodes is returned that has the highest number of edges. As there might be several, we choose the minimal according to the order \( < \) which is a strict total order on the domain that is available in most answer-set solvers.

\[
\text{nrEdges(\text{C1?,Nr?): \text{Nr?} = \{edge(\text{C1?, C2?): C2?comps(\_)}; \text{C1?comps(\_)}} \}
\]

\[
\text{notReturn(\text{C1?): \text{nrEdges(\text{C1?,Nr1?), \text{Nr1? < Nr2?; \text{nrEdges(\text{C2?,Nr2?)}}}}}}} \}
\]

\[
\text{notReturn(\text{C1?): \text{nrEdges(\text{C1?,Nr?), \text{C1? > C2?; \text{nrEdges(\text{C2?,Nr?)}}}}}}} \}
\]

\[
\text{return N?: \text{N?Node(\text{C?)}}), \text{not notReturn(\text{C?)}}} \]

This completes the rules that suffice to describe the desired objects from the example problem. Next, we show how to access the specification from Java code to obtain the specified graph.

```java
Component[] comps = {c1,c2,c3,c4,c5,c6};
NetworkSpec spec = new NetworkSpec();
spec.evaluate(comps,9, 1);
if(spec.hasSolution()){
    Node res = (Node)spec.getSolutions().get(0);
}
```

First, the array `comps` is created that contains six components assumed to be initialised earlier. Then, an object representing the `NetworkSpec` specification is instantiated. We call its method `evaluate` that takes as arguments the parameters of the specification, `comps` and `nrCables`, and an additional `int` parameter determining the number of desired solutions, here 1 (0 stands for all). The `hasSolution` method checks whether a desired graph exists, which might not be the case for some amounts of sockets and types of components. If one exists, it is assigned to the variable `res`. Note that the problem could be solved in only 23 lines of combined ASP and Java code.

There is also the possibility to employ optimisation statements for ASP solvers, e.g., adding the statement “#minimize{edge(C1?,C2?)}.” to the specification allows for searching solutions with a minimal number of edges. Moreover, if random objects are needed, techniques from SAT can be used to get a near uniformly distributed selection from the set of all specified objects [16].

We developed a prototype implementation of our approach which translates a specification into a Java class (the `NetworkSpec` class in the example). Instances of that class realise the dynamic creation of complex objects that satisfy the specification at any point of the execution of the Java program, i.e., in response to results of other computations and user input. To this end, it invokes Clasp as an external solver, converts given parameters into facts, and instantiates and configures the desired objects according to the obtained answer sets. Here, Java objects are internally represented by automatically generated identifiers.

Features that are already supported but not part of the example is the creation of arrays from specified or parameters objects, nested constructor and method calls, and the possibility to specify an order of execution for calls using an `exe` atom.

### 4 Related Work

Constraint programming (CP) is a declarative programming paradigm often used by imperative languages through respective CP libraries. Typically, constraints are formulated for variables over primitive data type domains. The embedding in object-oriented languages is mostly realised by wrapper classes for variables, constraints, and solvers. Output in CP is given as vectors of variable assignments.
satisfying the constraints, which is opposed to structured information in answer sets that we exploit for building up complex objects.

Declarative specification of complex structures is often used for testing. E.g., ALLOY [17] is a declarative first-order language for specifying objects for bounded-exhaustive testing, i.e., all objects that do not exceed a given size are used as test input for a piece of code. The TestEra framework [18] provides means to translate ALLOY instances to JAVA objects. As ALLOY structures are generated offline, specifications do not allow for runtime parameters as in our approach, other than size limitations. Moreover, as the target is to consider all solutions up to some size, there is no support for getting optimal ones. Another JAVA test input generator is KORAT [19], which is, however, based on procedural specifications. Object trees are generated as solution candidates which are then checked against a checking method that accepts when the structure is a solution. Hence, individual constraints cannot help to prune the search space. Indeed, we see testing as one application for our combined language, e.g., red-black trees can be concisely specified in our approach that are a popular example for test input generation in the testing literature.

5 Future Prospects

There are many ways to continue work from here. For one, the current specification language still misses important JAVA features, like direct field assignments and static method calls that could be easily integrated.

Currently, when the result of a call to an object method needs to be considered for solving, like when using C1?.getNrSock(), the objects considered for C1? has to exist before solving. Moreover, in our current implementation, we need to state that getNrSock() has to be precomputed for all objects in the comps array. The reason is that potential values for C1? are determined only during solving. Here, allowing the solver to interact with the JAVA-runtime can help. In this respect, a tight integration of a solver and a virtual machine would be desirable as this not only allows for executing JAVA code during solving, but also for exploiting the same data structure, e.g., pointers of objects as their identifiers, and reducing overhead for external calls.

Another possibility is to develop a tool that translates specifications without parameters into JAVA code generating the specified objects without calling a solver at runtime.

We understand this work as a first proposal towards a tight integration of JAVA and ASP. One issue that is—in our perspective—very important and requires extensive effort is the design of a simpler syntax for specifications that is intuitive for programmers familiar with JAVA. Currently, the semantics of our approach is implicitly given by that of JAVA, Clasp, and our prototype. The definition of a formal semantics is left for future work.

In conclusion, we see a high potential for significantly reducing the effort that has to be spend in a software project for writing and testing involved imperative code by integrating declarative specifications.
References

A Denotational Semantics for Weak Encapsulation in Curry

Jan Christiansen¹, Michael Hanus¹, Fabian Reck¹, and Daniel Seidel²

¹ Christian-Albrechts-Universität Kiel, Institut für Informatik, Olshausenstraße 40, D-24098 Kiel, Germany
{jac,mh,fre}@informatik.uni-kiel.de
² Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik, Römerstraße 164, D-53117 Bonn, Germany
ds@iai.uni-bonn.de

Abstract. Encapsulated search is a key feature of (functional) logic languages. It allows to access and process different results of a non-deterministic computation within a program. Unfortunately, there is no straightforward definition of the semantics of encapsulated search due to the advanced operational features of functional logic languages (lazy evaluation, partial values, infinite structures). As a consequence, various proposals and implementations are available but a rigorous definition covering all semantical aspects does not exist. In this paper, we analyze the requirements of encapsulated search in a functional logic language like Curry and provide a comprehensive definition that covers weak encapsulation, a natural evaluation-order independent form of encapsulation, and nested applications of search operators. Thus, this semantics is the basis to implement search operators in functional logic languages.

1 Introduction

Functional logic languages combine the most appealing features of the functional world, like algebraic data types, higher-order and lazy evaluation, with features from the logical world, like non-determinism and free variables (see [3,12] for recent surveys). The functional logic language Curry [17], which is considered in this paper, is based on a demand-driven (lazy) evaluation strategy that is optimal for a wide class of programs [2]. Due to its logic programming features, an expression in Curry might have multiple results that are non-deterministically evaluated. These results are shown in typical Curry systems in a read-eval-print-loop (REPL): the user enters an expression and the system displays the computed values in some order determined by the search strategy. For instance, the Curry system KiCS2 [5] supports depth-first, breadth-first, iterative deepening, or parallel search strategies.

In actual applications, however, the computation of non-deterministic values should be encapsulated: instead of processing (and eventually printing) all results of a subcomputation, one wants to select a single result, e.g., the “best” one, such as computing the shortest itinerary among all possible itineraries between two cities [4]. Therefore, many non-deterministic languages provide primitives to compute the set of all non-deterministic results of an expression or goal. In this way non-deterministic results can
be related, e.g., by computing the smallest of the results or by accumulating them. For instance, Prolog has a primitive \texttt{findall} \cite{21} to compute the list of all answers to a goal.

In this work we specify the semantics of an encapsulation primitive \texttt{allValues} so that \texttt{allValues}(e) denotes the set of all values of the expression \( e \). However, the precise meaning of this primitive is not obvious due to the combination of laziness and non-determinism:

1. Does the result set contain fully evaluated terms or only head normal forms?
2. Which non-deterministic computations are encapsulated? For instance, consider
   \begin{verbatim}
   let \texttt{x = e1 in allValues(e2)}
   \end{verbatim}
   where \( x \) occurs in \( e_2 \) and both \( e_1 \) and \( e_2 \) result in non-deterministic computations. Is the non-determinism of \( e_1 \) encapsulated by \texttt{allValues}(e2)?
3. If there are nested applications, as in \texttt{allValues(e1 allValues(e2) e3)}, which non-deterministic computations are encapsulated by which search operator? In particular, if some subcomputation fails, how does it effect the encapsulation?

Lazy evaluation complicates the answers to these questions. For instance, the evaluation of \( e_2 \) might trigger the evaluation of some subexpression of \( e_1 \) caused by shared variables. The various proposals for encapsulating search in functional logic languages (e.g., \cite{4,7,8,16,18,19}) demonstrate that there are no clear answers to these questions.

In order to define the intended results of expressions and to be independent of implementation details, we propose a denotational semantics for the encapsulation primitive \texttt{allValues}, inspired by a denotational semantics without capsules \cite{10}. As we will see, a precise description of nested capsules (item 3 above) requires the introduction of different failure values in the semantical domain so that one can distinguish between non-termination and various sources of failures. This distinction is not made by semantical descriptions without capsules (e.g., \cite{11}) where failure and non-termination are often identified.

Our semantics helps to understand the detailed meaning and semantical consequences of such a primitive, e.g., we can show that particular program transformations used in the implementation of purely functional languages are no longer valid in the presence of such an encapsulation primitive. Furthermore, our semantics is the basis to implement advanced search operators in the recent Curry implementation KiCS2 \cite{5}.

The main focus of this paper is to provide a formal foundation for encapsulated search in a functional logic language. Therefore, we introduce a kernel language for Curry programs that features primitives to calculate and test the set of non-deterministic results of a computation (Section 2). After discussing the intended behavior of encapsulated search (Section 3), we define its denotational semantics in Section 4. Section 5 shows our semantics at work before we conclude in Section 6.

2 (Flat) Curry

The syntax of Curry is very similar to the syntax of the functional language Haskell \cite{23}. In contrast to Haskell, Curry additionally provides free variables and the means to introduce non-determinism. However, many tools that process Curry programs, such
as compilers and analyzers, actually work on a kernel language, called Flat Curry, into which all Curry programs can be translated by eliminating syntactic sugar from source programs. Christiansen et al. [10] present a semantics for a variant of this intermediate language, called TFLC.

On the one hand, we extend TFLC with a construct for encapsulation, a primitive data type for sets and operations on sets. On the other hand, we exclude polymorphic types and restrict let-expressions to bind only a single variable. These restrictions are reasonable since polymorphism does not add extra insights and non-recursive let-expressions with more than one binding can easily be split up into a series of bindings. Furthermore, Christiansen et al. [10] point out that recursive let-expressions cannot be handled by a compositional semantics like the one we head for. We may also omit let-expressions completely because they can be simulated by function calls, but they are useful for example calculations. Thus, the syntax of our extended kernel language TFLC is defined as follows.

\[
\begin{align*}
\tau & ::= \tau_1 \to \tau_2 \mid \mathbb{N} \mid \text{Bool} \mid \{\tau\} \\
P & ::= D \mid P \mid e \\
D & ::= f : \tau; \ f(x_n) = e \\
e & ::= x \mid n \mid e_1 + e_2 \mid \text{Nil}_\tau \mid \text{Cons}(e_1,e_2) \mid \text{True} \mid \text{False} \mid e_1 \ ? \ e_2 \\
& \quad \mid \text{case } e \text{ of } \{\text{True } \to e_1; \text{False } \to e_2\} \\
& \quad \mid \text{apply}(e_1,e_2) \mid \text{unknown}_\tau \\
& \quad \mid \text{case } e \text{ of } \{\text{Nil } \to e_1; \text{Cons}(x,xs) \to e_2\} \\
& \quad \mid \text{let } x : : \tau = e_1 \text{ in } e_2 \mid \text{failed}, \\
& \quad \mid \text{allValues}_\tau(e) \mid \text{size}_\tau(e) \mid \text{isEmpty}_\tau(e)
\end{align*}
\]

Similarly to TFLC, \(\tau\) denotes a type, \(P\) a program, \(D\) a function definition, and \(e\) an expression. Furthermore, \(x\) stands for an expression variable and \(n\) for a natural number. A bar, as in \(x_n\), denotes the sequence \(x_1, \ldots, x_n\). An expression of the form \((e_1 \ ? \ e_2)\) denotes a non-deterministic choice between \(e_1\) and \(e_2\), \(\text{apply}(e_1,e_2)\) expresses the application of a functional expression \(e_1\) to an argument \(e_2\) (to cover higher-order features), \(\text{unknown}_\tau\) denotes a free variable of type \(\tau\) and \(\text{failed}\), a finite failure. However, some restrictions apply: free variables of types involving \(\to\) and \(\{\tau\}\) are not permitted and, as already told, let-expressions have to be non-recursive.

To deal with encapsulated search, we extend TFLC by the primitive \text{allValues}\ and the operations \text{size} and \text{isEmpty} on sets. Further set operations may be added provided that they do not allow to observe the order of the elements of a set. We disallow sets of functional types, since implementations are not able to test functions for (extensional) equality. Therefore, \text{allValues} may not be applied to an expression of functional type.

The typing rules for the additional primitives of TFLC\(^e\) are given by Figure[1] The first rule states that the encapsulation primitive that is applied to an expression of type \(\tau\) yields a result of type \(\{\tau\}\). The other two rules state that the set operations can only be applied to sets and yield a natural number and a Boolean, respectively. The typing

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash e :: \tau) &amp; (\Gamma \vdash \text{allValues}_\tau(e) :: \tau)</td>
<td>(\Gamma \vdash \text{size}<em>\tau(e) :: \text{Nat}) &amp; (\Gamma \vdash \text{isEmpty}</em>\tau(e) :: \text{Bool})</td>
</tr>
</tbody>
</table>
context $\Gamma$ provides type information for unbound variables. Typing rules for the remaining language constructs can be found in [10]. In example programs, we often omit type subscripts if they are determined by the context. As a further notational simplification, we write applications of a function name $f$ to an expression $e$ as $f(e)$ instead of $\text{apply}(f, e)$ (and similarly for more than one argument).

3 Requirement Analysis

As mentioned above, there are various proposals to encapsulate non-deterministic computations in functional logic programs, e.g., [4, 7, 8, 16, 18, 19]. The most recent approach [4] proposes set functions. With set functions, for every function $f$ in a Curry program, there is a function $f_S$ that yields the set of all results of $f$ applied to some deterministic input. Hence, set functions encapsulate only the non-determinism that is introduced by the function’s definition but not the non-determinism brought in via arguments. This way the result of a set function does not depend on the order of evaluation—a desirable property for declarative programming. Therefore, our approach should conform to these basic ideas of set functions, i.e., using our primitive $\text{allValues}$, for every Curry function $f(x_1, \ldots, x_n) = e$ we can define the corresponding set function as $f_S(x_1, \ldots, x_n) = \text{allValues}(e)$. However, as discussed below, the semantics of set functions that is presented in [4] is underspecified in the presence of finite failures and nested applications of set functions. Thus, a proposed implementation [6] of set functions does not yield the intended results for the motivating example of [4].

In order to obtain a comprehensive and reasonable definition of the meaning of $\text{allValues}$, we consider in this section the requirements concerning encapsulating non-deterministic computations.

3.1 Normal Form Encapsulation

Consider the following TFLC program where $\text{coin}$ denotes a non-deterministic value:

\begin{verbatim}
  coin :: Nat; coin() = 0 ? 1
  numCoinLists :: Nat; numCoinLists() = size(\text{allValues}(\text{Cons}(\text{coin}, \text{Nil})))
\end{verbatim}

What is the intended result of a call to $\text{numCoinLists}$? There are two possible choices. If $\text{allValues}$ does not evaluate the elements of the list $\text{Cons}(\text{coin}, \text{Nil})$ (since the values of the individual elements are not demanded), the non-determinism in $\text{coin}$ is not uncovered so that we have a set with only one list and, thus, 1 is returned as its size. If $\text{allValues}$ evaluates its argument completely (similarly to the top-level REPL of a Curry system), we yield a set with $\text{Cons}(0, \text{Nil})$ and $\text{Cons}(1, \text{Nil})$ as elements, and, thus, 2 is returned as its size.

The first option is called head normal form encapsulation. All elements of the set that $\text{allValues}$ returns are in head normal form (i.e., without a defined function as the outermost symbol) but not necessarily in normal form, i.e., completely evaluated. This form of encapsulation implies that the result of an encapsulated expression may still contain nested non-determinism, as seen by the example.
The second option is called *normal form encapsulation*. In this case the argument to `allValues` is fully evaluated, i.e., the set contains only deterministic values in normal form. Thus, all non-determinism is capsuled.

We choose the second option since this is conform with the “REPL view of non-determinism”: the values in an encapsulation set are also the values shown by the REPL of a Curry system. This option is also used for set functions [4]. Moreover, head normal form encapsulation might cause weird effects in combination with sharing when further evaluation of encapsuled entries is enforced. This could be possible if we allow to process specific elements from a set (which is not possible in our kernel calculus). Finally, we do not want to sweep a technical detail under the carpet: In the semantics we set up, we cannot model head normal form encapsulation.

### 3.2 Weak Encapsulation

A key feature of declarative languages is *referential transparency* [24], i.e., the value of an expression depends solely on the values of its sub-expressions. In particular, the evaluation of a sub-expression does not influence the result of another sub-expression. Referential transparency is most important in conjunction with lazy evaluation where the order of evaluation is not easily predictable by the user. Therefore, the demand that `allValues` should retain referential transparency is quite natural.

Braßel et al. [7] investigate various approaches to encapsulated search. They distinguish two concepts, strong encapsulation and weak encapsulation. These concepts differ when non-deterministic expressions are introduced outside an encapsulation primitive but evaluated inside. For example, consider the following TFLC\(^*\) program:

\[
\begin{align*}
\text{coin} &:: \text{Nat}; \quad \text{coin} = 0 ? 1 \\
\text{allCoin} &:: \{\text{Nat}\}; \quad \text{allCoin} = \text{let } x :: \text{Nat} = \text{coin} \text{ in } \text{allValues}(x)
\end{align*}
\]

The operation `allCoin` contains a non-deterministic sub-expression `coin` that is bound to `x`. Since `coin` textually occurs outside of the encapsulation primitive `allValues`, we will consider it as *introduced outside*. With *strong encapsulation*, any non-determinism that is evaluated by an encapsulation primitive is encapsulated, no matter where it is introduced. Thus, `allCoin` would yield the set \(\{0, 1\}\) w.r.t. strong encapsulation. The Curry system PAKCS [14] implements this kind of encapsulation.

In contrast, *weak encapsulation* [7] only encapsulates the non-determinism that is introduced inside the encapsulation primitive, e.g., the non-determinism in the expression `allValues(coin)` is considered to be introduced inside. In the case of weak encapsulation, `allCoin` yields non-deterministically one of the sets \(\{0\}\) or \(\{1\}\). The Münster Curry Compiler (MCC) [20] provides a primitive `findall` that implements weak encapsulation. However, a function like `allValues` cannot be defined by means of `findall`.

Braßel et al. [7] show that strong encapsulation violates referential transparency. While they restrict the use of encapsulation to preserve referential transparency, we opt to implement weak encapsulation. Again, our choice is in line with set functions [4].
3.3 Completeness of Results

Encapsulation is meant to provide access to the set of all results of an encapsulated expression. Thus, if \( v \) is a value (i.e., normal form) of some expression \( e \), then we have \( v \in S \) for some result \( S \) of \( \text{allValues}(e) \) and vice versa. Furthermore, we want two different results of an expression \( e \) to be in the same result \( S \) of \( \text{allValues}(e) \), if and only if they only differ by different choices made for non-determinism that was introduced inside \( e \). For example, the expression \( \text{coin} \) yields the results 0 and 1. Therefore, there has to be a result of \( \text{allValues(coin)} \) that contains 0 and a result that contains 1. No other elements are allowed. Since the non-determinism is introduced inside \( \text{allValues} \), both elements have to be in the same result set. Thus, \( \{0, 1\} \) is the only valid result of \( \text{allValues(coin)} \). However, if we consider the expression \( \text{let } x :: \text{Nat} = \text{coin} \text{ in } \text{allValues}(x) \), there also has to be a result that contains 0 and a result that contains 1. Furthermore, since non-determinism is introduced outside of \( \text{allValues} \), 0 and 1 need to be in different result sets. Thus, \( \{0\} \) and \( \{1\} \) are the only valid results of \( \text{let } x :: \text{Nat} = \text{coin} \text{ in } \text{allValues}(x) \). This requirement is also formally stated for set functions in [4].

3.4 Finite Failures

So far, the exact semantics of our primitive \( \text{allValues} \) is not completely specified. We stated which results have to reside in the same sets, yet sets may also be empty. Empty result sets are of special interest because they allow for programming with \textit{negation as failure}, similarly to logic programming [18]. Consider the following TFLC\textsuperscript{c} program:

\begin{align*}
\text{nilP} :: \text{Nat} \to \text{Bool} \\
\text{nilP}(x) & = \text{case } x \text{ of } \{ \text{Nil } \to \text{True}; \text{Cons}(y, ys) \to \text{failed}_{\text{Bool}} \} \\
\text{consP} :: \text{Nat} \to \text{Bool} \\
\text{consP}(z) & = \text{isEmpty}(\text{allValues(\text{nilP}(z)))}
\end{align*}

The predicate \( \text{nilP} \) yields \text{True} if its argument is the empty list and fails otherwise. The function \( \text{consP} \) is intended to yield \text{True} exactly if its argument is a non-empty list. It is implemented by means of negation as failure using the predicate \( \text{nilP} \). If \( \text{nilP} \) is applied to a non-empty list, it fails and, hence, the result of the encapsulated expression is the empty set so that \( \text{consP} \) yields \text{True}.

What happens if \( \text{consP} \) is applied to an expression \( fl \) whose evaluation fails? In a language with a demand-driven evaluation strategy, \( fl \) is not immediately evaluated but only when pattern matching is performed in the body of \( \text{nilP} \). This leads to two options:

1. Since the evaluation of \( \text{nilP}(fl) \) fails, the encapsulation primitive returns an empty set and, therefore, the evaluation of \( \text{consP}(fl) \) yields \text{True}.
2. Since the failing expression \( fl \) has been created outside the encapsulated expression, its failure is not covered by \( \text{allValues} \) so that the evaluation of \( \text{consP}(fl) \) fails.

Thus, failures of expressions created outside of an encapsulation primitive are not covered by \( \text{allValues} \), similarly to the behavior regarding non-determinism.
The first option leads to the problem that it is hard to tell whether a failure introduced outside an encapsulated expression will result in an empty set, since it is very difficult to reason about the control flow in a lazy language.

Therefore, we prefer the second option which also means that failures of expressions created outside allValues do not influence the result of the encapsulation. For instance, the result of \( \text{let } x :: \text{Bool} = \text{failed}_\text{Bool} \text{ in allValues}(x \ ? \ True) \) is the set \( \{\text{True}\} \) rather than a failure. Otherwise, the property of the completeness of results would be violated: the value True is a result of \( x \ ? \ True \), independently of the value of \( x \), and, thus, is should appear in a result set of allValues(\( x \ ? \ True \)).

Note that [4] does not specify the handling of failures, although it contains an example for programming with negation as failure that only works with our interpretation of failures in encapsulated computations. A precise definition of this interpretation of failures demands for a sophisticated semantics where different kinds of failures can be distinguished. We introduce this kind of semantics in the following section. As a positive side effect, we also specify the meaning of nested applications of search primitives, which is practically relevant but has not been formally covered in previous approaches for weak encapsulation.

4 Semantics

Christiansen et al. [10] presented a set-based denotational semantics for Flat Curry. We present a slightly modified version of this semantics and extend it with constructs for encapsulation. The semantics is based on a multialgebraic view on functions. That is, functions map single elements to sets of elements. This view models call-time choice as implemented in Curry. Furthermore, it closely corresponds to the CRWL [11] approach, which is a well established logical foundation for functional logic languages. The denotational semantics is advantageous for our setting in comparison with CRWL as expressions in the denotational semantics are already set-valued whereas CRWL only states how expressions can be rewritten to obtain valid results.

4.1 Semantics of Types

Curry provides angelic (“don’t know”) non-determinism, which is modeled by the use of the Hoare powerdomain. We restrict ourselves to continuous directed-complete partial orders (dcpos) as domains. The Hoare powerdomain of a dcpo is the complete lattice of all its non-empty Scott-closed subsets. For theoretical background, consult the survey on domain theory from Abramsky and Jung [2], in particular section 6.2, Theorem 6.2.13. For a continuous dcpo \( D = (D, \sqsubseteq) \), we denote its Hoare powerdomain by \( \mathcal{P}_H(D) \). Then infimum and supremum of \( M \subseteq \mathcal{P}_H(D) \) are defined by:

\[
\bigcap M = \{x \in D \mid \forall m \in M. x \in m\}
\]

\[
\bigcup M = \bigcap\{m \in \mathcal{P}_H(D) \mid \forall n \in M. n \sqsubseteq m\}.
\]
In our calculus, we interpret types as follows.

\[
\llbracket \text{Bool} \rrbracket = \{ \text{True}, \text{False} \} \perp
\]
\[
\llbracket \text{Nat} \rrbracket = \mathbb{N} \perp
\]
\[
\llbracket [\tau] \rrbracket = \text{lfp}(\lambda S. \{ [] \} \cup \{ a : b \mid a \in \llbracket [\tau] \rrbracket, b \in S \}) \perp
\]
\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \{ f : \llbracket \tau_1 \rrbracket \rightarrow \mathcal{P}_H(\llbracket \tau_2 \rrbracket) \mid f \text{ continuous} \} \perp
\]

This semantics of types slightly differs from the one introduced in [10]. First, we need no extra environment, because we abstain from polymorphism as mentioned before. Second, we directly add a \( \perp \)-element as least element to each dcpo via the lifting operator \((\cdot) \perp\), which is technically beneficial later on and is more similar to \([1]\). Regarding the order, the semantics of \text{Bool} and \text{Nat} are sets with the discrete order and \( \perp \) as least element. The semantics of lists is given as least fixpoint \((\text{lfp})\). Note that the entries of a list are elements (rather than sets of elements), and, as we are modelling a non-strict language, \( \perp \) is a valid entry as well. The order on lists is given by element-wise comparison. The order on the function space is point-wise and the functions need to be Scott-continuous (i.e., monotone and preserving suprema of directed sets) to ensure that the function space itself is a continuous dcpo.

We use \( \perp \) to model non-termination and do not use it to model finite failures because the semantics of the encapsulation primitive is supposed to distinguish non-termination from finite failure. Note that we have to distinguish failures that are introduced outside an encapsulated expression from those introduced inside. Since we also want to model nested encapsulations, we introduce new values for failures from different layers of encapsulation. These values are represented as \( \perp_i \) where \( i \in \mathbb{N} \). To obtain a dcpo, we impose an order on these values, defined by \( \perp_i \subseteq \perp_j \iff i \leq j \), and add an artificial greatest element \( \perp_{\infty} \). In order to keep our definitions simple, we identify \( \perp \) with \( \perp_0 \). Furthermore, we refer to \( \perp_i \) with \( i > 0 \) as \text{finite failure} in the following. Note that non-termination is the least element in the domain of failures and, therefore, holds less information than any finite failure.

As finite failures are supposed to be valid inputs for functions and elements in lists, we provide a modified version of the semantics of types that includes all the different \( \perp \)-values:

\[
\llbracket \text{Bool} \rrbracket^{\perp_{\infty}} = \{ \text{True}, \text{False} \} \perp_{\infty}
\]
\[
\llbracket \text{Nat} \rrbracket^{\perp_{\infty}} = \mathbb{N} \perp_{\infty}
\]
\[
\llbracket [\tau] \rrbracket^{\perp_{\infty}} = \text{lfp}(\lambda S. \{ [] \} \cup \{ a : b \mid a \in \llbracket [\tau] \rrbracket, b \in S \}) \perp_{\infty}
\]
\[
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket^{\perp_{\infty}} = \{ f : \llbracket \tau_1 \rrbracket \rightarrow \mathcal{P}_H(\llbracket \tau_2 \rrbracket) \mid f \text{ continuous} \} \perp_{\infty}
\]

In this semantics, the new lifting function \((\cdot) \perp_{\infty}\) extends a dcpo with finite failures and a least element. As before, \( \perp \) (which is short for \( \perp_0 \)) is added as least element. Additionally, \( \perp_i \) with \( i > 0 \) and \( \perp_{\infty} \) are added with the order given above. Note that finite failures are incomparable to any element of the unlifted dcpo. Figure 2 shows the structure of a dcpo after the lifting. In this modified type semantics, lists can also contain finite failures and functions are allowed to yield finite failures as results.
4.2 Semantics of Expressions

After adjusting the semantical domains, we are ready to assign meaning to expressions. As in \cite{10}, the semantic function $\llbracket \cdot \rrbracket_{\tau,n}^\sigma$ is a family of functions where $\sigma$ maps expression variables to semantic values. The index $\tau$ indicates the type of the expressions to which the semantic function is applied. Subsequently, we omit the type index if it is not relevant. The additional index $n$ identifies different levels of nested encapsulation.

For given $\sigma, \tau$ and $n$, $\llbracket \cdot \rrbracket_{\tau,n}^\sigma$ maps an expression of type $\tau$ (determined by type inference) to elements of $P_D(\llbracket \tau \rrbracket_{\perp,\infty}^\perp)$. Figure 3 shows the semantics for all TFLC expressions. In contrast to \cite{10}, the elements of the Hoare powerdomain may contain finite failures and must contain $\perp$. This entails also changes to the semantics of expressions.

As mentioned before, the elements of the Hoare powerdomain are Scott-closed sets. We ensure this property by the down-closure operation $\downarrow$. The down-closure is defined for every $A \subseteq D$, where $(D, \sqsubseteq)$ is a dcpo, as $A_{\downarrow} = \{ x \in D \mid \exists y \in A. x \sqsubseteq y \}$.

Because $\perp$ is now part of our dcpos, in contrast to the original semantics, a separate handling of $\sigma(x) = \perp$ is no longer necessary in the case of an expression variable. For the same reason, we now need to down-close the sets on the right-hand side of some of the equations. In non-failure cases, only $\perp$ and values including $\perp$ but no finite failures are added to the set. However, if there is a failure $\perp_j$ present, then all $\perp_i$, with $i < j$ are added. Since we can no longer rely on the property that the base operation $\perp$ is only applied to non-$\perp$ arguments, we use its strict extension $\perp^+$, which yields the leftmost failure if one of the arguments of $\perp$ is a failure and behaves like $\perp$ otherwise. A free variable, denoted by $\text{unknown}_\tau$, represents all values of its type. Therefore, we can simply use the type semantics defined earlier to specify the semantics of a free variable.

Note that we use the version of the type semantics that contains no finite failures, since a free variable that produces all kinds of finite failures would behave quite awkwardly. A finite failure with index $i$ would not necessarily stem from the $i$-th layer of encapsulation in this case.

Up to now, the index $i$ of the semantic function had no influence on the semantics of expressions. However, the semantics of $\text{allValues}$ depends on the index. We define:

$$\llbracket \text{allValues}(e) \rrbracket_{\sigma}^{\tau,i} = \begin{cases} S_{\downarrow} & \text{if } S = \text{nfs}_\tau(\llbracket e \rrbracket_{\sigma}^{\tau,i+1}) \text{ and } S \subseteq \{ \perp_i \}_{\downarrow} \\ \{ \{ \llbracket e \rrbracket_{\sigma}^{\tau,i+1} \} \}_{\downarrow} & \text{otherwise} \end{cases}$$

where, for every dcpo $(D, \sqsubseteq)$ and $A \subseteq D$, $(A)$ denotes the set of the compact (i.e., finite) elements in $A$ that are maximal in $D$ (w.r.t. $\sqsubseteq$). Furthermore, $\text{nfs}$ calculates normal
forms and is defined as the set lifting of \( nf \), where

\[
[nf]_\sigma^t = \{\sigma(x)\} \quad [n]_\sigma^t = \{n\} \quad [\text{True}]_\sigma^t = \{\text{True}\} \quad [\text{False}]_\sigma^t = \{\text{False}\} \quad [\text{Nil}]_\sigma^t = \{[]\} \quad [\text{Cons}(e_1, e_2)]_\sigma^t = \bigsqcup_{h \in ([e_1]_\sigma^t)} [h : t]_\sigma^t \quad [e_1 + e_2]_\sigma^t = \bigsqcup_{a \in \mathbb{N}} \bigsqcup_{b \in \{\bot\}} ([a + b]_\sigma^t) \quad [e_1 ? e_2]_\sigma^t = [e_1]_\sigma^t \cup [e_2]_\sigma^t \quad [\text{unknown}]_\sigma^t = [\tau] \quad [\text{failed}]_\sigma^t = \{\bot\} \quad [\text{apply}(e_1, e_2)]_\sigma^t = \bigsqcup_{e \in [e_1]_\sigma^t} \bigsqcup_{a \in [e_2]_\sigma^t} ([f \& a]_\sigma^t) \quad [\text{case}\ e\ \text{of}\ \{\text{Nil} \rightarrow e_1; \text{Cons}(x_1, x_2) \rightarrow e_2\}]_\sigma^t = \bigsqcup_{t \in [\sigma]} \begin{cases} \{\bot\} \downarrow & \text{if } t = \bot_j \\ [e_1]_\sigma^t & \text{if } t = [] \\ [e_2]_{[x_1 \mapsto t_1, x_2 \mapsto t_2]}_\sigma^t & \text{if } t = t_1 : t_2 \end{cases} \quad [\text{case}\ e\ \text{of}\ \{\text{True} \rightarrow e_1; \text{False} \rightarrow e_2\}]_\sigma^t = \bigsqcup_{t \in [\sigma]} \begin{cases} \{\bot\} \downarrow & \text{if } t = \bot_j \\ [e_1]_\sigma^t & \text{if } t = \text{True} \\ [e_2]_\sigma^t & \text{if } t = \text{False} \end{cases} \quad [\text{let}\ x : \tau = e_1\ \text{in}\ e_2]_\sigma^t = \bigsqcup_{t \in [\sigma]} [e_2]_{[x \mapsto t]}_\sigma^t
\]

![Fig. 3. Adjusted denotational semantics for TFLC expressions](image-url)

Let us recall the properties of allValues we are heading for (see Section 3). First of all, we do not want to encapsulate non-determinism that is introduced outside the capsule. The original semantics already satisfies this requirement. In Curry we can only introduce non-determinism outside the capsule by binding the non-determinism to a variable and using this variable inside the argument of allValues. However, in the semantics variables are bound to (deterministic) values by the environment \( \sigma \). Therefore, non-determinism from outside cannot have any effect inside a capsule.

As in the case of non-determinism, failures that are introduced outside of a capsule should have no effect on the encapsulation behavior. Failures are treated like values in the semantics. We use the index \( i \) to distinguish failures from the outside from those introduced inside the capsule. This index denotes the nesting level of encapsulation of the current expression. A finite failure \( \text{failed}_i \) is mapped to the element in the failure
domain that corresponds to this level, i.e., the one with the same index (cf. Figure 3).

To relate the nesting index $i$ to the actual nesting level, the encapsulated expression $e$ in $\text{allValues}(e)$ is evaluated with an increased index $i + 1$. The normal form function $\text{nf}$ is used to map a semantic value either to itself, if it is a normal form, or to the failure that is leftmost in its structure. If the resulting set contains only failures from outside the encapsulated expression, i.e., with indices less or equal to $i$, then the set representing the greatest of these failures is returned. Otherwise, the set containing the set of all maximal and compact elements of the resulting set is returned. There are two points worth noting. By using the normal form function $\text{nf}$, we fix the evaluation order that is used to evaluate expressions to normal form. Fixing the order is necessary since another order of evaluation may produce other kinds of finite failures and, thus, other results for encapsulated expressions. In the semantics we could also return all failures that are present, but this would lack a reasonable implementation. Furthermore, we now have actual sets as semantic values. Since the result of $\text{allValues}$ is intended to be a set, this is a natural way to model it. To fit these sets into our framework, we need to extend the semantics and the normal form function for the set type:

$$\|\{\tau\}\|_t = \mathcal{P}(\langle\{\tau\}\rangle), \quad \|\{\tau\}\|_t \downarrow = \mathcal{P}(\langle\{\tau\}\rangle) \downarrow, \quad \text{nf}_{\{\tau\}}(t) = t$$

To make the domain of sets a dcpo, we impose the discrete order on it. As before, $\bot$ is added as the least element and in the latter type semantics finite failures are added.

Last but not least, the semantics for the set operations $\text{size}$ and $\text{isEmpty}$ is missing. Their definitions are straightforward:

$$\|\text{size}(e)\|_t = \bigsqcup_{t \in [e]} \begin{cases} \{\bot\}\downarrow & \text{if } t = \bot \\ \{t\}\downarrow & \text{if } t \text{ is a finite set} \\ \{\bot\} & \text{otherwise} \end{cases}$$

$$\|\text{isEmpty}(e)\|_t = \bigsqcup_{t \in [e]} \begin{cases} \{\bot\}\downarrow & \text{if } t = \bot \\ \{\text{True}\}\downarrow & \text{if } t = \emptyset \\ \{\text{False}\}\downarrow & \text{otherwise} \end{cases}$$

Note that $|A|$ denotes the cardinality of set $A$.

### 5 Examples

Due to \cite{10}, Lemma 4.5 – 4.7, it suffices in most cases to calculate with the maximal elements of a Scott-closed set and to treat $\bigsqcup$ as set union. With this simplification we calculate some examples in the semantics. First, we compute the semantics of the expression $\text{allValues}(\text{coin})$:

$$\|\text{coin}\|_\emptyset = \{0\} \cup \{1\} \cup \{\bot\} = \{0, 1, \bot\}$$

$$\|\text{allValues}(\text{coin})\|_\emptyset = \{\{0, 1, \bot\}\}\downarrow = \{\{0, 1\}, \bot\}$$
As intended, the result is the down-closed set that contains the set \{0, 1\}. The weak encapsulation behavior is demonstrated by the following example:

\[
\begin{align*}
\text{let } x :: \text{Nat} = \text{coin} \text{ in } \text{allValues}(x) \\
&= \bigcup_{t \in \text{coin}} \text{allValues}(x)_{[x \mapsto t]} \\
&= \text{allValues}(x)_{[x \mapsto 0]} \cup \text{allValues}(x)_{[x \mapsto 1]} \\
&= \{}\{0\}, \{1\}\}\downarrow \cup \{\perp\} \\
&= \{}0, 1, \perp\}
\end{align*}
\]

In Section 3.4 we considered the encapsulation of failing computations and argued that the evaluation of the expression \text{consP(failed}_{\text{Nat}}) should fail. In order to calculate the semantics of \text{consP(failed}_{\text{Nat}}), we need to calculate the semantics of the function \text{nilP} applied to a variable that is bound to a finite failure from a lower level of encapsulation. Note that the finite failure has index \(i\) and the semantic function is indexed by \(i + 1\).

\[
\begin{align*}
\text{nilP}(z)_{[z \mapsto \perp]}^{i+1} &= \bigcup_{t \in \text{z}} \text{case } x \text{ of } \{\text{Nil} \rightarrow \text{True} ; \text{Cons}(y, ys) \rightarrow \text{failed}_{\text{Bool}}\}_{[x \mapsto t]}^{i+1} \\
&= \text{case } x \text{ of } \{\text{Nil} \rightarrow \text{True} ; \text{Cons}(y, ys) \rightarrow \text{failed}_{\text{Bool}}\}_{[x \mapsto \perp]}^{i+1} \\
&= \{\perp\}\downarrow
\end{align*}
\]

Since \(z\) is bound to a finite failure, the pattern matching in the body of \text{nilP} just returns this finite failure. When we apply our encapsulation primitive \text{allValues} to \text{nilP}(z), the failure is returned but not encapsulated since the failure is introduced outside the capsule:

\[
\begin{align*}
\text{allValues}(\text{nilP}(z))_{[z \mapsto \perp]}^i &= \begin{cases} 
S \downarrow & \text{if } S \subseteq \{\perp\}\downarrow \text{ and } S = \text{nf}_T(\text{nilP}(z)_{[z \mapsto \perp]}^{i+1}) \\
\{\text{allValues}(\text{nilP}(z))_{[z \mapsto \perp]}^{i+1}\}\downarrow & \text{otherwise}
\end{cases} \\
&= \{\perp\}\downarrow
\end{align*}
\]

Now we are ready to calculate the semantics of our initial expression:

\[
\begin{align*}
\text{consP(failed}_{\text{Nat}})_{\emptyset} &= \bigcup_{f \in \text{consP}} \bigcup_{a \in \text{failed}_{\text{Nat}}} (f \& a) \\
&= \text{isEmpty}(\text{allValues}(\text{nilP}(z)))_{[z \mapsto \perp]} \\
&= \begin{cases} 
\{\perp\}\downarrow & \text{if } t = \perp \\
\{\text{True}\}\downarrow & \text{if } t = \emptyset \\
\{\text{False}\}\downarrow & \text{otherwise}
\end{cases} \\
&= \{\perp\}\downarrow
\end{align*}
\]
As mentioned in the introduction, we can also show that particular program transformations used in purely functional languages are no longer valid in the presence of allValues. For instance, Peyton Jones et al. [22] argue that, for efficiency reasons, a compiler should be able to change the order of evaluation. In particular, the following two expressions should be interchangeable by a compiler:

```plaintext
    case x of
      (a,b) -> case y of
        (c,d) -> e
    case y of
      (c,d) -> case x of
        (a,b) -> e
```

The first expression enforces the evaluation of x before y. In the second expression the pattern matchings are switched and, thus, y is evaluated before x. A similar transformation is proposed in [13] to improve the performance of the Curry system KiCS2 [5]. However, we can show that this transformation is not semantics-preserving if calls to allValues occur in the transformed expressions. For this purpose, consider two binary functions \(g_1\) and \(g_2\). Both functions perform pattern matching by means of case-expressions on both of their arguments. The only difference is that \(g_1\) evaluates its first argument first while \(g_2\) evaluates its second argument first. Hence, we get

\[
\llbracket g_1(x,y) \rrbracket_{[x \mapsto \bot, y \mapsto \bot] + 1}^{i+1} = \{\bot_i\} \downarrow \quad \llbracket g_2(x,y) \rrbracket_{[x \mapsto \bot, y \mapsto \bot] + 1}^{i+1} = \{\bot_{i+1}\} \downarrow
\]

Consequently, exchanging \(g_1\) and \(g_2\) in a context with allValues might turn a correct result into a failure or vice versa. For example, we have (type annotations are omitted):

\[
\llbracket \text{let } x = \text{failed in allValues}(g_1(x, \text{failed}) \rrbracket_i^0
= \bigsqcup_{t \in \text{failed}} \llbracket \text{allValues}(g_1(x, \text{failed}) \rrbracket_{[x \mapsto t]}^{i}
= \llbracket \text{allValues}(g_1(x, \text{failed}) \rrbracket_{[x \mapsto \bot]}^{i}
= \begin{cases} S \downarrow & \text{if } S \subseteq \{\bot\} \downarrow \text{ and } S = \text{nf}(\llbracket g_1(x, \text{failed}) \rrbracket_{[x \mapsto \bot]}^{i+1}) \\ \{\langle \llbracket g_1(x, \text{failed}) \rrbracket_{[x \mapsto \bot]}^{i+1} \rangle \downarrow & \text{otherwise} \\ \{\bot\} \downarrow \end{cases}
\]

However, replacing \(g_1\) by \(g_2\) yields a different semantic value:

\[
\llbracket \text{let } x = \text{failed in allValues}(g_2(x, \text{failed}) \rrbracket_i^0
= \bigsqcup_{t \in \text{failed}} \llbracket \text{allValues}(g_2(x, \text{failed}) \rrbracket_{[x \mapsto t]}^{i}
= \llbracket \text{allValues}(g_1(x, \text{failed}) \rrbracket_{[x \mapsto \bot]}^{i}
= \begin{cases} S \downarrow & \text{if } S \subseteq \{\bot\} \downarrow \text{ and } S = \text{nf}(\llbracket g_2(x, \text{failed}) \rrbracket_{[x \mapsto \bot]}^{i+1}) \\ \{\langle \llbracket g_2(x, \text{failed}) \rrbracket_{[x \mapsto \bot]}^{i+1} \rangle \downarrow & \text{otherwise} \\ \{\emptyset\} \downarrow \end{cases}
\]
6 Conclusion and Related Work

In this paper we have presented the first formal semantics for a primitive to encapsulate non-deterministic computations in a weak manner. Weak encapsulation is intended to distinguish non-deterministic and failing computations inside and outside the capsule. This distinction is essential to obtain a semantics that is independent of an evaluation-order, an important requirement for declarative languages with a non-strict semantics. We analyzed the various alternatives and requirements (normal form encapsulation, completeness of results, weak encapsulation of non-deterministic and failing computations) before we proposed an encapsulation primitive with an appropriate denotational semantics. As an application of our semantics, we showed that program transformations used in purely functional languages are not semantics preserving in our extended language framework.

The Curry compiler MCC [20] supports a weak encapsulation primitive findall [19] but lacks a formal foundation. Moreover, Braßel et al. [7] point out that apparently equal expressions yield different results with this primitive.

An operational semantics for a strong encapsulation operator getSearchTree was presented in [7]. Since strong encapsulation is known to lack referential transparency in specific cases, the authors proposed to disallow these cases by means of program analysis. In a subsequent work [8], Braßel and Huch presented a simplified operational semantics for getSearchTree that features sharing across non-determinism.

Computations with failures in functional logic programs have also been considered in [18]. The authors define the semantics of an operator to check finitely failed computations by an extension of the rewriting calculus CRWL [11]. However, they do not provide an operator to collect all non-failing results as it is presented here.

Antoy and Hanus [4] proposed set functions to express encapsulated search. Set functions can be considered as weak encapsulation that is restricted to encapsulate the body of function definitions. This restriction does not limit their expressiveness but rather eases the comprehension of encapsulation. While the properties of set functions are the foundation for our work, Antoy and Hanus did not specify the semantics of set functions in the presence of finite failures or nested applications.

Braßel [6] presented an implementation idea for set functions. While this idea conforms with the specification of set functions, its treatment of finite failures does not conform with the intended results of the motivating example given by Antoy and Hanus [4]. Therefore, our work can also be interpreted as a complete specification of set functions that builds the foundation of a practical implementation available in the current implementation of KiCS2 [15]. Due to lack of space, details about this implementation have to be excluded.

References
